Making Databases Work

ACM Books

Editor in Chief

M. Tamer Özsu, *University of Waterloo*

ACM Books is a new series of high-quality books for the computer science community, published by ACM in collaboration with Morgan & Claypool Publishers. ACM Books publications are widely distributed in both print and digital formats through booksellers and to libraries (and library consortia) and individual ACM members via the ACM Digital Library platform.

Making Databases Work: The Pragmatic Wisdom of Michael Stonebraker

Editor: Michael L. Brodie
2018

The Handbook of Multimodal-Multisensor Interfaces, Volume 2: Signal Processing, Architectures, and Detection of Emotion and Cognition

Editors: Sharon Oviatt, *Monash University*
Björn Schuller, *University of Augsburg and Imperial College London*
Philip R. Cohen, *Monash University*
Daniel Sonntag, *German Research Center for Artificial Intelligence (DFKI)*
Gerasimos Potamianos, *University of Thessaly*
Antonio Krüger, *Saarland University and German Research Center for Artificial Intelligence (DFKI)*
2018

Declarative Logic Programming: Theory, Systems, and Applications

Editors: Michael Kifer, *Stony Brook University*
Yanhong Annie Liu, *Stony Brook University*
2018

The Sparse Fourier Transform: Theory and Practice

Haitham Hassanieh, *University of Illinois at Urbana-Champaign*
2018

The Continuing Arms Race: Code-Reuse Attacks and Defenses

Editors: Per Larsen, *Immunant, Inc.*
Ahmad-Reza Sadeghi, *Technische Universität Darmstadt*
2018

Frontiers of Multimedia Research

Editor: Shih-Fu Chang, *Columbia University*
2018

Shared-Memory Parallelism Can Be Simple, Fast, and Scalable
Julian Shun, *University of California, Berkeley*
2017

Computational Prediction of Protein Complexes from Protein Interaction Networks
Sriganesh Srihari, *The University of Queensland Institute for Molecular Bioscience*
Chern Han Yong, *Duke-National University of Singapore Medical School*
Limsoon Wong, *National University of Singapore*
2017

The Handbook of Multimodal-Multisensor Interfaces, Volume 1: Foundations, User Modeling, and Common Modality Combinations
Editors: Sharon Oviatt, *Incaa Designs*
Björn Schuller, *University of Passau and Imperial College London*
Philip R. Cohen, *Voicebox Technologies*
Daniel Sonntag, *German Research Center for Artificial Intelligence (DFKI)*
Gerasimos Potamianos, *University of Thessaly*
Antonio Krüger, *Saarland University and German Research Center for Artificial Intelligence (DFKI)*
2017

Communities of Computing: Computer Science and Society in the ACM
Thomas J. Misa, Editor, *University of Minnesota*
2017

Text Data Management and Analysis: A Practical Introduction to Information Retrieval and Text Mining
ChengXiang Zhai, *University of Illinois at Urbana–Champaign*
Sean Massung, *University of Illinois at Urbana–Champaign*
2016

An Architecture for Fast and General Data Processing on Large Clusters
Matei Zaharia, *Stanford University*
2016

Reactive Internet Programming: State Chart XML in Action
Franck Barbier, *University of Pau, France*
2016

Verified Functional Programming in Agda
Aaron Stump, *The University of Iowa*
2016

The VR Book: Human-Centered Design for Virtual Reality
Jason Jerald, *NextGen Interactions*
2016

Ada’s Legacy: Cultures of Computing from the Victorian to the Digital Age
Robin Hammerman, *Stevens Institute of Technology*
Andrew L. Russell, *Stevens Institute of Technology*
2016

Edmund Berkeley and the Social Responsibility of Computer Professionals
Bernadette Longo, *New Jersey Institute of Technology*
2015

Candidate Multilinear Maps
Sanjam Garg, *University of California, Berkeley*
2015

Smarter Than Their Machines: Oral Histories of Pioneers in Interactive Computing
John Cullinane, *Northeastern University; Mossavar-Rahmani Center for Business and Government, John F. Kennedy School of Government, Harvard University*
2015

A Framework for Scientific Discovery through Video Games
Seth Cooper, *University of Washington*
2014

Trust Extension as a Mechanism for Secure Code Execution on Commodity Computers
Bryan Jeffrey Parno, *Microsoft Research*
2014

Embracing Interference in Wireless Systems
Shyamnath Gollakota, *University of Washington*
2014

Making Databases Work

The Pragmatic Wisdom of Michael Stonebraker

Michael L. Brodie
Massachusetts Institute of Technology

ACM Books #22

Making Databases Work: The Pragmatic Wisdom of Michael Stonebraker

Michael L. Brodie, editor

books.acm.org
www.morganclaypoolpublishers.com

ISBN: 978-1-94748-719-2 hardcover
ISBN: 978-1-94748-716-1 paperback
ISBN: 978-1-94748-717-8 eBook
ISBN: 978-1-94748-718-5 ePub

Series ISSN: 2374-6769 print 2374-6777 electronic

DOIs:

10.1145/3226595	Book
10.1145/3226595.3226596	Foreword/Preface
10.1145/3226595.3226597	Introduction
10.1145/3226595.3226598	Part I
10.1145/3226595.3226599	Part II/Chapter 1
10.1145/3226595.3226600	Part III/Chapter 2
10.1145/3226595.3226601	Part IV/Chapter 3
10.1145/3226595.3226602	Chapter 4
10.1145/3226595.3226603	Chapter 5
10.1145/3226595.3226604	Chapter 6
10.1145/3226595.3226605	Part V/Chapter 7
10.1145/3226595.3226606	Chapter 8
10.1145/3226595.3226607	Chapter 9
10.1145/3226595.3226608	Part VI/Chapter 10
10.1145/3226595.3226609	Chapter 11
10.1145/3226595.3226610	Chapter 12
10.1145/3226595.3226611	Chapter 13
10.1145/3226595.3226612	Part VII/Chapter 14
10.1145/3226595.3226613	Part VII.A/Chapter 15
10.1145/3226595.3226614	Chapter 16
10.1145/3226595.3226615	Chapter 17
10.1145/3226595.3226616	Chapter 18
10.1145/3226595.3226617	Chapter 19
10.1145/3226595.3226618	Chapter 20
10.1145/3226595.3226619	Chapter 21
10.1145/3226595.3226620	Chapter 22
10.1145/3226595.3226621	Chapter 23
10.1145/3226595.3226622	Part VII.B/Chapter 24
10.1145/3226595.3226623	Chapter 25
10.1145/3226595.3226624	Chapter 26
10.1145/3226595.3226625	Chapter 27
10.1145/3226595.3226626	Chapter 28
10.1145/3226595.3226627	Chapter 29
10.1145/3226595.3226628	Chapter 30
10.1145/3226595.3226629	Chapter 31
10.1145/3226595.3226630	Part VIII/Chapter 32
10.1145/3226595.3226631	Chapter 33
10.1145/3226595.3226632	Chapter 34
10.1145/3226595.3226633	Chapter 35
10.1145/3226595.3226634	Chapter 36
10.1145/3226595.3226635	Part IX/Paper 1
10.1145/3226595.3226636	Paper 2
10.1145/3226595.3226637	Paper 3
10.1145/3226595.3226638	Paper 4
10.1145/3226595.3226639	Paper 5
10.1145/3226595.3226640	Paper 6
10.1145/3226595.3226641	Collected Works
10.1145/3226595.3226642	References/Index/Bios

A publication in the ACM Books series, #22
Editor in Chief: M. Tamer Özsu, *University of Waterloo*

This book was typeset in Arnhem Pro 10/14 and Flama using ZzTEX.

First Edition

10 9 8 7 6 5 4 3 2 1

This book is dedicated to Michael Stonebraker, Jim Gray, Ted Codd, and Charlie Bachman, recipients of the ACM A.M. Turing Award for the management of data, one of the world's most valuable resources, and to their many collaborators, particularly the contributors to this volume.

Contents

Data Management Technology Kairometer: The Historical Context

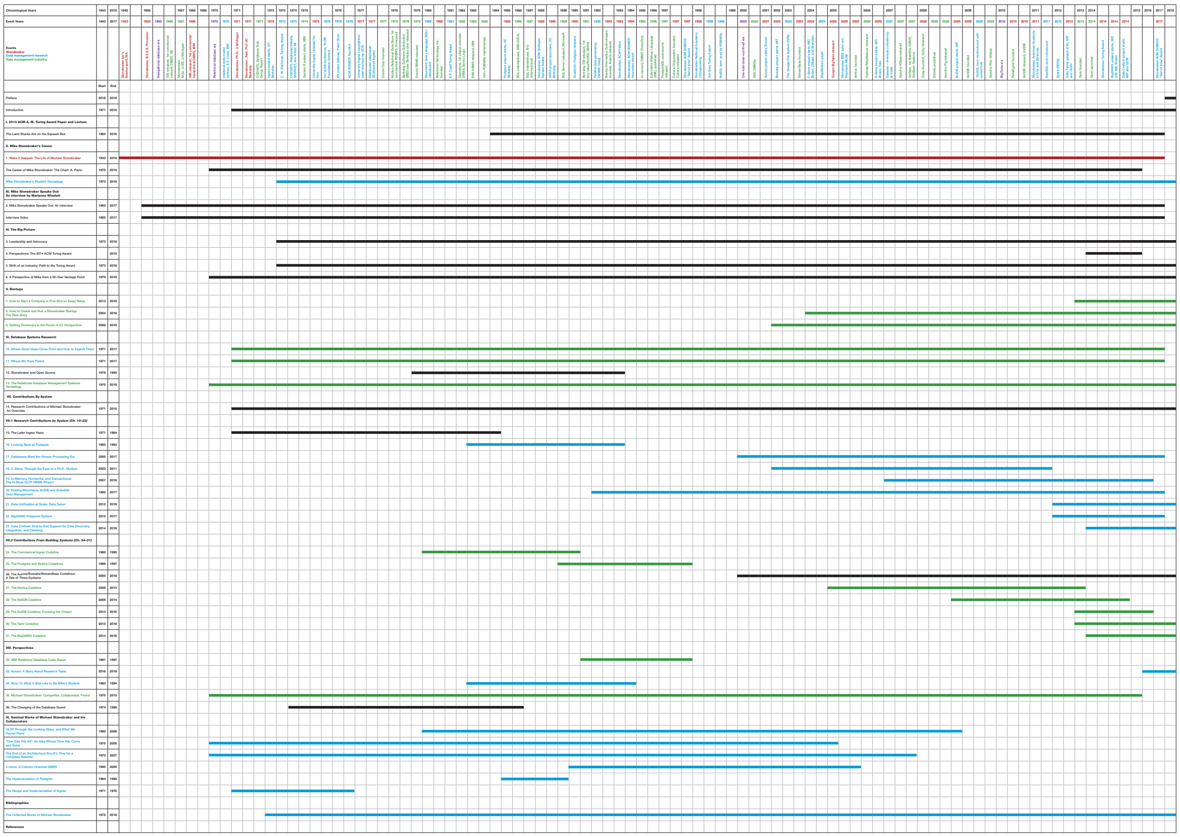
The stories in this book recount significant events in the development of data management technology relative to Michael Stonebraker's achievements, over his career, for which he received the 2014 ACM A.M. Turing Award. To appreciate Mike's contributions to data management technology, it helps to understand the historical context in which the contributions were made. A Data Management Technology Kairometer[1] (available at http://www.morganclaypoolpublishers.com/stonebraker/) answers the questions: *What significant data management events were going on at that time in research, in industry, and in Mike's career?*

Over the years covered by this book (1943 to 2018), the Data Management Technology Kairometer lays out, left to right, the significant events in data management research and industry interspersed with the events of Mike's career. Against these timelines, it presents (top to bottom) the stories ordered by the book's contents, each with its own timeline. A glance at the kairometer tells you how the timelines of the various stories relate in the context of data management research, industry, and Stonebraker career events. When did that event occur relative to associated events?

Our stories recount three types of event (color-coded): data management research events (blue), such as the emergence of in-memory databases; data management industry events (green), such as the initial release of IBM's DB2; and

[1] While a chronometer records the passage of all events, a kairometer records the passage of significant events. "Kairos ($\kappa\alpha\iota\rho\acute{o}\varsigma$) is an Ancient Greek word meaning the right, critical, or opportune moment. The ancient Greeks had two words for time: chronos ($\chi\rho\acute{o}\nu o\varsigma$) and kairos. The former refers to chronological or sequential time, while the latter signifies a proper or opportune time for action. While chronos is quantitative, kairos has a qualitative, permanent nature." (source: http://en.wikipedia.org/wiki/Kairos) We welcome ideas to extend the kairometer; contact michaelbrodie@michaelbrodie.com.

milestones in Mike Stonebraker's career (red), such as his appointment as an assistant professor at UC Berkeley. Stories in black involve multiple event types. Events are separated into the four data management eras described in the book's introduction (purple): navigational, relational, one-size-does-not-fit-all, and Big Data. The data management kairometer provides an historical context for each story relative to the significant data management research and industry events and Mike's career.



Foreword

The A.M. Turing Award is ACM's most prestigious technical award and is given for major contributions of lasting importance to computing. Sometimes referred to as the "Nobel Prize of computing," the Turing Award was named in honor of Alan M. Turing (1912–1954), a British mathematician and computer scientist. Alan Turing is a pioneer of computing who made fundamental advances in various aspects of the field including computer architecture, algorithms, formalization, and artificial intelligence. He was also instrumental in British code-breaking work during World War II.

The Turing Award was established in 1966, and 67 people have won the award since then. The work of each of these awardees has influenced and changed computing in fundamental ways. Reviewing the award winners' work gives a historical perspective of the field's development.

ACM Books has started the Turing Award Series to document the developments surrounding each award. Each book is devoted to one award and may cover one or more awardees. We have two primary objectives. The first is to document how the award-winning works have influenced and changed computing. Each book aims to accomplish this by means of interviews with the awardee(s), their Turing lectures, key publications that led to the award, and technical discussions by colleagues on the work's impact. The second objective is to celebrate this phenomenal and well-deserved accomplishment. We collaborate with the ACM History Committee in producing these books and they conduct the interviews.

Our hope is that these books will allow new generations to learn about key developments in our field and will provide additional material to historians and students.

M. Tamer Özsu
Editor-in-Chief

Preface

The ACM A.M. Turing Award

This book celebrates Michael Stonebraker's accomplishments that led to his 2014 ACM A.M. Turing Award "*For fundamental contributions to the concepts and practices underlying modern database systems.*" [ACM 2016]

When Barbra Liskov, Turing Award committee chair, informed Mike that he had been awarded the 2014 Turing Award, he ". . . teared up. The recognition and validation for my lifetime work was incredibly gratifying." [Stonebraker 2015b]

The book describes, for the broad computing community, the unique nature, significance, and impact of Mike's achievements in advancing modern database systems over more than 40 years. Today, data is considered the world's most valuable resource,[1] whether it is in the tens of millions of databases used to manage the world's businesses and governments, in the billions of databases in our smartphones and watches, or residing elsewhere, as yet unmanaged, awaiting the elusive next generation of database systems. Every one of the millions or billions of databases includes features that are celebrated by the 2014 Turing Award and are described in this book.

Why should I care about databases? What is a database? What is data management? What is a database management system (DBMS)? These are just some of the questions that this book answers, in describing the development of data management through the achievements of Mike Stonebraker and his over 200 collaborators. In reading the stories in this book, you will discover core data management concepts that were developed over the two greatest eras—so far—of data management technology. *Why do we need database systems at all? What concepts were added? Where did those concepts come from? What were the drivers? How did they evolve? What failed and why? What is the practice of database systems?* And, *why do those achievements warrant a Turing Award?*

[1] The *Economist,* May 6, 2017.

While focus of this book is on Michael Stonebraker, the 2014 Turing Award winner, the achievements that the award honors are not just those of one person, no matter how remarkable s/he may be. The achievements are also due to hundreds of collaborators—researchers, students, engineers, coders, company founders and backers, partners, and, yes, even marketing and sales people. *Did all of the ideas come from Mike?* Read on, especially Mike's chapter "Where Good Ideas Come from and How to Exploit Them" (chapter 10).

I have had the great privilege of working with more than my fair share of Turing Award recipients starting as an undergraduate taking complexity theory from Steve Cook of $\mathsf{P} = \mathsf{NP}$ fame. No two Turing Award winners are alike in topic, approach, methods, or personality. All are remarkably idiosyncratic. Mike is, to say the least, idiosyncratic, as you will discover in these pages.

This book answers questions, like those in *italics*, in 30 stories, each by storytellers who were at the center of the story. The stories involve technical concepts, projects, people, prototype systems, failures, lucky accidents, crazy risks, startups, products, venture capital, and lots of applications that drove Mike Stonebraker's achievements and career. Even if you have no interest in databases at all,[2] you'll gain insights into the birth and evolution of Turing Award-worthy achievements from the perspectives of 39 remarkable computer scientists and professionals.

Making Databases Work: The Pragmatic Wisdom of Michael Stonebraker

The theme of this book is modern database *systems*. The 2014 A.M. Turing Award was conferred "*For fundamental contributions to the concepts and practices underlying modern database systems*." It is 1 of only 4 Turing Awards given for databases, and 1 of only 2 out of 51 given for computer systems.

Mike addressed the systems theme in his Turing Award lecture (typically intended to summarize Turing-worthy achievements) in terms of the challenges that he faced and the approach he took to systems research, in four steps. "The first was to try to explain why system software is so hard to build, and why good teams screw it up on a regular basis. Second, it takes real perseverance to "stick it out" and make something actually work. The third was to talk about the start-up experience, and why venture capitalists usually deserve their reputation as "land sharks." Lastly, it is clear that luck plays a significant role in successful startups, and I wanted to explain that. The overarching theme was to use a significant physical challenge as a meta-

[2] Is that possible?

phor for system software development. Over the years, the physical challenge has varied between our cross-country bike ride in 1988, and my climbing all forty-eight 4,000-foot mountains in New Hampshire." [Stonebraker 2015b]

This description contains the seeds of answers to the previous *italicized* questions that are elaborated throughout the book.

The computer systems theme is pursued in the book by stories told from the research perspective: *What were the core database concepts? How did they develop? Why were they significant?* And stories told from the computer systems perspective: *What are the development or engineering challenges? What challenges arise in implementing a research idea? How are they overcome? Do essential research contributions arise from systems engineering?* As you read these stories ask yourself: *What is the relationship between research and systems engineering? Why build prototype systems at all?* Having proven concepts in research and in a prototype system, *why build a product*? (Spoiler alert: While money plays a significant role, it was by no means the goal.)

Acknowledging 39 Remarkable Contributors

This book is a collection of 36 stories written by Mike[3] and 38 of his collaborators: 23 world-leading database researchers, 11 world-class systems engineers, and 4 business partners. They were aided by an editor and 4 professional publishers and editors.

It is my great pleasure to acknowledge the fascinating contributions of all of these remarkable people. They responded with enthusiasm to recount their collaborations with Mike, looking for the essential contributions and how they emerged, all mixed with concern for accurately remembering the crucial facts for you, the reader—in some cases reaching back four decades. *What was important? What seemed to matter vs. what really mattered?* Each contributor, like Mike, is idiosyncratic and strongly opinionated, as you will see. Their achievements reflect the state of the technology and data management demands of the time. Everyone had to be reminded to reflect disagreements with Mike (showing the normal give-and-take of computing research and product development), as well as to state why Mike's contributions warranted the Turing Award. Interestingly, few authors felt comfortable praising Mike, perhaps reflecting the personalities of computer scientists.

[3] Mike wrote some of the most interesting chapters; he did not review other chapters so as not to influence the authors' voices.

Mike can be intimidating. He has made a career of making bold, black-and-white statements to challenge and to inspire himself and the database community to greater accomplishments, as Phil Bernstein recounts so well.[4] It's a sign of maturity for a Ph.D., postdoc, or collaborator of any kind to stand up and refute Mike—and such a pleasure to experience, by Mike included. You will see, in each story, the efforts of the contributors to pass this benchmark.

A theme of Mike's career has been to question *conventional wisdom,* specifically as it ages and as new challenges and concepts arise or as it is undermined by poor practices. The most obvious example is Mike's claim that "one-size-does-*not*-fit-all" in databases, which is a complete contradiction of the claims of the Elephants, Mike's affectionate term for the DBMSs that dominate their market. Yet, Mike was the chief proponent of "one-size-fits-all" in the relational database era. It has been fascinating to watch Mike's contributions become conventional wisdom, which Mike then questions toward the next level of achievement.

If you are an aspiring researcher, engineer, or entrepreneur you might read these stories to find these turning points as practice to tilt at your own computer-science windmills, to spur yourself to your next step of innovation and achievement.

Janice Brown, Our Amanuensis

My greatest acknowledgement, for her contributions to this book, is for Janice L. Brown, technology writer/editor, startup consultant, and frequent Stonebraker collaborator, of Janice Brown & Associates, Inc. In many ways this is Janice's book. (If you dream about the book, it's yours.) Janice was our amanuensis: editor, copywriter, enthusiast, critic,[5] and berger du chats (cat herder) extraordinaire et malheureusement, très necessaire.

Michael Brodie
October 2018

[4] See Chapter 3, "Leadership and Advocacy."

[5] "Truly a great story, yet perhaps you didn't really mean that; how about . . . "

Introduction

Michael L. Brodie

Our story begins at University of California, Berkeley in 1971, in the heat of the Vietnam War. A tall, ambitious, newly minted assistant professor with an EE Ph.D. in Markov chains asks a seasoned colleague for direction in shaping his nascent career for more impact than he could imagine possible with Markov chains.[1] Professor Eugene Wong suggests that Michael Stonebraker read Ted Codd's just-published paper on a striking new idea: *relational databases*. Upon reading the paper, Mike is immediately convinced of the potential, even though he knows essentially nothing about databases. He sets his sights on that pristine relational mountaintop. The rest is history. Or rather, it is the topic of this book. Ted's simple, elegant, relational data model and Mike's contributions to *making databases work* in practice helped forge what is today a $55 billion industry. But, wait, I'm getting ahead of myself.

A Brief History of Databases

What's a database? What is data management? How did they evolve?

Imagine accelerating by orders of magnitude the discovery of cancer causes and the most effective treatments to radically reduce the 10M annual deaths worldwide. Or enabling autonomous vehicles to profoundly reduce the 1M annual traffic deaths worldwide while reducing pollution, traffic congestion, and real estate wasted on vehicles that on average are parked 97% of the time. These are merely two examples of the future potential positive impacts of using Big Data. As with all technical advances, there is also potential for negative impacts, both unintentional—in error—and intentional such as undermining modern democracies (allegedly well

[1] "I had to publish, and my thesis topic was going nowhere." —Mike Stonebraker

under way). Using data means managing data efficiently at scale; that's what data management is for.

In its May 6, 2017 issue, The *Economist* declared data to be the *world's most valuable resource*. In 2012, data science—often AI-driven analysis of data at scale—exploded on the world stage. This new, data-driven discovery paradigm may be one of the most significant advances of the early 21st century. Non-practitioners are always surprised to find that 80% of the resources required for a data science project are devoted to data management. The surprise stems from the fact that data management is an infrastructure technology: basically, unseen plumbing. In business, data management "just gets done" by mature, robust database management systems (DBMSs). But data science poses new, significant data management challenges that have yet to be understood, let alone addressed.

The above potential benefits, risks, and challenges herald a new era in the development of data management technology, the topic of this book. *How did it develop in the first place? What were the previous eras? What challenges lie ahead?* This introduction briefly sketches answers to those questions.

This book is about the development and ascendancy of data management technology enabled by the contributions of Michael Stonebraker and his collaborators. Novel when created in the 1960s, data management became the key enabling technology for businesses of all sizes worldwide, leading to today's $55B[2,3] DBMS market and tens of millions of operational databases. The average Fortune 100 company has more than 5,000 operational databases, supported by tens of DBMS products.

Databases support your daily activities, such as securing your banking and credit card transactions. So that you can buy that Starbucks latte, Visa, the leading global payments processor, must be able to simultaneously process 50,000 credit card transactions per second. These "database" transactions update not just your account and those of 49,999 other Visa cardholders, but also those of 50,000 creditors like Starbucks while simultaneously validating you, Starbucks, and 99,998 others for fraud, no matter where your card was swiped on the planet. Another slice of your financial world may involve one of the more than 3.8B trade transactions that occur daily on major U.S. market exchanges. DBMSs support such critical functions not just for financial systems, but also for systems that manage inventory, air traffic control, supply chains, and all daily functions that depend on data and data transactions. Databases managed by DBMSs are even on your wrist and in your pocket

[2] http://www.statista.com/statistics/724611/worldwide-database-market/

[3] Numbers quoted in this chapter are as of mid-2018.

if you, like 2.3B others on the planet, use an iPhone or Android smartphone. If databases stopped, much of our world would stop with them.

A database is a logical collection of data, like your credit card account information, often stored in records that are organized under some data model, such as tables in a relational database. A DBMS is a software system that manages databases and ensures persistence—so data is not lost—with languages to insert, update, and query data, often at very high data volumes and low latencies, as illustrated above.

Data management technologies have evolved through four eras over six decades, as is illustrated in the Data Management Technology Kairometer (see page xxvii).

In the inaugural *navigational era* (1960s), the first DBMSs emerged. In them, data, such as your mortgage information, was structured in hierarchies or networks and accessed using record-at-a-time navigation query languages. The navigational era gained a database Turing Award in 1973 for Charlie Bachman's "*outstanding contributions to database technology*."

In the second, *relational era (* 1970s–1990s), data was stored in tables accessed using a declarative, set-at-a-time query language, SQL: for example, "Select name, grade From students in Engineering with a B average." The relational era ended with approximately 30 commercial DBMSs dominated by Oracle's Oracle, IBM's DB2, and Microsoft's SQL Server. The relational era gained two database Turing Awards: one in 1981 for Codd's "*fundamental and continuing contributions to the theory and practice of database management systems, esp. relational databases*" and one in 1998 for Jim Gray's "*seminal contributions to database and transaction processing research and technical leadership in system implementation*."

The start of Mike Stonebraker's database career coincided with the launch of the relational era. Serendipitously, Mike was directed by his colleague, UC Berkeley professor Eugene Wong, to the topic of data management and to the relational model via Ted's paper. Attracted by the model's simplicity compared to that of navigational DBMSs, Mike set his sights, and ultimately built his career, on *making relational databases work*. His initial contributions, Ingres and Postgres, did more to make relational databases work in practice than those of any other individual. After more than 30 years, Postgres—via PostgreSQL and other derivatives—continues to have a significant impact. PostgreSQL is the third[4] or fourth[5] most popular (used) of hundreds of DBMSs, and all relational DBMSs, or RDBMSs for short, implement the object-relational data model and features introduced in Postgres.

[4] http://www.eversql.com/most-popular-databases-in-2018-according-to-stackoverflow-survey/

[5] http://db-engines.com/en/ranking

In the first relational decade, researchers developed core RDBMS capabilities including query optimization, transactions, and distribution. In the second relational decade, researchers focused on high-performance queries for data warehouses (using column stores) and high-performance transactions and real-time analysis (using in-memory databases); extended RDBMSs to handle additional data types and processing (using abstract data types); and tested the "one-size-fits-all" principle by fitting myriad application types into RDBMSs. But complex data structures and operations—such as those in geographic information, graphs, and scientific processing over sparse matrices—just didn't fit. Mike knew because he pragmatically exhausted that premise using real, complex database applications to push RDBMS limits, eventually concluding that "one-size-does-*not*-fit-all" and moving to special-purpose databases.

With the rest of the database research community, Mike turned his attention to special-purpose databases, launching the third, "*one-size-does-not-fit-all*" era (2000–2010). Researchers in the "one-size-does-not-fit-all" era developed data management solutions for "one-size-does-not-fit-all" data and associated processing (e.g., time series, semi- and un-structured data, key-value data, graphs, documents) in which data was stored in special-purpose forms and accessed using non-SQL query languages, called NoSQL, of which Hadoop is the best-known example. Mike pursued non-relational challenges in the data manager but, claiming inefficiency of NoSQL, continued to leverage SQL's declarative power to access data using SQL-like languages, including an extension called NewSQL.

New DBMSs proliferated, due to the research push for and application pull of specialized DBMSs, and the growth of open-source software. The "one-size-does-not-fit-all" ended with more than 350 DBMSs, split evenly between commercial and open source and supporting specialized data and related processing including (in order of utilization): relational, key-values, documents, graphs, time series, RDF, objects (object-oriented), search, wide columns, multi-value/dimensional, native XML, content (e.g., digital, text, image), events, and navigational. Despite the choice and diversity of DBMSs, the market remained dominated by five relational DBMSs: the original three plus Microsoft Access and Teradata, which Mike came to call collectively "*the Elephants*." Although the Elephants all supported Mike's object-relational model, they had become "conventional wisdom" that lagged new data management capabilities. A hallmark of Mike's career is to perpetually question conventional wisdom, even of his own making. At the end of the "one-size-does-not-fit-all" era, there was a significant shift in the DBMS market away from the RDBMS Elephants and to less-expensive open-source DBMSs.

The relational and "one-size-does-*not*-fit-all" eras gained a database Turing Award for Stonebraker's "*concepts and practices underlying modern database sys-*

tems." It is the stories of these relational and "one-size-does-*not*-fit-all" developments that fill these pages. You will find stories of their inception, evolution, experimentation, demonstration, and realization by Mike and his collaborators through the projects, products, and companies that brought them to life.

This brings us to the fourth and current **Big Data era** (2010–present), characterized by data at volumes, velocities, and variety (heterogeneity) that cannot be handled adequately by existing data management technology. Notice that, oddly, "Big Data" is defined in terms of data management technology rather than in terms of the typically real-world phenomena that the data represents. Following the previous eras, one might imagine that the Big Data era was launched by the database research community's pursuit of addressing future data management requirements. That is not what happened. The database community's focus remained on relational and "one-size-does-*not*-fit-all" for three decades with little concern for a grander data management challenge—namely managing *all* data as the name "data management" suggests. The 2012 annual IDC/EMC Digital Universe study [Gantz and Reinsel 2013] estimated that of all data in the expanding digital universe, less than 15% was amenable to existing DBMSs. Large enterprises like Yahoo! and Google faced massive data management challenges for which there were no data management solutions in products or research prototypes. Consequently, the problem owners built their own solutions, thus the genesis of Hadoop, MapReduce, and myriad NoSQL Big Data managers. In 2009, Mike famously criticized MapReduce to the chagrin of the Big Data community, only to be vindicated five years later when its creators disclosed that Mike's criticism coincided with their abandonment of MapReduce and Hadoop for yet another round of Big Data management solutions of their own making. This demonstrates the challenges of the Big Data era and that data management at scale is hard to address without the data management underpinnings established over six decades. Retrospectively, it illustrates the challenges faced in the previous database eras and validates that the solutions warranted database Turing Awards.

We appear to be entering a golden age of data, largely due to our expectations for Big Data: that data will fuel and accelerate advances in every field for which adequate data is available. Although this era started in 2010, there has been little progress in corresponding data management solutions. Conventional DBMS "wisdom," as Stonebraker loves to say, and architectures do not seem to apply. Progress, at least progress Stonebraker-style (as we will see throughout the book), is hindered by a paucity of effective use cases. There are almost no (1) reasonably well-understood Big Data management applications that are (2) owned by someone with a pain that no one else has addressed, with (3) the willingness to provide access to their data and to be open to exploring new methods to resolve the pain.

The bright side of this is the explosion of data-intensive applications leading to, as Michael Cafarella and Chris Ré of Stanford University say in Cafarella and Ré [2018], a "blindingly bright future" for database research. The dominant class of data-intensive processing is data science, itself just emerging as a domain; thus, its use above as an example of the need for effective new database management technologies. As the data management technology evolution story continues, the overwhelming interest in data requires a distinction between core data management technology, the topic of this book, and its development to support activities in every human endeavor.

If you are a young researcher or engineer contemplating your future in data management or in computer systems, you now find yourself at the dawn of the Big Data era, much as Mike found himself at the beginning of the relational era. Just as Mike was new to the then-new idea of relational databases, so too you must be new to the as-yet-undefined notion of a Big Data system and data manager. These pages tell the stories of Mike's career as seen from the many different perspectives of his primary collaborators. These stories may give you a historical perspective and may provide you guidance on your path: what issues to pursue, how to select them, how to pursue them, how to collaborate with people who complement your knowledge, and more. These stories recount not only challenges, technology, methods, and collaboration styles, but also people's attitudes that, perhaps more so than technology, contributed to the state of data management today, and specifically to the achievements of Mike Stonebraker and his collaborators. However, the world now is not as it was in 1971 as Mike launched his career, as Mike discusses in Chapter 11, "Where We Have Failed."

Preparing to Read the Stories and What You Might Find There

To explore and understand Lithuania on a trip, few people would consider visiting every city, town, and village. So how do you choose what to see? Lonely Planet's or Eyewitness' travel guides to Estonia, Latvia, and Lithuania are fine, but they are 500 pages each. It's better to be motivated and have worked out the initial questions that you would like answered. Here is a method for visiting Lithuania and for reading the stories in this book.

The book was written by computer systems researchers, engineers, developers, startup managers, funders, Silicon Valley entrepreneurs, executives, and investors for people like themselves and for the broad computing community. Let's say you aspire to a career in, or are currently in, one of those roles: *What would you like to learn from that perspective?*

Say, for example, that you are a new professor in Udine, Italy, planning your career in software systems research. As Mike Stonebraker was at UC Berkeley in 1971, you ask: *What do I want to do in my career? How do I go about realizing that career?* Develop a list of questions you would like to pursue such as the 20 or so italicized questions in this Introduction and the Preface. Select the stories that appear to offer answers to your questions in a context in which you have some passion. Each story results in significant successes, perhaps on different topics that interest you, or in a different culture and a different time. Nonetheless, the methods, the attitudes, and the lessons are generally independent of the specific story. The contributors have attempted to generalize the stories with the hindsight and experience of as much as 40 years.

Choose your role, figure out the questions on which you would like guidance, choose your own perspective (there are 30 in the book), and set off on your journey, bailing when it is not helpful. Become your own software systems ciceroni.[6]

A Travel Guide to Software Systems Lessons in Nine Parts

The 30 stories in this book are arranged into 9 parts.

Part I, "2014 ACM A.M. Turing Award Paper and Lecture," contains the paper in which Turing awardees typically describe the achievements for which the award was conferred. The paper is given as a lecture at ACM conferences during the award year. True to Mike's idiosyncratic nature, he used the paper and lecture as a platform for what he considered his most important message for the community, as opposed to the already published technical achievements. Mike writes of challenges posed by software systems, the theme of this book, as explained in the Preface. Mike described the nature of the challenges by analogy with significant physical challenges—something most audiences can understand—from Mike's own personal life. The paper is reproduced from the *Communications of the ACM*. The lecture was given initially at the Federated Computing Research Conference, June 13, 2015, and can be viewed online.[7]

Part II, "Mike Stonebraker's Career," lays out Mike's career in Sam Madden's biography and in two graphic depictions. Chart 1 lists chronologically Mike's Ph.D. students and postdocs. Chart 2 illustrates the academic projects and awards and

[6] Expert tourist guide, who makes the trip worthwhile, derived from Marcus Tullius Cicero, Roman politician and orator known for guiding through complex political theses, diminished today to restaurants.

[7] http://www.youtube.com/watch?v=BbGeKi6T6QI

the creation and acquisition of his companies. On April 12, 2014, Mike Carey (University of California, Irvine), David DeWitt (then at University of Wisconsin), Joe Hellerstein (University of California, Berkeley), Sam Madden (MIT), Andy Pavlo (Carnegie Mellon University), and Margot Seltzer (Harvard University) organized a Festschrift for Mike: a day-long celebration of Mike Stonebraker at 70.[8] More than 200 current and former colleagues, investors, collaborators, rivals, and students attended. It featured speakers and discussion panels on the major projects from Mike's 40-plus year career. Chart 2, "The Career of Mike Stonebraker," was produced for the Festschrift by Andy Pavlo and his wife.

Part III, "Mike Stonebraker Speaks Out: An Interview with Marianne Winslett," is a post-Turing-Award interview in the storied series of interviews of database contributors by Marianne Winslett. A video of the interview can be seen online http://www.gmrtranscription.com.

In Part IV, "The Big Picture," world-leading researchers, engineers, and entrepreneurs reflect on Mike's contributions in the grander scope of things. Phil Bernstein, a leading researcher and big thinker, reflects on Mike's leadership and advocacy. James Hamilton, a world-class engineer and former lead architect of DB2—an Elephant—reflects on the value of Turing contributions. Jerry Held, Mike's first Ph.D. student, now a leading Silicon Valley entrepreneur, recounts experiences collaborating and competing with Mike. Dave DeWitt, comparable to Mike in his data management contributions, reflects on 50 years as a mentee, colleague, and competitor.

Part V, "Startups," tells one of the 21st century's hottest stories: how to create, fund, and run a successful technology startup. As the *Data Management Technology Kairometer* (see page xxvii) illustrates, Mike has co-founded nine startups. The startup story is told from three distinctly different points of view: from that of the technical innovator and Chief Technology Officer (Mike), from that of the CEO, and from that of the prime funder. These are *not* mere *get rich quick* or *get famous quick* stories. Startups and their products are an integral component of Mike Stonebraker's database technology research and development methodology, to ensure that the results have impact. This theme of industry-driven and industry-proven database technology research pervades these pages. If you get only one thing from this book, let this be it.

Part VI, "Database Systems Research," takes us into the heart of database systems research. Mike answers: *Where do ideas come from? How to exploit them?* Take a

[8] For photographs see http//stonebraker70.com.

master class with a Turing Award winner on how to do database systems research. Mike's invitation to write the "Failures" chapter was: *So, Mike, what do you really think?* His surprising answers lay out challenges facing database systems research and the database research community. Mike Olson, a Ph.D. student of Mike and an extremely successful co-founder and Chief Strategy Officer of Cloudera (which has data management products based on Hadoop), lays out Mike's contributions relative to the open-source movement that was launched after Ingres was already being freely distributed. Part VI concludes with Felix Naumann's amazing Relational Database Management Genealogy, which graphically depicts the genealogy of hundreds of RDBMSs from 1970 to the present showing how closely connected RDBMSs are in code, concepts, and/or developers.

Part VII, "Contributions by System," describes the technical contributions for which the Turing award was given. They are presented chronologically in the context of the nine projects, each centered on the software system in which the contributions arose. Each system is described from a research perspective in Part VII.A, "Research Contributions by System," and from a systems perspective, in a companion story in Part VII.B, "Contributions from Building Systems." Chapter 14 synopsizes the major technical achievements and offers a map to contributions and their stories.

The stories in "Research Contributions by System" focus on the major technical achievements that arose in each specific project and system. They do not repeat the technical arguments from the already published papers that are all cited in the book. The research chapters explain the technical accomplishments: their significance, especially in the context of the technology and application demands of the time; and their value and impact in the resulting technology, systems, and products that were used to prove the ideas and in those that adopted the concepts. Most stories, like Daniel Abadi's Chapter 18 tell of career decisions made in the grips of challenging and rewarding research. The first seven systems—Ingres, Postgres, Aurora, C-Store, H-Store, SciDB, and Data Tamer—span 1972–2018, including the relational, "one-size-does-*not*-fit-all", and now the Big Data eras of data management. All seven projects resulted in successful systems, products, and companies. Not included in the book are two systems that Mike does not consider successful. (But if some didn't fail, he wasn't trying hard enough.) The final two projects, BigDAWG and Data Civilizer, are under way at this writing as two of Mike's visions in the Big Data world.

The stories in "Contributions from Building Systems" are a little unusual in that they tell seldom-told stories of heroism in software systems engineering. The development of a software system, e.g., a DBMS, is a wonderful and scary experience for individuals, the team, and the backers! There is a lot of often-unsung drama: inevitable disasters and unbelievable successes, and always discoveries, sometimes new but often repeated for the umpteenth time. Key team members of some of the best-known DBMSs in the world (DB2, Ingres, Postgres) tell tales of the development of the codelines we have come to know as DB2, Ingres, StreamBase, Vertica, VoltDB, and SciDB, and the data unification system Tamr.

These stories were motivated, in part, by an incidental question that I asked James Hamilton, former lead architect, IBM DB2 UDB, which (with Stonebraker's Ingres) was one of the first relational DBMSs and became an Elephant. I asked James: "Jim Gray told me some fascinating stories about DB2. What really happened?" What unfolded was a remarkable story told in Chapter 32, "IBM Relational Database Code Bases," which made it clear that all the codeline stories must be told. One sentence that got me was: "Instead of being a punishing or an unrewarding 'long march,' the performance improvement project was one of the best experiences of my career." This came from one of the world's best systems architects, currently, Vice President and Distinguished Engineer, Amazon Web Services.

But, more importantly, these stories demonstrate the theme of the book and Mike's observation that "*system software is so hard to build, and why good teams screw it up on a regular basis.*"

From the beginning of Mike's database career, the design, development, testing, and adoption of prototype and commercial systems have been fundamental to his research methodology and to his technical contributions. As he says in Chapter 9 "Ingres made an impact mostly because we persevered and got a real system to work." Referring to future projects, he says, "In every case, we built a prototype to demonstrate the idea. In the early days (Ingres/Postgres), these were full-function systems; in later days (C-Store/H-Store) the prototypes cut a lot of corners." These systems were used to test and prove or disprove research hypotheses, to understand engineering aspects, to explore details of real use case, and to explore the adoption, hence impact, of the solutions. As a result, research proceeded in a virtuous cycle in which research ideas improved systems and, in turn, systems and application challenges posed research challenges.

Part VIII, "Perspectives," offers five personal stories. James Hamilton recounts developing one of the world's leading DBMSs, which included the highlights of his storied engineering career. Raul Castro Fernandez recounts how, as a Stonebraker postdoc, he learned how to do computer systems research—how he gained research taste. Marti Hearst tells an engaging story of how she matured from a student to a researcher under a seemingly intimidating but actually caring mentor. Don Haderle, a product developer on IBM's Systems R, the alleged sworn enemy of the

Ingres project, speaks admiringly of the competitor who became a collaborator and friend. In the final story of the book, I recount meeting Mike Stonebraker for the first time in 1974. I had not realized until I reflected for this story that the 1974 pre-SIGMOD (Special Interest Group on Management of Data) conference that hosted the much anticipated CODASYL-Relational debate marked a changing of the database guard: not just shifting database research leadership from the creators of navigational DBMSs to new leaders, such as Mike Stonebraker, but also foreshadowing the decline of navigational database technology and the rise of the relational and subsequent eras of database technology.

Part IX, "Seminal Works of Michael Stonebraker and His Collaborators," reprints the six papers that, together with the 2014 ACM A.M. Turing Award Paper in Chapter 1, constitute the papers that present Mike's most significant technical achievements. Like most of the stories in this book, these seminal works should be read in the context of the technology and challenges at the time of their publication. That context is exactly what the corresponding research and systems stories in Part VII provide. Until now, those papers lacked that context, now given by contributors who were central to those contributions and told from the perspective of 2018.

Mike's 7 seminal papers were chosen from the approximately 340 that he has authored or co-authored in his career. Mike's publications are listed in "Collected Works of Michael Stonebraker," page 607.

PART I

2014 ACM A.M. TURING AWARD PAPER AND LECTURE

The Land Sharks Are on the Squawk Box

Michael Stonebraker

It turns out riding across America is more than a handy metaphor for building system software.

—Michael Stonebraker

KENNEBAGO, ME, SUMMER 1993. The “Land Sharks” are on the squawk box, Illustra (the company commercializing Postgres) is down to fumes, and I am on a conference call with the investors to try to get more money. The only problem is I am in Maine at my brother’s fishing cabin for a family event while the investors are on a speakerphone (the squawk box) in California. There are eight of us in cramped quarters, and I am camped out in the bathroom trying to negotiate a deal. The conversation is depressingly familiar. They say more-money-lower-price; I say less-money-higher-price. We ultimately reach a verbal handshake, and Illustra will live to fight another day.

Negotiating with the sharks is always depressing. They are superb at driving a hard bargain; after all, that is what they do all day. I feel like a babe in the woods by comparison.

This article interleaves two stories (see Figure 1). The first is a cross-country bike ride my wife Beth and I took during the summer of 1988; the second is the design, construction, and commercialization of Postgres, which occurred over a 12-year

Originally published in *Communications of the ACM,* 59(2): 74–83, 2016. Original DOI: 10.1145/2869958

Key Insights

- Explained is the motivation behind Postgres design decisions, as are "speed-bumps" encountered.
- Riding a bicycle across America and building a computer software system are both long and difficult affairs, constantly testing personal fortitude along the way.
- Serendipity played a major role in both endeavors.

period, from the mid-1980s to the mid-1990s. After telling both stories, I will draw a series of observations and conclusions.

Off to a Good Start

Anacortes, WA, June 3, 1988. Our car is packed to the gills, and the four of us (Beth; our 18-month-old daughter Leslie; Mary Anne, our driver and babysitter; and me) are squished in. It has been a stressful day. On the roof is the cause of it all—our brand-new tandem bicycle. We spent the afternoon in Seattle bike shops getting it repaired. On the way up from the Bay Area, Mary Anne drove into a parking structure lower than the height of the car plus the bike. Thankfully, the damage is repaired, and we are all set to go, if a bit frazzled. Tomorrow morning, Beth and I will start riding east up the North Cascades Scenic Highway; our destination, some 3,500 miles away, is Boston, MA. We have therefore christened our bike "Boston Bound."

It does not faze us that we have been on a tandem bike exactly once, nor that we have never been on a bike trip longer than five days. The fact we have never climbed mountains like the ones directly in front of us is equally undaunting. Beth and I are in high spirits; we are starting a great adventure.

Berkeley, CA, 1984. We have been working on Ingres for a decade. First, we built an academic prototype, then made it fully functional, and then started a commercial company. However, Ingres Corporation, which started with our open source code base four years ago in 1980, has made dramatic progress, and its code is now vastly superior to the academic version. It does not make any sense to continue to do prototyping on our software. It is a painful decision to push the code off a cliff, but at that point a new DBMS is born. So what will Postgres be?

One thing is clear: Postgres will push the envelope on data types. By now I have read a dozen papers of the form: "The relational model is great, so I tried it on [pick a vertical application]. I found it did not work, and to fix the situation, I propose we add [some new idea] to the relational model."

Anacortes, WA: Day 1 – June 4, 1988

Some chosen verticals were geographic information systems (GISs), computer-aided design (CAD), and library information systems. It was pretty clear to me that the clean, simple relational model would turn into a complete mess if we added random functionality in this fashion. One could think of this as "death by 100 warts."

The basic problem was the existing relational systems—specifically Ingres and System R—were designed with business data processing users in mind. After all, that was the major DBMS market at the time, and both collections of developers were trying to do better than the existing competition, namely IMS and Codasyl, on this popular use case. It never occurred to us to look at other markets, so RDBMSs were not good at them. However, a research group at the University of California at Berkeley, headed by Professor Pravin Varaiya, built a GIS on top of Ingres, and we saw firsthand how painful it was. Simulating points, lines, polygons, and line groups on top of the floats, integers, and strings in Ingres was not pretty.

It was clear to me that one had to support data types appropriate to an application and that required user-defined data types. This idea had been investigated earlier by the programming language community in systems like EL1, so all I had to do was apply it to the relational model. For example, consider the following SQL update to a salary, stored as an integer

```
Update Employee set (salary = salary + 1000) where name = ‘George'
```

To process it, one must convert the character string `1000` to an integer using the library function string-to-integer and then call the integer + routine from the C library. To support this command with a new type, say, `foobar`, one must merely add two functions, `foobar-plus` and `string-to-foobar`, and then call them at the appropriate times. It was straightforward to add a new DBMS command, `ADDTYPE`, with the name of the new data type and conversion routines back and forth to ASCII. For each desired operator on this new type, one could add the name of the operator and the code to call to apply it.

The devil is, of course, always in the details. One has to be able to index the new data type using B-trees or hashing.

Indexes require the notion of less-than and equality. Moreover, one needs commutativity and associativity rules to decide how the new type can be used with other types. Lastly, one must also deal with predicates of the form:

```
not salary < 100
```

This is legal SQL, and every DBMS will flip it to

```
salary ≥ 100
```

So one must define a negator for every operator, so this optimization is possible.

We had prototyped this functionality in Ingres [8], and it appeared to work, so the notion of abstract data types (ADTs) would clearly be a cornerstone of Postgres.

Winthrop, WA, Day 3. My legs are throbbing as I lay on the bed in our motel room. In fact, I am sore from the hips down but elated. We have been riding since 5 a.m. this morning; telephone pole by telephone pole we struggled uphill for 50 miles. Along the way, we rose 5,000 feet into the Cascades, putting on every piece of clothing we brought with us. Even so, we were not prepared for the snowstorm near the top of the pass. Cold, wet, and tired, we finally arrived at the top of the aptly named Rainy Pass. After a brief downhill, we climbed another 1,000 feet to the top of Washington Pass. Then it was glorious descent into Winthrop. I am now exhausted but in great spirits; there are many more passes to climb, but we are over the first two. We have proved we can do the mountains.

Berkeley, CA, 1985–1986. Chris Date wrote a pioneering paper [1] on referential integrity in 1981 in which he defined the concept and specified rules for enforcing it. Basically, if one has a table

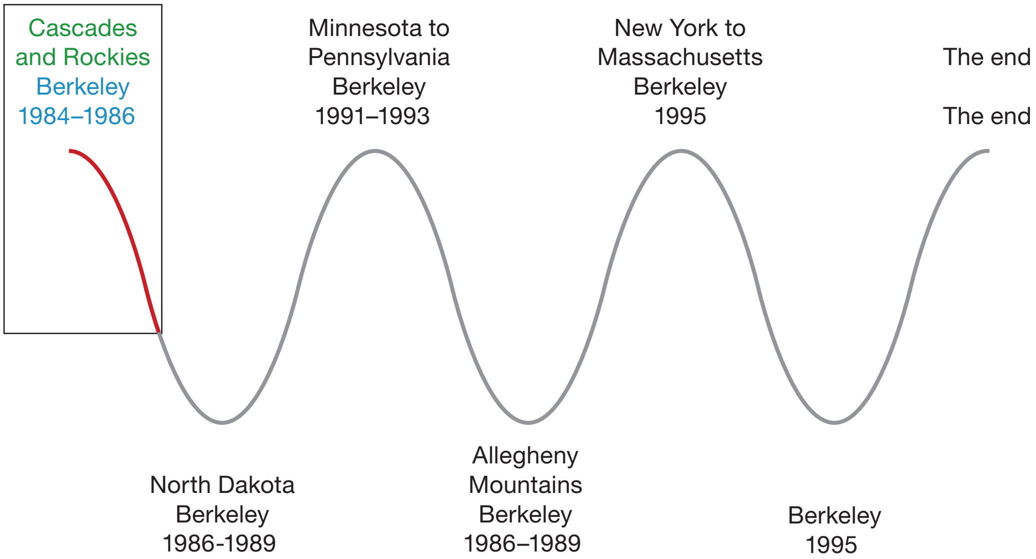


Figure 1 The two timelines: Cross-country bike ride and Illustra/Postgres development.

```
Employee (name, salary, dept, age) with primary key "name"
```

and a second table

```
Dept (dname, floor) with a primary key "dname"
```

then the attribute dept in Employee is a foreign key; that is, it references a primary key in another table; an example of these two tables is shown in Figure 2. In this case, what happens if one deletes a department from the dept table?

For example, deleting the candy department will leave a dangling reference in the Employee table for everybody who works in the now-deleted department. Date identified six cases concerning what to do with insertions and deletions, all of which can be specified by a fairly primitive if-then rule system. Having looked at programs in Prolog and R1, I was very leery of this approach. Looking at any rule program with more than 10 statements, it is very difficult to figure out what it does. Moreover, such rules are procedural, and one can get all kinds of weird behavior depending on the order in which rules are invoked. For example, consider the following two (somewhat facetious) rules:

```
If Employee.name = ‘George'
Then set Employee.dept = ‘shoe'
If Employee.salary > 1000 and Employee.dept = ‘candy'
```

dname	floor	sq. ft.	budget
Shoe	3	500	40,000
Candy	2	800	50,000

name	dept	salary	age
Bill	Shoe	2,000	40
Art	Candy	3,000	35
Sam	Shoe	1,500	25
Tom	Shoe	1,000	23

Figure 2 Correlated data illustrating why data users need referential integrity.

```
Then set Employee.salary = 1000
```

Consider an update that moves George from the shoe department to the candy department and updates his salary to 2000. Depending on the order the two rules are processed, one will get different final answers. Notably, if the rules are executed in the order here, then George will ultimately have a salary of 2000; if the rule order is reversed, then his ending salary will be 1000. Having order-dependent rule semantics is pretty awful.

A fundamental tenet of the relational model is the order of evaluation of a query, including the order in which records are accessed, is up to the system. Hence, one should always give the same final result, regardless of the query plan chosen for execution. As one can imagine, it is trivial to construct collections of rules that give different answers for different query plans—obviously undesirable system behavior.

I spent many hours over a couple of years looking for something else. Ultimately, my preferred approach was to add a keyword `always` to the query language. Hence, any utterance in the query language should have the semantics that it appears to be continually running. For example, if Mike must have the same salary as Sam, then the following `always` command will do the trick

```
Always update Employee, E
set salary = E.salary
where Employee.name = ‘Mike' and E.name = ‘Sam'
```

Whenever Mike receives a salary adjustment, this command will kick in and reset his salary to that of Sam. Whenever Sam gets a raise, it will be propagated to

Marias Pass, MT: Day 15

Mike. Postgres would have this `always` command and avoid (some of) the ugliness of an if-then rules system. This was great news; Postgres would try something different that has the possibility of working.

Marias Pass, MT, Day 15. I cannot believe it. We round a corner and see the sign for the top of the pass. We are at the Continental Divide! The endless climbs in the Cascades and the Rockies are behind us, and we can see the Great Plains stretching out in front of us. It is now downhill to Chicago! To celebrate this milestone, we pour a small vial of Pacific Ocean water we have been carrying since Anacortes to the east side of the pass where it will ultimately flow into the Gulf of Mexico.

Berkeley, CA, 1986. My experience with Ingres convinced me a database log for recovery purposes is tedious and difficult to code. In fact, the gold standard specification is in C. Mohan et al. [3]. Moreover, a DBMS is really two DBMSs, one managing the database as we know it and a second one managing the log, as in Figure 3. The log is the actual system of record, since the contents of the DBMS can be lost. The idea we explored in Postgres was to support time travel. Instead of updating a data record in place and then writing both the new contents and the old contents into the log, could we leave the old record alone and write a second record with the new contents in the actual database? That way the log

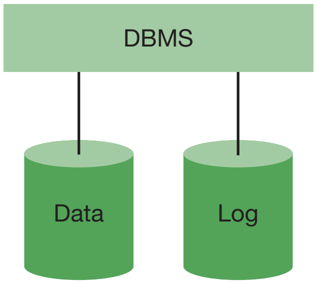


Figure 3 Traditional DBMS crash recovery.

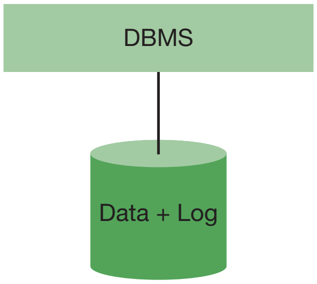


Figure 4 Postgres picture: No overwrite.

would be incorporated into the normal database and no separate log processing would be required, as in Figure 4. A side benefit of this architecture is the ability to support time travel, since old records are readily queryable in the database. Lastly, standard accounting systems use no overwrite in their approach to record keeping, so Postgres would be compatible with this tactic.

At a high level, Postgres would make contributions in three areas: an ADT system, a clean rules system based on the `always` command, and a time-travel storage system. Much of this functionality is described in Stonebraker and Rowe [6,7]. For more information on the scope of Postgres, one can consult the video recording of the colloquium celebrating my 70th birthday [2]. We were off and running with an interesting technical plan.

First Speedbumps

Drake, ND, Day 26. We are really depressed. North Dakota is bleak. The last few days have been the following monotony:

Drake, ND: Day 26

See the grain elevator ahead that signifies the next town
Ride for an hour toward the elevator
Pass through the town in a few minutes
See the next grain elevator . . .

However, it is not the absence of trees (we joke the state tree of North Dakota is the telephone pole) and the bleak landscape that is killing us. Normally, one can simply sit up straight in the saddle and be blown across the state by the prevailing winds, which are typically howling from west to east. They are howling all right, but the weather this summer is atypical. We are experiencing gale-force winds blowing east to west, straight in our faces. While we are expecting to be blown along at 17–18 miles per hour, we are struggling hard to make 7. We made only 51 miles today and are exhausted. Our destination was Harvey, still 25 miles away, and we are not going to make it. More ominously, the tree line (and Minnesota border) is still 250 miles away, and we are not sure how we will get there. It is all we can do to refuse a ride from a driver in a pickup truck offering to transport us down the road to the next town.

The food is also becoming problematic. Breakfast is dependable. We find a town, then look for the café (often the only one) with the most pickup trucks. We eat from the standard menu found in all such restaurants. However, dinner is getting really boring. There is a standard menu of fried fare; we yearn for pasta and salad, but it is never on the menu.

We have established a routine. It is in the 80s or 90s Fahrenheit every day, so Beth and I get on the road by 5 a.m. Mary Anne and Leslie get up much later; they hang around the motel, then pass us on the road going on to the town where we will spend the night. When we arrive at the new motel, one of us relieves Mary Anne while the other tries to find someplace with food we are willing to eat. Although we have camping equipment with us, the thought of an air mattress after a hard day on the road is not appealing. In fact, we never camp. Leslie has happily accommodated to this routine, and one of her favorite words, at 18-months old, is "ice machine." Our goal is 80 miles a day in the flats and 60 miles a day in the mountains. We ride six days per week.

Berkeley, CA, 1986. I had a conversation with an Ingres customer shortly after he implemented `date` and `time` as a new data type (according to the American National Standards Institute specification). He said, "You implemented this new data type incorrectly." In effect, he wanted a different notion of time than what was supported by the standard Gregorian calendar. More precisely, he calculated interest on Wall Street-type financial bonds, which give the owner the same amount of interest, regardless of how long a month is. That is, he wanted a notion of `bond time` in which March 15 minus February 15 is always 30 days, and each year is divided into 30-day months. Operationally, he merely wanted to overload temporal subtraction with his own notion. This was impossible in Ingres, of course, but easy to do in Postgres. It was a validation that our ADTs are a good idea.

Berkeley, CA, 1986. My partner, the "Wine Connoisseur," and I have had a running discussion for nearly a year about the Postgres data model. Consider the `Employee-Dept` database noted earlier. An obvious query is to join the two tables, to, say, find the names and floor number of employees, as noted in this SQL command:

```
Select E.name, D.floor
From Employee E, Dept D
Where E.dept = D.dname
```

In a programming language, this task would be coded procedurally as something like (see code section 1).

A programmer codes an algorithm to find the desired result. In contrast, one tenet of the relational model is programmers should state what they want without having to code a search algorithm. That job falls to the query optimizer, which must decide (at scale) whether to iterate over `Employee` first or over `Dept` or to hash both tables on the join key or sort both tables for a merge or . . .

My Ingres experience convinced me optimizers are really difficult, and the brain surgeon in any database company is almost certainly the optimizer specialist. Now we were considering extending the relational model to support more complex types. In its most general form, we could consider a column whose fields were pointers to arrays of structures of . . . I could not wrap my brain around designing a query optimizer for something this complex. On the other hand, what should we discard? In the end, The Wine Connoisseur and I are depressed as we choose a design point with rudimentary complex objects. There is still a lot of code to support the notion we select.

Berkeley, CA, 1987. The design of time travel in Postgres is in Stonebraker [5]. Although this is an elegant construct in theory, making it perform well in practice is tricky. The basic problem is the two databases in the traditional architecture of Figure 3 are optimized very differently. The data is "read-optimized" so queries are fast, while the log is "write-optimized" so one can commit transactions rapidly. Postgres must try to accomplish both objectives in a single store; for example, if 10 records are updated in a transaction, then Postgres must force to disk all the pages on which these records occurred at commit time. Otherwise, the DBMS can develop "amnesia," a complete no-no. A traditional log will group all the log records on a small collection of pages, while the data records remain read-optimized. Since we are combining both constructs into one storage structure, we have to address a tricky record placement problem to try to achieve both objectives, and our initial implementation is not very good. We spend a lot of time trying to fix this subsystem.

Berkeley, CA, 1987. The Wine Connoisseur and I had written Ingres in C and did not want to use it again. That sounded too much like déjà vu. However, C++

Code Section 1.

```
For E in Employee {
         For D in Dept {
                  If (E.dept = D.dname) then add-to-result;
         }
}
```

was not mature enough, and other language processors did not run on Unix. By this time, any thought of changing operating systems away from Unix was not an option; all the Berkeley students were being trained on Unix, and it was quickly becoming the universal academic operating system. So we elected to drink the artificial intelligence Kool-Aid and started writing Postgres in Lisp.

Once we had a rudimentary version of Postgres running, we saw what a disastrous performance mistake this was—at least one-order-of-magnitude performance penalty on absolutely everything. We immediately tossed portions of the code base off the cliff and converted everything else to C. We were back to déjà vu (coding in C), having lost a bunch of time, but at least we had learned an important lesson: Do not jump into unknown water without dipping your toe in first. This was the first of several major code rewrites.

Berkeley, CA, 1988. Unfortunately, I could not figure out a way to make our `always` command general enough to at least cover Chris Date's six referential integrity cases. After months of trying, I gave up, and we decided to return to a more conventional rule system. More code over the cliff, and more new functionality to write.

In summary, for several years we struggled to make good on the original Postgres ideas. I remember this time as a long "slog through the swamp."

Another High

Carrington, ND, the next afternoon. It is really hot, and I am dead tired. I am on "Leslie duty," and after walking though town, we are encamped in the ubiquitous (and air-conditioned) local Dairy Queen. I am watching Leslie slurp down a soft serve, feeling like "God is on our side," as serendipity has intervened in a big way today. No, the wind is still blowing at gale force from east to west. Serendipity came in the form of my brother. He has come from Maine to ride with us for a week. Mary Anne picked him and his bicycle up at the Minot airport yesterday afternoon. He is fresh and a very, very strong rider. He offers to break the wind for us, like you see in bicycle races. With some on-the-job training (and a couple of excursions into the wheat fields when we hit his rear wheel), Beth and I figure out how to ride six inches behind his rear wheel. With us trying to stay synchronized with a faster-slower-faster dialog, we rode 79 miles today. It is now clear we are "over the hump" and will get out of North Dakota, a few inches behind my brother's wheel, if necessary.

Battle Lake, MN, July 4, 1988, Day 30. We are resting today and attending the annual 4th of July parade in this small town. It is quite an experience—the local band, clowns giving out candy, which Leslie happily takes, and Shriners in their little cars. It is a slice of Americana I will never forget. Rural America has taken very good care of us, whether by giving our bike a wide berth when passing, willingly cashing our travelers checks, or alerting us to road hazards and detours.

Berkeley, CA, 1992. In my experience, the only way to really make a difference in the DBMS arena is to get your ideas into the commercial marketplace. In theory, one could approach the DBMS companies and try to convince them to adopt something new. In fact, there was an obvious "friendly" one—Ingres Corporation—although it had its own priorities at the time.

I have rarely seen technology transfer happen in this fashion. There is a wonderful book by Harvard Business School professor Clayton Christiansen called *The Innovators Dilemma.* His thesis is technology disruptions are very challenging for the incumbents. Specifically, it is very difficult for established vendors with old technology to morph to a new approach without losing their customer base. Hence, disruptive ideas do not usually find a receptive audience among the established vendors, and launching a startup to prove one's ideas is the preferred option.

By mid-1992 I had ended my association with Ingres and a sufficient amount of time had passed that I was free of my non-compete agreement with the company. I was ready to start a commercial Postgres company and contacted my friend the "Tall Shark." He readily agreed to be involved. What followed was a somewhat torturous negotiation of terms with the "Head Land Shark," with me getting on-the-job training in the terms and conditions of a financing contract. Finally, I understood what I was being asked to sign. It was a difficult time, and I changed my mind more than once. In the end, we had a deal, and Postgres had $1 million in venture capital to get going.

Right away two stars from the academic Ingres team—"Quiet" and "EMP1"—moved over to help. They were joined shortly thereafter by "Triple Rock," and we had a core implementation team. I also reached out to "Mom" and her husband, the "Short One," who also jumped on board, and we were off and running, with the Tall Shark acting as interim CEO. Our initial jobs were to whip the research code line into commercial shape, convert the query language from QUEL to SQL, write documentation, fix bugs, and clean up the "cruft" all over the system.

Emeryville, CA, 1993. After a couple of naming gaffes, we chose Illustra, and our goal was to find customers willing to use (and hopefully pay for) a system from a startup. We had to find a compelling vertical market, and the one we chose to focus on was geographic data. Triple Rock wrote a collection of abstract data types for points, lines, and polygons with the appropriate functions (such as distance from a point to a line).

After an infusion of capital from new investors, including the "Entrepreneur-Turned-Shark," we again ran out of money, prompting the phone call from Kennebago noted earlier. Soon thereafter, we were fortunate to be able to hire the "Voice-of-Experience" as the real CEO, and he recruited "Smooth" to be VP of sales, complementing "Uptone," who was previously hired to run marketing. We had a real company with a well-functioning engineering team and world-class executives. The future was looking up.

Luddington, MI, Day 38. We walk Boston Bound off the Lake Michigan ferry and start riding southeast. The endless Upper Midwest is behind us; it is now less than 1,000 miles to Boston! Somehow it is reassuring that we have no more more water to cross. We are feeling good. It is beginning to look like we might make it.

The High Does Not Last

Ellicottville, NY, Day 49. Today was a very bad day. Our first problem occurred while I was walking down the stairs of the hotel in Corry, PA, in my bicycle cleats. I slipped on the marble floor and wrenched my knee. Today, we had only three good legs pushing Boston Bound along. However, the bigger problem is we hit the Alleghany Mountains. Wisconsin, Michigan, and Ohio are flat. That easy riding is over, and our bicycle maps are sending us up and then down the same 500 feet over and over again. Also, road planners around here do not seem to believe in switchbacks; we shift into the lowest of our 21 gears to get up some of these hills, and it is exhausting work. We are not, as you can imagine, in a good mood. While Beth is putting Leslie to bed, I ask the innkeeper in Ellicottville a simple question, "How do we get to Albany, NY, without climbing all these hills?"

Emeryville, CA, 1993. Out of nowhere comes our first marketing challenge. It was clear our "sweet spot" was any application that could be accelerated through ADTs. We would have an unfair advantage over any other DBMS whenever this was true. However, we faced a Catch-22 situation. After a few "lighthouse" customers, the more cautious ones clearly said they wanted GIS functionality from the major GIS vendors (such as ArcInfo and MapInfo). We needed to recruit application companies in specific vertical markets and convince them to restructure the inner core of their software into ADTs—not a trivial task. The application vendors naturally said, "Help me understand why we should engage with you in this joint project." Put more bluntly, "How many customers do you have and how much money can I expect to make from this additional distribution channel for my product?" That is, we viewed this rearchitecting as a game-changing technology shift any reasonable application vendor should embrace. However, application vendors viewed it as merely a new distribution channel. This brought up the Catch-22: Without ADTs we could not get customers, and without customers we could not get ADTs. We were pondering this depressing situation, trying to figure out what to do, when the next crisis occurred.

Oakland, CA, 1994. We were again out of money, and the Land Sharks announced we were not making good progress toward our company goals. Put more starkly, they would put up additional capital, but only at a price lower than the previous financing round. We were facing the dreaded "down round." After the initial (often painful) negotiation, when ownership is a zero-sum game between the company team and the Land Sharks, the investors and the team are usually on the same side of the table. The goal is to build a successful company, raising money when necessary at increasing stock prices. The only disagreement concerns the "spend." The investors naturally want you to spend more to make faster progress, since that would ensure them an increasing percentage ownership of the company. In contrast, the team wants to "kiss every nickel" to minimize the amount of capital raised and maximize their ownership. Resolving these differences is usually pretty straightforward. When a new round of capital is needed, a new investor is typically brought in to set the price of the round. It is in the team's interest to make this as high as possible. The current investors will be asked to support the round, by adding their pro-rata share at whatever price is agreed on.

However, what happens if the current investors refuse to support a new round at a higher price? Naturally, a new investor will follow the lead of the current ones, and a new lower price is established. At this point, there is a clause in most financing agreements that the company must ex post facto reprice the previous financing round (or rounds) down to the new price. As you can imagine, a down round is incredibly dilutive financially to the team, who would naturally say, "If you want us to continue, you need to top up our options." As such, the discussion becomes a three-way negotiation among the existing investors, the new investors, and the team. It is another painful zero-sum game.

When the dust settled, the Illustra employees were largely made whole through new options, the percentage ownership among the Land Sharks had changed only slightly, and the whole process left a bitter taste. Moreover, management had been distracted for a couple of months. The Land Sharks seemed to be playing some sort of weird power game with each other I did not understand. Regardless, Illustra will live to fight another day.

The Future Looks Up (Again)

Troy, NY, Day 56. The innkeeper in Ellicottville tells us what was obvious to anybody in the 19th century moving goods between the eastern seaboard and the middle of the country. He said, “Ride north to the Erie Canal and hang a right.” After a pleasant (and flat) ride down the Mohawk Valley, we arrive at Troy and see our first road sign for Boston, now just 186 miles away. The end is three days off! I am reminded of a painted sign at the bottom of Wildcat Canyon Road in Orinda, CA, at the start of the hill that leads back to Berkeley from the East Bay. It says simply “The Last Hill.” We are now at our last hill. We need only climb the Berkshires to Pittsfield, MA. It is then easy riding to Boston.

Oakland, CA, 1995. Shortly after our down round and the Catch-22 on ADTs, serendipity occurred once more. The Internet was taking off, and most enterprises were trying to figure out what to do with it. Uptone executes a brilliant repositioning of Illustra. We became the “database for cyberspace,” capable of storing Internet data like text and images. He additionally received unbelievable airtime by volunteering Illustra to be the database for “24 Hours in Cyberspace,” a worldwide effort by photojournalists to create one Web page per hour, garnering a lot of positive publicity. Suddenly, Illustra was “the new thing,” and we were basking in reflected glory. Sales picked up and the future looked bright. The Voice-of-Experience stepped on the gas and we hired new people. Maybe this was the beginning of the widely envied “hockey stick of growth.” We were asked to do a pilot application for a very large Web vendor, a potentially company-making transaction. However, we were also in a bake-off with the traditional RDBMSs.

The Good Times Do Not Last Long

Oakland, CA, 1995. Reality soon rears its ugly head. Instead of doing a benchmark on a task we were good at (such as geographic search or integrating text with structured data and images), the Web vendor decided to compare us on a traditional bread-and-butter transaction-processing use case, in which the goal is to perform as many transactions per second as you can on a standard banking application. It justified its choice by saying, “Within every Internet application, there is a business data-processing sub-piece that accompanies the multimedia requirements, so we are going to test that first.”

There was immediately a pit in my stomach because Postgres was never engineered to excel at online transaction processing (OLTP). We were focused on ADTs, rules, and time travel, not on trying to compete with current RDBMSs on the turf for which they had been optimized. Although we were happy to do transactions, it was far outside our wheelhouse. Our performance was going to be an order-of-magnitude worse than what was offered by the traditional vendors we were competing against. The problem is a collection of architectural decisions I made nearly a decade earlier that are not easy to undo; for example, Illustra ran as an operating system process for each user. This architecture was well understood to be simple to implement but suffers badly on a highly concurrent workload with many users doing simple things. Moreover, we did not compile query plans aggressively, so our overhead to do simple things was high. When presented with complex queries or use cases where our ADTs were advantageous, these shortcomings are not an issue. But when running simple business data processing, we were going to lose, and lose badly.

We were stuck with the stark reality that we must dramatically improve transaction-processing performance, which will be neither simple nor quick. I spent hours with the Short One trying to find a way to make it happen without a huge amount of recoding, energy, cost, and delay. We drew a blank. Illustra would have to undergo a costly rearchitecting.

The Stories End

Sutton, MA, Day 59. Massachusetts roads are poorly marked, and we have never seen more discourteous drivers. Riding here is not pleasant, and we cannot imagine trying to navigate Boston Bound into downtown Boston, let alone find someplace where we can access the ocean. We settle instead for finishing at Wollaston Beach in

Wollaston Beach, MA: Day 59

Quincy, MA, approximately 10 miles south of Boston. After the perfunctory dragging of our bike across the beach and dipping the front wheel in the surf, we are done. We drink a glass of champagne at a beachside café and ponder what happens next.

Oakland, CA, February 1996. Serendipity occurs yet again. One of the vendors we competed against on the Web vendor's benchmark has been seriously threatened by the benchmark. It saw Illustra would win a variety of Internet-style benchmarks hands-down, and Web vendors would have substantial requirements in this area. As a result, it elected to buy Illustra. In many ways, this was the answer to all our issues. The company had a high-performance OLTP platform into which we could insert the Illustra features. It was also a big company with sufficient "throw-weight" to get application vendors to add ADTs to its system. We consummated what we thought was a mutually beneficial transaction and set to work putting Illustra features into its engine.

I will end the Illustra story here, even though there is much more to tell, most of it fairly dark—a shareholder lawsuit, multiple new CEOs, and ultimately a sale of the company. The obvious takeaway is to be very careful about the choice of company you agree to marry.

Why a Bicycle Story?

You might wonder why I would tell this bicycling story. There are three reasons. First, I want to give you an algorithm for successfully riding across America.

```
Until (Ocean) {
       Get up in the morning;
       Ride east;
       Persevere, and overcome any obstacles that arise;
       }
```

It is clear that following this algorithm will succeed. Sprinkle in some serendipity if it occurs. Now abstract it a bit by substituting `goal` for `Ocean` and `Appropriate Action` for `Ride east`

```
Until (Goal) {
       Get up in the morning;
       Appropriate action;
       Persevere, and overcome any obstacles that arise;
       }
```

Since I will be using this algorithm again, I will make it a macro

```
Make-it-happen (Goal);
```

With this preamble, I can give a thumbnail sketch of my résumé, circa 1988.

```
Make-it-happen (Ph.D.);
Make-it-happen (Tenure);
Make-it-happen (Ocean);
```

In my experience, getting a Ph.D. (approximately five years) is an example of this algorithm at work. There are ups (passing prelims), downs (failing quals the first time), and a lot of slog through the swamp (writing a thesis acceptable to my committee). Getting tenure (another five years) is an even less pleasant example of this algorithm at work.

This introduces the second reason for presenting the algorithm. The obvious question is, “Why would anybody want to do this bicycle trip?” It is long and very difficult, with periods of depression, elation, and boredom, along with the omnipresence of poor food. All I can say is, “It sounded like a good idea, and I would go again in a heartbeat.” Like a Ph.D. and tenure, it is an example of `make-it-happen` in action. The obvious conclusion to draw is I am programmed to search out `make-it-happen` opportunities and get great satisfaction from doing so.

I want to transition here to the third reason for telling the bicycle story. Riding across America is a handy metaphor for building system software. Let me start by writing down the algorithm for building a new DBMS (see code section 2).

The next question is, “How do I come up with a new idea?” The answer is, “I don’t know.” However, that will not stop me from making a few comments. From personal experience, I never come up with anything by going off to a mountaintop to think. Instead, my ideas come from two sources: talking to real users with real problems and then trying to solve them. This ensures I come up with ideas that somebody cares about and the rubber meets the road and not the sky. The second source is to bounce possibly good (or bad) ideas off colleagues that will challenge them. In summary, the best chance for generating a good idea is to spend time in the real world and find an environment (like MIT/CSAIL and Berkeley/EECS) where you will be intellectually challenged.

Code Section 2.

```
Until (it works) {
          Come up with a new idea;
          Prototype it with the help of superb computer scientists;
          Persevere, fixing whatever problems come up; always
          remembering that it is never too late to throw everything
          away;
          }
```

Code Section 3.

```
Until (first few customers) {
         With shoe leather, find real world users who will say,
         "If you build this for real, I will buy it";
         }
Recruit seasoned start-up executives;
Recruit crack engineers;
Until (success or run-out-of-money) {
         Persevere, fixing whatever problems come up;
         }
```

If your ideas hold water and you have a working prototype, then you can proceed to phase two, which has a by-now-familiar look (see code section 3).

As with other system software, building a new DBMS is difficult, takes a decade or so, and involves periods of elation and depression. Unlike bicycling across America, which takes just muscles and perseverance, building a new DBMS involves other challenges. In the prototype phase, one must figure out new interfaces, both internal and to applications, as well as to the operating system, networking, and persistent storage. In my experience, getting them right the first time is unusual. Unfortunately, one must often build it first to see how one should have built it. You will have to throw code away and start again, perhaps multiple times. Furthermore, everything influences everything else. Ruthlessly avoiding complexity while navigating a huge design space is a supreme engineering challenge. Making the software fast and scalable just makes things more difficult. It is a lot like riding across America.

Commercialization adds its own set of challenges. The software must really work, generating the right answer, never crashing, and dealing successfully with all the corner cases, including running out of any computer resource (such as main memory and disk). Moreover, customers depend on a DBMS to never lose their data, so transaction management must be bulletproof. This is more difficult than it looks, since DBMSs are multi-user software. Repeatable bugs, or “Bohrbugs,” are easy to knock out of a system, leaving the killers, nonrepeatable errors, or “Heisenbugs.” Trying to find nonrepeatable bugs is an exercise in frustration. To make matters worse, Heisenbugs are usually in the transaction system, causing customers to lose data. This reality has generated a severe pit in my stomach on several occasions. Producing (and testing) system software takes a long time and costs a lot of money. The system programmers who are able to do this have my admiration. In summary, building and commercializing a new DBMS can be characterized by

```
Have a good idea (or two or three);
Make-it-happen--for a decade or so;
```

This brings up the obvious question: “Why would anybody want to do something this difficult?” The answer is the same as with a Ph.D., getting tenure, or riding across America. I am inclined to accept such challenges. I spent a decade struggling to make Postgres real and would do it again in a heartbeat. In fact, I have done it multiple times since Postgres.

The Present Day

I will finish this narrative by skipping to 2016 to talk about how things ultimately turned out. For those of you who were expecting this article to be a commentary on current good (and not-so-good) ideas, you can watch my IEEE International Conference on Data Engineering 2015 talk on this topic at http://kdb.snu.ac.kr/data/stonebraker_talk.mp4 or the video that accompanies this article in the ACM Digital Library.

Moultonborough, NH, present day. Boston Bound arrived in California the same way it left, on the roof of our car. It now sits in our basement in New Hampshire gathering dust. It has not been ridden since that day at Wollaston Beach.

I am still inclined to accept physical challenges. More recently, I decided to climb all 48 mountains in New Hampshire that are over 4,000 feet. In a softer dimension, I am struggling to master the five-string banjo.

Leslie is now Director of Marketing for an angel-investor-backed startup in New York City, whose software incidentally runs on Postgres. She refused to major in computer science.

Illustra was successfully integrated into the Informix code base. This system is still available from IBM, which acquired Informix in 2001. The original Illustra code line still exists somewhere in the IBM archives. The academic Postgres code line got a huge boost in 1995 when “Happy” and “Serious” replaced the QUEL query language with a SQL interface. It was subsequently adopted by a dedicated pick-up team that shepherd its development to this day. This is a shining example of open source development in operation. For a short history of this evolution, see Momjian [4]. This open source code line has also been integrated into several current DBMSs, including Greenplum and Netezza. Most commercial DBMSs have extended their engines with Postgres-style ADTs.

I now want to conclude with three final thoughts. First, I want to mention the other DBMSs I have built—Ingres, C-Store/Vertica, H-Store/VoltDB, and SciDB—all have development stories similar to that of Postgres. I could have picked any one of them to discuss in this article. All had a collection of superstar research programmers, on whose shoulders I have ridden. Over the years, they have turned my ideas into working prototypes. Other programming superstars have converted the prototypes into bulletproof working code for production deployment. Skilled startup executives have guided the small fragile companies with a careful hand. I am especially indebted to my current business partner, "Cueball," for careful stewardship in choppy waters. Moreover, I want to acknowledge the Land Sharks, without whose capital none of this would be possible, especially the "Believer," who has backed multiple of my East Coast companies.

I am especially indebted to my partner, Larry Rowe, and the following 39 Berkeley students and staff who wrote Postgres: Jeff Anton, Paul Aoki, James Bell, Jennifer Caetta, Philip Chang, Jolly Chen, Ron Choi, Matt Dillon, Zelaine Fong, Adam Glass, Jeffrey Goh, Steven Grady, Serge Granik, Marti Hearst, Joey Hellerstein, Michael Hirohama, Chin-heng Hong, Wei Hong, Anant Jhingren, Greg Kemnitz, Marcel Kornacker, Case Larsen, Boris Livshitz, Jeff Meredith, Ginger Ogle, Mike Olson, Nels Olsen, LayPeng Ong, Carol Paxson, Avi Pfeffer, Spyros Potamianos, Sunita Surawagi, David Muir Sharnoff, Mark Sullivan, Cimarron Taylor, Marc Teitelbaum, Yongdong Wang, Kristen Wright, and Andrew Yu.

Second, I want to acknowledge my wife, Beth. Not only did she have to spend two months looking at my back as we crossed America, she also gets to deal with my goal orientation, desire to start companies, and, often, ruthless focus on "the next step." I am difficult to live with, and she is long-suffering. I am not sure she realizes she is largely responsible for keeping me from falling off my own personal cliffs.

Third, I want to acknowledge my friend, colleague, and occasional sounding board, Jim Gray, recipient of the ACM A.M. Turing Award in 1998. He was lost at sea nine years ago on January 28, 2007. I think I speak for the entire DBMS community when I say: Jim: We miss you every day.

References

[1] Date, C. Referential integrity. In *Proceedings of the Seventh International Conference on Very Large Data Bases Conference* (Cannes, France, Sept. 9–11). Morgan Kaufmann Publishers, 1981, 2–12.

[2] Madden, S. *Mike Stonebraker's 70th Birthday Event.* MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, Apr. 12, 2014; http:// webcast.mit.edu/ spr2014/csail/12apr14/

[3] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. Aries: A transaction recovery method supporting fine granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems 17*, 1 (Mar. 1992), 94–162.

[4] Momjian, B. *The History of PostgreSQL Open Source Development;* https://momjian.us/ main/writings/pgsql/history.pdf

[5] Stonebraker, M. The design of the Postgres storage system. In *Proceedings of the 13th International Conference on Very Large Data Bases Conference* (Brighton, England, Sept. 1–4). Morgan Kaufmann Publishers, 1987, 289–300.

[6] Stonebraker, M. and Rowe, L. The design of Postgres. In *Proceedings of the 1986 SIGMOD Conference* (Washington, D.C., May 28–30). ACM Press, New York, 1986, 340–355.

[7] Stonebraker, M and Rowe, L. The Postgres data model. In *Proceedings of the 13th International Conference on Very Large Data Bases Conference* (Brighton, England, Sept. 1–4). Morgan Kaufmann Publishers, 1987, 83–96.

[8] Stonebraker, M., Rubenstein, B., and Guttman, A. Application of abstract data types and abstract indices to CAD databases. In *Proceedings of the ACM-IEEE Workshop on Engineering Design Applications* (San Jose, CA, May). ACM Press, New York, 1983, 107–113.

Michael Stonebraker (stonebraker@csail.mit.edu) is an adjunct professor in the MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA.

PART II

MIKE STONEBRAKER'S CAREER

1

Make it Happen: The Life of Michael Stonebraker

Samuel Madden

Make it happen.

—Michael Stonebraker

Synopsis

Michael Stonebraker is an American computer scientist, teacher, inventor, technology entrepreneur, and intellectual leader of the database field for the last nearly 40 years. With fellow professor Eugene Wong of the University of California at Berkeley, he developed the first relational database management system (RDBMS) prototype (Ingres) that, together with IBM's System R and Oracle's Oracle database, proved the viability of the RDBMS market, and made many lasting contributions to the design of RDBMS systems, most notably developing the Object-Relational model that became the de facto way of extending database systems with abstract data types (ADTs), as a part of his post-Ingres Postgres project.

A leading contributor to both research and industry, Stonebraker took a distinctively different approach to database research: He emphasized targeting real-life problems over more abstract, theoretical research. His research is notable for its focus on open, working academic prototypes and on his repeated efforts to prove out his ideas through commercialization: founding or co-founding nine companies based on his research. Stonebraker has of this book produced more than 300 research papers[1] (see "The Collected Works of Michael Stonebraker," p. 607), influenced hundreds of students, and advised 31 Ph.D. students (see "Michael Stonebraker's Student Genealogy," p. 52), many of whom went on to successful academic careers or to found successful startup companies on their own.

More than any other person, Stonebraker made Edgar (Ted) Codd's vision [Codd 1970] of data independence and the relational database model a reality, leading to the $55 billion-plus market that exists today. Stonebraker-originated ideas appear in virtually every relational database product on the market, as he and others have taken his open-source code, refined it, built on it, and extended it. As a result of this, Stonebraker has had a multiplier effect on the relational database market. For his pioneering ideas, Stonebraker received the 2014 ACM A.M. Turing Award citing his "fundamental contributions to the concepts and practices underlying modern database systems," many of which were developed as a part of the Ingres and Postgres projects.

Early Years and Education[2]

Michael "Mike" Stonebraker was born on October 11, 1943, in Newburyport, Massachusetts, the middle of three sons born to an engineer and a schoolteacher. He grew up in Milton Mills, New Hampshire, near the Maine border. His parents placed a high emphasis on education; when Stonebraker was ten, his father moved the family to Newbury, Massachusetts, home to the prestigious Governor's Academy (formerly Governor Dummer Academy). As Newbury had no local high school at the time, the town would pay day-student tuition for any local residents who could qualify academically, resulting in all three Stonebraker boys attending the school.

Stonebraker excelled at mathematics and the sciences in high school, and upon graduating in 1961, enrolled in Princeton University. He graduated with a bachelor's degree in Electrical Engineering from Princeton in 1965. There were no computer classes at Princeton nor a computer science major at the time.

As a young man graduating college in 1965 during the Vietnam War, Stonebraker recalls in the "Oral History of Michael Stonebraker" [Grad 2007] that he had four life choices: "go to Vietnam, go to Canada, go to jail, or go to graduate school." The decision was obvious: backed by a National Science Foundation (NSF) fellowship, he enrolled in graduate school at the University of Michigan at Ann Arbor, joining

[1] Source: DBLP (http://dblp.uni-trier.de/pers/hd/s/Stonebraker:Michael. Last accessed April 8, 2018.

[2] This section and the next draw on the excellent resource by Grad [2007].

Table 1.1 The academic positions of Michael Stonebraker

Assistant Professor of Computer Science	University of California at Berkeley	1971–1976
Associate Professor	University of California at Berkeley	1976–1982
Professor	University of California at Berkeley	1982–1994
Professor of the Graduate School	University of California at Berkeley	1994–1999
Senior Lecturer	Massachusetts Institute of Technology	2000–2001
Adjunct Professor	Massachusetts Institute of Technology	2002–Present

the Computer Information and Control Engineering (CICE) program, a joint program in engineering focused on computer science. He received a M.Sc. in CICE in 1967.

With his options no better in 1967, he decided to stay on at Ann Arbor to get his Ph.D., which he received in 1971 for his doctoral dissertation "The Reduction of Large Scale Markov Models for Random Chains" [Stonebraker 1971c]—which he describes as theoretical research with limited applications. (Indeed, one wonders if his focus on applicability in his later work wasn't a reaction to the lack of applicability of his early research.)

Academic Career and the Birth of Ingres

In 1971, Stonebraker was hired as an assistant professor at the University of California at Berkeley to work on the application of technology to public systems in the Electrical Engineering and Computer Science (EECS) Department. He would go on to spend the next 28 years as an influential and highly productive professor at UC Berkeley (see "Michael Stonebraker's Student Genealogy," p. 52), retiring as professor of the graduate school in 1999 before moving east to join MIT (see Table 1.1).

As a new assistant professor, he soon discovered that getting data for his public-systems work was very hard, and that his interest in Urban Dynamics—modeling and applying data to predicting growth in urban areas—wasn't going to help him "get famous." [Grad 2007]

While Stonebraker was casting around for more fame-carrying material, Professor Eugene Wong suggested that he read Ted Codd's seminal paper. Stonebraker also read the CODASYL (Conference on Data Systems Languages) report [Metaxides et al. 1971] but dismissed the latter as far too complicated. He had a better idea.

In the “Oral History of Michael Stonebraker” [Grad 2007], he recalled:

> . . . I couldn’t figure out why you would want to do anything that complicated and Ted’s work was simple, easy to understand. So it was pretty obvious that the naysayers were already saying nobody who didn’t have a Ph.D. could understand Ted Codd’s predicate calculus or his relational algebra. And even if you got past that hurdle, nobody could implement the stuff efficiently.
>
> And even if you got past that hurdle, you could never teach this stuff to COBOL programmers. So it was pretty obvious that the right thing to do was to build a relational database system with an accessible query language. So Gene [Wong] and I set out to do that in 1972. And you didn’t have to be a rocket scientist to realize that this was an interesting research project.

This project eventually became the Ingres system (see Chapters 5 and 15). The challenges faced by Stonebraker in the Ingres project were daunting: they amounted to nothing less than developing automatic programming techniques to convert declarative query specifications into executable algorithms that could be evaluated as efficiently as code written by skilled programmers on the leading commercial systems of the day—all of this over a new, unproven data model. Remarkably, Stonebraker at the time was an assistant professor at UC Berkeley, starting the project just two years after completing his Ph.D., and, along with the System R team at IBM (see Chapter 35), developing the ideas and approaches that made relational databases a reality. Many of the Ingres ideas and approaches are still used by every relational database system today, including the use of views and query rewriting for data integrity and access control, persistent hash tables as a primary access method, primary-copy replication control, and the implementation of rules/triggers in database systems. Additionally, experimental evaluation within the Ingres project provided critical insights into issues involved in building a locking system that could provide satisfactory transaction performance.

Because the Ingres and System R teams communicated closely during the development of these systems, it is sometimes difficult to tease apart the individual contributions; they both received the ACM Software System Award in 1988 for their pioneering work. One of the central ideas invented by Stonebraker in Ingres was the use of query modification to implement views. A view is a virtual table in a database that is not physically present but is instead defined as a database query. The idea of views appears nearly concurrently in papers from Ingres and the System R project, but Stonebraker developed the algorithms that made their implementation practical and that are still the way databases support them today. He also showed that these techniques could be used for preserving database access control and integrity.

Stonebraker's ideas first appear in his 1974 paper "Access Control in a Relational Database Management System By Query Modification" [Stonebraker and Wong 1974b] and are later developed in his 1976 paper "Implementation of Integrity Constraints and Views By Query Modification" [Stonebraker 1975]. His key idea is to implement views using the rewriting technique he called "interaction modification"—essentially rewriting queries over views into queries over the physical tables in the database. This is an extremely powerful and elegant implementation idea that has been adopted for many other features besides views, including authorization, integrity enforcement, and data protection.

The Post-Ingres Years

After Ingres, Stonebraker and his students began working on the follow-on Postgres project in the 1980s (see Chapter 16). Like Ingres, Postgres was hugely influential. It was the first system to support the Object-Relational data model, which allows the incorporation of abstract data types into the relational database model. This enabled programmers to "move code to data," embedding sophisticated abstract data types and operations on them directly inside of the database.

Stonebraker described this model, including extensions to the language and necessary modifications to the database implementation, in his 1986 paper "Object Management in Postgres Using Procedures" [Stonebraker 1986c], which included insights learned through earlier experiments with "user-defined functions" in Ingres in the early 1980s. In contrast to the prevailing ideas of the time that proposed integrating persistent objects into object-oriented programming languages, Stonebraker's idea allowed the relational model to thrive while obtaining the benefits of a variety of rich new data types. Again, this idea is used in every modern database system. It has also become increasingly important in recent years, with all the buzz about "Big Data" as database vendors have grown their systems into "analytic platforms" that support complex statistics, machine learning, and inference algorithms. These features are provided using the Object-Relational interfaces pioneered by Stonebraker. Like Ingres, the Postgres system explored a number of other radical ideas that were well before their time, including notions of immutable data and historical "time travel" queries in databases, and the use of persistent memory to implement lightweight transaction schemes.

In both Ingres and Postgres, the impact of Stonebraker's ideas was amplified tremendously by the fact that they were embodied in robust implementations that were widely used. These systems were so well engineered that they form the basis of many modern database systems. For example, Ingres was used to build Sybase SQL Server, which then became Microsoft SQL Server, and Postgres has been used as the basis for many of the new commercial databases developed over the last 20 years, including those from Illustra, Informix, Netezza, and Greenplum. For a complete view of Ingres' impact on other DBMSs, see the RDBMS genealogy in Chapter 13.

Industry, MIT, and the New Millennium

After the Postgres project, Stonebraker became heavily involved in industry, most notably at Illustra Information Technologies, Inc., turning ideas from the Postgres project into a major commercial database product. In 1997, Illustra was acquired by Informix, Inc., which brought on Stonebraker as CTO until his departure in 2000.

In 1999, Stonebraker moved to New Hampshire, and in 2001 started as an adjunct professor at MIT. Despite already having compiled an impressive array of academic and commercial successes, Stonebraker launched a remarkable string of research projects and commercial companies at the start of the millennium, beginning with the Aurora and StreamBase projects (see Chapter 17), which he founded with colleagues from Brandeis University, Brown University, and MIT. The projects explored the idea of managing streams of data, using a new data model and query language where the focus was on continuously arriving sequences of data items from external data sources such as sensors and Internet data feeds. Stonebraker co-founded StreamBase Systems in 2003 to commercialize the technology developed in Aurora and Borealis. StreamBase was acquired by TIBCO Software Inc. in 2013.

In 2005, again with colleagues from Brandeis, Brown, and MIT, Stonebraker launched the C-Store project, where the aim was to develop a new type of database system focused on so-called data analytics, where long-running, scan-intensive queries are run on large, infrequently updated databases, as opposed to transactional workloads, which focus on many small concurrent reads and writes to individual database records. C-Store was a shared-nothing, column-oriented database designed for these workloads (see Chapter 18). By storing data in columns, and accessing only the columns needed to answer a particular query, C-Store was much more input/output-efficient (I/O-efficient) than conventional systems that stored data in rows, offering order-of-magnitude speedups vs. leading commercial systems at the time. That same year, Stonebraker co-founded Vertica Systems, Inc., to commercialize the technology behind C-Store; Vertica Systems was acquired in 2011 by Hewlett-Packard, Inc. Although Stonebraker wasn't the first to propose column-oriented database ideas, the success of Vertica led to a proliferation of commercial systems that employed column-oriented designs, including the Microsoft Parallel Data Warehouse project (now subsumed into Microsoft SQL Server as Column Store indexes) and the Oracle In-Memory Column Store.

After C-Store, Stonebraker continued his string of academic and industrial endeavors, including the following.

- In 2006, the Morpheus project, which became Goby, Inc., focused on a data integration system to transform different web-based data sources into a unified view with a consistent schema.
- In 2007, the H-Store project, which became VoltDB, Inc., focused on online transaction processing (see Chapter 19).
- In 2008, the SciDB project, which became Paradigm4, Inc., focused on array-oriented data storage and scientific applications (see Chapter 20).
- In 2011, the Data Tamer project, which became Tamr Inc., focused on massive-scale data integration and unification (see Chapter 21).

Stonebraker's Legacy

Stonebraker's systems have been so influential for two reasons.

First, Stonebraker engineered his systems to deliver performance and usability that allowed them to still live on 40 years later. The engineering techniques necessary to achieve this are succinctly described in two extremely influential systems papers, the 1976 paper "The Design and Implementation of Ingres" [Stonebraker et al. 1976b] and the 1986 paper "The Implementation of Postgres" [Stonebraker et al. 1990b]. These papers have taught many generations of students and practitioners how to build database systems.

Second, Stonebraker systems were built on commodity Unix platforms and released as open source. Building these systems on "low end" Unix machines required careful thinking about how database systems would use the operating system. This is in contrast to many previous database systems that were built as vertical stacks on "bare metal." The challenges of building a database system on Unix, and a number of proposed solutions, were described in Stonebraker's 1986 paper "Operating System Support for Data Management" [Stonebraker and Kumar 1986]. The Ingres and Postgres prototypes and this seminal paper pushed both OS implementers and database designers toward a mutual understanding that has become key to modern OS support of the long-running, I/O-intensive services that underpin many modern scalable systems. Furthermore, by releasing his systems as open source, Stonebraker enabled innovation in both academia and industry, as many new databases and research papers were based on these artifacts. For more on the impact of Stonebraker's open-source systems, see Chapter 12.

In summary, Stonebraker is responsible for much of the software foundation of modern database systems. Charlie Bachman, Ted Codd, and Jim Gray made monumental contributions to data management, earning each of them the Turing Award. Like them, Stonebraker developed many of the fundamental ideas used in every modern relational database system. However, more than any other individual, it was Stonebraker who showed that it was possible to take the relational model from theory to practice. Stonebraker's software artifacts continue to live on as open-source and commercial products that contain much of the world's important data. His ideas continue to impact the design and features of many new data processing systems—not only relational databases but also the "Big Data" systems that have recently gained prominence. Remarkably, Stonebraker continues to be extremely productive and drive the research agenda for the database community (see Chapter 3), contributing innovative research (see Chapters 22 and 23), and having massive intellectual and commercial impact through his work on stream processing, column stores, scientific databases, and transaction processing. These are the reasons Stonebraker won the ACM A.M. Turing Award.

Companies

As of this writing, Stonebraker had founded/co-founded nine startup companies to commercialize his academic systems.

- Relational Technology, Inc., founded 1980; became Ingres Corporation (1989); acquired by Ask (1990), acquired by Computer Associates (1994); spun out as a private company Ingres Corp. (2005); acquired VectorWise (2010); name change to Actian (2011); acquired Versant Corporation (2012); acquired Pervasive Software (2013); acquired ParAcceJ (2013). In 2016, Actian phased out ParAccel, VectorWise, and DataFlow, retaining Ingres.
- Illustra Information Technologies, founded 1992; acquired in 1996 by Informix, where Stonebraker was Chief Technology Officer 1996–2000, and was later acquired by IBM.
- Cohera Corporation, founded 1997 (acquired by PeopleSoft).
- StreamBase Systems, founded 2003 (acquired by TIBCO in 2013).
- Vertica Systems, founded 2005 (acquired by HP in 2011).

- Goby, founded in 2008 (acquired by Telenauv in 2011).
- VoltDB, founded in 2009.
- Paradigm4, founded in 2010.
- Tamr, founded in 2013.

Awards and Honors

See Table 1.2 for further information on Michael Stonebraker's awards and honors.

Service

See Table 1.3 for further details on Michael Stonebraker's service.

Table 1.2 Awards and honors of Michael Stonebraker

ACM Software System Award	1988
ACM SIGMOD Innovation Award	1994
ACM Fellow	1993
ACM SIGMOD "Test of Time" Award (best paper, 10 years later)	1997, 2017
National Academy of Engineering	Elected 1998
IEEE John von Neumann Medal	2005
American Academy of Arts and Sciences	2011
Alan M. Turing Award	2014

Table 1.3 Service of Michael Stonebraker

Chairman of ACM SIGMOD	1981–1984
General Chairperson, SIGMOD 1987	1987
Program Chairperson SIGMOD 1992	1992
Organizer, the Laguna Beach Workshop	1988
Organizer, the Asilomar DBMS Workshop	1996
SIGMOD Awards Committee	2001–2005
Co-founder, CIDR Conference on Visionary DBMS Research (with David J. DeWitt and Jim Gray)	2002
ACM System Software Awards Committee	2006–2009

Advocacy

In addition to his academic and industrial accomplishments, Stonebraker has been a technical iconoclast, acted as the de facto leader of the database community, and served as a relentless advocate for relational databases. Sometimes controversially, he's been unafraid to question the technical direction of a variety of data-oriented technologies, including:

- Advocated in favor of the Object-Relational data model over the object-oriented approach pursued by many other research groups and companies in the mid-1980s.
- Argued against the traditional idea of the "one-size-fits-all" database in favor of domain specific designs like C-Store, H-Store, and SciDB [Stonebraker and Çetintemel 2005].
- Criticized weaker data management solutions in the face of vocal opposition, including NoSQL [Stonebraker 2010a, Stonebraker 2011b], MapReduce [DeWitt and Stonebraker 2008, Stonebraker et al. 2010], and Hadoop [Barr and Stonebraker 2015a].
- Co-founded the biennial Conference on Innovative Data Systems Research (CIDR) to address shortcomings of existing conferences, and created many special-purpose workshops.

Personal Life

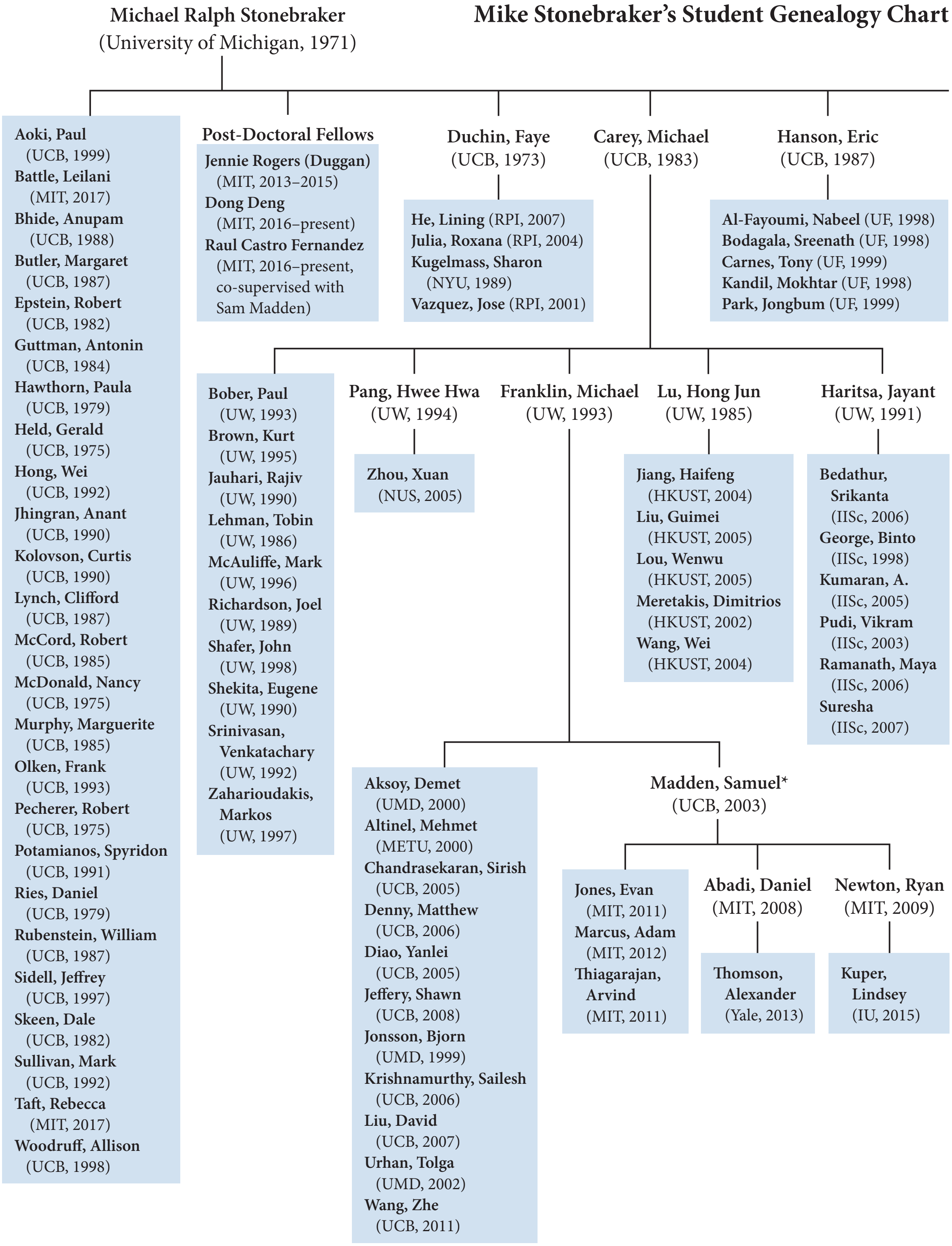
Stonebraker is married to Beth Stonebraker,[3] née Rabb. They have two grown daughters, Lesley and Sandra. They have homes in Boston's Back Bay (where he bicycles between home, MIT CSAIL, and his Cambridge startups) and on Lake Winnipesaukee in New Hampshire, where he has climbed all forty-eight 4,000-foot mountains and plies the lake waters in his boat. They are contributors to many causes in both communities. He plays the five-string banjo and (if you ask nicely) he may arrange a performance of his bluegrass pickup band, "Shared Nothing," with fellow musicians and computer scientists John "JR" Robinson and Stan Zdonik (the latter of Brown University).

Acknowledgments

The author wishes to acknowledge Janice L. Brown for her contributions to this chapter.

[3] Stonebraker's co-pilot in the cross-country tandem-bicycle trip recounted this in Stonebraker [2016].

Mike Stonebraker's Student Genealogy Chart



Michael Ralph Stonebraker
(University of Michigan, 1971)

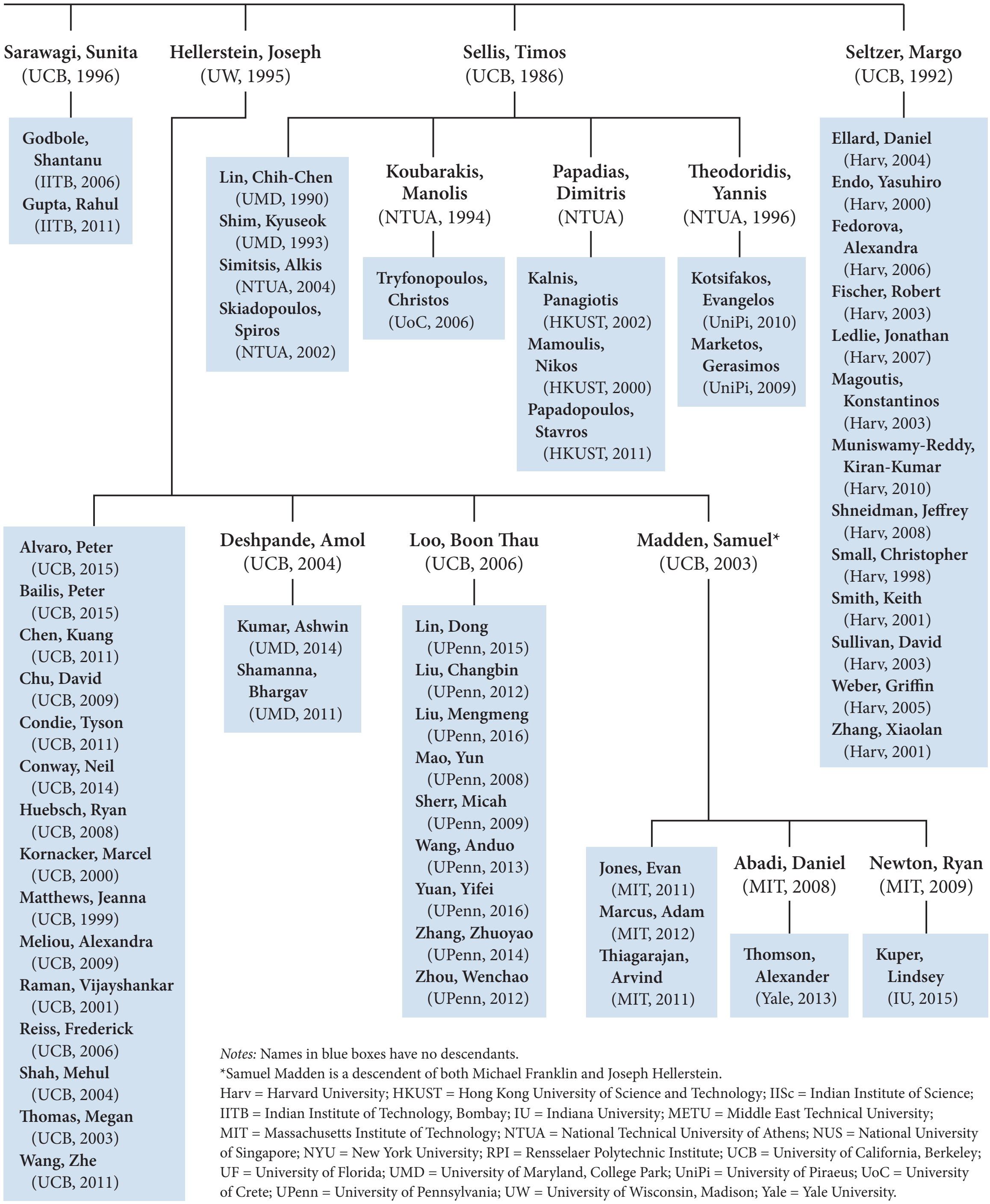
- Aoki, Paul (UCB, 1999)
- Battle, Leilani (MIT, 2017)
- Bhide, Anupam (UCB, 1988)
- Butler, Margaret (UCB, 1987)
- Epstein, Robert (UCB, 1982)
- Guttman, Antonin (UCB, 1984)
- Hawthorn, Paula (UCB, 1979)
- Held, Gerald (UCB, 1975)
- Hong, Wei (UCB, 1992)
- Jhingran, Anant (UCB, 1990)
- Kolovson, Curtis (UCB, 1990)
- Lynch, Clifford (UCB, 1987)
- McCord, Robert (UCB, 1985)
- McDonald, Nancy (UCB, 1975)
- Murphy, Marguerite (UCB, 1985)
- Olken, Frank (UCB, 1993)
- Pecherer, Robert (UCB, 1975)
- Potamianos, Spyridon (UCB, 1991)
- Ries, Daniel (UCB, 1979)
- Rubenstein, William (UCB, 1987)
- Sidell, Jeffrey (UCB, 1997)
- Skeen, Dale (UCB, 1982)
- Sullivan, Mark (UCB, 1992)
- Taft, Rebecca (MIT, 2017)
- Woodruff, Allison (UCB, 1998)

Post-Doctoral Fellows
- Jennie Rogers (Duggan) (MIT, 2013–2015)
- Dong Deng (MIT, 2016–present)
- Raul Castro Fernandez (MIT, 2016–present, co-supervised with Sam Madden)

Duchin, Faye (UCB, 1973)
- He, Lining (RPI, 2007)
- Julia, Roxana (RPI, 2004)
- Kugelmass, Sharon (NYU, 1989)
- Vazquez, Jose (RPI, 2001)

Carey, Michael (UCB, 1983)
- Bober, Paul (UW, 1993)
- Brown, Kurt (UW, 1995)
- Jauhari, Rajiv (UW, 1990)
- Lehman, Tobin (UW, 1986)
- McAuliffe, Mark (UW, 1996)
- Richardson, Joel (UW, 1989)
- Shafer, John (UW, 1998)
- Shekita, Eugene (UW, 1990)
- Srinivasan, Venkatachary (UW, 1992)
- Zaharioudakis, Markos (UW, 1997)
- Pang, Hwee Hwa (UW, 1994)
  - Zhou, Xuan (NUS, 2005)
- Franklin, Michael (UW, 1993)
  - Aksoy, Demet (UMD, 2000)
  - Altinel, Mehmet (METU, 2000)
  - Chandrasekaran, Sirish (UCB, 2005)
  - Denny, Matthew (UCB, 2006)
  - Diao, Yanlei (UCB, 2005)
  - Jeffery, Shawn (UCB, 2008)
  - Jonsson, Bjorn (UMD, 1999)
  - Krishnamurthy, Sailesh (UCB, 2006)
  - Liu, David (UCB, 2007)
  - Urhan, Tolga (UMD, 2002)
  - Wang, Zhe (UCB, 2011)
  - Madden, Samuel* (UCB, 2003)
    - Jones, Evan (MIT, 2011)
    - Marcus, Adam (MIT, 2012)
    - Thiagarajan, Arvind (MIT, 2011)
    - Abadi, Daniel (MIT, 2008)
      - Thomson, Alexander (Yale, 2013)
    - Newton, Ryan (MIT, 2009)
      - Kuper, Lindsey (IU, 2015)
- Lu, Hong Jun (UW, 1985)
  - Jiang, Haifeng (HKUST, 2004)
  - Liu, Guimei (HKUST, 2005)
  - Lou, Wenwu (HKUST, 2005)
  - Meretakis, Dimitrios (HKUST, 2002)
  - Wang, Wei (HKUST, 2004)
- Haritsa, Jayant (UW, 1991)
  - Bedathur, Srikanta (IISc, 2006)
  - George, Binto (IISc, 1998)
  - Kumaran, A. (IISc, 2005)
  - Pudi, Vikram (IISc, 2003)
  - Ramanath, Maya (IISc, 2006)
  - Suresha (IISc, 2007)

Hanson, Eric (UCB, 1987)
- Al-Fayoumi, Nabeel (UF, 1998)
- Bodagala, Sreenath (UF, 1998)
- Carnes, Tony (UF, 1999)
- Kandil, Mokhtar (UF, 1998)
- Park, Jongbum (UF, 1999)



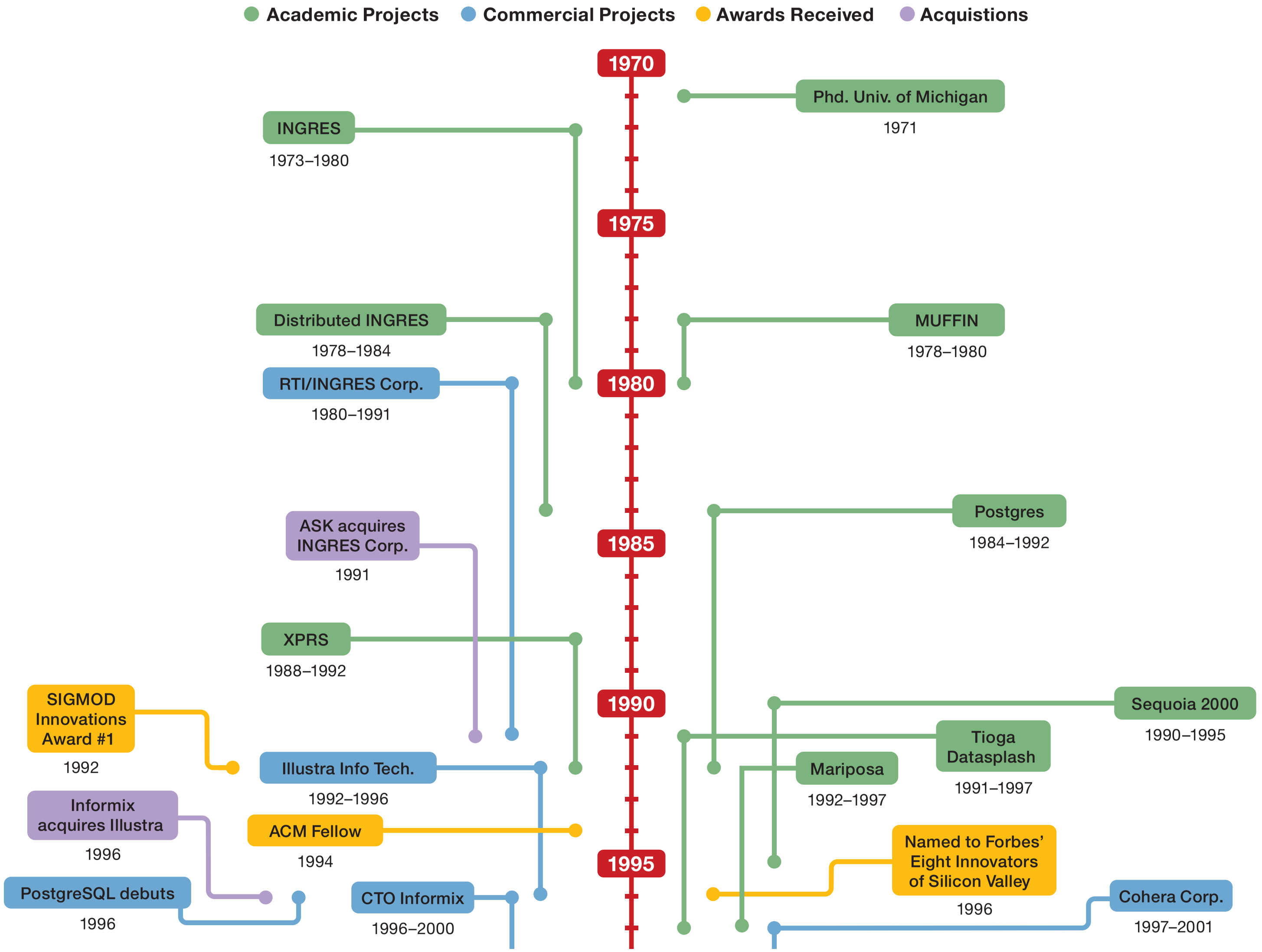
- Sarawagi, Sunita (UCB, 1996)
  - Godbole, Shantanu (IITB, 2006)
  - Gupta, Rahul (IITB, 2011)
- Hellerstein, Joseph (UW, 1995)
  - Alvaro, Peter (UCB, 2015)
  - Bailis, Peter (UCB, 2015)
  - Chen, Kuang (UCB, 2011)
  - Chu, David (UCB, 2009)
  - Condie, Tyson (UCB, 2011)
  - Conway, Neil (UCB, 2014)
  - Huebsch, Ryan (UCB, 2008)
  - Kornacker, Marcel (UCB, 2000)
  - Matthews, Jeanna (UCB, 1999)
  - Meliou, Alexandra (UCB, 2009)
  - Raman, Vijayshankar (UCB, 2001)
  - Reiss, Frederick (UCB, 2006)
  - Shah, Mehul (UCB, 2004)
  - Thomas, Megan (UCB, 2003)
  - Wang, Zhe (UCB, 2011)
  - Deshpande, Amol (UCB, 2004)
    - Kumar, Ashwin (UMD, 2014)
    - Shamanna, Bhargav (UMD, 2011)
  - Loo, Boon Thau (UCB, 2006)
    - Lin, Dong (UPenn, 2015)
    - Liu, Changbin (UPenn, 2012)
    - Liu, Mengmeng (UPenn, 2016)
    - Mao, Yun (UPenn, 2008)
    - Sherr, Micah (UPenn, 2009)
    - Wang, Anduo (UPenn, 2013)
    - Yuan, Yifei (UPenn, 2016)
    - Zhang, Zhuoyao (UPenn, 2014)
    - Zhou, Wenchao (UPenn, 2012)
  - Madden, Samuel* (UCB, 2003)
    - Jones, Evan (MIT, 2011)
    - Marcus, Adam (MIT, 2012)
    - Thiagarajan, Arvind (MIT, 2011)
    - Abadi, Daniel (MIT, 2008)
      - Thomson, Alexander (Yale, 2013)
    - Newton, Ryan (MIT, 2009)
      - Kuper, Lindsey (IU, 2015)
- Sellis, Timos (UCB, 1986)
  - Lin, Chih-Chen (UMD, 1990)
  - Shim, Kyuseok (UMD, 1993)
  - Simitsis, Alkis (NTUA, 2004)
  - Skiadopoulos, Spiros (NTUA, 2002)
  - Koubarakis, Manolis (NTUA, 1994)
    - Tryfonopoulos, Christos (UoC, 2006)
  - Papadias, Dimitris (NTUA)
    - Kalnis, Panagiotis (HKUST, 2002)
    - Mamoulis, Nikos (HKUST, 2000)
    - Papadopoulos, Stavros (HKUST, 2011)
  - Theodoridis, Yannis (NTUA, 1996)
    - Kotsifakos, Evangelos (UniPi, 2010)
    - Marketos, Gerasimos (UniPi, 2009)
- Seltzer, Margo (UCB, 1992)
  - Ellard, Daniel (Harv, 2004)
  - Endo, Yasuhiro (Harv, 2000)
  - Fedorova, Alexandra (Harv, 2006)
  - Fischer, Robert (Harv, 2003)
  - Ledlie, Jonathan (Harv, 2007)
  - Magoutis, Konstantinos (Harv, 2003)
  - Muniswamy-Reddy, Kiran-Kumar (Harv, 2010)
  - Shneidman, Jeffrey (Harv, 2008)
  - Small, Christopher (Harv, 1998)
  - Smith, Keith (Harv, 2001)
  - Sullivan, David (Harv, 2003)
  - Weber, Griffin (Harv, 2005)
  - Zhang, Xiaolan (Harv, 2001)

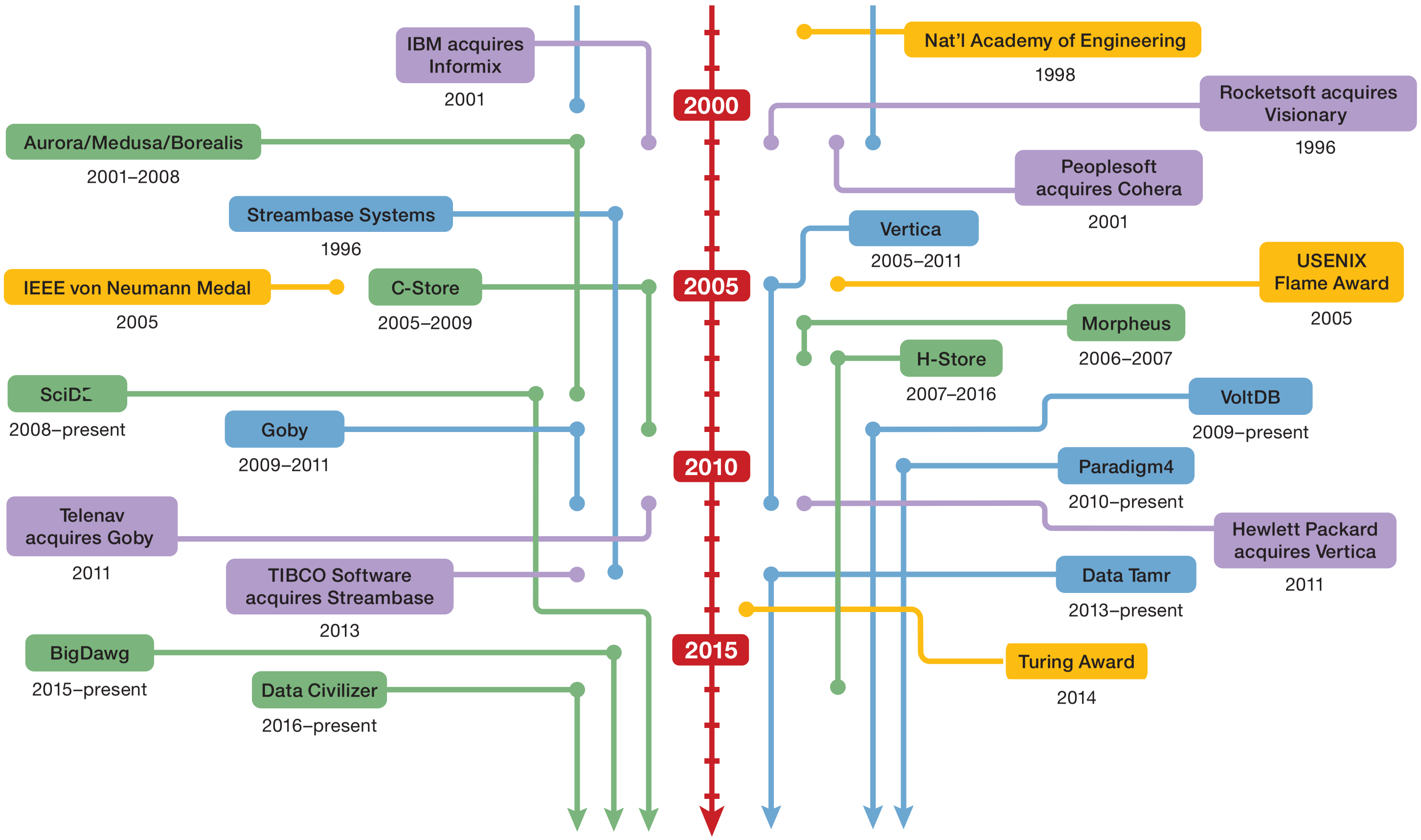
Notes: Names in blue boxes have no descendants.

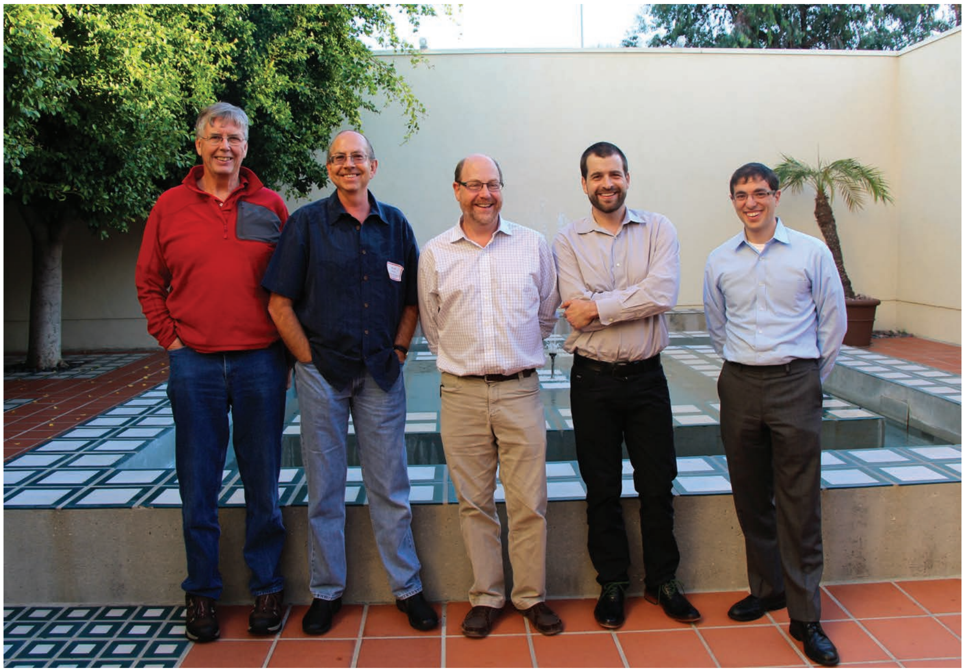
*Samuel Madden is a descendent of both Michael Franklin and Joseph Hellerstein.

Harv = Harvard University; HKUST = Hong Kong University of Science and Technology; IISc = Indian Institute of Science; IITB = Indian Institute of Technology, Bombay; IU = Indiana University; METU = Middle East Technical University; MIT = Massachusetts Institute of Technology; NTUA = National Technical University of Athens; NUS = National University of Singapore; NYU = New York University; RPI = Rensselaer Polytechnic Institute; UCB = University of California, Berkeley; UF = University of Florida; UMD = University of Maryland, College Park; UniPi = University of Piraeus; UoC = University of Crete; UPenn = University of Pennsylvania; UW = University of Wisconsin, Madison; Yale = Yale University.

The Career of Mike Stonebraker: The Chart (by A. Pavlo)




2000
2005
2010
2015
IBM acquires Informix
2001
Aurora/Medusa/Borealis
2001–2008
Streambase Systems
1996
IEEE von Neumann Medal
2005
C-Store
2005–2009
SciDB
2008–present
Goby
2009–2011
Telenav acquires Goby
2011
TIBCO Software acquires Streambase
2013
BigDawg
2015–present
Data Civilizer
2016–present
Nat'l Academy of Engineering
1998
Rocketsoft acquires Visionary
1996
Peoplesoft acquires Cohera
2001
Vertica
2005–2011
USENIX Flame Award
2005
Morpheus
2006–2007
H-Store
2007–2016
VoltDB
2009–present
Paradigm4
2010–present
Hewlett Packard acquires Vertica
2011
Data Tamr
2013–present
Turing Award
2014



Five generations of database researchers from the Stonebraker Student Genealogy chart are captured in this photo, taken at UC Irvine's Beckman Center (NAE West) in 2013. From left are Michael Stonebraker; Michael J. Carey of UC Irvine (Ph.D. student of Stonebraker when he was at UC Berkeley); Michael J. Franklin, Chairman of the University of Chicago's Department of Computer Science (Ph.D. student of Carey when he was at University of Wisconsin); Samuel Madden of MIT CSAIL (Ph.D. student of Franklin when he was at UC Berkeley); and Daniel Abadi of the University of Maryland, College Park (former Ph.D. student of Madden). The first three plan to work their way down the chain to get the others to legally change their names to "Mike."

PART III

MIKE STONEBRAKER SPEAKS OUT: AN INTERVIEW WITH MARIANNE WINSLETT

2 Mike Stonebraker Speaks Out: An Interview

Marianne Winslett

Welcome to ACM SIGMOD Record's series of interviews with distinguished memberstex of the database community.[1] I'm Marianne Winslett, and today we are at Lake Winnipesaukee in New Hampshire, USA. I have here with me Michael Stonebraker, who is a serial entrepreneur and a professor at MIT, and before that for many years at Berkeley. Mike won the 2014 Turing Award for showing that the relational model for data was not just a pipe dream, but feasible and useful in the real world. Mike's Ph.D. is from the University of Michigan. So, Mike, welcome!

Michael Stonebraker: Thank you, Marianne.

Marianne Winslett: Thirty-five years ago, you told a friend that winning the Turing Award would be your proudest moment. While ambition was hardly the only factor in your success, I think that being so ambitious would have made a huge difference from day one.

Stonebraker: I think that if you decide to become an assistant professor, you've got to be fanatically ambitious, because it's too hard otherwise. If you're not just driven—people who aren't really driven fail. The professors I know are really driven to achieve. Those who aren't go do other things.

Winslett: Would that be specific to Berkeley and MIT? Or you think for computer science professors in general?

Stonebraker: I think that if you're at any big-name university—Illinois is no exception—that it's publish or perish, and the only way to get tenure is to really be driven. Otherwise it's just too hard.

[1] A video version of this conversation is also available at www.gmrtranscription.com.

Winslett: That's true, but publishing is not the same thing as having impact, and you've had a lot of impact. Are there other character traits that you see in students, like competitiveness, that have been a big factor in the impact they've had in their careers?

Stonebraker: My general feeling is that you have to be really driven. Furthermore, I think if you're not at one of two or three dozen big-name universities, it's hard to really have impact because the graduate students you have aren't that good. I think you've got to have good graduate students or it's very difficult to succeed.

Anyone who works for me has to learn how to code, even though I'm horrible at coding, because I make everybody actually do stuff rather than just write theory. In our field, it's really hard to have an impact just doing paper-and-pencil stuff.

Winslett: You couched your advice in terms of advice for professors. Would it be different for people in industry or at a research lab in industry?

Stonebraker: There are some exceptions, but I think by and large, the people who've made the biggest impact have been at universities.

Industrial research labs have two main problems. The first one is that the best way to build prototypes is with a chief and some Indians, and that generally doesn't exist at industrial research labs. I think it's a marvel that System R managed to put together nearly a dozen chiefs and get something to work.

Problem two is that if you're at a big-name university, if you don't bring in money you can't get anything done. You have to be entrepreneurial, you've got to be a salesman, you've got to raise money, and those are characteristics you don't have to have at an industrial research lab. The really aggressive people self-select themselves into universities.

Winslett: As one of my other jobs (co-editor-in-chief of *ACM Transactions* on the Web), I go through the major conferences that are related in any way to the Web and look at the best paper prize winners. It is amazing how many of those now come from industry.

Stonebraker: Essentially all the contributions to Web research involve big data, and the Internet companies have all the data and they don't share it with academia. I think it's very difficult to make significant contributions in Web research without being at a Web company. In hardware, in the Web—there are definitely areas where it's hard to make a contribution in academia.

Winslett: But in the infrastructure side of databases, you think it's still possible to have strong impact as an academic researcher. You don't have to go where the data is.

Stonebraker: Right. The thing I find really interesting is that if you look at the contributions that have come from the database companies, they're few and far between. Good ideas still come primarily from universities. However, there are storm clouds on the horizon. For whatever reason, companies are happy to let the vendor look at their data, but they don't want to share it with anybody else. My favorite example is that I was looking for data on database crashes—why do database systems crash?

I had a very large whale who was willing to share their logs of database crashes. That went down the tubes because number one, the company didn't want people to know how low their uptime was, and number two, their vendor didn't want people to know how often they crashed. I think the trouble with operational data is that it tends to put egg on somebody's face and that makes it difficult.

Winslett: I agree completely, so how do you still manage to have an impact coming from the academic side?

Stonebraker: I think that the easiest way to have an impact is to do something interesting and then get venture capital backing to turn it into something real. Ingres actually really worked. Postgres really worked, but every system I've built since then just barely limped along because it got too difficult. You get a prototype that just barely works and then you get VC money to make it real and then you go and compete in the marketplace against the elephants.

There's a fabulous book by Clayton Christensen called The Innovator's Dilemma. Basically, it says that if you're selling the old technology, it's very difficult to morph selling the new technology without losing your customer base. This makes the large database companies not very interested in new ideas, because new ideas would cannibalize their existing base. If you want to make a difference, you either try to interest a database company in what you're doing, or you do a startup. If you don't do one or the other, then I think your impact is limited. Everyone I know is interested in starting a company to make a difference. To get your ideas really into the world, that's the only way to do it.

Winslett: With the Turing Award already in hand, what else would you like to accomplish in your career?

Stonebraker: At this point I'm 73 years old. I don't know of any 80-year-old researchers who are still viable, and so my objective is very simple, to stay viable as long as I can, and to hopefully realize when I've fallen off the wagon and gracefully retire to the sidelines. I'm just interested in staying competitive.

Winslett: Regarding staying competitive: One of your colleagues says that "Mike is notorious for only liking his own ideas, which is certainly justifiable because he is often right." Tell me about a time you changed your mind on a major technical matter.

Stonebraker: I think my biggest failure was that I was a huge fan of distributed databases in the 1970s and the '80s and even in the '90s, and there's no commercial market for that stuff. Instead there's a gigantic market for parallel database systems, which are distributed database systems with a different architecture, and I didn't realize that that was where the big market was. I just missed that completely. I could've written Gamma, but I didn't. That was a major theme that I missed completely, and it took me a very long time to realize that there really is no market for distributed database systems for all kinds of good reasons. At the end of the day, the real world is the ultimate jury and I was slow to realize that there was no market.

Winslett: You spent decades pooh-poohing specialized data management tools such as object databases and vertical stores. Then in the 2000s, you started arguing that one size does not fit all. Why did you change your mind?

Stonebraker: In the 1980s, there was only one market for databases. It was business data processing, and for that market the relational model seems to work very well. After that, what happened was that all of a sudden there were scientific databases. All of a sudden there were Web logs. These days, everyone on the planet needs a database system. I think the market has broadened incredibly since the '80s and in the non-business-data-processing piece of the market, sometimes relational databases are a good idea and sometimes they're not. That realization was market-driven. I changed my mind based on the market being very different.

Winslett: Was there a particular moment when that happened? Something you saw that made you think, "We have to diversify?"

Stonebraker: Let me generalize that question a bit: Where do good ideas come from? I have no clue. They just seem to happen. I think the way to make them happen best is to hang around smart people, talk to lots and lots of people, listen to what they say. Then slowly something sinks in and then something happens.

For example, a couple of years before we wrote H-Store, I had talked to a VC who said, "Why don't you propose a main memory database system for OLTP?" And I said, "Because I don't have a good idea for how to do it." But that generated the seed, the realization that somebody was interested in that topic. Eventually the ideas came, and we built H-Store.

I don't know when this happens or how it happens. I live in terror of not having any more good ideas.

Winslett: How many different forms of data platform would be too many?

Stonebraker: There will certainly be main memory OLTP systems that are mostly going to be row stores, and there will certainly be column stores for the data warehouse market.

My suspicion is that the vast majority of scientific databases are array-oriented and they're doing complex codes on them. My suspicion is that relational database systems are not going to work out very well there and that it would be something else, maybe an array store, who knows? In complex analytics, singular value decomposition, linear regression, all that stuff, which is the operations those kinds of folks want to do on largely array-oriented data, the jury is out as to how that's going to be supported.

I'm not a huge fan of graph-based database systems, because it's not clear to me that a graph-based system is any faster than simulating a graph either on a tabular system or an array system. I think we'll see whether graph-based systems make it. XML is yesterday's big idea and I don't see that going anywhere, so I don't see doing an XML store as a worthwhile thing to try.

Winslett: What about specialized stores for log data?

Stonebraker: It seems to me that most of the log processing works fine with data warehouses. But there's no question that stream processing associated with the front end of the log will either be specialized stream processing engines like Kafka or main memory database systems like VoltDB. The jury is out as to whether there's going to be a special category called streaming databases that's different from OLTP.

We might need half a dozen specialized data stores, but I don't think we need 20. I don't even think we need ten.

Winslett: You say that there's no query Esperanto. If so, why have you been working on polystores and BigDAWG?

Stonebraker: Polystores to me mean support for multiple query languages, and BigDAWG has multiple query languages, because I don't think there is a query language Esperanto. Among the various problems with distributed databases: First, there isn't a query language Esperanto. Second, the schemas are never the same on independently constructed data. Third, the data is always dirty, and everybody assumes that it's clean. You've got to have much more flexible polystores that support multiple query languages and integrate data cleaning tools and can deal with the fact that schemas are never the same.

Winslett: That's a good lead-in to talking about your project Data Civilizer, which aims to automate the grunt work of finding, preparing, integrating, and cleaning data. How well can we solve this problem?

Stonebraker: The Data Civilizer project comes from an observation made by lots of people who talk to a data scientist who is out in the wild doing data science. No one claims to spend less than 80% of their time on the data munging that has to come in advance of any analytics. A data scientist spends at most one day a week doing the job for which she was hired, and the other four days doing grunt work. Mark Schreiber, who's the chief data scientist for Merck, claims it's 98% time in grunt work, not 80%! So, the overwhelming majority of your time, if you're a data scientist, is spent doing mung work. In my opinion, if you worry about data analytics, you're worrying about the spare-change piece of the problem.

If you want to make a difference, you have to worry about automating the mung work. That's the purpose of Data Civilizer. Mark Schreiber's using the system that we have, and he likes what he sees, so at least we can make some difference. How much we can cut down this 80 to 90% remains to be seen. As a research community, we worked on data integration 20 years ago and then it got kind of a bad name, but the problems in the wild are still there and if anything, they're much, much worse. I'd encourage anybody who wants to make a difference to go work in that area.

Winslett: You said that your Merck guy likes what he sees. Can you quantify that?

Stonebraker: At the top level, Merck has about 4,000 Oracle databases. They don't actually know how many they've got. That's in addition to their data lake, on top of uncountable files, on top of everything imaginable. For a starter, if you were to say, "I'm interested in finding a dataset that can be used to figure out whether Ritalin causes weight gain in mice," your first problem is to identify a dataset or datasets that actually might have the data you're interested in. So, there's a discovery problem.

Merck is running the discovery component of Data Civilizer, which lets you ask questions like, "I'm interested in Ritalin, tell me about some datasets that contain Ritalin." They're using that and they like what they see.

Beyond discovery, data cleaning is a huge problem. We're working on that using Merck and others as a test case. This ties back to what we said earlier: To make a difference in data integration and data cleaning, you've got to find a real-world problem, find an enterprise that actually wants your problem solved.

For instance, in doing data integration, the overwhelming majority of the products I've seen have Table 1 over here, Table 2 over here, you draw some lines to hook stuff up. That doesn't help anybody that I know of. For example, in the commercialization of the original Data Tamer system, GlaxoSmithKline is a customer. They've got 100,000 tables and they want to do data integration at that scale, and anything that manually draws lines is a non-starter.

As a research community, it absolutely behooves us to do the shoe leather, to go out and talk to people in the wild and figure out exactly what their data problems are and then solve them, as opposed to solving problems that we make up.

Winslett: Definitely, but data integration has been on that top ten list of problems of those Laguna Beach-type reports . . .

Stonebraker: Forever.

Winslett: Always.

Stonebraker: Yes.

Winslett: How have things changed that we can finally get some traction?

Stonebraker: Let me give you a quick example. I assume you know what a procurement system is?

Winslett: Sure.

Stonebraker: How many procurement systems do you think General Electric has?

Winslett: Maybe 500?

Stonebraker: They have 75, which is bad enough. Let's suppose you're one of these 75 procurement officers and your contract with Staples comes up for renewal. If you can figure out the terms and conditions negotiated by your other 74 counterparts and then just demand most-favored-nation status, you'll save General Electric something like $500 million a year.

Winslett: Sure, but that was already true 25 years ago, right?

Stonebraker: Yeah, but enterprises are in more pain now than they were back then, and modern machine learning can help.

The desire of corporations to integrate their silos is going up and up and up, either because they want to save money, or they want to do customer integration. There's a bunch of things that companies want to do that all amount to data integration. If you realize that there's $500 million on the table, then it leads you to not be very cautious, to try wild and crazy ideas. The thing about Tamr that just blows me away is that GE was willing to run what I would call a pre-alpha product just because they were in so much pain. Generally, no one will even run version 1.0 of a database system. If you're in enough pain, then you'll try new ideas.

In terms of data integration, it's very, very simple. You apply machine learning and statistics to do stuff automatically because anything done manually, like drawing lines, is just not going to work. It's not going to scale, which is where the problem is. Data integration people hadn't applied machine learning, but it works like a charm . . . well, it works well enough that the return on investment is good!

Winslett: You've said many harsh words about AI in the past. Was there a moment when AI finally turned a corner and became useful?

Stonebraker: I think *machine learning* is very useful, and it's going to have a gigantic impact. Whether it's conventional or deep learning, the stuff works. I'm much less interested in other kinds of AI. Google pioneered deep learning in a way that actually works for image analysis, and I think it works for natural language processing too. There's a bunch of areas where it really does work. Conventional machine learning, based on naive Bayes models, decision trees, whatever, also works well enough in a large number of fields to be worth doing.

The standard startup idea, at least from three or four years ago, was to pick some area, say choosing pricing for hotel rooms. One startup said, "Okay, that's what I want to do. I'll try it in the Las Vegas market first," and they got all the data they could find on anything that might relate to hotel rooms. They ran an ML model and they found out that you should set hotel prices based on arrivals at McCarran Airport, which sounds like a perfectly reasonable thing to do. If you apply this kind of technology to whatever your prediction problem is, chances are some version of ML is going to work, unless of course there's no pattern at all.

But in lots of cases there is a pattern, it's just fairly complicated and not obvious to you and I. ML will probably find it. I think applications of ML are going to have a big impact.

Winslett: From your perspective as a database researcher, what are the smartest and dumbest things you've seen a hardware vendor do in the last few years?

Stonebraker: A million years ago, Informix was losing the relational database wars to Oracle. A succession of CEOs thought the solution to the problem was to buy some startup, so they bought Illustra, which was the company I worked for. After that they bought a company called Red Brick Systems, and after that they bought a company whose name I can't remember who built Java database systems. They thought that the salvation was going to be to buy somebody.

I think that's almost always a dumb idea, because in all these cases, the company really didn't have a plan for how to integrate what they were buying, how to train their sales force on how to sell it, how to sell it in conjunction with stuff they already had. When the rubber meets the road, I read somewhere that three-quarters of the acquisitions that companies make fail. So, my recommendation is to be a lot more careful about what you decide to acquire, because lots of times it doesn't work out very well. Getting value from an acquisition means integrating the sales force, integrating the product, etc., etc., and lots and lots of companies screw that up.

When HP bought Vertica, the biggest problem was that HP really couldn't integrate the Vertica sales force with the HP sales force, because the HP sales force knew how to sell iron and iron guys couldn't sell Vertica. It was a totally different skill set.

Winslett: Are there advances in hardware in recent years that you think have been really good for the database world?

Stonebraker: I think GPUs for sure will be interesting for a small subset of database problems. If you want to do a sequential scan, GPUs do great. If you want to do singular value decomposition, that's all floating-point calculations, and GPUs are blindingly fast at floating point calculations. The big caveat, though, is that your dataset has to fit into GPU memory, because otherwise you're going to be network-bound on loading it. That will be a niche market.

I think non-volatile RAM is definitely coming. I'm not a big fan of how much impact it's going to have, because it's not fast enough to replace main memory and it's not cheap enough to replace solid-state storage or disk. It will be an extra level in the memory hierarchy that folks may or may not choose to make use of. I think it's not going to be a huge game changer.

I think RDMA and InfiniBand will be a huge, huge, huge deal. Let me put it generally: Networking is getting faster at a greater rate than CPUs and memory are getting beefier. We all implemented distributed systems such as Vertica with the assumption that we were network-bound, and that's not true anymore. That's going to cause a fair amount of rethinking of most distributed systems. Partitioning databases either makes no sense anymore or it makes only limited sense. Similarly, if you're running InfiniBand and RDMA, then Tim Kraska demonstrated that new kinds of concurrency control systems are perhaps superior to what we're currently doing, and that is going to impact main memory database systems.

The networking advances make a big difference, but I think on top of this, James Hamilton, who is one super smart guy, currently estimates that Amazon can stand up a server node at 25% of your own cost to do it. Sooner or later that's going to cause absolutely everybody to use cloud-based systems, whether you're letting Amazon run your dedicated hardware or you're using shared hardware or whatever. We're all going to move to the cloud. That's going to be the end of raised floor computer rooms at all universities and most enterprises. I think that's going to have an unbelievable impact and sort of brings us back to the days of time-sharing. What goes around comes around.

I think that that in turn is going to make it difficult to do computer architecture research, because if there are half a dozen gigantic cloud vendors running 10 million nodes, and the rest of us have a few nodes here and there, then you pretty much have to work for one of the giants to get the data to make a difference.

Winslett: What do you wish database theory people would work on now?

Stonebraker: Here's something that I would love somebody to work on. We professors write all the textbooks, and on the topic of database design, all the textbooks say to build an entity relationship model, and when you're happy with it, push a button and it gets converted to third normal form. Then code against that third normal form set of tables, and that's the universal wisdom. It turns out that in the real world, nobody uses that stuff. Nobody. Or if they use it, they use it for the greenfield initial design and then they stop using it. As near as I can tell, the reason is that the initial schema design #1 is the first in an evolution of schemas as business conditions change.

When you move from schema #1 to schema #2, the goal is never to keep the database as clean as possible. Our theory says, "Redo your ER model, get a new set of tables, push the button." That will keep the schema endlessly in third normal form, a good state. No one uses that because their goal is to minimize application maintenance, and so they let the database schema get as dirty as required in order to keep down the amount of application maintenance.

It would be nice if the theory guys could come up with some theory of database application coevolution. That's clearly what the real world does. My request to the theory guys is that they find a real-world problem that somebody's interested in that your toolkit can be used to address. Please don't make up artificial problems and then solve them.

Winslett: That's good advice for *any* researcher.

What lessons can the database community learn from MapReduce's success in getting a lot of new people excited about big data?

Stonebraker: I view MapReduce as a complete and unmitigated disaster. Let me be precise. I'm talking about MapReduce, the Google thing where there's a map and a reduce operation that was rewritten by Yahoo and called Hadoop. That's a particular user interface with a map operation and a reduce operation. That's completely worthless. The trouble with it is that no one's problem is simple enough that those two operations will work.

If you're Cloudera, you've now got a big problem because you've been peddling MapReduce and there's no market for it. Absolutely no market. As a result, Cloudera very carefully applied some marketing and said, "Hadoop doesn't mean Hadoop anymore. It means a three-level stack with HDFS at the bottom, MapReduce in the middle, and SQL at the top. That's what Hadoop means now: it's a stack." However, you still have a problem because there is no market for the MapReduce piece of the Cloudera stack. So, the next Cloudera action was to deprecate the MapReduce piece by implementing a relational SQL engine, Impala, which drops out MapReduce completely and does SQL on top of HDFS. In effect, Cloudera realized that 75% of the "Hadoop market" is SQL and that MapReduce is irrelevant.

In an SQL implementation, there is no place for a MapReduce interface. None of the data warehouse products use anything like that, and Cloudera Impala looks exactly like the other data warehouse guys' products. In my opinion, the "Hadoop market" is actually a SQL data warehouse market. May the cloud guys and the Hadoop guys and the traditional database vendors duke it out for who's got the best implementation.

Winslett: But didn't MapReduce get a lot of potential users excited about what they might be able to do with their data?

Stonebraker: Yes.

Winslett: It's a gateway drug, but you still don't approve of it.

Stonebraker: Lots of companies drank the MapReduce Kool-Aid, went out and spent a lot of money buying 40-node Hadoop clusters, and they're now trying to figure out what the heck to do with them. Some poor schmuck has to figure out what in the world to do with an HDFS file system running on a 40-node cluster, because *nobody wants MapReduce*.

Never to be denied a good marketing opportunity, the Hadoop vendors said, "Data lakes are important." A data lake is nothing but a junk drawer where you throw all of your data into a common place, and that ought to be a good thing to do. The trouble with data lakes is that if you think they solve your data integration problem, you're sadly mistaken. They address only a very small piece of it. I'm not opposed to data lakes at all, if you realize that they are just one piece of your toolkit to do data integration. If you think that all data integration needs are a MapReduce system, you're sadly mistaken.

If you think that the data lake is your data warehouse solution, the problem is that right now, the actual truth that Cloudera doesn't broadcast is that Impala doesn't really run on top of HDFS. The last thing on the planet you want in a data warehouse system is a storage engine like HDFS that does triple-redundancy but without transactions, and that puts your data all over everywhere so that you have no idea where it is. Impala actually drills through HDFS to read and write the underlying Linux files, which is exactly what all the warehouse products do.

In effect, the big data market is mostly a data warehouse market, and may the best vendor win. We talked about ML earlier, and I think that complex analytics are going to replace business intelligence. Hopefully that will turn this whole discussion into how to support ML at scale, and whether database systems have a big place in that solution. Exactly what that solution is going to be, I think, is a very interesting question.

Winslett: What do you think of the database technology coming out of Google, like Cloud Spanner?

Stonebraker: Let's start way back when. The first thing Google said was that MapReduce was a purpose-built system to support their Web crawl for their search engine, and that MapReduce was the best thing since sliced bread. About five years went by and all the rest of us said, "Google said MapReduce is terrific, so it must be good, because Google said so," and we all jumped on the MapReduce bandwagon. At about the same time Google was getting rid of MapReduce for the application for which it was purpose-built, namely, Web search. MapReduce is completely useless, and so Google has done a succession of stuff. There's BigTable, BigQuery, there's Dremel, there's Spanner . . . I think, personally, Spanner is a little misguided.

For a long time, Google was saying eventual consistency is the right thing to do. All their initial systems were eventual consistency. They figured out maybe in 2014 what the database folks had been saying forever, which is that eventual consistency actually creates garbage. Do you want me to explain why?

Winslett: No.

Stonebraker: Okay. Essentially everybody has gotten rid of eventual consistency because it gives no consistency guarantee at all. Eventual consistency was another piece of misdirection from Google that just was a bad idea. These were bad ideas because Google didn't have any database expertise in house. They put random people on projects to build stuff and they built whatever they wanted to without really learning the lessons that the database folks had learned over many, many years.

Google takes the point of view in Spanner of, "We're not going to do eventual consistency. We're going to do transactional consistency, and we're going to do it over wide area networks." If you control the end-to-end network, meaning you own the routers, you own the wires, you own everything in between here and there, then I think Spanner very, very cleverly figured out that you could knock down the latency to where a distributed commit worked over a wide area network.

The problem is that you and I don't control the end-to-end network. We have no way to knock the latency down to what Google can do. I think the minute you're not running on dedicated end-to-end iron, the Spanner ideas don't knock the latency down enough to where real-world people are willing to use it.

I will be thrilled when distributed transactions over the wide area networks that you and I can buy will be fast enough that we're willing to run them. I think that will be great. In a sense, Spanner leads the way on totally dedicated iron.

Winslett: What low-hanging fruit is there for machine learning in solving database problems?

Stonebraker: We've been building a database for supporting autonomous vehicles. Right now, AV folks want to keep track of whether there's a pedestrian in a particular image, whether there's a bicycle in a particular image. So far, they want to keep track of half a dozen things, but the number of things you might want to keep track of is at least 500. Stop signs, free parking spaces, emergency vehicles, unsafe lane changes, sharp left-hand turns . . . Assume there are 500 things you might want to index and then figure out which ones to actually index. For instance, you might want to index cornfields. In Urbana, that's probably a really good idea.

Winslett: Because the corn might get up and walk in front of the car?

Stonebraker: Well, because . . .

Winslett: I'd rather see deer indexed. And kangaroos, because they seem to have a death wish too.

Stonebraker: That'd be fine. I'm just saying there's a lot of things that might be worth indexing, and they're very situational. Cornfields are a good example because there are lots of them in Illinois, but there aren't hardly any inside Route 128 in Massachusetts. You've got to figure out what's actually worth indexing. You can probably apply machine learning to watch the queries that people do and start by indexing everything and then realize that some things are just so rarely relevant that it isn't worth continuing to index them. Applying ML to do that, rather than have it be a manual thing, probably makes a lot of sense.

Winslett: Do you see a broader role for ML in query optimization, or has it just become a kind of black art?

Stonebraker: It's certainly worth a try. It's perfectly reasonable to run a plan, record how well it did, choose a different plan next time, build up a plan database with running times and see if you can run ML on that to do better. I think it's an interesting thing to try.

Winslett: You've been stunningly successful in pulling together a bunch of universities to work on an integrated project. Was this your solution to the lone gunslinger mentality that you found when you moved to MIT?

Stonebraker: I had no choice. I was alone. There were no faculty, no students, no courses, no nothing. The only strategy was to reach out to the other universities in the Boston area. That strategy wouldn't work very well in Urbana because there aren't enough close-by universities, but in major metropolitan areas it's different. In Boston, there are six or eight universities, each with one or two database people. In aggregate, you can be a very, very strong distributed group.

Winslett: It's very hard to make a distributed collaboration work, but you made it work. It seems like physical proximity still played a role. How often did you get together?

Stonebraker: I drove to Brown once a week. In effect, we held real group meetings once a week and people drove to them. That only works if you have geographic proximity.

Winslett: Other key ingredients in making the distributed collaboration work?

Stonebraker: I think I had the great advantage that people were willing to listen to me and pretty much do what I suggested. The general problem is that there's a cacophony of ideas with no way to converge. There's got to be some way to converge, and either that takes a lead gunslinger, or it takes a program monitor from DARPA who's willing to knock heads. There's got to be some way to converge people, and I've managed to do that, pretty much.

Winslett: Any other ingredients worth mentioning, beyond those two?

Stonebraker: I also have a big advantage that I don't need any more publications, and so I'm happy to write papers that other people are the first author on. It helps to be willing to have no skin in the publication game. It generates a lot of goodwill to make sure that you're the last author and not the first author.

Winslett: One of the great joys of Postgres is that it allowed people to experiment with database components—join algorithms, index structures, optimization techniques—without having to build the rest of the system. What would be an equally open software system for today?

Stonebraker: A distributed version of Postgres.

Winslett: Who's going to build that?

Stonebraker: I know! There is no open source multi-node database system I'm aware of that's really good, and how one could be built remains to be seen. The big problem is that building it is a tremendous amount of work. It could come from Impala over time. It could come from one of the commercial vendors. The trouble with the commercial vendors is that the standard wisdom is to have a teaser piece of the system that's open-source and then the rest of the system is proprietary. It's exactly the distributed layer that tends to be proprietary. The vendors all want freemium pricing models and that makes a bunch of their system proprietary.

I don't think such a system can come from academia, it's just too hard. I think the days of building systems like Ingres and Postgres in universities are gone. The average Ph.D. student or postdoc has to publish a huge amount of stuff in order to get a job, and they're not willing to code a lot and then write just one paper, which was the way Ingres and Postgres got written. We had grad students who coded a lot and published a little, and that's no longer viable as a strategy.

Winslett: Could you do it with master's students?

Stonebraker: Maybe. Let's assume in round numbers that getting this distribution layer to be fast, reliable, and really work takes 10 man-years' worth of work. Maybe it's more, but the point is that it's a lot of work. A master's student is around for a maximum of two years, and you get maybe one year of productive work out of that person, assuming that they're good (the average may be six months). So that means you need 20 of these people. That means it occurs over a decade. It isn't like a cadre of 20 of them show up and say, "Here, manage me."

Ingres and Postgres were both written with one full-time person and three or four or five grad students, no postdocs. Back then, you could get something built in a few years with that scope of a team. Today it's just much harder to get stuff to work.

Winslett: The big data world has startup fever. We get it: often, it's easier to raise money from the investment community than from traditional sources of academic funding. How can we maintain the transparency required to move a field forward scientifically if so many ideas are hidden inside the IP of startups?

Stonebraker: I know! I find it distressing that the success rate for National Science Foundation proposals is down to something like 7%. It's getting incredibly hard to raise money in the traditional open-source, open-IP kinds of worlds. I think it's a huge problem.

The way I look at it is that the number of faculty in any given discipline of computer science is up by an order of magnitude over what it was 20 years ago. The number of mouths to feed is up by at least that same amount, and funding has not kept pace at all. I think we're starving.

The solution when you get disgusted with trying to raise money is that you leave the university and go to Google or another company. I wouldn't be surprised if the brain-drain out of universities gets to be significant.

Winslett: Why is it a bad idea to bootstrap a startup using your own money?

Stonebraker: Look at Vertica, Illustra, any of the companies that I've started. In round numbers, they required $20 or $30 million to get a reliable, stable, sellable product out the door. If you're writing an iPhone app, that's a different situation. But writing enterprise software takes a lot of money, and getting to something that you can release as version 1 is usually $5 to $10 million. Unless you're independently wealthy, that's not possible with self-funding.

The self-funded companies I've seen that have succeeded have had a sugar daddy, a corporation that said, "I'll pay for version 1 because I need it as an application, as long as you write something that I want." If you have that, you're basically having the customer fund the development.

If you're actually going to fund it out of your own checking account, the trouble is that you and five of your friends agree to write code at nights and weekends, because you've got to have day jobs to keep the bill collectors away. It just takes forever if you're writing code nights and weekends.

Another trouble with self-funding is that if your own money is invested, you make very, very cautious decisions relative to what VCs would make. In other words, they're much better businessmen than you are and will make much better decisions about money than you will. Mortgaging your house to fund a startup is something I would never do . . . that's a clear way to break up your marriage.

Winslett: You recommend that startup founders focus on great engineering, but when I consider the giant corporations in the data world, I get the impression that focusing on great marketing has been a more effective route to build market share.

Stonebraker: All true. You don't have to look any further than Oracle and Ingres. One had great engineering, one had great marketing, and look who won.

The trouble is that your first objective must be to build something that's reliable enough that your first five customers will buy it. If you don't have really good engineering, chances are you're not going to get to that milestone. If you just threw stuff together, chances are it's going to have serious reliability problems, which are going to be very expensive to fix. Chances are that's going to impact whether you can get revenue out of your first five customers.

So, worrying about superb engineering at the beginning is a really good idea. After that, making sure you have the world's best VP of marketing is a terrific strategy.

Winslett: Your research group has often implemented full DBMSs, and Andy Pavlo has written about the challenges this raises. Do you still feel that this is the best way to advance the state of the art?

Stonebraker: Calling them full DBMSs is a big stretch. As has been pointed out by Martin Kersten, C-Store ran about 10 queries. It was not a complete implementation at all. We marketed it as a complete system, but it really didn't have an optimizer. It hard-coded how to do the queries in our benchmarks. We cut a lot of corners on it.

H-Store was more complete than C-Store, but the academic version didn't have a replication system. What Andy Pavlo did was heave most of the H-Store executor and replace it by the open source of the VoltDB executor. H-Store got better mostly because he swiped open-source commercial code. Since Postgres, we really haven't produced what you would call a full-function well-functioning system. I think we've written pieces of such a system.

Winslett: Software is often free now, and not just phone apps. For example, when was the last time you bought a compiler? Will database software go the same route?

Stonebraker: I think that's an interesting question, because right now, the model used by most of the recent DBMS startups is freemium. It isn't really open source. It has a teaser piece that's open source, but anyone who's going to run it in production is going to get the non-free piece, and support only comes with the non-free piece. I think that the freemium model works fine, but it isn't really a complete open-source system.

I think it will be interesting to see whether the big cloud vendors will have complete free open-source systems, which probably will only run on their hardware so they can get the rental income from their iron. Right now, I'm hard pressed to think of a complete open-source system that you'd actually put into production use.

Winslett: You biked across the country and climbed all forty-eight 4,000-plus foot mountains in New Hampshire. How has athletics affected your professional life?

Stonebraker: I'm wired to aggressively attempt to achieve what's hard. That's true in physical stuff, that's true in professional stuff. That's just the way I'm wired. There's nothing purposeful there.

Winslett: You were one of the founders of CIDR. Has CIDR been a success or a failure?

Stonebraker: We started CIDR because SIGMOD was turning down practical papers. I think CIDR has proved to be a great venue for practical stuff over the years. The major conferences have attempted with some success to get more pragmatic papers, through their industrial tracks. But they still turn down my papers that are pragmatic, and that still pisses me off.

CIDR has caused the major conferences to change some, but in my opinion not enough. So I think CIDR continues to be a great outlet for practical papers that the major conferences won't touch. As long as we stick to our knitting, CIDR will be very viable long-term. Every time it's held, we have to close registration because it's over-subscribed.

Winslett: Are minimum publishable units still a big issue in our community?

Stonebraker: That's my favorite pet peeve. When I graduated with a Ph.D., I had zero publications. When I came up for tenure five years later, I had a handful, maybe six, and that was the norm. David DeWitt was the same way. That was typical back then. Now you have to have an order of magnitude more to get an assistant professor job or get tenure. That forces everybody to think in terms of Least Publishable Units (LPUs), which create a dizzying sea of junk that we all have to read. I think it's awful. I don't know how anybody keeps up with the deluge of publications. All of us tell our grad students to go read this or that paper and tell us what it says, because no one can physically read all that stuff.

My favorite strategy which might work is to get the top, say, 20 U.S. universities to say, "If you send us an application for an assistant professor position in computer science, list three publications. We're not going to look at any more. Pick three. We don't care if you have more. Pick three. When you come up for tenure, pick ten. If you publish more, we don't want to look at them." If you got the top universities to enforce that discipline, it might knock down the publication rates and start getting people to consolidate more LPUs into bigger and better papers.

Winslett: You might have the biggest family tree in the database field. Has that been an important factor in your success?

Stonebraker: I don't think so. I think success is determined by having good ideas and having good graduate students and postdocs to realize them. I think the fact that I know a lot of people, some of whom are my students and some of whom are other people's students, is not all that significant.

Winslett: When you pick a new problem to work on, how do you balance intellectual satisfaction, your industry buddies' depth of desire for a solution, and the gut feeling that you might be able to turn it into a startup?

Stonebraker: I don't think in those terms at all. I think mostly in terms of finding a problem that somebody has in the real world, and working on it. If you solve it in a way that's commercializable, that's downstream. You don't get any brownie points at my university for doing startups.

I think the biggest mistakes I've made have been when we had a prototype and a student really wanted to do a startup, but I was very reluctant. Usually I was right. I've created startups that have failed and usually I didn't think it would work at the beginning. I find it hard to resist when you have a Ph.D. student pleading with you, "Please do a startup."

Winslett: Were you more productive at Berkeley or at MIT?

Stonebraker: I think I've been much more productive at MIT.

Winslett: And why is that?

Stonebraker: I have no idea. If you look at the data, I did 3 startups in 25 years at Berkeley, and I did 6 startups in 16 years at MIT.

Winslett: Which set do you think had more impact, though?

Stonebraker: Number one was probably Postgres and number two was probably Vertica, so it's not obvious one way or the other.

Winslett: What successful research idea do you most wish had been your own?

Stonebraker: Parallel databases, like we talked about earlier.

Winslett: If you had the chance to do one thing over again, what would it be?

Stonebraker: I would have worked on parallel databases in the '70s.

Winslett: A recurring theme! Did you ever have an idea that the research community rejected but that you still believe in fervently and may pursue again?

Stonebraker: If a paper gets rejected, as near as I can tell, everybody, including me, rewrites it until it gets accepted. I don't know of any papers that actually went onto the cutting room floor. We all modify them until they get accepted.

Winslett: If you were a first-year data-oriented graduate student, what would you pick to work on?

Stonebraker: If you have a good idea on how to do cloud stuff . . . everybody's going to move to the cloud. That's going to be a huge amount of disruption. We're going to run database systems where we share a million nodes.

If you have a good idea on how to make data integration work, it's an unbelievably hard problem, and unbelievably important. If you have a good idea about database design, I would work on that. If you have a good idea on data cleaning, by all means, work on that. Find some problem in the wild that you can solve and solve it.

Winslett: Are there any hot topics in database research right now that you think are a waste of time?

Stonebraker: What we used to think of as database core competency is now a very minuscule portion of what appears in SIGMOD and VLDB. The field is basically fragmented beyond recognition. Very few papers on core database stuff appear these days. I think we're doing a lot of non-cutting-edge research in all these fragmented different fields, and I wonder what the long-term impact of that is going to be.

I'm kind of a little bit worried, because the database guys are all publishing ML papers under the guise of scalability. That isn't our community. There is an ML community that worries about ML. Database people don't publish in pure ML conferences, mostly I suspect because we're second-rate researchers there.

As near as I can tell, SIGMOD and VLDB are 300 or so researchers, and everything they and their grad students are doing is a huge spectrum of stuff. Deciding what's workable and what's not workable becomes very, very diffuse.

Winslett: Isn't that inevitable in an expanding field like ours?

Stonebraker: Yeah, but I think if you look at the operating system guys, they're starting to write database papers in their venues. When you get a lot of fragmentation, at some point it seems to me that we probably ought to reorganize computer science. CMU and Georgia Tech have schools of computer science that seem much better able to organize around this diffuse nature of things. MIT doesn't. The universities that don't have schools of computer science will be disadvantaged long-run, long-term. That's a political hot potato at MIT and elsewhere.

Winslett: Which of your technical projects have given you the most personal satisfaction?

Stonebraker: Vertica and Postgres. I think Vertica was the most satisfying because Postgres we rewrote and then we rewrote it again. We started off implementing Postgres in LISP, which was the biggest disaster on the planet. That's probably my biggest technical mistake ever. Postgres eventually got it more or less right. Vertica got it pretty much right the first time, which I thought was remarkable. Usually, you rewrite everything when you realize you screwed it up, and then you rewrite it again when you realize you still screwed it up. Vertica did pretty well the first time, which I thought was pretty remarkable.

Winslett: What was the most difficult part of your cross-country bike trip?

Stonebraker: The Turing Award lecture is set in North Dakota, and North Dakota was awful. Absolutely awful. Not so much because it's flat and boring and you spend your day looking up ahead 10 miles, seeing the grain elevator in the next town, riding toward it for three-quarters of an hour, passing through the town in a couple of minutes, and then 10 miles up ahead is the next grain elevator. That's really monotonous and boring, but what made it impossibly hard was that we were fighting impossible headwinds all the way across North Dakota. It is so demoralizing when you're struggling to make seven miles an hour and you realize that it's a 500-mile-wide state.

Winslett: What was the North Dakota of your career?

Stonebraker: I think it was by no means a slam dunk that I was going to get tenure at Berkeley. I think the department probably went out on a limb to make that happen. At the time, databases were this little backwater, and somebody had enough vision to promote me.

The stress associated with getting tenure I think is awful for everybody, universally, and the year that you're up for tenure is horrible, no matter who you are. I personally think we shouldn't subject assistant professors to that kind of stress. We should invent a better tenure system or gradual tenure system or something else. The stress level we subject assistant professors to is awful.

Winslett: If you were retired now, what would you be doing?

Stonebraker: That's equivalent to the question, "What do I do when I'm no longer competitive as a researcher?"

Winslett: The definition of "retired" is when you're no longer competitive as a researcher?

Stonebraker: Yeah. I'm going to work. I'm going to do what I'm doing until I'm not competitive doing it. I wake up in the morning and I like what I do. The only aspect of my job that I hate is editing student papers. Students by and large can't write worth a darn. Like everybody else, I'm stuck fixing their papers. I hate that.

The real question you youngsters don't really have to face, or you probably don't think about, is what the state of my physical health is when I retire. If I'm impaired, then life's in a whole different ballpark. If I'm full function, I will hike a lot more, I will bike a lot more. I always threaten to do woodworking. Dave DeWitt laughs at me because I have a woodworking shop that is essentially unused. I would spend more time with my wife. I would spend more time up here in New Hampshire.

I think the real answer is that I would probably become a venture capitalist of sorts if I'm not viable with my own ideas. I'm very good at helping people start companies, so I would probably do a whole bunch of that.

Winslett: I'm told that I should ask you to sing "The Yellow Rose of Texas."

Stonebraker: Only after many, many beers or glasses of wine.

I don't know where that question came from . . . the only place that question could have come from was from when the Illustra guys invited me to one of their sales reward meetings, and everybody had to get up and do karaoke. I don't think I did "The Yellow Rose of Texas," but maybe I did. That's the only time I can remember . . . I wonder, where did that question come from?

Winslett: I would never give away my sources, even if I remembered who contributed that one, which I don't.

Stonebraker: Anyway, I'm the world's worst singer. I sing in a great monotone.

Winslett: I hear you play the banjo for a bluegrass band. What got you interested in the banjo and bluegrass music?

Stonebraker: When my first wife and I separated in 1975, I went out and bought a banjo within a couple months, then asked the guy who sold the banjo, "What kind of music do you play with this?" I have no idea why I chose the banjo. There's nowhere in my history that anybody ever had a banjo, so I don't have any idea why I decided to take it up.

Having kids got in the way of having time to play, but after the kids were adults, then I started playing again. I'm now in a band of sorts, called Shared Nothing. Not Shared Nothings (plural), it's Shared Nothing (singular), exactly like the technical distributed database term *shared nothing*. We jam every couple of weeks but calling us a band is a bit of a stretch. Our goal is to start playing at assisted living centers because those places are always looking for something for their residents to do.

Winslett: Have you reached that level of expertise yet?

Stonebraker: I know a friend whose father is in an assisted living facility, and he pointed me to their entertainment director. I said, "Can we come play for your folks?" She said to send her a tape. So, we made a tape, but that was the last we heard from her. We're not yet at the level of playing at assisted living centers.

Winslett: It's something to aspire to.

We're good enough to play in front of my Ph.D. students, who are a captive audience.

Winslett: I hear that you wear a lot of red shirts.

Yep.

Winslett: Why, and how many do you own?

Stonebraker: Approximately 15. I like red. I have a red boat. For a long time, I drove a red car. Red is my favorite color for whatever reason and I wear red shirts, although not today.

Winslett: To avoid the draft for the Vietnam War, you had to go straight to grad school after college. You have said that this forced you into a career path prematurely, without time to explore other options. In hindsight, what path do you think you would have taken if there hadn't been a draft?

Stonebraker: When I graduated from college in 1965, my choices were go to graduate school, go to Vietnam, go to jail, or go to Canada. Those were the exact choices. If I went to graduate school, it was right at the time of the post-Sputnik science craze, so I would have a full-ride fellowship to sit out the war in graduate school. Why wouldn't you do that? You had to sit in graduate school until you were 26 and the government didn't want you for the war anymore. That forced me to get a Ph.D. that I don't think I ever would have gotten without that kind of pressure. The threat of the draft was a powerful motivator.

You're probably not old enough to remember the TV show called Route 66. You ever heard of that?

Winslett: Yes, but I haven't watched it.

Stonebraker: It's these two guys who drive around the country in a Corvette and have great experiences. When I graduated from college, I had no idea what I wanted to do, and so I would have done the Route 66 thing if I could. I have no idea where that would have led, but my life would have certainly been very different.

Winslett: Do young computer scientists even need a computer science degree today?

Stonebraker: In my opinion, yes. We talked about Google earlier. Google had a bunch of database projects that they assigned to people who were skilled at other stuff, and they screwed them up. They implemented short-term systems that weren't viable long-term. There's a lot of theory and pragmatics that we've accumulated over the years that is useful to know. I don't know a better way to do it than by studying computer science.

By and large, if you look at people in surrounding disciplines who actually end up doing computer science, like students in most of the physical sciences, and you ask what contribution they have really made to computer science, the answer is it's not very dramatic. For whatever reason, computer science advances tend to come from people who are trained as computer scientists.

Winslett: Do you have a philosophy for advising graduate students?

Stonebraker: The simple answer is that as a faculty member, your charge is to make your students successful. When you take someone on, it's basically an agreement to make them successful, and if they drop out then you're a failure. I try very, very hard to make my students successful. Usually that means feeding them good ideas when they don't have any of their own, and pushing them hard, saying, "The VLDB deadline is in three weeks, and you can, in fact, get a paper in. Progress will have to be exponential in the distance to the deadline, but you can do it."

Be a cheerleader and push your students to get stuff done at a rate much faster than they think they can. When they go off the rails, as they always do, pull them back onto the rails. This does take a lot of time. I meet with all my students once a week or more. My job is to be the cheerleader, idea generator, and encourager.

Winslett: If you magically had enough extra time to do one additional thing at work that you're not doing now, what would it be?

Stonebraker: If I have a good idea I start working on it, even if I don't have any extra time. I fit it in. So, I don't have a good idea just sitting, waiting for some time to work on it. I don't know what I'd do if I had extra time. Getting up in the morning and having nothing that I have to do drives me crazy. I stay very busy, and I don't know what I would do with free time.

Winslett: If you could change one thing about yourself as a computer science researcher, what would it be?

Stonebraker: I'd learn how to code.

Winslett: That's what you said the last time we did an interview, but obviously, since you've achieved high success without being able to code, it must not be necessary.

Stonebraker: I know, but it's embarrassing. It's something that takes a lot of time to learn, time that I don't have. If I could magically create a lot of time, I'd learn how to code.

Winslett: Maybe while you were going across North Dakota in the headwinds, you could have been practicing on a little keyboard on the handlebars. That would have made it less painful.

Stonebraker: I don't think you've ever been to North Dakota.

Winslett: I have, I have. And from your description, it sounds a lot like Illinois. The secret is to ride with the wind behind you. Maybe you could have gone to the far end of the state and ridden across it in the reverse direction. You still would have crossed North Dakota, but the wind would have been helping you.

Stonebraker: There you go.

Winslett: Thank you very much for talking with me today.

PART IV

THE BIG PICTURE

3 Leadership and Advocacy

Philip A. Bernstein

More than anyone else, Mike Stonebraker has set the research agenda for database system implementation architecture for the past 40 years: relational databases, distributed databases, object-relational databases, massively distributed federated databases, and specialized databases. In each of these cases, his was the groundbreaking research effort, arguing for a different system-level architecture type of database system. He proposed the architecture system type, justified its importance, evangelized the research agenda to create a new topic within the database community, and built a successful prototype that he later moved into the commercial world as a product, Ingres and Postgres being the most well known and influential examples. It is for these efforts that he richly deserves the ACM Turing Award.

I have been following Mike's work since we first met at the 1975 SIGMOD Conference. Despite having crossed paths a few times per year ever since then, we've never collaborated on a research project. So, unlike most authors of chapters of this book, I don't have personal experiences of project collaborations to recount. Instead, I will focus on his ideas, roughly in chronological order. I will first describe the systems, then mechanisms, and finally his advocacy for the database field.

Systems

The story starts with the Ingres project. At the project's outset in 1973, the first generation of database system products were already well established. They used record-at-a-time programming interfaces, many of which followed the proposed CODASYL database standard, for which Charles Bachman received the 1973 Turing Award. Vendors of these products regarded the relational model as infeasible, or at least too difficult to implement—especially targeting a 16-bit minicomputer (a PDP-11/40), as the Ingres project did. Moreover, database management was a topic for business schools, not computer science departments. It was loosely associated with COBOL programming for business data processing, which received no respect in the academic computer science research community. In those days, IBM had a dominant share of the computer market and was heavily promoting its hierarchical database system, IMS. The only ray of hope that the relational model was worth implementing was that IBM Research had spun up the IBM System R project (see Chapter 35). In the world of 1973, Mike Stonebraker and Gene Wong were very brave in making Ingres the focus of their research.

As they say, the rest is history. The Ingres project (see Chapter 15) was a big research success, generating early papers on many of the main components of a database system: access methods, a view and integrity mechanism, a query language, and query optimization. Many of the students who worked on the system became leading database researchers and developers. Moreover, Ingres was unique among academic research projects in that its prototype was widely distributed and was used by applications (see Chapter 12). Ultimately, Ingres itself became a successful commercial product.

In 1984, Mike and his UC Berkeley colleague Larry Rowe started a follow-on project, Postgres (see Chapter 16), to correct many of the functional limitations in Ingres. By that time, it was apparent to the database research community that the first generation of relational databases was not ideal for engineering applications, such as computer-aided design and geographical information systems. To extend the reach of relational systems to these applications, Mike and Larry proposed several new features for Postgres, the most important of which was user-defined datatypes.

The notion of abstract data type was a well-understood concept at that time, having been pioneered by Barbara Liskov and Stephen Zilles in the mid-1970s [Liskov and Zilles 1974]. But it had not yet found its way into relational systems. Mike had his students prototype an abstract data type plugin for Ingres, which he then reapplied in Postgres [Stonebraker 1986b, Stonebraker 1986c]. This was among the earliest approaches to building an extensible database system, and it has turned out to be the dominant one, now commonly called a user-defined datatype. It led to another startup company, which developed the Illustra system based on Postgres. Illustra's main feature was extensibility using abstract data types, which they called "data blades." Illustra was acquired in 1996 by Informix, which was later acquired by IBM.

In the mid-1990s, Mike led the development of his next big system, Mariposa, which was a geo-distributed database system for heterogeneous data. It was layered on a network of independently managed database systems whose resources Mariposa could not control. Therefore, Mariposa introduced an economic model where each database bids to execute part of a query plan. The global optimizer selects bids that optimize the query and fit within the user's budget. It is a novel and conceptually appealing approach. However, unlike Ingres and Postgres, Mariposa was not a major commercial success and did not create a new trend in database system design. It did lead to a startup company, Cohera Corporation, which ultimately used Mariposa's heterogeneous query technology for catalog integration, which was a pressing problem for business-to-business e-commerce.

Starting in 2002, Mike embarked on a new wave of work on database systems specialized for particular usage patterns: stream databases for sensors and other real-time data sources, column stores for data warehousing, main memory databases for transaction processing, and array databases for scientific processing. With this line of work, he again set the agenda for both the research field and products. He argued that relational database products had become so large and difficult to modify that it was hopeless to expect them to respond to the challenges presented by each of these workloads. His tag line was that "one size does not fit all." In each case, he showed that a new system that is customized for the new workload would outperform existing systems by orders of magnitude.

For each of these workloads, he followed the same playbook. First, there were research papers showing the potential of a system optimized for the workload. Next, he led a project to develop a prototype. Finally, he co-founded a startup company to commercialize the prototype. He founded startups in all of these areas. The Aurora research prototype led to the founding of StreamBase Systems for stream processing. The C-Store research prototype led to the founding of Vertica Systems (acquired by HP and now owned by Micro Focus) for column stores. The H-Store research prototype led to the founding of VoltDB (http://www.voltdb.com/) for main memory transaction processing. And the SciDB project, a database system for array processing, started as an open-source project with contributions by researchers at many different institutions, and led to the founding of Paradigm4 (http://www.paradigm4.com/).

Some vendors have picked up the gauntlet and shown how to modify existing products to handle these workloads. For example, Microsoft now offers a column store component of SQL Server ("Apollo"), and a main memory transactional database component of SQL Server ("Hekaton"). Perhaps the changes of workload and improvements in hardware made it inevitable that vendors would have developed these new features for existing products. But there is no doubt that Mike greatly accelerated the development of this functionality by pushing the field to move these challenges to the top of its priority list.

One of the most difficult problems in database management is integration of heterogeneous data. In 2006, he started the Morpheus project in collaboration with Joachim Hammer, which developed a repertoire of data transformations for use in data integration scenarios. This led to the Goby startup, which used such data transformations to integrate data sources in support of local search; the company was acquired by Telenauv in 2011. Also in 2011, he started the Data Tamer project, which proposed an end-to-end solution to data curation. This project became the basis for another startup, Tamr, which is solving data integration problems for many large enterprises (http://tamr.com), especially unifying large numbers of data sources into a consistent dataset for data analytics.

Mechanisms

In addition to making the above architectural contributions as a visionary, Mike also invented important and innovative approaches to building the system components that enable those architectures and that are now used in all major database products. There are many examples from the Ingres system, such as the following.

- Query modification (1975)—The implementation of views and integrity constraints can be reduced to ordinary query processing by expressing them as queries and substituting them into queries as part of query execution [Stonebraker 1975]. This technique, now known as view unfolding, is widely used in today's products.
- Use of B-trees in relational databases (1976–1978)—A highly influential technical report listed problems with using B-trees in relational database systems. This problem-list defined the research agenda for B-tree research for the following several years, which contributed to its use as today's standard access method for relational database systems. Given the multi-year delay of journal publication in those days, by the time the paper appeared in CACM (Communications of the ACM) [Held and Stonebraker 1978], many of the problems listed there had already been solved.
- Primary-copy replication control (1978)—To implement replicated data, one copy of the data is designated as the primary to which all updates are applied [Stonebraker 1978]. These updates are propagated to read-only replicas. This is now a standard replication mechanism in all major relational database system products.
- Performance evaluation of locking methods (1977–1984)—Through a sequence of Ph.D. theses at UC Berkeley, he showed the importance of fine-

tuning a locking system to gain satisfactory transaction performance [Ries and Stonebraker 1977a, Ries and Stonebraker 1977b, Ries and Stonebraker 1979, Carey and Stonebraker 1984, Bhide and Stonebraker 1987, Bhide and Stonebraker 1988].

- Implementing rules in a relational database (1982–1988, 1996)—He showed how to implement rules in a database system and advocated this approach over the then-popular rule-based AI systems. He initially built it into Ingres [Stonebraker et al. 1982a, Stonebraker et al. 1983c, Stonebraker et al. 1986] and incorporated a more powerful design into Postgres [Stonebraker et al. 1987c, Stonebraker et al. 1988a, Stonebraker et al. 1989, Stonebraker 1992a, Chandra et al. 1994, Potamianos and Stonebraker 1996]. He later extended this to larger-scale trigger systems, which are popular in today's database products.
- Stored procedures (1987)—In the Postgres system, he demonstrated the importance of incorporating application-oriented procedures inside the database system engine to avoid context-switching overhead. This became the key feature that made Sybase successful, led by his former student Bob Epstein, and is an essential feature in all database system products.

Advocacy

In addition to the above architectural efforts that resulted in new database system product categories, Mike was an early advocate—often the leading advocate—for focusing attention on other critical system-level data management problems and approaches. These included the integration of legacy applications, ensuring distributed database systems will scale out, avoiding the inefficiency of too many layers of database middleware, enabling relational databases to be customized for vertical applications, and circumventing the inflexibility of one-size-fits-all database systems. For his entire career, he has been the database field's technical conscience, an iconoclast who continually asks whether we are working on the right problems and using the best system architectures to address the most expensive data engineering problems of the day. Examples include shared-nothing distributed database architecture (today's dominant approach) [Stonebraker 1985d, Stonebraker 1986d], adopting object-relational databases in preference over object-oriented databases [Stonebraker et al. 1990c, Stonebraker et al. 1990d], an incremental approach to migrating legacy applications [Brodie and Stonebraker 1995a], merging application servers and enterprise application integration (EAI) systems [Stonebraker 2002], and replacing one-size-fits-all database systems with more specialized database engines [Stonebraker and Çetintemel 2005, Stonebraker 2008b].

In 2002, Mike, David DeWitt, and Jim Gray lamented the difficulty of publishing novel ideas with a system focus in database conferences. To address this problem, they created the Conference on Innovative Data Systems Research (http://cidrdb.org/), with Mike as program committee chair of the first conference in 2003, co-PC chair of the second, and co-general chair of the third. As its website says, "CIDR especially values innovation, experience-based insight, and vision." It's my favorite database conference for two reasons: it has a very high density of interesting ideas that are major breaks from the past, and it is single-track, so I hear presentations in all areas of data management, not just topics I'm working on.

In 1989, Mike and Hans Schek led a workshop attended by many leaders of the database research community to review the state of database research and identify important new areas [Bernstein et al. 1998a]. Since then, Mike has been instrumental in ensuring such workshops run periodically, by convening an organizing committee and securing funding. There have been eight such workshops, originally convened every few years and, since 1998, every five years. Each workshop produces a report [Silberschatz et al. 1990, Bernstein et al. 1989, Bernstein et al. 1998b, Abiteboul et al. 2003, Gray et al. 2003, Abiteboul et al. 2005, Agrawal et al. 2008, 2009, Abadi et al. 2014, 2016]. The report is intended to help database researchers choose what to work on, help funding agencies understand why it's important to fund database research, and help computer science departments determine in what areas they should hire database faculty. In aggregate, the reports also provide a historical record of the changing focus of the field.

Mike is always thinking about ways that the database research field can improve by looking at new problems and changing its processes. You can read about many of his latest concerns in his chapter (Chapter 11 in this book. But I'd bet money that by the time this book is published, he'll be promoting other new issues that should demand our attention.

4

Perspectives: The 2014 ACM Turing Award

James Hamilton

Academic researchers work on problems they believe to be interesting and then publish their results. Particularly good researchers listen carefully to industry problems to find real problems, produce relevant work, and then publish the results. True giants of academia listen carefully to find real problems, produce relevant results, build real systems that actually work, and then publish the results.

The most common mistake in academic research is choosing the wrong problem to solve. Those that spend time with practitioners, and listen to the problems they face, produce much more relevant results. Even more important and much more time-consuming, those that build real systems have to understand the problems at an even deeper level and have to find solutions that are practical, can actually be implemented by mortals, and aren't exponential in computational complexity. It's much harder and significantly more time-consuming to build real implementations, but running systems are where solutions are really proven. My favorite researchers are both good listeners and great builders.

Michael Stonebraker takes it a step further. He builds entire companies on the basis of the systems research he's done. We've all been through the experience of having a great idea where "nobody will listen." Perhaps people you work with think they tried it back in 1966 and it was a failure. Perhaps some senior engineer has declared that it is simply the wrong approach. Perhaps people just haven't taken the time to understand the solution well enough to fully understand the value. But, for whatever reason, there are times when the industry or the company you work for still doesn't implement the idea, even though you know it to be a good one and good papers have been published with detailed proofs. It can be frustrating, and I've met people who end up a bit bitter from the process.

In the database world, there was a period when the only way any good research idea could ever see the light of day was to convince one of the big three database companies to implement it. This group has millions of lines of difficult-to-maintain code written ten years ago, and customers are paying them billions every year whether they implement your ideas or not. Unsurprisingly, this was a very incremental period when a lot of good ideas just didn't see the light of day.

Stonebraker lovingly calls this group of three innovation gatekeepers "the Elephants." Rather than wasting time ranting and railing at the Elephants (although he did some of that as well), he just built successful companies that showed the ideas worked well enough that they actually could sell successfully against the Elephants. Not only did he build companies but he also helped break the lock of the big three on database innovation and many database startups have subsequently flourished. We're again going through a golden age of innovation in the database world. And, to a large extent, this new period of innovation has been made possible by work Stonebraker did. To be sure, other factors like the emergence of cloud computing also played a significant part in making change possible. But the approach of building real systems and then building real companies has helped unlock the entire industry.

Stonebraker's ideas have been important for years, his lack of respect for the status quo has always been inspirational, and the database research and industry community have all changed greatly due to his influence. For this and a long history of innovation and contribution back to the database industry and research communities, Michael Stonebraker has won the 2014 ACM Turing Award, the most prestigious and important award in computer science. From the ACM announcement:

> Michael Stonebraker is being recognized for fundamental contributions to the concepts and practices underlying modern database systems. Stonebraker is the inventor of many concepts that were crucial to making databases a reality and that are used in almost all modern database systems. His work on INGRES introduced the notion of query modification, used for integrity constraints and views. His later work on Postgres introduced the object-relational model, effectively merging databases with abstract data types while keeping the database separate from the programming language.
>
> Stonebraker's implementations of Ingres and Postgres demonstrated how to engineer database systems that support these concepts; he released these systems as open software, which allowed their widespread adoption and incorporation of their code bases into many modern database systems. Since the path-breaking work on INGRES and Postgres, Stonebraker has continued to be

> a thought leader in the database community and has had a number of other influential ideas including implementation techniques for column stores and scientific databases and for supporting on-line transaction processing and stream processing.

This chapter was previously published in James Hamilton's Perspectives blog in June 2015. http://perspectives.mvdirona.com/2015/06/2014-acm-turing-award/. Accessed February 5, 2018.

5

Birth of an Industry; Path to the Turing Award

Jerry Held

The year was 1973. Having started my career at RCA Sarnoff Labs in New Jersey a few years earlier, I was working in the field of computer-aided design (CAD) for semiconductors. I was very fortunate to win a Sarnoff Fellowship to pursue a Ph.D. for two years at the university of my choice. My search eventually landed me at UC Berkeley with a plan to do research in CAD (or so I thought).

Although I didn't realize it at the time, it also landed me in the delivery room for the birth of the relational database industry, where the ambition, insatiable curiosity, and almost uncanny luck[1] of an audacious assistant professor named Michael Stonebraker changed computer science and would earn him computing's highest honor, the A.M. Turing Award, 42 years later.

For me, this is the story of a dynamic professional collaboration and friendship that persists to this day and one of the more unique entrepreneurship stories in my nearly 50 years of helping build technology companies. It's also the story of someone who has managed to couple sustained academic curiosity with unending entrepreneurship, which has led to a huge collection of academic work, a long line of successful students, and the creation of numerous companies.

Birth of an Industry (1970s)

In support of my CAD work at RCA, we decided to use a new database system based on work emanating from an industry/government consortium formed to promote standardized ways to access data: CODASYL (short for "Conference on Data Systems Languages"). BFGoodrich, the tire company, had done a huge project to create

[1] As Mike himself will tell you.

an early implementation of CODASYL.[2] RCA was one of the first to purchase the software (which later became Cullinet). Part of my job was to dig into the guts of it and learn how these database systems really worked.

At Berkeley, I was looking for a Ph.D. advisor under whom I could do research in CAD. I was close to picking another professor when I was introduced to Mike, who was just starting his work in databases with a more senior professor, Eugene Wong. As I recall, it took only one meeting to realize that Mike and Gene were starting a very exciting project (dubbed Ingres[3]) and I wanted to be part of it.

I now had my thesis advisor. Mike's ambition (and audacity) were blossoming, as he turned from his own, self-described obscure Ph.D. thesis and area of expertise (applied operations research) to more famous and tenure-making material. Gene was brilliant and also knew the ropes of getting things done at Berkeley—the beginning of what would be an uncanny run of luck for Mike.

There were four keys to the success of the Ingres project:

- timing
- team
- competition
- platform

Ingres—Timing

Gene had introduced Mike to Ted Codd's seminal 1970 paper [Codd 1970] applying a relational model to databases. In Codd's paper, Mike had found his Good Idea[4] (more on this below). At that point, everyone was buzzing about Ted's paper and its potential, particularly when compared to CODASYL and IBM's IMS. Mike had of course read the CODASYL report but dismissed the specification as far too complicated. In the "Oral History of Michael Stonebraker," [Grad 2007] Mike recalled:

> . . . I couldn't figure out why you would want to do anything that complicated and Ted's work was simple, easy to understand. So it was pretty obvious that the naysayers were already saying nobody who didn't have a Ph.D. could understand Ted Codd's predicate calculus or his relational algebra. And even if you got past that hurdle, nobody could implement the stuff efficiently.

[2] CODASYL was supposed to be the industry standard to compete against IMS, IBM's market-dominant, hierarchical database system.

[3] For its originally intended application, an INteractive Graphics REtrieval System.

[4] For more information on how Mike finds his ideas, see Chapter 10.

> And even if you got past that hurdle, you could never teach this stuff to COBOL programmers. So it was pretty obvious that the right thing to do was to build a relational database system with an accessible query language. So Gene [Wong] and I set out to do that in 1972. And you didn't have to be a rocket scientist to realize that this was an interesting research project.

In a headiness that today reminds me of the early days of the commercial Internet (at that point 20-plus years in the future), there was a lot going on. There wasn't just our project, Ingres, and IBM's System R project [5] the two that turned out to be the most prominent, but a dozen other projects going on at universities around the world (see Chapter 13. There were conferences and papers. There was debate. There was excitement. People were excited about the possibilities of databases with a relational model and accessible query languages, and all the things one might be able to do with these.

It was a brave new world. Little did we know that we were all taking the first steps to build a huge new database industry and that it would be thriving some 50 years later. Had we started a little earlier, we might have chosen to do research around the CODASYL model and reached a dead end. By starting later, we likely would have missed the chance to be pioneers. Timing (again) was luckily just right.

Ingres—Team

In my first conversations with Mike and Gene, we discussed my experiences with commercial database systems and agreed to build Ingres with underpinnings that might support real-world database applications.

Mike and Gene put together a small team of students. At the time, there were three Ph.D. students: Nancy MacDonald, Karel Youseffi, and me. There were two master's students: Peter Kreps and Bill Zook. And four undergraduate students: Eric Allman, Richard Berman, Jim Ford, and Nick Whyte. Given my industry experience, I ended up being the chief programmer/project lead on this ragtag group's race to build the first implementation of Codd's vision.

It was a great group of people with diverse skills who really came together as a team. The good fortune of building this team should not be underestimated since, unlike today, most students then had no exposure to computer programming prior to entering university-level study.

[5] For more on the IBM System R project, read Chapter 35 by IBM Fellow and "father of DB2" Don Haderle.

Of course, had we any idea what we were undertaking, we would never have done it; it was far too big and challenging for a group of students.[6] But that's how many great things happen: you don't think about possible failure, you just go for it. (Or, in Mike-speak: "Make it happen.")

Because we were dealing with a completely new space, none of us really knew what relational databases could do. Fortunately, Mike's academic curiosity kept constantly pushing us to think about the possibilities. In parallel with writing code and building the system, the team would have regular meetings in which Mike would lead a discussion, for example, on data integrity or data security. We explored many ideas, wrote concept papers, and laid the groundwork for future years of implementations while concentrating on getting the first version working.

Ingres—Competition

Although there were many other good research projects going on at universities around the world, they were almost entirely academically focused and therefore didn't create much competition for a system that could support real applications.

Meanwhile, just an hour's drive from Berkeley, IBM had created the System R project and assembled a stellar, well-funded team of researchers and computer scientists including the late, wildly talented Jim Gray: a terrific guy with whom both Mike and I later became lifelong friends and collaborators. This was real competition!

Although Ted Codd worked for IBM, he spent time with both teams and kept up with the rapid progress being made. The whole IBM System R team was very collaborative with us even though we were clearly competing.

So why is it that Ingres was built and widely distributed before the System R work saw the light of day? Again, it was our good luck that IBM suffered from "the innovator's dilemma," as described in the famous book of the same name by Clayton Christensen [Christensen 1997].

At the time, IBM dominated every aspect of computing—computer systems, software, operating systems and, yes, databases. IBM had IMS, which was THE database system in the industry, period. If you didn't need a database system, they had ISAM and VSAM, file systems to store everything else. IBM clearly believed that it was not in its interest for this new "relational" database model to take hold and possibly disrupt its dominant position.

[6] As Mike affirmed in an interview with Marianne Winslett [M. Winslett. June 2003. Michael Stonebraker speaks out. *ACM SIGMOD Record*, 32(2)].

And so, between the many university projects and the very well-funded IBM project, Ingres wound up as the first widely available relational database capable of supporting real-world applications.

Ingres—Platform

Maybe our greatest stroke of luck was the choice of platform for building Ingres. Mike and Gene were able to obtain one of the first copies of UNIX, which at the time was a completely unknown operating system. Little did we know that UNIX would become wildly popular; Ingres followed UNIX around the world, and lots of people started using our work. We acquired lots of users, which provided great feedback and accelerated our progress.

Mike made two other key decisions that would change Ingres' fortunes. First, he put the academic code into the public domain using a license that broadly enabled others to use the code and build on it (making him an accidental pioneer in the "open source" movement.)[7] Second, in 1980, he, Gene, and Larry Rowe created a company, Relational Technology, Inc. (RTI) to commercialize Ingres, believing that the best way to ensure that Ingres continued to prosper and work at scale for users was to back it with commercial development.

The Ingres model—open-source academic work leading to a commercial entity —became the formula for the rest of Mike's career (Chapter 7).

Adolescence with Competition (1980s and 1990s)

Meanwhile, I had left Berkeley in 1975 with my newly minted Ph.D. I went on to help start Tandem Computers, taking a lot of what I learned at Berkeley and building out a commercial database with some unique new properties (leading to NonStop SQL[8]). His commercial curiosity piqued, Mike came and did some consulting for us—his first exposure to seeing a startup in action. We had built a great team at Tandem, including people like Jim Gray, Franco Putzolu, Stu Schuster, and Karel Youssefi, and were exploring other aspects of the database world like transactions, fault tolerance, and shared nothing architectures. I think that this experience helped Mike get a taste of entrepreneurship, for which he would later become well known.

[7] For more on this topic, see Chapter 12.

[8] http://en.wikipedia.org/wiki/NonStop_SQL (Last accessed January 4, 2018).

Competing with Oracle

As a part of the Ingres project, we had created the QUEL language [Held et al. 1975] to allow users to query and update the database. The IBM team had defined a different language originally called SEQUEL but later shortened to SQL. A few years before Mike had co-founded RTI to commercialize Ingres, Larry Ellison had started Oracle Corporation with the idea of implementing IBM's SQL query language before IBM brought a product to market.

With IBM's great technology stuck behind the wall, RTI (with Ingres' QUEL language) and Oracle (with SQL) were the leading contenders to provide a commercial relational database system. In the early days of competition, many observers would say that Ingres had the best technology; however, a major lesson for Mike to learn was that "the best technology doesn't always win." Oracle ended up with a significant advantage by adopting IBM's SQL query language; it might not have been as good as QUEL,[9] but IBM's reputation and ability to create both de facto and industry standards won the day. Also, Mike had (and probably still has) a heavy focus on technology, while Ellison was able to win over many customers with strong sales and marketing.

Many more details of the commercial Ingres effort are well documented throughout this book.[10]

Competing with Oracle (Again)

In the meantime, back at Berkeley, Mike's academic curiosity continued post-Ingres. He started the Postgres project to explore complex and user-defined data types—a decade-long project that Stonebraker student/collaborator Joe Hellerstein calls "Stonebraker's most ambitious project—his grand effort to build a one-size-fits-all database system."[11]

Following the five-step model he had used for Ingres, Mike in 1992 formed Illustra Information Technologies, Inc., a company to commercialize Postgres, becoming its Chief Technology Officer. After a few years, Illustra was acquired by Informix, whose major competitor was Oracle.

[9] For an interesting view on this, read Chapter 35 by Don Haderle, who worked on the IBM System R research project.

[10] In particular, see Chapter 15.

[11] For details about Postgres—including its many enduring contributions to the modern RDBMS industry and its code lines—read Joe Hellerstein's detailed Chapter 16.

In 1993, I had left Tandem and gone to what Mike would call "the Dark Side": Oracle, where I ran the database group. With the Illustra acquisition, Mike had become CTO of Informix and we became competitors. During this time, the press was full of articles covering our often-heated debates on the best way to extend the relational database model. Although the Postgres/Illustra Object-Relational technology may have been better than the Oracle 8 Object-Relational technology, Mike again learned that the best technology doesn't always win.

Through it all, Mike and I remained friends—which queued up the next phase of our relationship: commercial collaborators.

Maturity with Variety (2000s and 2010s)

The 1980s and 1990s had been a period of building general-purpose database systems (Ingres and Postgres) and competing across a broad front with Oracle and other general-purpose systems. As the new millennium dawned, however, Mike entered into a long period of focus on special-purpose database systems that would excel at performing specific tasks. This is his "one size does not fit all" period [Stonebraker and Çetintemel 2005, Stonebraker et al. 2007a].

Streaming data, complex scientific data, fast analytics, and high-speed transactional processing were just a few of the areas crying out for Mike's five-step approach to system ideation, development, and technology transfer—and his continued ambition to build companies.

In 2000, Mike retired from Berkeley and headed east to a new base of operations at MIT CSAIL (yet another new frontier, as MIT had no database research group to speak of at the time) and a new home for his family in the Greater Boston area (close to his beloved White Mountains in New Hampshire).

As described elsewhere in this book,[12] Mike proceeded to help create a world-class database research group at MIT while leading research into each of these specialty areas. He also drew on the formidable database expertise at other area universities, such as Brandeis (Mitch Cherniack) and Brown University (Stan Zdonik).

Around the same time that Mike was moving east, I left Oracle to concentrate on what I (finally) realized I loved to do most: help companies get off the ground. I spent a year at Kleiner Perkins and got a world-class education in the venture capital world. After helping to start a couple of Kleiner-backed companies, I set

[12] See Chapter 1, Research Contributions by System; and Part 7.1 Contributions from building systems describing more than a half-dozen special-purpose systems in current or emerging use.

out on my own as a board member and mentor to a long list of startups (and large public companies). This eventually led me back to Mike.

Vertica

Both Mike and I had separately done some consulting with a startup that was attempting to build a special-purpose database system for high-speed analytics. The consulting engagements were both very short, as neither of us was enamored with the company's approach. Mike, however, was intrigued with the general problem and started the C-Store project at MIT [Stonebraker et al. 2005a] to investigate the possible use of column stores for high-speed analytic databases (versus the traditional row store approach). As usual, Mike turned the academic project into a company (Vertica Systems, Inc.) with the help of Andy Palmer as the founding CEO.[13] In 2006, I joined them as chairman of Vertica. Mike and Andy were able to build a great team and the Vertica product was able to produce 100× performance advantage over general-purpose database systems and demonstrate that "one size doesn't fit all." Vertica had a reasonably good outcome as it was acquired by HP in 2011, becoming the company's "Big Data" analytics platform.[14]

VoltDB

While we were building Vertica, Mike started the H-Store research project at MIT with the idea of building a high-performance row store specialized for transaction processing (see Chapter 19 "H-Store/VoltDB"). To get a commercial version off the ground, we incubated a team inside of Vertica. Although there were clear benefits to a specialized system for transaction processing, the market for a very high-performance transaction system was much more limited than on the analytics side and it was significantly more difficult for a startup to pursue these typically mission-critical applications.

VoltDB, Inc. was spun out of Vertica in 2009 and has continued with moderate success. Interestingly, over the past three years, I have been chairman of MemSQL, a company that started with an in-memory row store (à la VoltDB), added a column store (à la Vertica), and has had significant success going after the emerging real-time database market.

[13] See Chapter 18, for more on the company's successful commercial implementation of the technology.

[14] For an inside look at a Stonebraker startup, read Andy Palmer's Chapter 8.

Tamr

Mike realized that the effectiveness of all of the database systems that he had worked on over the decades depended on the quality of the data going into them. He started the Data Tamer project at MIT [Stonebraker et al. 2013b]—with collaborators at QCRI, (Qatar Computing Research Institute) Brandeis University, and Brown University—to investigate how a combination of Machine Intelligence and human data experts could efficiently unify the disparate data sources that feed ever-expanding databases. When it came time to start commercialization of the technology, Mike turned again to Andy Palmer as CEO and me as chairman. Tamr, Inc., was founded in 2013.

Tamr is similar but also very different from previous Stonebraker startups. The Tamr system[15] is similar in that it followed Mike's basic formula for turning academic research into a commercial operation, but modified for a product that *prepares* data for a DBMS instead of *being* the DBMS. It differs from all but the original Ingres project in that it explored a completely new problem (enterprise data unification) instead of being an incremental investigation of different aspects of database technology. We're still in the middle of the Tamr journey; however, many large enterprises are realizing great results and the company is growing nicely.

The Bottom Line

Beyond his ambition to help keep the relational database industry vibrant, "honest," and focused on customers' evolving data management problems, Mike (in my view) was long driven by the ambition to win the Turing Award—the pre-eminent award in the computer science world. In exploring so many different research areas in an academic setting and then proving these ideas in a commercial setting, he steadily built his résumé in a most unique manner to achieve this goal.

In all of his commercial ventures, Mike has taken the role of Chief Technical Officer and led the high-level product architecture. But as importantly, in that CTO role, he has had the opportunity to have many deep interactions with customers and gain an understanding of what the products can and cannot do. This insight has led not only to improvements in the current company's products but also to ideas for new academic research, completing a virtuous circle leading to his next entrepreneurial efforts.

Mike continues to imagine, inquire, investigate, innovate, inspire, and (yes) irritate today, even after having received the 2014 Turing Award and having just

[15] For more on the Tamr system platform, read Chapter 21.

entered his 74th year (at the time I write this). Although there are others who have been more successful in building large, extremely successful database companies (especially Mike's "buddy" Larry Ellison), and there may be someone who has written more academic database papers (but I'm not sure who), there is certainly no one who comes close to Mike as a combined academic and entrepreneur in what has been one of the most important parts of the software world: database systems.

So, Mike, what's next?

6

A Perspective of Mike from a 50-Year Vantage Point

David J. DeWitt

Fall 1970—University of Michigan

When I arrived in Ann Arbor in September 1970 to start a graduate degree in computer engineering, the city was just months removed from a series of major protests against the Vietnam War that would continue until the war was finally ended. Having spent the previous year at various protest marches in Chicago and Washington, D.C., I felt right at home. I was actually fortunate to be in graduate school at all because military deferments for graduate school had been terminated by then and my lottery number was sufficiently low to pretty much guarantee that I was headed for the rice paddies of Vietnam. Had I not failed my Army physical, I could easily have been headed to Vietnam instead of Michigan.

In the late 1960s, very few universities offered undergraduate degrees in computer science (CS). I was actually a chemistry major but had taken three or four CS seminars as an undergraduate. I was definitely not well prepared to undertake a rigorous graduate program. One of the classes I took my first semester as a graduate student was an introductory computer architecture course. So, there I was, a scared, ill-prepared first-year graduate student with this towering guy as our TA (teaching assistant). That TA turned out to be a guy named Mike Stonebraker who would turn out to have a huge impact on my professional career over the next 48 years. It was eye opening to discover that my TA knew less about the subject than I did as an incoming graduate student, despite his height. This gave me hope that I could successfully make the transition to graduate school.

The following Spring, Mike finished his thesis under Arch Naylor and headed off to Berkeley. Among the graduate students, there were rumors of epic battles between Mike and Arch over his thesis. I never learned whether those battles were over content or style (Arch was a real stickler when it came to writing). However, Arch never took on another graduate student between then and when he retired in 1994. Looking back, it is still hard to fathom how a graduate student whose thesis explored the mathematics of random Markov chains and who had no experience building software artifacts and limited programming ability (actually none, I believe) was going to end up helping to launch an entire new area of CS research as well as a $50B/year industry.

From 1971–1976 while I was working on my Ph.D., Mike and I lost contact with one another, but it turned out that both of us had discovered database systems during the intervening period. I took an incredibly boring class that covered the IMS and CODASYL data models and their low-level procedural manipulation languages. As far as I can recall the class never mentioned the relational data model. Almost simultaneously, in 1973 Mike and Gene Wong started the Ingres project—clearly a huge risk since (a) Mike was an untenured assistant professor in an EE (electrical engineering) department, (b) the techniques for building a relational database system were totally unknown, and (c) he really did not know anything about building large software artifacts. Furthermore, the target platform for the first version of Ingres was a PDP 11/45 whose 16-bit address space required building the system as four separate processes. The highly successful outcome of Mike's first "big" bet clearly laid the groundwork for the other big bets he would make over the course of the next 50 years.

In the Spring of 1976, I finished my Ph.D. in computer architecture and took a job as an assistant professor at Wisconsin. A month before classes were to start, I was assigned to design and teach a new class on database systems and strongly encouraged to switch the focus of my research program from computer architecture to database systems even though I knew nothing about this new area of relational database systems. Fortunately, I knew the "tall guy" who, in the intervening six years, had established Berkeley as the academic leader of this new field of research.

Fall 1976—Wisconsin

While Mike and I had not been good friends at Michigan, for reasons I still do not understand Mike decided to become my mentor once we reconnected. He provided me with an early copy of Ingres to use in the classroom and invited me to attend the Second Berkeley Workshop on Distributed Data Management and Computer Networks in May 1977 (Chapter 12). where he introduced me to what was then a very small community of researchers in the database systems field. It was my first exposure to the database research community. It was also the venue where Mike published his first paper on a distributed version of Ingres

[Stonebraker and Neuhold 1977]. Given that Mike and Gene had just gotten Ingres to work, attempting to build a distributed version of Ingres in that timeframe was a huge challenge (there were no "off-the-shelf" networking stacks in Unix until Bill Joy released BSD Unix for the VAX in the early 1980s). While this project did not turn out to have the same commercial impact that Ingres did (nor did R*, the IBM competitor), the numerous technical challenges in access control, query processing, networking, and concurrency control provided a rich set of challenges for the academic research community to tackle. The impact of these two projects, and that of System R, on our field cannot be underestimated. By demonstrating the viability of a DBMS based on the relational data model and through technical leadership at SIGMOD and VLDB conferences, the Ingres and System R projects provided a framework for the nascent database research community to explore, a voyage that continues to this day.

With the tenure clock ticking and bored with the line of research from my Ph.D. thesis (exploitation of functional-level parallelism at the processor level), I decided to try to build a parallel database system and launched the DIRECT project in early 1978 [DeWitt 1979a, 1979b]. While others had started to explore this idea of processing relational queries in parallel—notably the RAP project at Toronto, the RARES project at Utah, the CASSM project at Florida and the DBC project at Ohio State University—I had one critical advantage: I knew the "tall" guy and had, by that time, studied the Ingres source code extensively.

Since obtaining a copy of Ingres required signing a site license and paying a small fee ($50) to cover the cost of producing a tape, some will dispute whether or not Ingres was the first open source piece of software (Chapter 12. My recollection is that the copy of Ingres that I received in the Fall of 1976 included the source code for Ingres—predating the first release of Berkeley Unix by at least two years [Berkeley Software Distribution n.d., BSD licenses n.d.]. Apparently lost to history is what copyright, if any, the early Ingres releases had. We do, however, still have access to the source code for Ingres Version 7.1 (dated February 5, 1981), which was distributed as part of the BSD 4.2 Unix release.[1] Only two files have copyright notices in them: parser.h and tutorial.nr. The copyright notice in these two files is reproduced below:

```
/*
** COPYRIGHT
**
** The Regents of the University of California
**
```

[1] http://highgate.comm.sfu.ca/pups/4BSD/Distributions/4.2BSD/ingres.tar.gz

```
** 1977
**
** This program material is the property of the
** Regents of the University of California and
** may not be reproduced or disclosed without
** the prior written permission of the owner.
*/
```

While this copyright is more restrictive than the BSD copyright used by Ingres in later releases, one could argue that, since none of the other .h or .c files contained any copyright notice, the early versions of Ingres were truly open source.

Whether or not Ingres should be considered as the very first example of open-source software, it was truly the first database system to be released in source-code form and hence provided the nascent database community the first example of a working DBMS that academic researchers could study and modify. The source code for Ingres played a critical role in the implementation of DIRECT, our first effort to build a parallel database system.

As mentioned above, the initial versions of Ingres were implemented as four processes, as shown in Figure 6.1.

Process 1 served as a terminal monitor. Process 2 contained a parser, catalog support, concurrency control, and code to implement query modification to enforce integrity constraints. Once a query had been parsed it was passed to Process 3 for execution. Utility commands to create/drop tables and indices were executed by Process 4.

Since we had access to the Ingres source code, our strategy for implementing DIRECT was to reuse as much of the Ingres code as possible. Other than process 3, we were able to use the other Ingres processes with little or no modification. Our version of process 3 took the query, translated it into a form that DIRECT could execute, and then sent it DIRECT's backend controller for parallel execution on a collection of four PDP 11/23s.

While access to Ingres source code was essential to the project, Mike went much further and made the entire Ingres team available to help me with questions about how Ingres worked, including Bob Epstein (who went on to start Britton

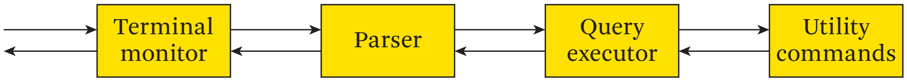


Figure 6.1 Ingres process structure.

Lee and then Sybase) and Eric Allman (of Unix BSD and Sendmail fame). While DIRECT was largely a failure (it ran QUEL queries in parallel but not very effectively; [Bitton et al. 1983]), without Mike's generous support it would not have succeeded at all. Not only would I have not gotten tenure, but also the lessons learned from doing DIRECT were critical to being able to be more successful when I started the Gamma project in early 1984 [DeWitt et al. 1986].

Fall 1983—Berkeley

Having been tenured in the Spring of 1983, it was time for me to take a sabbatical. Mike invited me to come to Berkeley for the Fall 1983 semester, found us a house to rent in Berkeley Hills, and even lent us some speakers for the stereo system that came with the house. While I spent a lot of the semester continuing the Wisconsin benchmarking effort that had started a year earlier, Mike organized a weekly seminar to study novel database algorithms that could take advantage of large amounts of main memory. The resulting SIGMOD 1984 publication [DeWitt et al. 1984] made a number of seminal contributions, including the hybrid hash join and group commit algorithms.

This was also the time that the first commercial version of Ingres (RTI Ingres) appeared and Mike graciously let us benchmark it on one of RTI's computers running VMS. While he had some minor quibbles about the results, he was more than placated by the fact that the commercial version of Ingres was significantly faster than its commercial rival Oracle for essentially all the queries in the benchmark—results that launched a sequence of benchmark wars between the two rival products.

1988–1995—No Object Oriented DBMS Detour for Mike

Frustrated by the impedance mismatch between the limited type system of the relational model and the languages used to develop database applications and inspired by the 1984 SIGMOD paper titled "Making Smalltalk a Database System" by Copeland and Maier [1984], the academic database and startup communities took a detour in an attempt to develop a new generation of database systems based on an object-oriented data model. Mike, unconvinced that a switch in data models was either wise or warranted, crushed the rebellion with a combination of Postgres, the quad chart shown in Figure 6.2, a book to explain the quad chart (*Object-Relational DBMSs: Tracking the Next Great Wave*, with Paul Brown) [Stonebraker and Moore 1996], and a fantastic bumper sticker: "The database for cyber space."

Following the same approach that he had used so successfully with Ingres (and would continue to use numerous times in the future), Mike used the Postgres

Query	Relational DBMS	Object Relational DBMS
No query	File system	Object-Oriented DBMS
	Simple data	Complex data

Figure 6.2 One of Mike's more famous quad charts.

code to start Illustra Information Technologies, Inc., which he sold to Informix in early 1996. The thing I remember most about the sale was that Jim Gray, having discovered the sale price in some SEC EDGAR report, called me about 2 A.M. to tell me. Seems the quad chart was worth about $200M.

One interesting thing to reflect on is that the Illustra acquisition began the long slow decline of Informix, as integrating the two code bases proved to be technically much more difficult than originally assumed. The other interesting outcome is that Postgres to this day remains one of the most popular open-source database systems. In recent years, while Mike has pushed the "one size fits none" mantra to justify a series of application-specific database systems (Vertica, Paradigm4, and VoltDB), I always remind myself that Postgres, by far his most successful DBMS effort, mostly definitely falls into the "one-size-fits-all camp."

2000—Project Sequoia

In the mid-1990s, NASA launched an effort to study the earth called "Mission to Planet Earth" (MTPE) using a series of remote sensing satellites designed to obtain "data on key parameters of global climate change." As part of this effort, NASA issued an RFP for alternative designs for the data storage and processing components that would be needed to store and analyze the terabytes of data that the satellites were expected to generate over their lifetimes. This RFP inspired Mike, along with Jim Gray, Jeff Dozier, and Jim Frew, to start a project called Sequoia 2000 to design and implement a database-centric approach (largely based on Postgres). While NASA eventually rejected their approach (instead they picked a design based on CORBA—remember CORBA?), the effort inspired Gray to work with Alex Szalay to use SQL Server for the Sloan Digital Sky Survey, which proved to be a huge success in the space science community.

The Sequoia 2000 project also inspired myself and Jeff Naughton to start the Paradise project at Wisconsin. While we were convinced of the merits of a database-centric approach, we did not believe that a single Postgres instance would be sufficient and set out to build a parallel database system from scratch targeted at the technical challenges of the MTPE data storage and processing pipeline. While Paradise reused many of the parallel database techniques that we had developed as part of the Gamma project, it had several unique features, including a full suite of parallel algorithms for spatial operations (e.g., spatial selections and joins) and integrated support for storing and processing satellite imagery on tertiary storage.

2003—CIDR Conference Launch

In June 2002, having had all our papers rejected by Mike Franklin, the program committee chair of the 2002 SIGMOD conference, Mike, Jim Gray, and I decided to start a new database system conference. We felt strongly that SIGMOD had lost its way when it came to evaluating and accepting systems-oriented database research papers (a situation we find ourselves in again in 2018 (see Chapter 11). We felt the only solution that was going to work was to start a new conference with an objective of accepting only papers that had the potential to advance the state of the art. We specifically did not want people to submit polished pieces of research, preferring instead half-baked ideas. To this day, CIDR, the Conference on Innovative Data Systems Research, continues to thrive as an outlet with far more submissions than a single-track conference can accept.

2005—Sabbatical at MIT

I was fortunate to spend the 2005–2006 academic year at MIT. This was soon after Mike had started Vertica to build a scalable data warehousing platform based on a column store paradigm. While the idea of storing tables in a column-oriented format dates back to Raymond Lorie's XRM project at IBM and had been explored for use in main memory database systems by the MonetDB project at CWI, Vertica was the first parallel database system that used tables stored as columns exclusively in order to dramatically improve the performance of decision-support queries operating against large data warehouses. It was an amazing opportunity to have a close-up look at Mike launching and running one of his numerous startups. While Vertica has had modest commercial success, it has had a huge impact on the DBMS field. Every major database platform today either uses a columnar layout exclusively or offers a columnar layout as a storage option. ORC and Parquet, the two major HDFS file formats for "big data," also both use columnar storage layout.

2008—We Blog about "MapReduce"

In 2007, the CS field was abuzz about MapReduce, Google's paradigm for processing large quantities of data. Frankly, we were amazed about the hype it was generating and decided to write a blog post for the Vertica website with our reactions [DeWitt and Stonebraker 2008]. The key point we were trying to make was that while the fault tolerant aspects of MapReduce were novel, the basic processing paradigm had been in use by parallel database systems for more than 25 years. We also argued abandoning the power of a declarative language like SQL, however flawed the language might be, for a procedural approach to querying was a really bad idea. The blog post would probably not have attracted much attention except that it got "slashdotted." The reaction of the non-database community was incredible hostile. "Idiots" was one of the kinder adjectives applied to us.

It is interesting to reflect on this blog post ten years later. While perhaps a few hardcore hackers use MR today, every major big data platform (Hive, Presto, Impala, Cloudera, Red Shift, Spark, Google BigQuery, Microsoft SQL DW, Cosmos/Scope, etc.) all use SQL as their interface. While early versions of Hive used MapReduce as its execution engine, Tez, the current Hive executor, uses a vectorized executor in combination with standard parallel query mechanisms first developed as part of the Gamma project in the early 1980s.

Where are the MR fan boys today? They owe us an apology.

2014—Finally, a Turing Award

In 1998, I led the effort to nominate Jim Gray for a Turing Award. When I started that effort, I spent a fair amount of time debating with myself whether to nominate Jim or Mike first. By that time, both had had a huge impact on our field and it was not at all obvious to me who should be nominated first (System R and Ingres had been jointly recognized with the ACM Software Systems Award in 1988). While Mike's publication record was clearly broader, Jim's pioneering work on transactions [Eswaran et al. 1976] seemed to me to be ideal to base a Turing Award nomination on. I still recall thinking that a successful nomination for Mike would follow shortly after Jim. Soon after Jim received the Turing Award, he was appointed to join the Turing Award selection committee and, while serving, felt it inappropriate to support a nomination for Mike. He kept promising that once he was off the committee he would strongly support a nomination for Mike but unfortunately was lost at sea in January 2007 before that came to pass. It is sad that the Turing Award process is so political that it took 16 years to recognize Mike's major contributions to both the research and industrial database communities. It was never my intention for it to turn out this way. Had I known in 1998 what I know now, I probably would have done things differently.

2016—I Land at MIT

Almost 50 years since we first met at Michigan, Mike and I again find ourselves at the same place, this time at MIT. While we are both nearing the end of our careers (well, at least I am) our hope is that this will be an opportunity to do one last project together.

2017

At every important juncture of my career, Mike was there to give me assistance and advice. I owe him a huge debt of gratitude. But I also owe thanks to whoever scheduled Mike to be the TA for that introductory computer architecture class in the Fall of 1970. That chance encounter with Mike turned out to have a profound impact on both my career and my life.

PART V

STARTUPS

7

How to Start a Company in Five (Not So) Easy Steps

Michael Stonebraker

Introduction

This chapter describes the methodology I have used in starting nine venture capital-backed companies. It assumes the following.

(a) I have experience in starting system software companies. The procedure is quite different for hardware companies and applications such as those in biotech. Nothing herein applies outside of system software.

(b) It occurs in a university context, whereby one is trying to commercialize an idea from a university research project.

(c) One requires capital from the venture capital (VC) community. I have no experience with angel investors. I am not a fan of self-funded startups. Unless you are independently wealthy, you will need a day job, and your startup will be nights and weekends only. In system software, a part-time effort is really difficult because of the amount of code that typically needs to be written.

This chapter is divided into five steps, to be performed in order. At the end, there are some miscellaneous comments. Along with each step, I present, as an example, how we accomplished the step in the formation of my latest company, Tamr, Inc.

Step 1: Have a Good Idea

The first step in starting a company is to have a good idea. A good idea is specific—in other words, it is capable of prototype implementation, without further specification. An idea like “I would like to do something in medical wearables” is not specific enough. At MIT, where I hang out, good ideas are presented in many of the faculty lunches. Hence, my advice is to situate yourself at a research institution like

CSAIL/MIT or CS/Berkeley or . . . . Good ideas seem to flow out of such environments.

So what do you do if you are not in an idea-fertile place? The answer is: travel! For example, one of my collaborators is a CS professor and researcher at another university. However, he spends a considerable amount of time at CSAIL, where he interacts with the faculty, students, and postdocs at MIT. Most universities welcome such cross-fertilization. If you are in the hinterlands, then find a fertile place and spend airplane tickets to hang out there.

How does one decide if an idea is worthy of commercialization? The answer is "shoe leather." Talk to prospective users and get their reaction to your proposal. Use this feedback to refine your ideas. The earlier you can reinforce or discard an idea, the better off you are because you can focus your energy on ideas that have merit.

If you come up with an "idea empty hole," then consider joining somebody else who has a good idea. Startups are always a team effort, and joining somebody else's team is a perfectly reasonable thing to do.

Once you have a good idea, you are ready to advance to step 2.

In the case of Tamr, Joey Hellerstein of UC Berkeley was spending a sabbatical at Harvard, and we started meeting to discuss possible projects. We quickly decided we wanted to explore data integration. In a previous company (Goby), I had encountered the issue of data integration of very large amounts of data, a topic that arises in large enterprises that want to combine data from multiple business units, which typically do not obey any particular standards concerning naming, data formatting, techniques for data cleaning, etc. Hence, Joey and I quickly homed in on this. Furthermore, MIT was setting up a collaboration with the Qatar Computing Research Institute (QCRI). Ihab Ilyas and George Beskales from QCRI started collaborating with us. Goby's major problem was deduplicating data records from multiple sources that represented the same entity, for example deduplicating the various public data sources with information about Mike Stonebraker. The QCRI team focused on this area, leaving MIT to work on schema matching. Last, Stan Zdonik (from Brown) and Mitch Cherniack (from Brandeis) were working with Alex Pagan (an MIT graduate student) on expert sourcing (i.e., crowdsourcing, but applied inside the enterprise and assuming levels of expertise). They agreed to apply their model to the Goby data. Now we had the main ideas and could focus on building a prototype.

Step 2: Assemble a Team and Build a Prototype

By now, you hopefully have a few friends who have agreed to join you in your endeavor. If not, then recruit them. Where should you look? At the idea factory where you hang out! If you cannot find competent programmers to join your team, then you should question whether you have a good idea or not.

This initial team should divide up the effort to build a prototype. This effort should be scoped to take no more than three months of your team's effort. If it takes more than three months, then revise the scope of the effort to make your prototype simpler. It is perfectly OK to hard-code functionality. For example, the C-Store prototype that turned into Vertica ran exactly seven queries, whose execution plans were hard-coded!

In other words, your prototype does not need to do much. However, make sure there is a graphical user interface (GUI); command line interfaces will make the eyes of the VCs glaze over.

VCs need to see something that demonstrates your idea. Remember that they are business people and not technologists. Your prototype should be simple and crisp and take no more than five minutes to demonstrate the idea. Think "Shark Tank," not a computer science graduate class.

Your prototype will almost certainly be total throwaway code, so don't worry about making the code clean and maintainable. The objective is to get a simple demo running as quickly as possible.

In the case of Tamr, we merely needed to whack out the code for the ideas discussed above, which was a team effort between QCRI and MIT. Along the way, we found two more use cases.

First, Novartis had been trying to integrate the electronic lab notebooks for about 10,000 bench scientists. In effect, they wanted to integrate 10,000 spreadsheets and had been trying various techniques over the previous 3 years. They were happy to make their data structures available for us to work on. This gave us a schema integration use case. Last, through the MIT Industrial Liaison Program (ILP), we got in touch with Verisk Health. They were integrating insurance claim data from 30-plus sources. They had a major entity consolidation problem, in that they wanted to aggregate claims data by unique doctor. However, there was substantial ambiguity: two doctors with the same last name at the same address could be a father-and-son practice or a data error. We had another, "in the wild," entity consolidation problem. In Data Tamer, we did not focus on repair, only on detection. Verisk Health had a human-in-the-loop to correct these cases.

Our prototype worked better than the Goby handcrafted code and equaled the results from the automatic matching of a professional data integration service on Verisk Health data. Last, the prototype appeared to offer a promising approach to the Novartis data. Hence, we had a prototype and three use cases for which it appeared to work.

With a prototype, you can move to step 3.

Step 3: Find a Lighthouse Customer

The first question a VC will ask you is, "Who is your customer?" It helps a lot to have an answer. You should go find a couple of enterprises that will say, "If you build this for real, then I will consider buying it." Such lighthouse customers must be real, that is, they cannot be your mother-in-law. VCs will ask to talk to them, to make sure that they see the same value proposition that you do.

What should you do if you can't find a lighthouse customer? One answer is to "try harder." Another answer is to start questioning whether you have a good idea. If nobody wants your idea, then, by definition, it is not a good idea.

If you lack contacts with business folks, then try hanging out at "meetups." There is no substitute for network, network, network. After trying hard, if you still can't find a lighthouse customer, then you can continue to the next step of the process, but it will make things much harder.

In the case of Tamr, we had three lighthouse customers, as noted in the previous section. All were happy to talk to interested people, which is what the next step is all about.

Step 4: Recruit Adult Supervision

VCs will look askance at a team composed of 23-year-old engineers. In general, VCs will want some business acumen on your team. In other words, they will want somebody who has "done it before." Although there are exceptions (such as Mark Zuckerberg and Facebook), as a general rule a team must have a business development or sales executive. In other words, somebody has to be available to run the company, and the VCs will not entrust that to somebody with no experience. Although your team may have an MBA type, VCs will look askance at a 23-year-old. VCs are risk-averse and want to entrust execution to somebody who has "been around the block a few times."

So how do you find a seasoned executive? The answer is simple: shoe leather. You will need to network extensively. Make sure you find somebody you can get along with. If the dynamics of your relationship are not terrific on Day 1, they are likely just to get worse. You will be spending a lot of time with this person, so make sure it is going to work.

Also, make sure this person has a business development or sales pedigree. You are not particularly looking for a VP/Engineering, although one of those would be nice. Instead, you are looking for someone who can construct a go-to-market strategy, write a business plan, and interact with the VCs.

What happens if you can't find such a person? Well, do not despair; you can continue to the next step without adult supervision. However, the next step will be tougher . . . .

In the case of Tamr, I reached out to Andy Palmer, who had been the CEO of a previous company (Vertica) that we co-founded. Andy worked at Novartis at the time and verified the importance of data integration with Novartis engineers. In addition, he invited Informatica (a dominant player in the data-integration software market) to talk about how they could solve the Novartis problem. It became clear that Novartis really wanted to solve its data integration challenge and that Informatica could not do so. Armed with that information, Andy agreed to become CEO of Tamr.[1]

Step 5: Prepare a Pitch Deck and Solicit the VCs

By this point in the process, you hopefully will have one or more lighthouse customers and a team that includes software developers and at least one "adult." You are now ready to pitch the VCs. Your adult should be in charge of the presentation; however, you should remember that VCs have a short attention span—no more than 30 minutes. Your deck should not have more than 15 slides, and your pitch should include a 5-minute demo.

How do you find VCs to pitch? Ask around: in other words, network. In the hotspots (Silicon Valley, Seattle, Boston, New York, etc.), VCs are literally on every street corner. If you are in the hinterlands, then you should consider moving to a hotspot. Moreover, it is unlikely that you can find good adult supervision in the hinterlands.

Check out the reputation of any VC with whom you interact. Each VC has a reputation that precedes him or her, good or bad. Run away from anybody who does not have a stellar reputation for fairness. I have heard enough horror stories of entrepreneurs getting taken advantage of by VCs or by CEOs brought in by the VCs, and I have had one painful experience myself in this area. My strong advice: If it doesn't feel right, you should run the other way.

Now, try out your pitch on a couple of "friendly" VCs. They will trash your pitch for sure, and you can now do the first of what will probably be several iterations. After a while your pitch deck will get better. Expect to pitch several-to-many VCs and to spend months on this process.

Every VC will give you one of three reactions.

[1] See Chapter 21 for the story of this system.

- "I will get back to you." This is code for, "I am not interested."
- "I am not interested, because [insert some usually irrelevant reason]." This is a slightly more genuine way of saying, "I am not interested."
- A "rock fetch." The VC will ask you to try out your pitch on:
  - possible executives (to beef up step 4)
  - possible customers (to beef up step 3)
  - their friends—to help them evaluate your proposal
  - companies in their portfolios—again to help them evaluate your proposal

In each case, the VC is gathering information on the validity of your business proposal. You should expect several-to-many rock fetches and the process to go on for weeks. Although rock fetches are very frustrating, you really have only two choices.

1. Do the i+1st[2] rock fetch.
2. Tell the VC you are not interested.

In general, you may have to turn over a lot of rocks to find a diamond. Enjoy the process as best you can. Hopefully, this ends with a VC saying he or she wants to give you a term sheet (details of his proposed investment). If this happens, then there may well be a "pile on." Previous VCs who were not interested may suddenly become very interested. In other words, there is a "follow the crowd" mentality. Unfortunately, this will not usually result in the price of the deal rising; there will just be more people around the table to split the deal. The above statements do not apply to "unicorns" (companies like Facebook or Uber), which have valuations in the stratosphere. The rest of us are stuck with down-to-earth values.

When you receive a term sheet, it is crucial that you find somebody who has "done it before" to help you negotiate the deal. The non-financial terms are probably more important than the financial ones, so pay close attention.

The most onerous term is "liquidation preference." VCs will routinely demand that they get their money back in any liquidity event before the common stockholders get anything. This is known as a 1× preference. However, I have seen term sheets that propose a 3× preference. Suppose you accept $20M of venture capital over multiple financing rounds. With a 3× preference, the VCs get the first $60M before the founders and employees get anything. If the VCs have 60% of the stock

[2] Where "i" is some big number and "+1" is yet another. In other words, a LOT of rock fetches.

and a 3× preference, then a liquidity event of $80M will mean the VCs get $72M and others get $8M. As you can see, this is not a terrific outcome. Hence, pay careful attention to this and similar terms.

The other thing about negotiating with the VCs is that they do deals on a regular basis and you don't. It is reminiscent of the automobile market before the Internet. VCs will throw around statements like "this salary is not market" or "this stock position is not market." It is difficult to argue since they have all the data and you don't. My only suggestion is to get somebody who has done it before to help with the negotiation, and to keep you out of all the assorted sand traps that you would otherwise encounter.

Expect the negotiation with the VC (or VCs) to be a back-and-forth process. My advice is to settle all the non-financial terms first. Second, make sure you have the amount that you want to raise. In general, this should be enough money to get your product into production use at one of your lighthouse customers. Don't forget that quality assurance (QA) and documentation must be included. When you think you have a number, then double it, because entrepreneurs are notoriously optimistic. Then, the negotiation boils down to a raise of $X in exchange for Y% of the company. In addition, the remainder must be split between the founders and an option pool for other employees to be hired in the first year. Hence, the only things to be negotiated are Y and the size of the option pool. My advice is to fix essentially everything and then negotiate the price of the deal (Y above).

In the case of Tamr, Andy and I pitched a friendly VC from New Enterprise Associates, who took the deal on the spot, as long as the terms were within what he could accept without asking his partners. He had the advantage that he knew the data integration space and the importance of what we were trying to do. Armed with his acceptance, we found it relatively easy to recruit another VC to supply the other half of the capital we needed.

Assuming you can come to terms with the VC (or VCs), you are off to the races. In the next section, I make a collection of random comments on topics not discussed above.

Comments

Spend No Money

Companies fail when they run out of money, and in no other way. Do not spend your capital on Class A office space, a receptionist, office furniture (buy from Ikea if you can't scrounge it from somewhere else), a car service, or somebody to make travel arrangements. Also, do not hire a salesperson—that is the job of the adult on your team. Your team should be as lean as possible! The adage is "kiss every nickel."

Intellectual Property

I am routinely asked, "What about intellectual property?" I am a huge fan of open-source software. If you did development at a university, then declare your efforts open source, and make a clean "toss the code over the wall" into your company. Otherwise, you are stuck negotiating with the Technology Licensing Office (TLO) of your university. I have rarely seen this negotiation go well. Your company can be open or closed source, and can submit patent applications on work done after the wall toss. I am also a fan of reserving a block of stock for your university, so they get a windfall if your company does well. Also, dictate the stock go to Computer Science, not to the general university coffers.

Your VCs will encourage you to submit patent applications. This is totally so you can say "patented XXX" to prospective customers, who think this means something. It costs about $25K to submit an application and a month of your time. Go along with the VCs on this one.

I have rarely seen a software patent stand up to scrutiny. There is always prior art or the company you are suing for patent infringement is doing something a bit different. In any case, startups never initiate such suits. It costs too much money. In my experience, software patents are instead used by big companies for business advantage.

In the case of Vertica, a large company against which we were competing (and routinely winning) sued us for patent infringement of one of their old patents. In round numbers, it cost us $1M to defend ourselves and it cost them $1M to push the suit. Their $1M was chump change since they were a large company; our $1M was incredibly valuable capital. Moreover, they told all of our prospects, "Don't buy Vertica, since they are being sued." Although we ultimately won, they certainly slowed us down and distracted management. In my opinion, the patent process is routinely abused and desperately needs a massive overhaul.

First Five Customers

It is the job of your adult to close the first five customers. Do not worry about getting lots of money from them. Instead, you just need them to say nice things to other prospective customers about your product.

Raising More Money

The general adage is “raise more money only when you don’t need it.” In other words, whenever you pass an oasis, stock up on water. In general, you can get the best price for your stock when you don’t need the money. If you can’t get a reasonable deal, then turn down the deal. If you are in danger of running out of money (say within six months of “cash out”), the VCs will hammer you on price or string you along until you are desperate and then hammer you on price. You can avoid this by raising money well before you run out.

Two VCs

In my opinion, if you have a single VC funding your company, you get a boss. If you have two, you get a Board of Directors. I much prefer two, if you can swing it.

Company Control

Unless you are a unicorn, the VCs will control the company. They invariably make sure they can outvote you if push comes to shove. Hence, get used to the fact that you serve at their pleasure. Again, I can’t overemphasize the importance of having a VC you can work with. Also, fundamentally your adult is running the company, so the VCs are actually backing that person. If you have a falling out with your adult, you will get thrown under the bus.

The worst problems will occur if (or when) your adult tires of the grind of running a startup. If he or she exits, then you and the VCs will recruit a new CEO. My experience is that this does not always go well. In two cases, the VCs essentially insisted on a new CEO with whom I did not get along. In both cases, this resulted in my departure from the company.

Secrecy

Some companies are very secretive about how their products work. In my opinion, this is usually counterproductive. Your company wins by innovating faster than the competition. If you ever fail to do this, you are toast. In my opinion, you should fear other startups, who are generally super smart and don’t need to learn from you. Large companies typically move slowly, and your competitive advantage is moving more quickly than they do. In my opinion, secrecy is rarely a good idea, because it doesn’t help your competitive position and keeps the trade press from writing about you.

Sales

It never ceases to gall me that the highest-priced executive in any startup is invariably the VP of Sales. Moreover, he or she is rarely a competent technologist, so you have to pair this person with a technical sales engineer (the so-called four-legged sales team). The skill this person brings to the table is an ability to read the political tea leaves in the companies of potential customers and to get customer personnel to like them. That is why they make the big bucks!

The most prevalent mistake I see entrepreneurs make is to build out a sales organization too quickly. Hire sales people only when your adult is completely overloaded and hire them very, very slowly. It is rare for a sales person to make quota in Year One, so the expense of carrying salespeople who are not delivering is dramatic.

Other Mistakes

There are two other mistakes I would like to mention.

First, entrepreneurs often underestimate how hard it is to get stuff done. Hence, they are often overoptimistic about how long it will take to get something done. As I noted above, double the amount of money you request as a way to mitigate this phenomenon. Also, system software (where I have spent all of my career, essentially) is notoriously hard to debug and make "production-ready." As a result, startups often run out of money before they manage to produce a saleable product. This is usually catastrophic. In the best case, you need to raise more money under very adverse circumstances.

The second mistake is trying to sell the product before it is ready. Customers will almost always throw out a product that does not work well. Hence, you spend effort in the sales process, and the net result is an unhappy customer!

Summary

Although doing a startup is nerve-racking, requires a lot of shoe leather and a lot of work, and has periods of extreme frustration, I have found it to be about the most rewarding thing I have done. You get to see your ideas commercialized, get to try your hand at everything from sales to recruiting executives, and get to see the joy of sales and the agony of sales failures firsthand. It is a very broadening experience and a great change from writing boring research papers for conferences.

8

How to Create and Run a Stonebraker Startup—The *Real* Story

Andy Palmer[1]

If you have aspirations to start a systems software company, then you should consider using the previous chapter as a guide.

In "How to Start a Company in Five (Not So) Easy Steps," (see Chapter 7). Michael Stonebraker has distilled the wisdom gained from founding 9 (so far) database startups over the last 40 years.

As the "designated adult supervision" (business co-founder and CEO) in two Stonebraker database startups (Vertica Systems and Tamr) and a founding BoD member/advisor to three others (VoltDB, Paradigm4,[2] Goby), I have lived and developed this approach with Mike through more than a dozen years marked by challenging economic climates and sweeping changes in business, technology, and society.

I am privileged to have had the opportunity to partner with Mike on so many projects. Our relationship has had a profound effect on me as a founder, as a software entrepreneur, and as a person.

In this chapter, I'll try to answer the question: "What's it like *running* a company with Mike Stonebraker?"

[1] As acknowledged by Mike Stonebraker in his Turing lecture, "The land sharks are on the squawk box." [Stonebraker 2016]

[2] See Chapters 27–30 on the research and development involving C-Store/Vertica, Tamr, H-Store/VoltDB, and SciDB/Paradigm4.

Figure 8.1 Andy Palmer and Mike Stonebraker at the 2014 Turing Award Ceremony, June 20, 2015. Photo credit: Amy Palmer.

An Extraordinary Achievement. An Extraordinary Contribution.

Founding nine startup companies is an extraordinary achievement for a computer scientist. As the 2014 ACM Turing Award citation noted:

> "Stonebraker is the only Turing award winner to have engaged in serial entrepreneurship on anything like this scale, giving him a distinctive perspective on the academic world. The connection of theory to practice has often been controversial in database research, despite the foundational contribution of mathematical logic to modern database management systems."

All of these companies were at the intersection of the academic and commercial worlds: Mike's tried-and-true formula [Palmer 2015b]. They involved convincing businesses across industries to provide their large working datasets—an achievement in itself—to help turn academic research and theory into breakthrough software that would work at scale.

Mike broke new ground not just in starting these companies, but also in *how* he has started them. Mike's methods are uniquely suited to bringing the best academic ideas into practice.

He focused on solving big "unsolvable" real-life problems versus making incremental improvements on current solutions. Incrementalism has all too often been the status quo among database software vendors, to the detriment of business and industry. A great example of how incrementalism plagued commercial database systems was the perpetuation of "row-oriented" database systems and incremental enhancements, such as materialized views, in the 1980s and 1990s. It took Mike's "One size does not fit all in database systems" paper [Stonebraker and Çetintemel 2005] to help shake things up and inspire the proliferation of innovation in database systems, including Vertica's column-oriented analytics database (acquired by HP (Hewlett-Packard) and now part of Micro Focus' software portfolio). The sheer number and variety of purpose-oriented databases on the market today is a testament to Mike's reluctance to settle for incrementalism.

He takes a disciplined, engineering-driven approach to systems software design and development. Mike always had a clear understanding of the differences between an academic code line, startup first release code, startup second release code, and code that would actually work at scale very broadly. He recognized how hard it was to build something rock solid, reliable, and scalable. He was able to suspend much of his academic hubris when it came time to build commercial product, respecting the requirements and costs of building real systems. He also recruited great people who had the patience and discipline to spend the years of engineering required to build a great system (even if he didn't have the patience himself). Mike is notorious for referring to *SMOC,* or "a Simple Matter of Code." But he always knew that it took many person-years (or decades) to write the code after working out the core design/algos—and how to achieve this optimally through strategic hiring and sticking to the right approach.

He takes a partnership approach to leadership. During the time I've worked with him, Mike has embraced a partnership approach, believing that success depends on both great technology and great business/sales—skills that don't typically coexist in one leader. I've always found him to actively want to be a partner, to work together to build something great. He believed that, through partnership and the open exchange of strong opinions, partners can get to the best answers for the project or company. Mike and I love to argue with each other. It's fun to go at a hard problem together. It doesn't matter if it's a business problem or a technical problem, we love to hash it out. I think that each of us secretly likes talking about the other's area of expertise.

Mike's track record in starting new companies is better than most venture capitalists. Of his startups to date, three have delivered greater than 5× returns to their investors (Ingres, Illustra/Postgres, and Vertica), three delivered less, and three are still "in flight." Blue-chip and leading technology companies—such as Facebook [Palmer 2013], Uber, Google, Microsoft, and IBM as well as cutting-edge mainstream industrial companies—such as GE and Thomson Reuters—use products from Stonebraker companies/projects. Stonebraker-inspired products have been in use for every second of every day across the world for nearly three decades.[3]

For these reasons, Mike's footprint extends far beyond the companies he founded, the students he taught, and the academic collaborators he inspired. The technology industry and the business world have learned a lot from him.

A Problem of Mutual Interest

A Happy Discovery

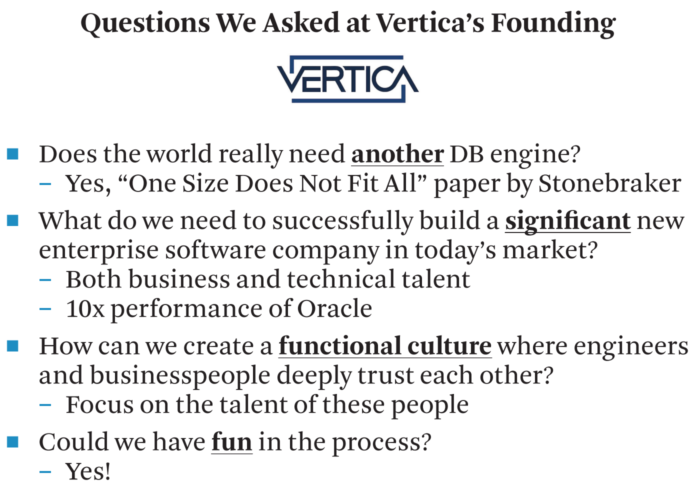
Mike and I were introduced in 2004 by Jo Tango,[4] a venture capitalist who was at Highland Capital Partners at the time. We were at a Highland event at the Greenbrier resort in West Virginia. Our wives met first and hit it off: so much so that they told us that we *had* to start a company together. Our meeting surfaced a problem of mutual interest, followed by a happy discovery that we really got along and saw startup stuff the same way. (Good thing for the preservation of marital bliss.)

I was then chief information and administrative officer for Infinity Pharmaceuticals, and we had just finished trying to implement a large data warehouse using Oracle RAC (an effort that essentially failed). Mike was working on the C-Store project at MIT [Stonebraker et al. 2005a], a scale-out *column-oriented database.* I had experienced firsthand the problems that Mike was trying to address with C-Store and was starting to think about doing another startup.

What really sealed the deal, however, is that Mike and I shared the same core values, particularly the partnership approach to founding a company. We both believed that the best companies are often made through partnerships between people who appreciate each other, have a sense of humility in their own areas, and

[3] For visual proof, see Chapter 13, by Naumann and Chart 2 of Part 1 by Pavlo

[4] Read Tango's Chapter 9.



Questions We Asked at Vertica's Founding

- Does the world really need **another** DB engine?
  - Yes, "One Size Does Not Fit All" paper by Stonebraker
- What do we need to successfully build a **significant** new enterprise software company in today's market?
  - Both business and technical talent
  - 10x performance of Oracle
- How can we create a **functional culture** where engineers and businesspeople deeply trust each other?
  - Focus on the talent of these people
- Could we have **fun** in the process?
  - Yes!

Figure 8.2 In founding Vertica, we asked ourselves several questions.

look to others to help them make the best decisions to achieve the best outcomes. We founded Vertica in 2005 (see Figure 8.2).

The Power of Partnership

Our partnership has worked over the years because our principles and core values are aligned. Our approach to starting companies is based in pragmatism and empirics, including:

1. a focus on great engineering, first and foremost;
2. build products designed to solve real customer problems [Palmer 2013];
3. deliver thoughtful system architecture: work in the "white space" instead of incrementally reinventing the proverbial wheel;
4. hire the best people (including "looking beyond the resume") and treat people respectfully (check titles and degrees at the door);
5. test all assumptions with many real customers, early and often;
6. take a capital-efficient approach, even in "flush" times, and consider the full lifecycle when seeking initial capital (think beyond the first round/next milestone); and
7. have fun in the process.

Vertica's success derived from our adherence to those core values and the hard work of many people over many years—most notably Colin Mahony, who still leads Vertica at Micro Focus; Shilpa Lawande, who led engineering for 10-plus years, and Chuck Bear, who was the core architect.

Many of Vertica's early prospective customers had expressed interest in a specific feature (materialized views). This was a classic example of customers asking for things that aren't in their own long-term best interests. Mike said that there was no way we were going to build materialized views; his exact words were "over my dead body"—another Mike-ism. A startup with a top-down, sales-oriented CEO would have probably just directed engineering to build materialized views as requested by customers. Instead, we had a healthy debate about it, but decided not to do it, which turned out to be absolutely the right thing. Another way to view this is that Vertica's entire system is a collection of materialized views—what we called "projections." And over time, our customers came to appreciate the fact that they didn't need to implement materialized views: they just needed to use Vertica.

Another example was pricing: I believed that we needed an innovative pricing model to differentiate our product. Instead of the conventional pricing model (which was then based on per-server or per-CPU), we went with a pricing model tied to the amount of data that people loaded into the system (per-terabyte). (Credit goes to product marketing pro Andy Ellicott for this idea.) This was counterintuitive at the time. Mike probably would have said, "Let's choose something that's easy for the customers." But I was really confident that we could pull this off and that it was right thing to do, and we talked it through. Mike supported the idea and in the end it was right. Much of the analytical database market subsequently migrated toward the per-terabyte pricing model.

The partnership model of entrepreneurship[5] takes work, but it's proven well worth it in the companies that we've co-founded.

After finding the right partner, it's vital to have a simple but firm partnership agreement, including: (1) a mutual understanding of the entire journey to build the company; (2) a willingness to challenge assumptions at each stage of its development; (3) a mutual appreciation and respect for both the technical and business sides of the company to prevent fingerpointing; and (4) and explicit alignment on core values. You should accept and plan for the overhead associated with effective

[5] Read more about the importance of a partnership model Chapter 7 "How to Start a Company in Five (Not So) Easy Steps" (Stonebraker) and 9 "Getting Grownups in the Room: A VC Perspective" (Tango).

shared leadership, including the time for healthy debates and navigating disagreements. Mike and I have gotten into some pretty heated discussions over the years, all of which made for better companies and an ever-stronger relationship.

Great co-founder relationships are more flexible—and more fun. Startups are inherently hard: a partner can help you through the tough times, and it's definitely more fun to celebrate with someone during the good times. Startups are also constantly changing, but with a complementary co-founder, it's much easier to roll with the changes. It's like a built-in support group.

At the same time, Mike and I aren't joined at the hip. We both have lots of interests and are always looking at lots of different things in our areas of interest. I have extracurricular interests in a whole bunch of different businesses, and Mike has interests in all kinds of different projects and technology. This brings invaluable outside perspective into our relationship and the projects we work on together. This is why—coupled with our deep trust in each other—we can do many different separate projects but always come back together and pick up exactly where we left off on our joint projects.

We're alike in that we both want to get lots of data and we appreciate external feedback. We're open to better ideas, regardless of where they come from.

Shared values and deep trust in each other saves a lot of time. Early in the history of Vertica, we were out raising money. Our financing terms were aggressive, but we were confident that we were on to something. We were in the offices of a large, well-known venture capital firm. There was a junior venture partner who wanted us to commit to certain revenue and growth numbers, but it was really early in the lifecycle of the company. Mike and I glanced at each other and then said, "No, we're not going to make commitments to numbers that are pure fiction. If you don't like it, then don't invest. We'll go right across Sand Hill Road and take money from somebody else." That's exactly what we did.

Fierce Pragmatism, Unwavering Clarity, Boundless Energy

There's a fierce pragmatism to Mike, which I share and which permeates our projects. While we like to work on really sophisticated, complex, technical problems, we pair that with a very pragmatic approach to how we start and run these companies. For example, from the beginning of Vertica, we agreed that our system had to be at least 10× faster and 50-plus% cheaper than the alternatives. If at any point in the project we hadn't been able to deliver on that, we would have shut it down. As it turned out, the Vertica system was 100× faster, a credit to the engineering team at Vertica. (You guys rock.)

Mike also sees things in a very clear way. For example: It's relatively easy for companies to go out and build new technology, but that doesn't mean they should. When Mike famously criticized MapReduce back in the late 2000s [DeWitt and Stonebraker 2008, Stonebraker et al. 2010], it was controversial. But at that point, there was a whole generation of engineers rebuilding things that had already been worked out either academically or commercially: they just had to do a bit more research. It's frustrating to watch people make the same mistakes over and over again. Today, because of Mike, there are a whole bunch of people doing things commercially who are not making the same mistakes twice. I always tell young engineers starting out in databases to read the Big Red Book (*Readings in Database Systems*, 5th ed.)

Mike's energy—physical and intellectual—is boundless. He's always pushing me hard, and I'm still struggling to keep up even though I'm 20 years younger. This energy, clarity, and pragmatism infuses the principles of our businesses at every level. Here are some more examples.

Our #1 principle (see list) is "focus on great engineering." This doesn't mean hiring a seasoned engineering VP from a big company who hasn't written code in decades. It starts and ends with real engineers who understand why things work and how they should work in great systems. We like hiring engineering leaders who want to write code first and manage second, and we don't get hung up on titles or resumes. (If they are reluctant managers, that's a good sign; if they want to manage rather than write code, that's usually a bad sign.)

After successfully resisting BoD pressure at Vertica to "hire a real VP of engineering," we promoted the amazing Shilpa Lawande to run engineering. Shilpa had a more integrated view of what needed to be built at Vertica than anyone. Promoting Shilpa was a risk as it was her first leadership opportunity, but we knew she was going to kill it. She kept writing code for the core system, but eventually (when the engineering team got to critical mass), she stepped up to the plate—and killed it, as predicted, leading the entire engineering team at Vertica for over a decade. (As of August 2017, Shilpa had started her own new company, an AI healthcare startup.)

Another pivotal "Mike moment" in Vertica's engineering-driven strategy was when Mike brought in noted computer scientist David DeWitt. Dave provided invaluable advice in system design *and* got right down in the trenches with our engineering team.

Another core principle is thoughtful system architecture, and Mike is our ace in the hole on this one. By knowing to the bits-on-disks level how systems work and having seen design patterns (good and bad) over and over again for the past 40-plus years, Mike knows almost instinctively how a new system should work.

Case in point: A lot of Vertica competitors building high-performance database systems believed that they needed to build their own operating systems (or at least methods to handle I/O). Early on, we made a decision to go with the Linux file system. The academic founding team led by Mike believed that Linux had evolved to be "good enough" to serve as a foundation for next-generation systems.

On the surface, it seemed like a big risk: Now, we were not only running on commodity hardware (Netezza was building its own hardware), but also running on a commodity (open source) operating system. Mike knew better: an instinctive decision for him, but the right one and highly leveraged. Had we gone the other way, the outcome of Vertica would be very different.

Fortunately, Mike talked me down from positioning Vertica as an alternative high-performance/read-oriented storage engine for MySQL. The idea was that, following InnoDB's acquisition by Oracle,[6] the MySQL community was looking for an alternative high-performance, read-oriented storage engine that would sit underneath the rest of the MySQL stack for SQL parsing, optimization, and so on. Mike said it would never work: high-performance applications wouldn't work without control from the top of the stack (the declarative language) on down. (Right again, Mike.)

But Mike isn't always right and he's not afraid of being wrong. His batting average is pretty good, though. He was famously early in embracing distributed systems, basing his early, early work (1970s) on his belief. It's taken 40-plus years for the industry to come around to these kinds of things. He was never a fan of using high-performance memory architectures, and he was wrong on that. A hallmark of Mike—and a key to his success—is that he's never afraid to have strong opinions about things (often things he knows nothing about) just to encourage debate. Sometimes very fruitful debates.

Mike's a great example of the belief that smart people are smart all the time. Venture capitalists and others who have tended to pigeonhole him as "just an academic" vastly underestimated him. Like MIT's Robert Langer in biotech, one doesn't start this many companies and have this kind of success commercially and academically without being really, really smart on many levels.

[6] InnoDB, which built the storage component of MySQL, had been acquired by Oracle, causing a temporary crisis in the open-source community.

Figure 8.3 Mike Stonebraker and his bluegrass band, "Shared Nothing" (what else?), entertain at the Tamr Summer Outing on Lake Winnipesaukee in July 2015. From left are Mike, software engineer John "JR" Robinson (of Vertica), and Professor Stan Zdonik (of Brown University). Photo Credit: Janice Brown.

A Final Observation: Startups are Fundamentally about People

Companies come and go; good relationships can last forever. Partnership extends to the people who work "with" you—whether it's the graduate students or Ph.D.s who help build early research systems or the engineers who develop and deploy commercial systems.

As a Mike Stonebraker co-founder, I believe that I work for the people who work with me, giving them (1) better career development opportunities than they could find elsewhere, by aligning their professional interests with "whatever it takes" to make the startup successful, and (2) *a healthier, more productive and more fun work environment* [Palmer 2015a] than they can find elsewhere.

One of the reasons Mike has such a broad and diverse "academic family" is that he invests tremendous time, energy, and effort in developing young people, giving them opportunities both academically as well as commercially. This may be his biggest gift to the world.

9

Getting Grownups in the Room: A VC Perspective

Jo Tango[1]

My First Meeting

"Your website's directions are all wrong," Mike said. "It led me to take the wrong freeway exit."

In the Summer of 2002 I was looking into new developments in the database space, and Sybase co-founder Bob Epstein suggested that I look up Mike Stonebraker, who had just moved from California to the Boston area.

So, I found Mike's email and reached out, proposing a meeting. Our receptionist showed Mike to the conference room. I walked in and said, "Hello."

I saw a tall man with penetrating eyes. A wrinkled shirt and shorts completed the ensemble.

Mike's first words (above) made me think: "This is going to be a different meeting!"

It turned out to be the start of what has been a long working relationship across a number of companies: Goby (acquired by NAVTEQ), Paradigm4, StreamBase Systems (TIBCO), Vertica Systems (Hewlett-Packard), and VoltDB. Most importantly, Mike and I have become friends.

Context

Venture capital is an interesting business. You raise money from institutions such as college endowments and pension funds, and you then strive to find

[1] Acknowledged as "Believer" by Mike Stonebraker in his Turing lecture, "The land sharks are on the squawk box" [Stonebraker 2016].

entrepreneurs to back. Being an early-stage VC, I look for entrepreneurs in emerging technologies just forming companies that warrant seed capital.

There are two interesting facets to venture capital.

First, it is the ultimate Trust Game. Institutional investors commit to a fund that is ten years in duration. There are very few ways for them to get out of a commitment, and there is no Board of Directors, as venture firms are run as private partnerships. So, they will invest only if they trust you.

You, as the VC, in turn invest in entrepreneurs, many of whom are quirky, like Mike. You go to board meetings, you try to influence strategy, but for 99% of the time, you and the entrepreneur are not in the same room. A founder can golf every day or engage in nefarious behavior, and you're often the last to know. So, you only invest in someone whom you trust.

Mike is someone I trust. Yes, I believe in him.

Second, you're paid to peer into the future with all the risks and imperfection that that entails. In 2002, I spent a lot of time thinking about, "What's next?" Through my work, I had built relationships with the CIOs and CTOs at Goldman Sachs, Morgan Stanley, Fidelity, Putnam, and MMC (Marsh & McLennan Companies).

In numerous one-on-one conversations with these industry leaders, we talked about the pressing problems facing their companies and what they would have liked to see if they could "wave the magic wand." Based on such insights, I started some seed projects with their sponsorship, often after getting Mike's take.

I noticed that there was a great deal going on with storage subsystems and networking technologies. But, there wasn't much going on in the database layer. Oracle seemed to have a lock on the space.

Hence, the call to Bob Epstein. I'm grateful that Bob suggested that I contact Mike.

StreamBase

A few months after our first meeting, Mike emailed me in "characteristic Mike" fashion: "I have a new idea—wanna take a look?" This time, Mike was wearing slacks, which I found to be a good sign!

He had a hypothesis that the database world was going to splinter, that "one size doesn't fit all." He proposed raising a small round, a "seed" financing for a company called "Grassy Brook." I proposed that we find a better name, to which Mike responded with a loud and hearty laugh. I love that laugh, and I've heard it many times since.

After doing some due diligence, Grassy Brook's founding team moved into some free space downstairs from my office. An early engineering team assembled.

Mike and I, in a way, started to "live together." We started to bump into each other at the office fairly often. We learned to work together.

Early in the investment, Mike suggested that we and our spouses grab dinner up near Lake Winnipesaukee, where I was renting a summer place for two weeks and where Mike had decided to buy a house on Grassy Pond Road. It was great to meet Beth, his spouse.

A bit later, Mike had the whole founding team and his board up that summer. It was great to meet Mike's daughters, Leslie and Sandy. Leslie did the first company logo.

GRASSY
BROOK

Some months later, I found for Mike a business-oriented CEO through my personal network.

I do remember early on some "email flame wars." I was part of many group email discussions, and, sometimes, someone would say something with which Mike would disagree. Then, the terse emails, sometimes in all caps, would come.

At other times, Mike was suspicious of what I or the other VCs were saying or doing, and he would react strongly. But, over time, I found that Mike was very consistent. He wanted to hear the truth, pure and simple. He wanted to test people's motivations and whether they were really saying what they meant and whether they would do what they said.

I was very comfortable with this. Mike was a lot like my father. He was a Truth Teller. And, he certainly expected others to do the same.

Now, don't get me wrong. Being at the end of a "Mike Firehose Email" dousing can be unpleasant. But, if you felt you were right and presented data, he would be the first to change his thinking.

"Color my face red," one email from Mike started, when he realized that he was incorrect in one of his assumptions and assertions.

Yes, a Truth Teller, one even willing to speak truth to himself and to have truth spoken to him. It was then that I respected Mike even more. It made me want to work even harder on the company, which was renamed StreamBase Systems.

streambase

A Playbook Is Set

By the time Mike started to think up Vertica, I felt he and I were in a groove. He would keep me informed of his ideas, and I would give him feedback. If I met good people, I introduced them to Mike.

VERTICA

You see, an early-stage VC is very much like an executive recruiter in a company's life. It is an extremely rewarding part of the job to connect together good people and watch a match happen. Key founding executives, including CEOs, joined StreamBase, Vertica, Goby, Paradigm4, and VoltDB in this way. Those are the Stonebraker companies with which it has been a pleasure to be involved as a VC.

When like-minded people with shared values partner together, such as Mike and Andy Palmer, good things tend to happen.[2]

So, our playbook is:

- Mike has good ideas.
- I invest in the ones that make me excited and make personal introductions to executives and potential customers.
- He hosts a kickoff party at his summer house.
- We work hard together, as partners, to do what is right for the company.

[2] See Mike and Andy's views in the other chapters in this section, Chapter 7 (by Michael Stonebraker) and Chapter 8 (by Andy Palmer).

This is the embryonic version of what Mike refined over nine startups to become what he described in a previous chapter as “five (not so) easy steps” (see Chapter 7.

I’m happy to state that over time, the email flame wars have become largely nonexistent. When we have conflict, Mike and I pick up the phone and talk it out. But, it is a lot easier now, in this Trust Game, for I believe that each of us has earned the other’s trust.

Mike’s Values

Before and after board meetings and calls, and over some dinners, I talked about personal things with Mike. I learned that he was from a low-income background, but that his life changed when Princeton gave him a spot.

I learned that he and Beth were very serious about supporting good causes, and they intentionally lived quite below their means. They wanted to be careful with how their children perceived money’s role in life.

I still chuckle when Mike in the summer wears shorts to board meetings. “Mike is Mike,” I often say to others. He is one-of-a-kind—and, in a very good way.

Through many interactions, Mike affected my style, too. He appreciates people being blunt and transparent, and so, in our interactions, I no longer filter. I just tell him what I think and why. It can be a bit intimidating to go toe-to-toe with Mike on difficult issues, but I have found that he is eminently reasonable, and, if you’re armed with data, he is very open minded.

So, like a pair of old friends, there is a give and take and mutual respect. We really do work well together, and there’s much trust there.

Why does one win the Turing Award? Honestly, I do not know. But, I feel I do know that Mike has unique abilities. He is comfortable with taking risks, working with a diverse group of people, and striving for something bold and interesting. He is a true entrepreneur.

A Coda

Sam Madden[3] called me up in 2014. He was throwing Mike a Festschrift—a 70th birthday celebration for a professor—at MIT. I rushed to be a personal sponsor, as did Andy Palmer, Intel, and Microsoft.

Sam did his usual job of leading the program, gathering a lineup of speakers and even handing out T-shirts in bright red.

[3] Read Sam Madden’s biography of Michael Stonebraker (Chapter 1) and his introduction to Mike’s research contributions by system (Chapter 14.)

A Celebration of Mike Stonebraker
Saturday April 12th, 2014
@ MIT Stata Center

It was a joyous event, with some great talks, many smiles, and a personal realization that I was in the midst of so many great people with large brains and sound values.

In particular, it was great to see Beth, as well as Leslie and Sandy, who had been young girls when I had met them the first time many years ago.

A Great Day

One time after a board meeting for Vertica, we somehow got to talking about the Turing Award. Being a liberal arts major, I asked: "So, what is it?'

"It is the greatest honor," Mike said. "I've been up for it in the past, from what I hear, but it's not something I like to think about. It is a crowning achievement for someone in computer science." He suddenly became quiet and looked down at the table.

I didn't dare ask any more questions.

Years later, when the world heard that Mike won the Turing Award, I was elated. I wasn't able to attend the ceremony, but I read his speech online. At the end, he referred to two people, "Cue Ball" and "The Believer." People told me that he was referring to Andy Palmer and me.

What an honor . . . .

> That was a very kind and greatly appreciated gesture, Mike. You've been a great business partner and friend. You are a great role model for me as a parent. I love you and am so happy for you!

PART VI

DATABASE SYSTEMS RESEARCH

10

Where Good Ideas Come From and How to Exploit Them

Michael Stonebraker

Introduction

I often get the question: “Where do good ideas come from?” The simple answer is “I don’t know.”

In my case, they certainly don’t come from lying on the beach or communing with nature on a mountaintop. As near as I can tell, there are two catalysts. The first is hanging around institutions that have a lot of smart, combative people. They have ideas that they want you to critique, and they are available to critique your ideas. Out of this back-and-forth, good ideas sometimes emerge. The second catalyst is talking to a lot of real-world users of DBMS technology. They are happy to tell you what they like and don’t like and the problems that they are losing sleep over. Out of talking to a lot of users often come problems to work on. From such problems, sometimes good ideas emerge.

However, I think these are often secondary effects. I will spend this chapter going through my career indicating where my ideas (both good and bad) came from. In a lot of cases, it was pure serendipity.

The Birth of Ingres

I arrived at Berkeley in 1971 as a new assistant professor. I knew my thesis was a bad idea or at best irrelevant. Berkeley hired me because I agreed to work on something called “urban systems.” This was around the time that the National Science Foundation (NSF) started a program called Research Applied to the National

Needs (RANN), and Rand Corporation was making headlines applying Operations Research (OR) ideas to locating firehouses and allotting police officers. For a while I tried to work in this area: I studied the Berkeley Municipal Court system and then built a land-use model for Marin County, California. I learned quickly how difficult these studies were and how bogged down they became because of bad data.

At about this time Gene Wong suggested we take a look at databases. In short order, we decided the CODASYL proposal was incomprehensible, and IMS was far too restrictive. Ted Codd's papers, of course, made perfect sense to us, and it was a no-brainer to start an implementation. We were not dissuaded by the fact that neither of us had ever written any software before nor managed a complex project. Several other research groups embarked on similar projects around the same time. Most (including us) got enough running so they could write a paper [Stonebraker et al. 1976b]. For totally unknown reasons, we persevered and got Ingres to work reasonably well, and it became widely used in the mid-1970s as the only RDBMS that researchers could get their hands on.[1] In effect, Ingres made an impact mostly because we persevered and got a real system to work. I view this decision as pure serendipity.

Abstract Data Types (ADTs)

The main internal user of the Ingres prototype was an urban economics group led by Pravin Varaiya, which were interested in Geographic Information Systems (GIS). Angela Go implemented a GIS system on top of Ingres, but it didn't work very well. Varaiya wanted polygon maps and operations like point-in-polygon and polygon-intersects-polygon. These are horrific to code in languages like QUEL and SQL and execute with dismal performance.

By this time, we understood how integers, floats, and strings worked in Ingres; however, simulating points, lines, and polygons on the Ingres-type system was very painful. It seemed natural at the time to ask, "Why not extend the built-in types in Ingres?" We built such a system (OngFS84) into Ingres in 1982–1983, and it seemed to work very well. Therefore, we used the same idea in Postgres. In my opinion, this was the major innovation (a good idea) in Postgres, to which we turn next.[2]

[1] See Ingres' impact in Chapter 13

[2] The use of Abstract Data Types, considered one of my most important contributions, is discussed in Chapters 3, 12, 15, and 16

Postgres[3]

Visitors to Berkeley always asked, "What is the biggest database that uses Ingres?" We always had to mumble, "Not very big at all." The reason can be illustrated by an example. In 1978, Arizona State University seriously considered running Ingres for its student records system, which covered 40,000 students. The project team could get around the fact that they had to run an unsupported operating system (Unix) and an unsupported DBMS (Ingres), but the project faltered when ASU found that there was no COBOL available for Unix (they were a COBOL shop). For these reasons, essentially anybody serious would not consider Ingres, and it was relegated to modest applications. At about the same time, Larry Ellison started claiming that Oracle was ten times faster than Ingres, a strange claim given that Oracle didn't even work yet.

It became obvious that, to make a difference, we had to move Ingres to a supported operating system, offer support, improve the documentation, implement a report writer, and so on. In short, we had to start a commercial company. I had no idea how to do this, so I went to talk to Jon Nakerud, then the Western Region sales manager for Cullinet Corp., which marketed IDMS (a CODASYL system). With Jon's help as CEO, we raised venture capital and started what turned into Ingres Corp.

This was a "trial by fire" on how to start a company, and I certainly learned a lot. Very quickly, the commercial version of Ingres became far better than the academic version. Although we implemented abstract data types (ADTs) in the academic version, the handwriting was on the wall: it made no sense to continue prototyping on the academic version. It was time to start a new DBMS codeline, and Postgres was born.

In my opinion, Postgres [Stonebraker and Rowe 1986] had one very good idea (ADTs) and a bunch of forgettable ideas (inheritance, rules using an "always" command, and initial implementation in Lisp, to name a few). Commercialization (as Illustra Corporation) fixed a lot of the problems; however, ADTs were somewhat ahead of their time, and Illustra struggled to get real-world users to adopt them. As such, Illustra was sold to Informix in 1996.

The bright legacy of Postgres was purely serendipitous. Two Berkeley grad students, Wei Hong and Jolly Chen, converted the academic version of Postgres in 1995 from QUEL to SQL. Then a dedicated pickup team of volunteers, with no relationship to me or Berkeley, shepherded the codeline over time. That is the

[3] See Postgres' impact in Chapter 13.

open source version of Postgres that you can download today off the Web from https://www.postgresql.org.

Distributed Ingres, Ingres*, Cohera, and Morpheus

My love affair with distributed databases occurred over 25 years (from the mid-1980s to the late 2000s). It started with Distributed Ingres, which federated the academic Ingres codeline. This system assumed that the schemas at multiple locations were identical and that the data was perfectly clean and could be pasted together. The code sort of worked, and the main outcome was to convince me that the commercial Ingres codeline could be federated in the same way. This project turned into Ingres* in the mid-1980s. There were essentially zero users for either system.

Undaunted, we built another distributed database prototype, Mariposa, in the early 1990s, which based query execution on an economic model. In effect, Mariposa still assumed that the schemas were identical and the data was clean, but relaxed the Ingres* assumption that the multiple sites were in the same administrative domain. There was little interest in Mariposa, but a couple of the Mariposa students really wanted to start a company. Against my better judgment, Cohera was born and it proved yet again that there was no market for distributed databases.

Still undaunted, we built another prototype, Morpheus. By talking to real-world users, we realized that the schemas were never the same. Hence, Morpheus focused on translating one schema into another. However, we retained the distributed database model of performing the translation on the fly. Again, we started a company, Goby, which focused on integration of Web data, using essentially none of the Morpheus ideas. Goby was in the business-to-consumer (B2C) space, in other words, our customer was a consumer. In B2C, one has to attract "eyeballs" and success depends on word-of-mouth and buying Google keywords. Again, Goby was not a great success. However, it finally made me realize that federating databases is not a big concern to enterprises; rather, it's performing data integration on independently constructed "silos" of data. Ultimately, this led to a prototype, Data Tamer [Stonebraker et al. 2013b], that actual users wanted to try.

In summary, I spent a lot of time on distributed/federated databases without realizing that there is no market for this class of products. Whether one will develop in the future remains to be seen. Not only did this consume a lot of cycles with nothing to show for it except a collection of academic papers, but it also made me totally miss a major multi-node market, which we turn to next.

Parallel Databases

I wrote a paper in 1979 proposing Muffin [Stonebraker 1979a], a shared-nothing parallel database. However, I did not pursue the idea further. A couple of years later, Gary Kelley, then an engineer at Sequent Computers, approached Ingres and suggested working together on a parallel database system. Ingres, which was working on Ingres* at the time, did not have the resources to pursue the project more than half-heartedly. Gary then went to Informix, where he built a very good parallel database system. All in all, I completely missed the important version of distributed databases where tightly coupled nodes have the same schema—namely parallel partitioned databases. This architecture enables much higher SQL performance, especially in the data warehouse marketplace.

Data Warehouses

Teradata pioneered commercial parallel database systems in the late 1980s with roughly the same architecture as the Gamma prototype built by Dave DeWitt [DeWitt et al. 1990] In both cases, the idea was to add parallelism to the dominant single-node technology of the time, namely row stores. In the late 1990s and early 2000s, Martin Kersten proposed using a column store and started building MonetDB [Boncz et al. 2008]. I realized the importance of column stores in the data warehouse market through a consulting gig around 2002. When the gig ended I started thinking seriously about C-Store [Stonebraker et al. 2005a], which turned into Vertica.[4] This codeline supported parallel databases, with an LSM-style (Log Structure Merge) storage infrastructure, a main memory row store to assemble tuples to be loaded, a sort engine to turn the row store into compressed columns, and a collection of so-called projections to implement indexes. Most of my other startups rewrote everything to fix the stuff that I got wrong the first time around. However, C-Store pretty much got it right, and I feel very proud of the Vertica codeline. To this day, it is nearly unbeatable in bakeoffs.

In summary, it was serendipitous to get the consulting gig, which got me to understand the true nature of the data warehouse performance problem. Without that, it is doubtful I would have ever worked in this area.

H-Store/VoltDB

David Skok, a VC with Matrix Partners, suggested one day that it would be great to work on a new OLTP architecture, different from the disk-based row stores of

[4] For the Vertica story, see Chapter 18.

the time. C-Store had convinced me that "one size does not fit all" [Stonebraker and Çetintemel 2005], so I was open to the general idea. Also, it was clear that the database buffer pool chews up a lot of cycles in a typical DBMS. When Dave DeWitt visited MIT, we instrumented his prototype, Shore [Carey et al. 1994], to see exactly where all the cycles went. This generated the "OLTP: Through the Looking Glass" paper [Harizopoulos et al. 2008]; all of us were shocked that multi-threading, the buffer pool, concurrency control, and the log consumed an overwhelming fraction of the CPU cycles in OLTP. This, coupled with the increasing size of main memory, led to H-Store and then to a company, VoltDB.[5]

The offhand remark from David Skok certainly stuck in my memory and caused me to look seriously a year or two later. Certainly, there was serendipity involved.

Data Tamer

Joey Hellerstein decided to visit Harvard in 2010–2011, and we agreed to brainstorm about possible research. This quickly evolved into a data integration project called Data Tamer. In a previous company, Goby, we had been trying to integrate the contents of 80,000 websites and had struggled with a custom code solution. Goby was willing to supply its raw data, that is, their crawl of the 80,000 sites. We decided to work on their data, and the project quickly evolved to trying to do scalable data integration. Goby data needed schema integration, data cleaning, and entity consolidation, which we started addressing. Talks on our project brought us two additional enterprise clients, Verisk Health and Novartis. Both were focused primarily on entity consolidation.

At about this time, MIT was setting up a relationship with the Qatar Computing Research Institute (QCRI), and Data Tamer became an early project in this collaboration. In effect, Data Tamer was focused on solving the data integration presented by Goby, Verisk Health, and Novartis. In a way, I think this is an ideal startup template: find some clients with a problem and then try to solve it.[6]

Again, there was much serendipity involved. Data Tamer would not have happened without Joey visiting Harvard and without Goby being willing to provide its data.

[5] See Chapter 19 for the VoltDB story.

[6] See Chapter 7 (Stonebraker) for a detailed description.

How to Exploit Ideas

In every case, we built a prototype to demonstrate the idea. In the early days (Ingres/Postgres), these were full-function systems; in later days (C-Store/H-Store), the prototypes cut a lot of corners. In the early days, grad students were happy to write a lot of code; these days, big implementations are dangerous to the publication health of grad students. In other words, the publication requirements of getting a good job are contrary to getting full-function prototypes to work!

In both cases, happy grad students are a requirement for success. I have only two points to make in this direction. First, I view it as my job to make sure that a grad student is successful, in other words, gets a good position following grad school. Hence, I view it as my job to spend as much time as necessary helping students be successful. To me, this means feeding good ideas to students to work on until they can get the hang of generating their own. In contrast, some professors believe in letting their students flounder until they get the hang of research. In addition, I believe in treating students fairly, spending as much time with them as necessary, teaching them how to write technical papers, and, in general, doing whatever it takes to make them successful. In general, this philosophy has produced energetic, successful, and happy students who have gone on to do great things.

Closing Observations

Good ideas are invariably simple; it is possible to explain the idea to a fellow researcher in a few sentences. In other words, good ideas seem to always have a simple "elevator pitch." It is wise to keep in mind the KISS adage: "Keep it Simple, Stupid." The landscape is littered with unbuildable ideas. In addition, never try to "boil the ocean." Make sure your prototypes are ultra-focused. Lastly, good ideas come whenever they come. Don't despair if you don't have a good idea today. This adage is especially true for me: At 74 years old, I keep worrying that I have "lost it." However, I still seem to get good ideas from time to time . . .

11

Where We Have Failed

Michael Stonebraker

In this chapter, I suggest three areas where we have failed as a research community. In each case, I indicate the various ramifications of these failures, which are substantial, and propose mitigations. In all, these failures make me concerned about the future of our field.

The Three Failures

Failure #1: We Have Failed to Cope with an Expanding Field

Our community spearheaded the elevation of data sublanguages from the record-at-a-time languages of the IMS and CODASYL days (1970s) to the set-at-a-time relational languages of today. This change, which occurred mostly in the 1980s along with the near-universal adoption of the relational model, allowed our community to investigate query optimizers, execution engines, integrity constraints, security, views, parallelism, multi-mode capabilities, and the myriad of other capabilities that are features of modern DBMSs.

SQL was adopted as the de facto DBMS interface nearly 35 years ago with the introduction of DB2 in 1984. At the time, the scope of the DBMS field was essentially business data processing. Pictorially, the state of affairs circa 1985 is indicated in Figure 11.1, with the expansion of scope in the 1980s indicated. The net result was a research community focused on a common set of problems for a business data processing customer. In effect, our field was unified in its search for better data management for business data processing customers.

Since then, to our great benefit, the scope of DBMSs has expanded dramatically as nearly everyone has realized they need data management capabilities. Figure 11.2 indicates our universe 30 years later. Although there is some activity in business data processing (data warehouses, OLTP), a lot of the focus has shifted to a variety

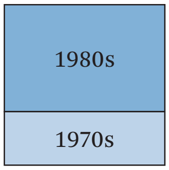


Figure 11.1 Our universe circa 1985: business data processing.

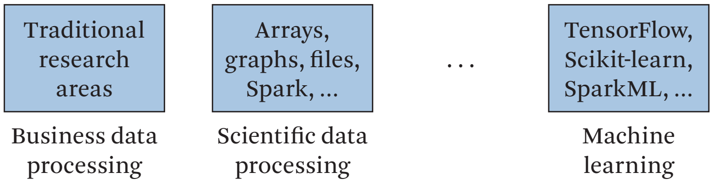

Traditional research areas
Arrays, graphs, files, Spark, ...
. . .
TensorFlow, Scikit-learn, SparkML, ...
Business data processing
Scientific data processing
Machine learning

Figure 11.2 Our universe now.

of other application areas. Figure 11.2 lists two of them: machine learning and scientific databases. In these areas, the relational data model is not popular, and SQL is considered irrelevant. The figure lists some of the tools researchers are focused on. Note clearly that the important topics and tools in the various areas are quite different. In addition, there is minimal intersection of these research thrusts.

As a result of the expanding scope of our field, the cohesiveness of the 1980s is gone, replaced by sub-communities focused on very different problems. In effect, our field is now composed of a collection of subgroups, which investigate separate topics and optimize application-specific features. Connecting these domain-specific features to persistent storage is done with domain-specific solutions. In effect, we have decomposed into a collection of N domain-specific groups with little interaction between them.

One might hope that these separate domains might be unified through some higher-level encompassing query notation. In other words, we might hope for a higher level of abstraction that would reunify the field. In business data processing, there have been efforts to develop higher-level notations, whether logic-based (Prolog, Datalog) or programming-based (Ruby on Rails, LINQ). None have caught on in the marketplace.

Unless we can find ways to generate a higher-level interface (which I consider unlikely at this point), we will effectively remain a loose coalition of groups with little research focus in common.

This state of affairs can be described as “the hollow middle.” I did a quick survey of the percentage of ACM SIGMOD papers that deal with our field as it was defined in 1977 (storage structures, query processing, security, integrity, query languages, data transformation, data integration, and database design). Today, I would call this the *core* of our field. Here is the result:

1977	100%	(21/21)
1987	93%	(40/43)
1998	68%	(30/44)
2008	52%	(42/80)
2017	47%	(42/90)

I include only research papers, and not papers from the industrial or demo tracks, when these tracks came into existence. Notice that the core is being “hollowed out,” as our researchers drift into working on what would have been called applications 40 years ago. In my opinion, the reason for this shift is that the historical uses of DBMS technology (OLTP, business data warehouses) are fairly mature. As a result, researchers have largely moved on to other challenges.

However, the fact remains that there is little or no commonality across the various applications areas. What is important in Natural Language Processing (NLP) is totally different from what is important in machine learning (ML) or scientific data processing. The net effect is that we have essentially “multi-furcated” into subgroups that don’t communicate with each other. This is reminiscent of the 1982 bifurcation that occurred when ACM SIGMOD and ACM PODS split.

My chief complaint is that we have failed to realize this, and our conference structures (mostly minor tweaks on 30-year-old ideas) are not particularly appropriate for the current times. In my opinion, the best solution is to recognize the hollow middle, and decompose the major DBMS conferences (SIGMOD, VLDB, ICDE) into multiple (say five or six) separate tracks with independent program committees. These can be co-located or organized separately. In other words, multi-furcate along the lines of the SIGMOD/PODS division many years ago.

If this does not happen, then the current conferences will remain “zoos” where it is difficult to impossible to find like-minded researchers. Also, reviewing will be chaotic (as discussed below), which frustrates researchers. The “systems folks” seem particularly upset by the current state of affairs. They are on the verge of declaring a divorce and starting their own conference. Other subgroups may follow. The result will be a collapse of the field as we know it.

You might ask, "Where is there room for cross-cultural research?" I have not seen people from other fields at SIGMOD conferences in quite a while. Equally, program committees do not have such multi-culturalism. The obvious answer is to have more cross-cultural conferences. In olden times, we used to have such things, but they have fallen out of favor in recent years.

Failure #2: We Have Forgotten Who Our Customer Is

Forty years ago, there was a cadre of "industry types" who came to our conferences. They were early users (evangelists) of DBMS technology and came from financial services, insurance, petroleum exploration, and so on. As such, they provided a handy reality check on whether any given idea had relevance in the real world. In effect, they were surrogates for our "customer." Hence, our mission was to provide better DBMS support for the broad customer base of DBMS technology, represented by the evangelists.

Over the years, these evangelists have largely disappeared from our conferences. As such, there is no customer-facing influence for our field. Instead, there are representatives of the large Internet vendors—I call them "the whales"—who have their own priorities and represent the largest 0.01% of DBMS users. Hence, they hardly represent the real world. In effect, our customer has vanished and been replaced by either the whales or a vacuum.

This loss of our customer has resulted in a collection of bad effects. First, there are no "real world" clients to keep us focused. As such, we are prone to think the next "silver marketing bullet" from the whales is actually a good idea. Our community has embraced and then rejected (when it became apparent that the idea was terrible) OLEDB, MapReduce, the Semantic WEB, Object Databases, XML, and data lakes, just to name a few.

We are very uncritical of systems created by the large Internet vendors to that solve application-specific problems, which have been written, until recently, by development teams with little background in DBMSs. As such, they have tended to reinvent the wheel. I am especially amused by Google's adoption and then rejection of MapReduce and eventual consistency.

Our community needs to become more assertive at pointing out flawed ideas and "reinventions of the wheel." Otherwise, the mantra "people who do not understand history will be condemned to repeat it" will continue to be true.

We desperately need to reconnect with the "real world." This could be done by giving free conference registrations to real-world users, organizing panels of real users, inviting short problem commentaries from real users, etc. I am also amused at the number of attendees at our conferences who have no practical experience in applying DBMS technology to real problems. Our field exists to serve a customer. If the customer is "us," then we have totally lost our way.

Failure #3: We Have Not Addressed the Paper Deluge

When I graduated from Michigan in 1971 with a Ph.D., my resume consisted of zero papers. Five years later, I was granted tenure at Berkeley with a resume of half a dozen papers. Others from my age cohort (e.g., Dave DeWitt) report similar numbers. Today, to get a decent job with a fresh Ph.D., one needs around 10 papers; to get tenure more like 40 is the goal. It is becoming common to take a two-year postdoc to build up one's resume before hitting the academic job market. Another common tactic these days is to accept an academic job and then delay starting the tenure clock by taking a postdoc for a year. This was done recently, for example, by Peter Bailis (now at Stanford) and Leilani Battle (now at Maryland). The objective in both situations is to get a head start on the paper deluge required for tenure.

Put differently, there has been an order of magnitude escalation in desired paper production. Add to this fact that there are (say) an order of magnitude more DBMS researchers today than 40 years ago, and we get paper output rising by two orders of magnitude. There is no possible way to cope with this deluge. There are several implications of this escalation.

First, the only way that I would ever read a paper is if somebody else said it was very good or if it was written by one of a handful of researchers that I routinely follow. As a result, we are becoming a "word of mouth" distribution system. That makes it nearly impossible for somebody from the hinterlands to get well known. In other words, you either work with somebody from the "in crowd" or you are in "Outer Siberia." This makes for an un-level tenure-track playing field.

Second, everybody divides their ideas into Least Publishable Units (LPUs) to generate the kind of volume that a tenure case requires. Generally, this means there is no seminal paper on a particular idea, just a bunch of LPUs. This makes it difficult to follow the important ideas in the field. It also ups the number of papers researchers must read, which makes us all grumpy.

Third, few graduate students are willing to undertake significant implementation projects. If you have to write ten papers in (say) four productive years, that is a paper every five months. You cannot afford the time for significant implementations. This results in graduate students being focused on "quickies" and tilts paper production toward theory papers. More on this later.

So how did this paper explosion occur? It is driven by a collection of universities, mostly in the Far East, whose deans and department chairpeople are too lazy to actually evaluate the contribution of a particular researcher. Instead they just count papers as a surrogate. This laziness also appears to exist at some second- and third-rate U.S. and European universities.

Our failure to deal with the deluge will allow this phenomenon to get worse off into the future. Hence, we should actively put a stop to it, and here is a simple idea. It would be fairly straightforward to get the department chairpeople of (say) the 25 best U.S. universities to adopt the following principle:

> Any DBMS applicant for an Assistant Professor position would be required to submit a resume with at most three papers on it. Anybody coming up for tenure could submit a resume with at most ten papers. If an applicant submitted a longer resume, it would be sent back to the applicant for pruning. Within a few years, this would radically change publication patterns. Who knows, it might even spread to other disciplines in computer science.

Consequences of Our Three Failures

Consequence #1: Reviewing Stinks

A consequence of the paper deluge and the "hollow middle" is the quality of reviewing, which, in general, stinks. In my experience, about half of the comments from reviewers are way off the mark. Moreover, the variance of reviews is very high. Of course, the problem is that a program committee has about 200 members, so it is a hit-or-miss affair. The biases and predispositions of the various members just increase the variance. Given the "hollow middle," the chances of getting three reviewers who are experts in the application domain of any given paper is low, thereby augmenting the variance. Add to this the paper deluge and you get very high volume and low reviewing quality.

So, what happens? The average paper is rewritten and resubmitted multiple times. Eventually, it generally gets accepted somewhere. After all, researchers have to publish or perish!

In ancient times, the program chairperson read all the papers and exerted at least some uniformity on the reviewing process. Moreover, there were face-to-face program committee meetings where differences were hashed out in front of a collection of peers. This is long gone—overrun by the size (some 800 papers) of the reviewing problem. In my opinion, the olden times strategy produced far better results.

The obvious helpful step would be to decompose the major DBMS conferences into subconferences as noted in Failure #1. Such subconferences would have (say) 75 papers. This would allow "old school" reviewing and program committees. This subdivision could be adopted easily by putting the current "area chairperson" concept on steroids. These subconferences could be co-located or not; there are pros and cons to both possibilities.

Another step would be to dramatically change the way paper selection is done. For example, we could simply accept all papers, and make reviews public, along with reviewers' scores. A researcher could then put on his or her resume his or her paper and the composite score he or she received. Papers would get exposure at a conference (long slot, short slot, poster) based on the scores. However, the best solution would be to solve Failure #3 (the paper deluge).

If the status quo persists, variance will just increase, resulting in more and more overhead for poorer and poorer results.

Consequence #2: We Have Lost our Research Taste

Faced with required paper production, our field has drifted into solving artificial problems, and especially into making 10% improvements on previous work (Least Publishable Units). The number of papers at major DBMS conferences that seem completely irrelevant to anything real seems to be increasing over time. Of course, the argument is that it is impossible to decide whether a paper will have impact at some later point in time. However, the number of papers that make a 10% improvement over previous work seems very large. A complex algorithm that makes a 10% improvement over an existing, simpler one is just not worth doing. Authors of such papers are just exhibiting poor engineering taste. I have generally felt that we were polishing a round ball for about the last decade. I would posit the following question: "What was the last paper that made a dramatic contribution to our field?" If you said a paper written in the last ten years, I would like to know what it is.

A possible strategy would be to require every assistant professor to spend a year in industry, pre-tenure. Nothing generates a reality check better than some time spent in the real world. Of course, implementing this tactic would first require a solution to Failure #3. In the current paper climate, it is foolhardy to spend a year not grinding out papers.

Consequence #3: Irrelevant Theory Is Taking Over

Given that our customer has vanished and given the required paper production, the obvious strategy is to grind out "quickies." The obvious way to optimize quickies is to include a lot of theory, whether relevant to the problem at hand or not. This has two major benefits. First, it makes for quicker papers, and therefore more volume. Second, it is difficult to get a major conference paper accepted without theorems, lemmas, and proofs. Hence, this optimizes acceptance probability.

This focus on theory, relevant or not, effectively guarantees that no big ideas will ever be presented at our conferences. It also guarantees that no ideas will ever be accepted until they have been polished to be pretty round.

My personal experience is that experimental papers are difficult to get by major conference reviewers, mostly because they have no theory. Once we move to polishing a round ball, then the quality of papers is not measured by the quality of the ideas, but by the quality of the theory. To put a moniker on this effect, I call this "excessive formalism," which is lemmas and proofs that do nothing to enhance a paper except to give it theoretical standing. Such "irrelevant theory" essentially guarantees that conference papers will diverge from reality. Effectively, we are moving to papers whose justification has little to nothing to do with solving a real-world problem. Because of this attitude, our community has moved from serving a customer (some real-world person with a problem) to serving ourselves (with interesting math). Of course, this is tied to Failure #3: Getting tenure is optimized by "quickies." I have nothing against theoretical papers, just a problem with irrelevant theory.

Of course, our researchers will assert that it is too difficult to get access to real-world problems. In effect, the community has been rendered sterile by the refusal of real enterprises to partner with us in a deep way. The likes of Google, Amazon, or Microsoft also refuse to share data with our community. In addition, my efforts to obtain data on software faults from a large investment bank were stymied because the bank did not want to make its DBMS vendor look bad, given the frequency of crashes. I have also been refused access to software evolution code by several large organizations, which apparently have decided that their coding techniques, their code, or both were proprietary (or perhaps may not stand up to scrutiny).

As a result, we deal primarily with artificial benchmarks (such as YCSB[1]) or benchmarks far outside the mainstream (such as Wikipedia). I am particularly reminded of a thread of research that artificially introduced faults into a dataset and then proved that the algorithms being presented could find the faults they injected. In my opinion, this proves absolutely nothing about the real world.

Until (and unless) the community finds a way to solve Failure #2 and to engage real enterprises in order to get real data on real problems, then we will live in the current theory warp. The wall between real enterprises and the research community will have to come down!

[1] Yahoo Cloud Serving Benchmark (https://research.yahoo.com/news/yahoo-cloud-serving-benchmark/). Last accessed March 2, 2018.

Consequence #4: We Are Ignoring the Hardest Problems

A big problem facing most enterprises is the integration of disparate data sources (data silos). Every large enterprise divides into semi-independent business units so business agility is enabled. However, this creates independently constructed "data silos." It is clearly recognized that data silo integration is hugely valuable, for cross-selling, social networking, single view of a customer, etc. But data silo integration is the Achilles' heel of data management, and there is ample evidence of this fact. Data scientists routinely say that they spend at least 80% of their time on data integration, leaving at most 20% for the tasks for which they were hired. Many enterprises report data integration (data curation) is their most difficult problem.

So, what is our community doing? There was some work on data integration in the 1980s as well as work on federated databases over the last 30 years. However, federating datasets is of no value unless they can be cleaned, transformed, and deduplicated. In my opinion, insufficient effort has been directed at this problem or at data cleaning, which is equally difficult.

How can we claim to have the research mandate of management of data if we are ignoring the most important management problem? We have become a community that looks for problems with a clean theoretical foundation that beget mathematical solutions, not one that tries to solve important real-world problems. Obviously, this attitude will drive us toward long-term irrelevance.

Of course, this is an obvious result of the necessity of publishing mountains of papers, in other words, don't work on anything hard, whose outcome is not guaranteed to produce a paper. It is equally depressing that getting tenure does not stop this paper grind, because your students still need to churn out the required number of papers to get a job. I would advise everybody to take a sabbatical year in industry and delve into data quality or data integration issues. Of course, this is a hollow suggestion, given the current publication requirements on faculty.

Data integration is not the only incredibly important issue facing our customers. Evolution of schemas as business conditions change (database design) is horribly broken, and real customers don't follow our traditional wisdom. It is also widely reported that new DBMS products require some $20M in capital to get to production readiness, as they did 20 years ago. For a mature discipline this is appalling. Database applications still require a user to understand way too much about DBMS internals to effectively perform optimization.

In other words, there is no shortage of very important stuff to work on. However, it often does not make for good theory or quickies and often requires massaging a lot of ugly data that is hard to come by. As a community, we need to reset our priorities!

Summary

I look out at our field with its hollow middle and increasing emphasis on applications with little commonality to each other. Restructuring our publication system seems desperately needed. In addition, there is increasing pressure to be risk averse and theoretical, so as to grind out the required number of publications. This is an environment of incrementalism, not one that will enable breakthrough research. In my opinion, we are headed in a bad direction.

Most of my fears are rectifiable, given enlightened leadership by the elders of our community. The paper deluge is addressable in a variety of ways, some of which were noted above. The hollow middle is best addressed in my opinion by multifurcating our conferences. Recruiting the real world, first and foremost, demands demonstrating that we're relevant to their needs (working on the right problems). Secondarily, it is a recruitment problem, which is easily addressed with some elbow grease.

This chapter is a plea for action, and quickly! If there is no action, I strongly suspect the systems folks will secede, which will not be good for the unity of the field. In my opinion, the "five-year assessment of our field," which is scheduled for the Fall of 2018 and organized by Magda Balazinska and Surajit Chaudhuri, should focus primarily on the issues in this chapter.

12

Stonebraker and Open Source

Mike Olson

The Origins of the BSD License

In 1977, Professor Bob Fabry at Berkeley began working with a graduate student, Bill Joy, on operating systems. Fabry and his colleagues created the Computer Systems Research Group (CSRG) to explore ideas like virtual memory and networking. Researchers at Bell Labs had created an operating system called UNIX™ that could serve as a good platform for testing out their ideas. The source code for UNIX was proprietary to the Labs' parent, AT&T, which carefully controlled access.

Because it wanted to enhance UNIX, CSRG depended on the Bell Labs source code. The group wanted to share its work with collaborators, so would somehow have to publish any new code it created. Anyone who got a copy of the Berkeley software needed to purchase a source code license from AT&T as well.

CSRG worked with the university's intellectual property licensing office to craft the "Berkeley Software Distribution (BSD)" license. This license allowed anyone to receive, modify, and further share the code they got from CSRG, encouraging collaboration. It placed no additional restrictions on the AT&T source code—that could continue to be covered by the terms of the AT&T source code license.

The BSD license was a really nifty hack. It protected the interests of AT&T, maintaining the good relationship Berkeley had with Bell Labs. It allowed the Berkeley researchers to share their innovative work broadly, and to take back contributions from others. And, significantly, it gave everyone a way to work together to build on the ideas in UNIX, making it a much better system.

BSD and Ingres

In 1976, before CSRG began working on UNIX, Mike Stonebraker had launched a research project with Eugene Wong (and, later, Larry Rowe) to test out some ideas published by Ted Codd [Codd 1970] and Chris Date. Codd and Date developed a "relational model," a way to think about database systems that separated the physical layout and organization of data on computers from operations on them. You could, they argued, describe your data, and then say what you wanted to do with it. Computers could sort out all the plumbing, saving people a lot of trouble and time.

The new project was called Ingres, short for INteractive Graphics REtrieval System.

Mike and his collaborators began to share their code with other institutions before CSRG finished its work on the BSD license. The earliest copies were shipped on magnetic tape with essentially no oversight by the university; the recipient would cover the cost of the tapes and shipping, and a grad student would mail a box of source code. There was no explicit copyright or licensing language attached.

The intellectual property office at Berkeley soon learned of this and insisted that Ingres adopt a new practice. Bob Epstein, a leader on the project at the time, sent a regretful email to the Ingres mailing list explaining that the software now carried a UC Berkeley copyright, and that further sharing or distribution of the software required written permission of the university. The Ingres team was disappointed with the change: they wanted widespread adoption and collaboration, and the new legal language interfered with both.

Notwithstanding that limitation, Ingres thrived for several years as a purely academic project. The research team implemented ideas, shipped code to collaborators, and got useful feedback. By 1980, the project had matured enough to be a credible platform for real query workloads. Companies that had been using older database systems were getting interested in Ingres as a possible alternative.

Mike and several of his colleagues decided to capitalize on the opportunity and created Relational Technology, Inc. (RTI) to commercialize the research they had done. The software, unfortunately, was under the restrictive UC Berkeley copyright, which required written permission by the university to reproduce or distribute. Mike made an audacious decision: he unilaterally declared the software to be in the public domain. RTI picked up the research code and used it as the foundation for its commercial offering.

In retrospect, it is hard to believe that it worked. Young professors do not often contradict the legal departments of their employers, especially for their own financial benefit. Mike has no clear explanation himself for how he got away with it. Most likely, quite simply, no one noticed.

Very soon afterward, the CSRG team finished its work with the university's legal department and published the BSD license. Once it existed, Mike quickly adopted it for the Ingres project source code as well. It satisfied all their goals—freely available to use, extend and enhance, and share further; no discrimination against commercial use or redistribution. Best of all, CSRG had already gotten the Berkeley lawyers to agree to the language, so there was no reasonable objection to its use for Ingres. This forestalled any challenge to Mike's brief public-domain insurrection.

Nobody remembers exactly when the first Ingres source code tapes went out under the BSD license, but it was an important day. It made Ingres the world's first complete, standalone substantial piece of systems software distributed as open source. CSRG was shipping BSD code, but it needed the AT&T-licensed code to build on; Ingres compiled and ran without recourse to third-party, proprietary components.

The Impact of Ingres

The Ingres project helped to create the relational database industry, which provided a foundation for all sorts of other technological innovations. Along with System R at IBM, Ingres turned the theory that Codd and Date espoused into practice. The history and detail of the Ingres project are available in Chapter 15, "The Ingres Years."

My own work was on the Postgres project (see Chapter 16 for more information). Like Ingres, Postgres used the BSD license for source code distributions. Postgres was a reaction to the success of Ingres in three important ways.

First, Ingres was a remarkably successful research project and open source database. By the mid- to late-1980s, however, it was clear that the interesting questions had pretty much been asked and answered. Researchers aim to do original work on tough problems. There just wasn't a lot more Ph.D.-worthy work to be done in Ingres.

Second, Mike had started RTI to bring the research project to the commercial market. The company and the research project coexisted for a while, but that could not continue forever. Neither the university nor the National Science Foundation was likely to fund development for the company. Mike had to separate his research from his commercial interests. The Ingres project had to end, and that meant Mike needed something new to work on.

Finally, and more fundamentally, Ingres had constrained Codd and Date's relational theory in important ways. The project team chose the data types and the operations most interesting to them, and those easiest to implement in the C programming language on UNIX. Together, Ingres and IBM's System R had served as reference implementations for all the relational database vendors that cropped up in the 1980s and 1990s (Chapter 13). They mostly chose the same datatypes, operations, and languages that those two research projects had implemented.

Post-Ingres

Mike argued that the relational model supported more than just those early data types and operations, and that products could do more. What if you could store graphs and maps, not just integers and dates? What if you could ask questions about "nearby" not just "less than?" He argued that the software itself could be smarter and more active. What if you could see not just current information, but the whole history of a database? What if you could define rules about data in tables, and the software enforced them?

Features like that are commonplace in database products today, but in the middle 1980s none of the commercial vendors were thinking about them. Mike created the Postgres project—for "post-Ingres," because college professors aren't always great at brand names—to explore those and other ideas.

As a research vehicle and as an engine for graduate degrees, Postgres was phenomenally successful. Like any real research project, it tried out some things that failed: The "no-overwrite" storage system and "time travel" [Stonebraker et al. 1990b, Stonebraker and Kemnitz 1991, Stonebraker 1987] were interesting, but never found a commercial application that pulled them into widespread use in industry. Other ideas did take hold, but not always in the way that Postgres did them: Mike's students implemented rules [Potamianos and Stonebraker 1996, Stonebraker et al. 1988a, Stonebraker et al. 1989] that applied to tables in a couple of different ways (the "query rewrite" rules system and the "tuple-level" rules system). Most databases support rules today, but they're built on the foundations of those systems, and not often in the ways that Postgres tried out. Some ideas, however, did get widespread adoption, much in the way that Postgres designed them. Abstract data types (ADTs) and user-defined functions (UDFs) in database systems today are based expressly on the Postgres architecture; support for spatial data and other complex types is commonplace.

In the early 1990s, the Ingres cycle repeated itself with Postgres. Most of the fundamental new research ideas had been explored. The project had been successful enough to believe that there would be commercial demand for the ideas. Mike started Montage (soon renamed Miro, then renamed again Illustra) in 1992 to build a product based on the project, and moved his research focus elsewhere. Illustra was acquired by Informix in 1996, and the bigger company integrated many of

Postgres' features into its Universal Database product. Informix was, in turn, later acquired by IBM.

The Impact of Open Source on Research

If you ask Mike today, he will tell you that the decision to use the BSD license for Ingres was just dumb luck. He was there at the time, so we should take him at his word. Whether he was an open-source visionary or not in the late 1970s, however, it's clear that he was an important figure in the early open-source movement. Ingres' success as open source changed the way that research is done. It helped, besides, to shape the technology industry. Mike now endorses using open source in both research (Chapter 10) and in startups (Chapter 7).

Just like the Ingres team did, those of us working on Postgres had an advantage that most graduate students lacked: we had real users. We'd publish a build on the project FTP site (no World Wide Web back then, youngsters!) and we'd watch people download it from all over the planet. We didn't offer any formal support, but we had a mailing list people could send questions and problems to, and we'd try to answer them. I still remember the blend of pride and angst we felt when we got a bug report from a Russian nuclear power facility—shouldn't they have been running code that had a quality assurance team?

The decision to publish research as open source established Berkeley as the first-class systems school it still is today. Grad students at other schools wrote papers. We shipped software. Oh, sure, we wrote papers, too, but ours were improved tremendously because we shipped that software. We could collaborate easily with colleagues around the globe. We learned how our ideas worked not just in theory, but also in practice, in the real world.

Code is a fantastic vector for the scientific method. Code makes it incredibly easy for others to test your hypothesis and reproduce your results. Code is easy for collaborators to enhance and extend, building on your original ideas with new ones of their own.

Ingres' success as the first large open-source systems software project influenced the thinking of faculty at universities around the world, but especially at Berkeley. Virtually all systems work at Berkeley today is open source. It's as fundamental to the university as free speech and Top Dog.[1]

[1] A long-standing (since 1966), not-so-healthy diner at Berkeley.

Stonebraker himself learned the lesson, of course. He had seen the benefits of the BSD license with Ingres and used it again on Postgres. Open source gave Postgres a tremendous impact on the industry.

For example, unlike Ingres, the project survived its shutdown by the university. Two former students, Jolly Chen and Andrew Yu, launched a personal project to replace Postgres' "postquel" query language with the by-then-industry-standard SQL. They rewrote the query parser and put up a new project page. In a nod to history and to the hard work they'd done, they named the new package "PostgreSQL."

Their work attracted the attention of folks outside the university. Today, PostgreSQL has a vibrant developer and user community around the world. PostgreSQL remains a proudly independent project, deployed widely, creating value and opportunity for an ecosystem of contributors, users, support organizations, consultants, and others. At the time I am writing this chapter, the project's development hub shows 44,000 commits against more than one million lines of code. At least two standalone companies ship commercial versions of it today. Besides that, virtually every version of Linux bundles a copy, and Amazon offers a hosted version for use in the cloud. All of that is just a sampling. There's a whole lot of PostgreSQL out there.

Getting taxpayer-funded research out of the university so that it can benefit citizens is important. Open source makes that easier. Mike, with Ingres, was the first person to create a substantial and innovative piece of open-source systems software with government funding, and then to start a company to commercialize precisely the IP created inside the university. The model worked well. He repeated it with Postgres and many other projects at Berkeley and other universities since.

That showed professors at Berkeley and elsewhere that they could work in the academy and still participate in the marketplace. Professors and grad students have long taken risks in research, and then started companies to bring products to market. Open source eliminates friction: the code is there, all set to go, permission granted. Whether we attract better young people to careers in research because of this is unknowable; certainly, we give those who choose to advance the state of the art a way to participate in the value that they create.

And it is not only Mike and colleagues like him who benefit financially.

Because of the permissive BSD license, Postgres and PostgreSQL were available to anyone who wanted to start a company based on the project. Many did. Netezza, Greenplum, Aster Data, and others adopted and adapted the code. Pieces of it—the query parser, for example—have found their way into other products. That saved many millions in upfront research and development costs. It made companies possible that might never have started otherwise. Customers, employees, and the investors of all those companies benefited tremendously.

My own career owes a great deal to the innovation in software licensing and the relational database industry that descended from the Ingres project. I have been part of two startups, Illustra and Sleepycat, created expressly to commercialize open-source UC Berkeley databases. My current company, Cloudera, builds on the lessons I've learned from Mike, using open-source data management technology from the consumer internet: the Apache Hadoop project for big data, and the rich ecosystem it has spawned.

More broadly, the entire database community—industry and academia—owes a great deal to Ingres, and to its implementation in open-source software. The ready availability of a working system meant that others—at universities that couldn't afford to build or buy their own systems, and at companies that couldn't afford to fund the blue-sky research that Mike did at Berkeley—could explore a real working system. They could learn from its innovations and build on its strengths.

In 2015, Dr. Michael Stonebraker won the 2014 A.M. Turing Award for lifetime contributions to the relational database community. There's no question that the very specific, very direct work that Mike led at Berkeley and elsewhere on Ingres, Postgres, columnar storage, and more deserves that award. The relational database market exists in no small part because of his work. That market generates hundreds of billions of dollars in commercial activity every year. His innovative use of a permissive open-source license for every meaningful project he undertook in his career amplified that work enormously. It allowed everyone—the commercial sector, the research community, and Mike himself—to create value on top of his innovation.

The choice of BSD for Ingres may well have been lucky, but in my experience, luck comes soonest to those who put themselves in its path. So much of research, so much of innovation, is just hard work. The inspired laziness of choosing the BSD license for Ingres—getting all the benefits of broad distribution, all the power of collaboration, without the trouble of a fight with the UC Berkeley legal team—put Ingres smack in the way of lucky.

We all learned a great deal from that first, hugely impactful, open-source systems project.

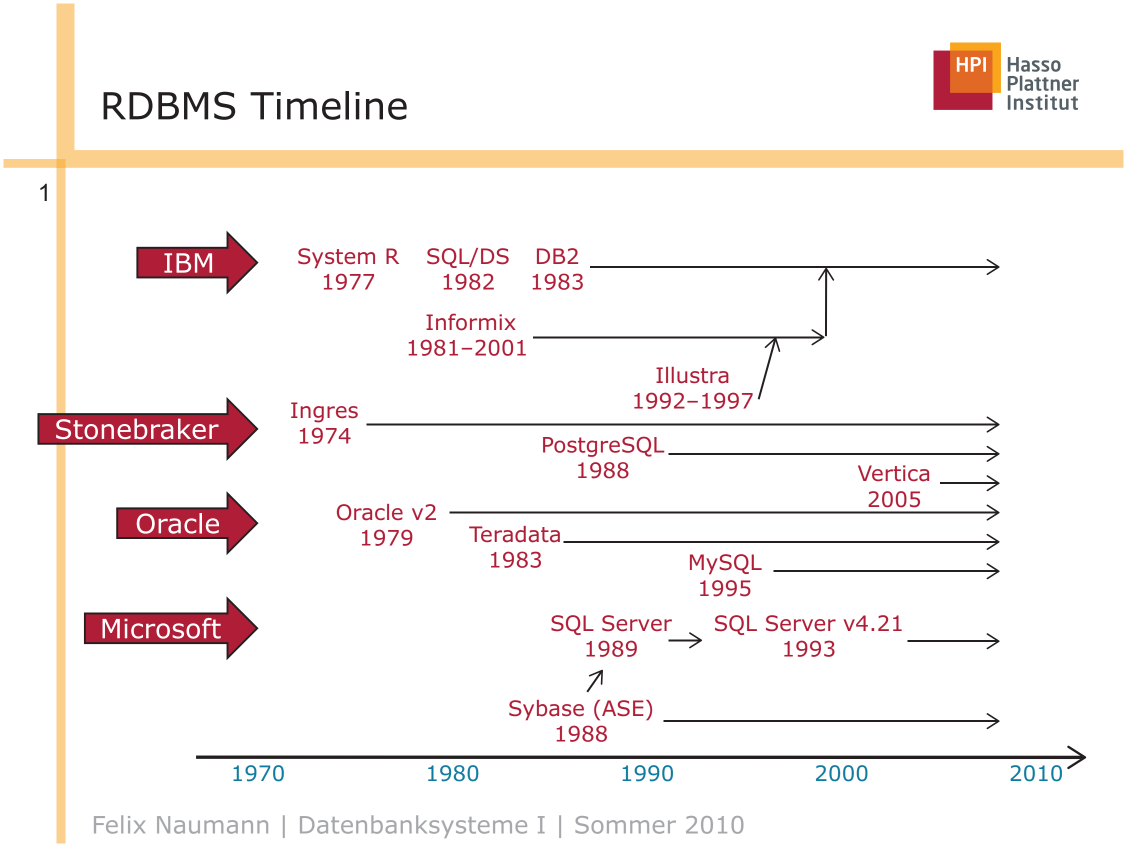
13

The Relational Database Management Systems Genealogy

Felix Naumann

The history of database systems, in particular the relational kind, reaches far back to the beginnings of the computer science discipline. Few other areas of computer science can look back as many decades and show how concepts, systems, and ideas have survived and flourished to the present day. Inspired by the database lectures of my Ph.D. advisor Christoph Freytag, who always included a short section on the history of DBMS in his courses, I included some such material in my own slides for undergraduate students (see Figure 13.1 from 2010). My limited view of DBMS history (and slide layout) is apparent.

Later, in the first week of my first database course as a professor, I had presented much introductory material, but lacked hard content to fill the exercise sheets and occupy the students. Thus, I let students choose a system from a long list of well-known DBMSs and asked them to research its history and collect data about its origin, dates, versions, etc. Together with my teaching assistants, I established an initial more-complete timeline, which anticipated the design of the latest version (see Figure 13.2). A graphic designer suggested that we apply a subway-map metaphor and in 2012 created the first poster version of the genealogy, as it appears today (Figure 13.3). Many years and versions later, and many systems and nodes more, the current 2017 genealogy is shown in Figure 13.4. Clearly, it has grown much denser for the present time (at the right of the chart), but also much more informative for the beginnings of RDBMS (at left), based on various discoveries of early RDBMSs.



RDBMS Timeline

HPI Hasso Plattner Institut

1

IBM

System R 1977

SQL/DS 1982

DB2 1983

Informix 1981–2001

Illustra 1992–1997

Stonebraker

Ingres 1974

PostgreSQL 1988

Vertica 2005

Oracle

Oracle v2 1979

Teradata 1983

MySQL 1995

Microsoft

SQL Server 1989

SQL Server v4.21 1993

Sybase (ASE) 1988

1970 1980 1990 2000 2010

Felix Naumann | Datenbanksysteme I | Sommer 2010

Figure 13.1 Genealogy slide from database lecture (2010).

Overall, the genealogy contains 98 DBMS nodes, 48 acquisitions, and 34 branches and 6 mergers. Many of the DBMSs are no longer in existence—19 are marked as discontinued.

Creating a genealogy like this is a somewhat unscientific process: the concrete start date for a system is usually undefined or difficult to establish. The same is also true for the points in time of almost all other events shown in the chart. Thus, time is treated vaguely—nodes are placed only approximately within their decades. Even more difficult is the treatment of branches, a feature that makes the chart especially interesting. We were very generous in admitting a branch: it could signify an actual code fork, a licensing agreement, or a concrete transfer of ideas, or it could simply reflect researchers and developers relocating to a new employer and re-establishing

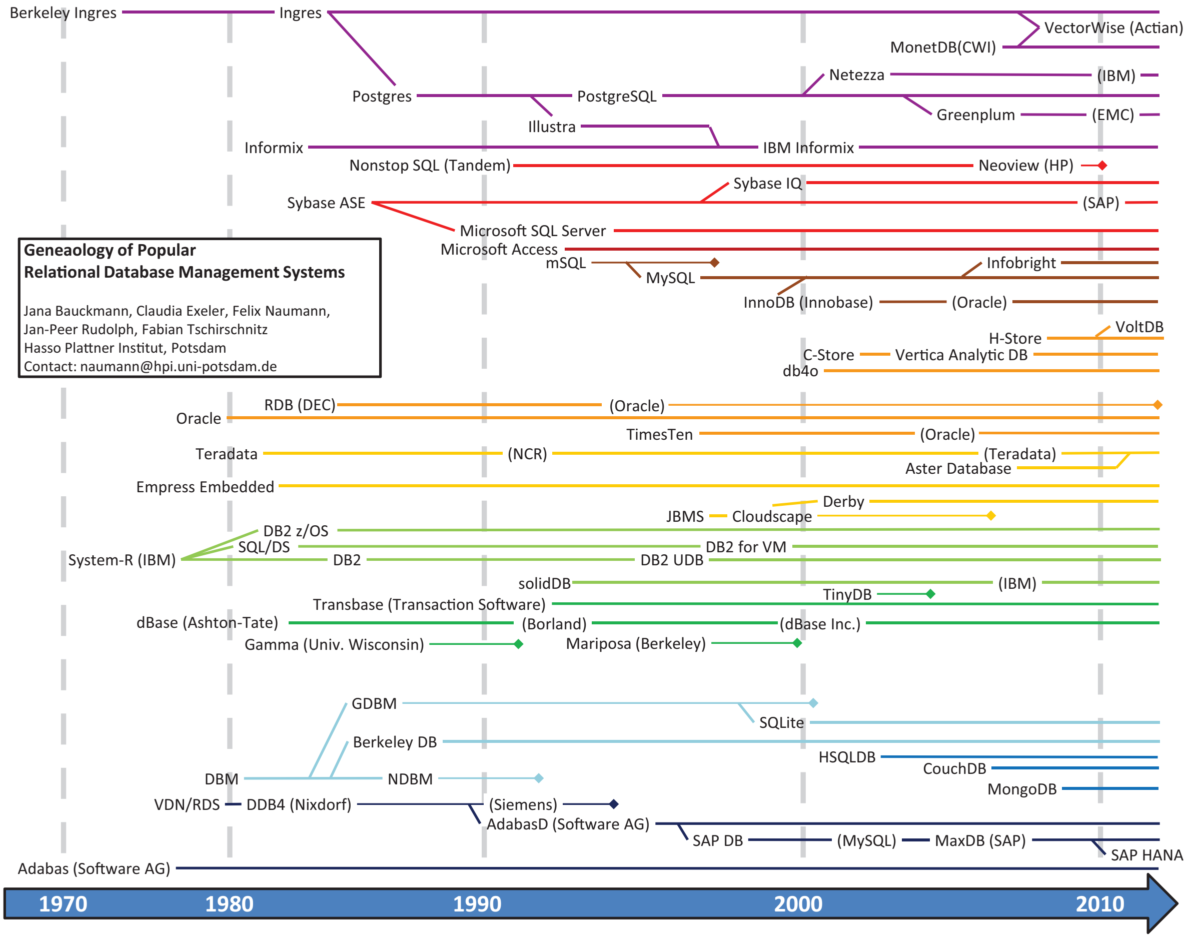


Figure 13.2 First poster version of the genealogy (2012).

the DBMS or its core ideas there. There is no structured source from which this chart is automatically created. Every node and every line are manually placed, carefully and thoughtfully.

Another important question was and remains which database systems to include. We have been strict about including only relational systems supporting at least some basic SQL capabilities. Hierarchical and object-oriented systems are excluded, as are XML or graph-databases or key-value stores. Another criterion for inclusion is that the system has at least some degree of distribution or some user base. A simple research prototype that was used only for research experiments

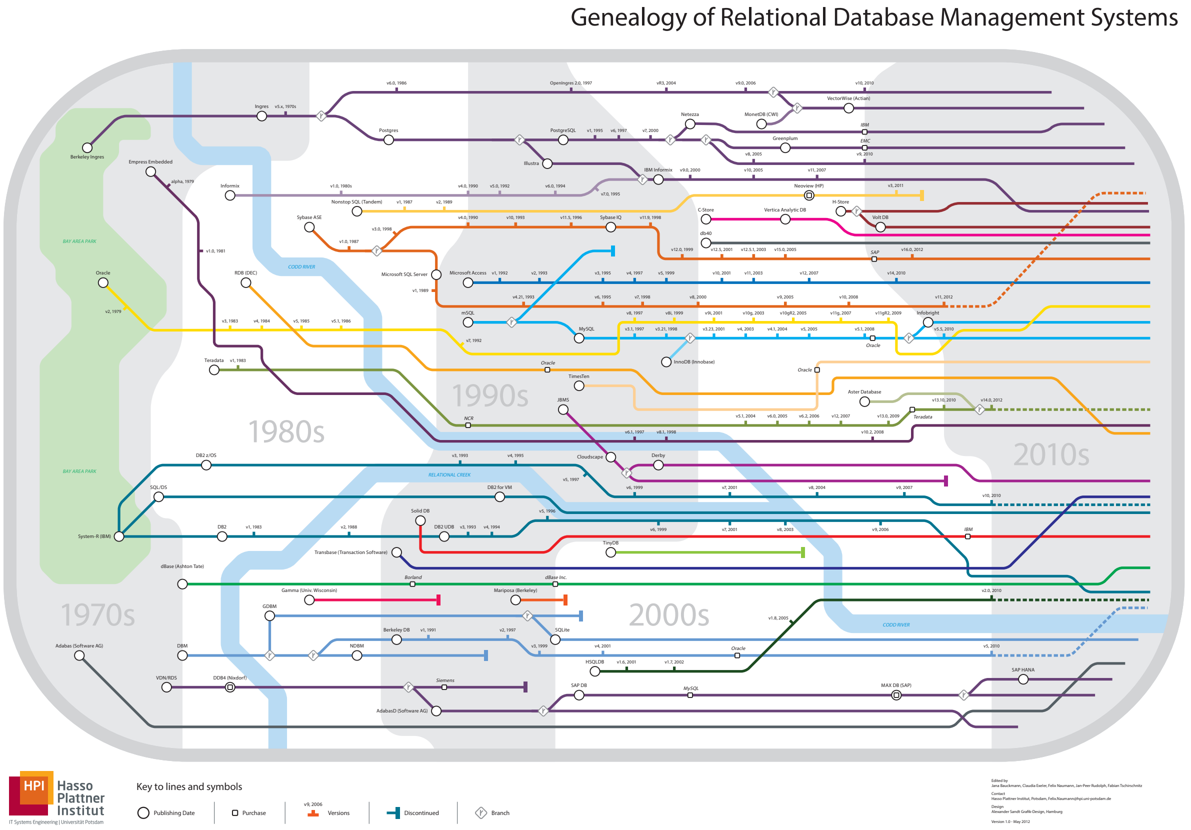


Figure 13.3 Version 1 of subway-map genealogy (2012).

would not fit that description. That being said, we took some liberty in admitting systems and still welcome any feedback for or against specific systems.

In fact, after the initial design and publication of the genealogy in 2012, the main source for additions, removals, and corrections was email feedback from experts, researchers, and developers after the announcement of each new version. Over the years, more than 100 persons contacted me, in parts with very many suggestions and corrections. The shortest email included nothing but a URL to some obscure DBMS; the most input by far came from David Maier. Especially for the early history of RDBMS, I relied on feedback from many other leading experts in database research and development, including (in chronological order of their input) Martin Kersten, Gio Wiederhold, Michael Carey, Tamer Öszu, Jeffrey Ullman, Erhard Rahm, Goetz Graefe, and many others. Without their experience and recollection, the genealogy would not exist in its current form, showing the impressive development of our field. For the most recent version, which is included in this book, Michael Brodie of MIT CSAIL initiated conversations with Joe Hellerstein, Michael Carey, David DeWitt, Kapali Eswaran, Michael Stonebraker, and several others, unearthing various new systems and new connections (sometimes scribbled on classroom notes), with a slight bias towards the numerous ones that can be traced back to Michael Stonebraker's work. These DBMSs can be found all over the genealogy, making Michael the great-great-grandfather of some of them. Starting from Ingres at the top left of that chart, you not only can reach many commercial and non-commercial systems, but also find smaller projects hidden in the genealogy, such as Mariposa, H-Store, and C-Store.

While I do not have download statistics, I have occasionally seen the printed poster in the background during television interviews with IT experts who had hung it on their office walls. Downloading, printing, and using the chart is free. Please find the latest version at http://hpi.de/naumann/projects/rdbms-genealogy.html. And, as always, additions and corrections are welcome.

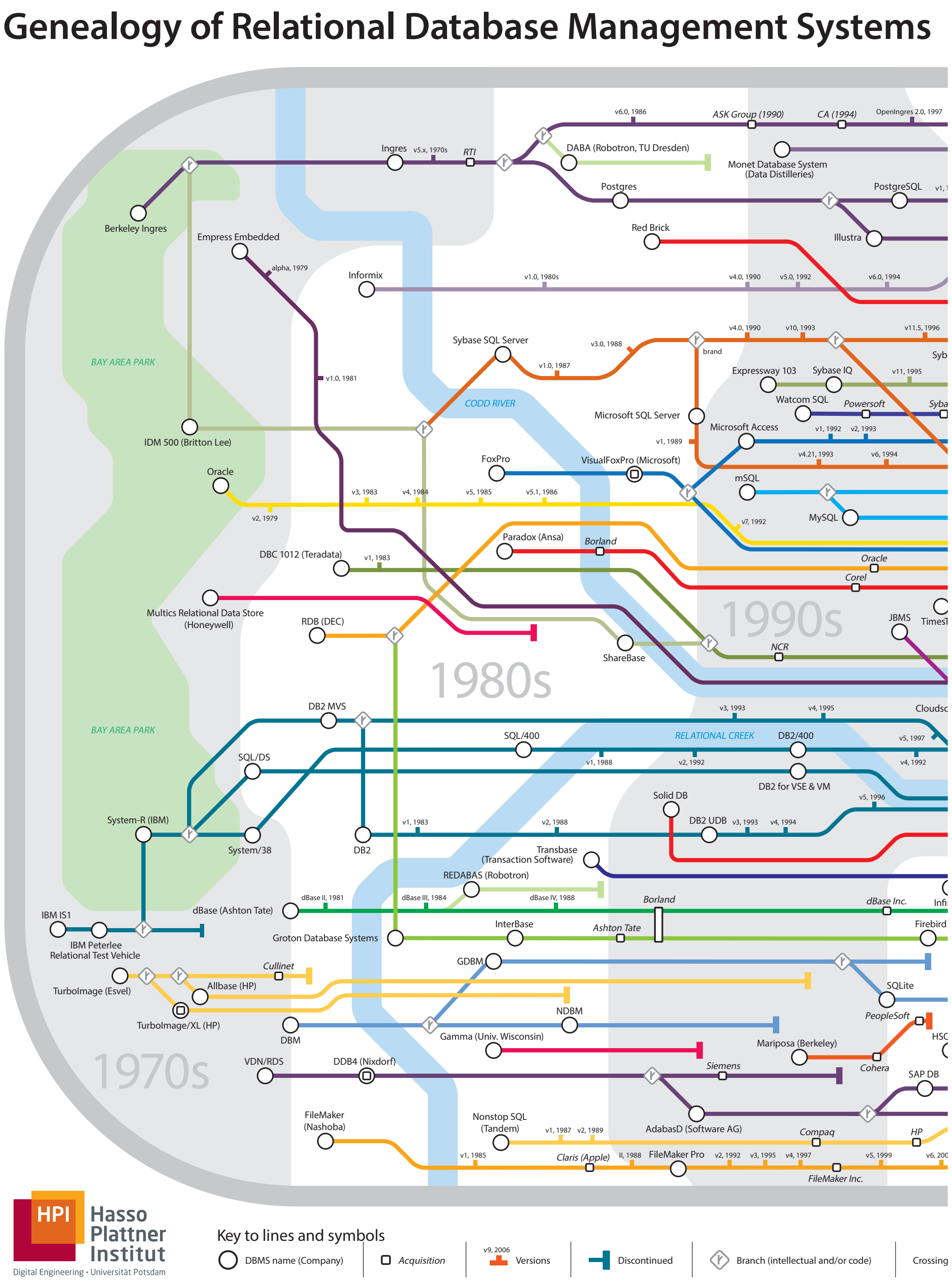
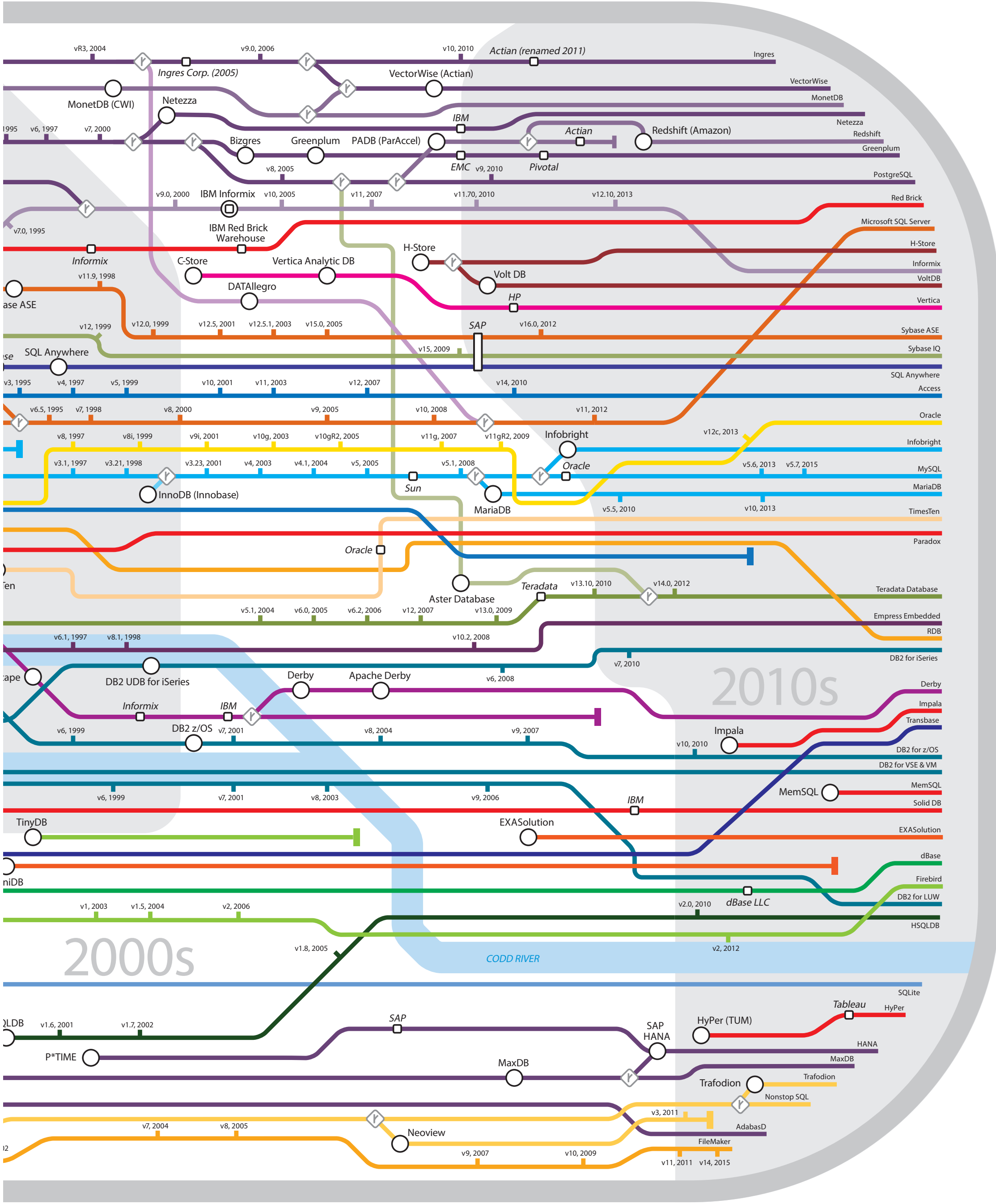


Figure 13.4 Subway-map genealogy today.



lines have no special semantics

Felix Naumann, Jana Bauckmann, Claudia Exeler, Jan-Peer Rudolph, Fabian Tschirschnitz
Contact - Hasso Plattner Institut, University of Potsdam, felix.naumann@hpi.de
Design - Alexander Sandt Grafik-Design, Hamburg
Version 6.0 - October 2017
https://hpi.de/naumann/projects/rdbms-genealogy.html

PART VII

CONTRIBUTIONS BY SYSTEM

Chapters in this part are in pairs. Research chapters in VII.A have corresponding systems chapters in VII.B. Research results described in Chapter 15 led to systems results (Ingres) described in Chapter 24; research results described in Chapter 16 led to systems results (Postgres) described in Chapter 25; and so forth.

14

Research Contributions of Mike Stonebraker: An Overview

Samuel Madden

As preceding chapters make clear, Mike Stonebraker has had a remarkable career, with at least (depending on how one counts) eight incredibly influential database systems, many of which were backed by commercial companies, spanning five decades of work. In the chapters in this section, Mike's collaborators on these projects and systems look at their technical contributions to computing. For each of his major systems, there are two chapters: one highlighting the intellectual, research, and commercial impact and Mike's role in crafting these ideas, and the other describing the software artifacts and codelines themselves. Like all large software systems, these projects were not Mike's alone, but in all cases their success and influence were magnified by Mike's involvement. Our goal in these writings is not to exhaustively recap the research contributions of the work, but instead capture a bit of what it was like to be there with Mike when the ideas emerged and the work was done.

Technical Rules of Engagement with Mike

Before diving into the technical details, it's worth reflecting a bit on the general technical rules of engagement when working with Mike.

First, Mike is an incredible collaborator. Even at 74, he comes to every meeting with new ideas to discuss and is almost always the first to volunteer to write up ideas or draft a proposal. Despite being the CTO of at least two companies, he fires off these research drafts seemingly instantaneously, leaving everyone else scrambling to keep up with him and his thinking.

Second, in research, Mike has a singular focus on database systems—he is not interested in other areas of computer science or even computer systems. This focus has served him well, magnifying his impact within the area and defining the scope of the systems he builds and companies he founds.

Third, Mike values simple, functional ideas above all else. Like all good systems builders, he seeks to eliminate complexity in favor of practicality, simplicity, and usability—this is part of the reason for his success, especially in commercial enterprises. He tends to dismiss anything that he perceives as complicated, and often prefers simple heuristics over complex algorithms. Frequently this works out, but it is not without pitfalls. For example, as described in Chapter 18 and Chapter 27, Mike felt strongly that C-Store and its commercial offspring should not use a conventional dynamic-programming-based algorithm for join ordering; instead, he believed in a simpler method that assumed that all tables were arranged in a "star" or "snowflake" schema, and only allowed joins between tables arranged in this way. Such schemas were common in the data warehouse market that Vertica was designed for, and optimizing such queries could be done effectively with simple heuristics. Ultimately, however, Vertica had to implement a real query optimizer, because customers demanded it, that is, the complicated design was actually the right one!

Fourth, Mike tends to see ideas in black and white: either an idea is great or it is awful, and more than one researcher has been taken aback by his willingness to dismiss their suggestions as "terrible" (see Chapter 21). But Mike is malleable: even after dismissing an idea, he can be convinced of alternative viewpoints. For example, he recently started working in the area of automated statistical and AI approaches to data integration and founded the company, Tamr (see Chapter 30), despite years of arguing that this problem was impractical to solve with AI techniques due to their complexity and inability to scale, as some of his collaborators describe later in this introduction.

Fifth, although Mike does form opinions quickly and doesn't shy away from sharing them—sometimes in controversial fashion—he's usually right. A classic example is his quip (in a 2008 blog post with David DeWitt [DeWitt and Stonebraker 2008]) that Hadoop was "a major step backwards," which resulted in some on the Internet declaring that Mike had "jumped the shark." However, the point of that post—that most data processing was better done with SQL-like languages—was prescient. Hadoop was rather quickly displaced. Today, most users of post-Hadoop systems, like Spark, actually access their data through SQL or an SQL-like language, rather than programming MapReduce jobs directly.

Finally, Mike is not interested in (to use a famous phrase of his) "zero-billion-dollar" ideas. Mike's research is driven by what real-world users (typically commercial users) want or need, and many of his research projects are inspired by what business users have told him are their biggest pain points.[1] This is a great strategy for finding problems, because it guarantees that research matters to someone and will have an impact.

Mike's Technical Contributions

In the rest of this introduction, we discuss technical contributions and anecdotes from collaborators on Mike's "big systems." The chapters in this section go in chronological order, starting with his time at UC Berkeley, with the Ingres and Postgres/Illustra projects and companies, and then going on to his time at MIT and the Aurora and Borealis/StreamBase, C-Store/Vertica, H-Store/VoltDB, SciDB/Paradigm4, and Data Tamer/Tamr projects/companies. Amazingly, this introduction actually leaves out two of his companies (Mariposa and Goby), because they are less significant both commercially and academically than many of his research projects, some of which you'll find mentioned elsewhere in the book. Finally, some of his current collaborators talk about what Mike has been doing in his most recent research.

The Berkeley Years

For the Ingres project, which Mike began when he got to Berkeley in 1971, Mike Carey writes about the stunning audacity of the project and its lasting impact (Chapter 15). Mike Carey was one of Mike's early Ph.D. students, earning his Ph.D. in 1983 and becoming one of the most influential database systems builders in his own right. He shares his perspective on how a junior professor with a background in math (thesis title: "The Reduction of Large Scale Markov Models for Random Chains") decided that building an implementation of Ted Codd's relational algebra was a good idea, and describes how Mike Stonebraker held his own competing against a team of 10-plus Ph.D.s at IBM building System R (Chapter 35). As Mike Carey describes, the Ingres project had many significant research contributions: a declarative language (QUEL) and query execution, query rewrites and view substitution algorithms, hashing, indexing, transactions, recovery, and many other ideas we now take for granted as components of relational databases. Like much

[1] Mike describes the use of such pain points that became the source of innovation and value in Ingres and other projects; see Chapter 7.

of Mike's work, Ingres was equally influential as an open-source project (Chapter 12) that became the basis of several important commercial database systems (illustrated in living color in Chapter 13).

In his write-up about Mike's next big Berkeley project, Postgres ("Post-Ingres"), Joe Hellerstein (Chapter 16) (who did his Master's work with Mike during the Postgres era and is another leading light in the database system area) describes Postgres as "Stonebraker's most ambitious research project." And indeed, the system is packed full of important and influential ideas. Most important and lastingly is support for abstract data types (aka user-defined types) in databases through the so-called "Object-Relational" model, as an alternative to the then-popular "Object-Oriented Database" model (which has since fallen out of favor). ADTs are now the standard way all database systems implement extensible types and are critically important to modern systems. Other important ideas included no-overwrite storage/time travel (the idea a database could store its entire history of deltas and provide a view as of any historical point in time) and rules/triggers (actions that could be performed whenever the database changed). As Joe describes, Postgres also led to the development of Mike's first big commercial success, Illustra (Chapter 25), as well as (eventually) the release of the hugely influential PostgreSQL open-source database, which is one of the "big two" open-source RDBMS platforms (with MySQL) still in use today (Chapter 12).

The Move to MIT

Mike left Berkeley in the late 1990s and a few years later moved to MIT as an adjunct professor. Even though he had already achieved more academically and commercially than most academics do in a career, he enthusiastically embarked on a remarkable series of research projects, beginning with the stream processing projects Aurora and Borealis (Chapter 17). These projects marked the beginning of a long series of collaborations among MIT, Brown, and Brandeis, which I would join when I came to MIT in 2004. In their chapter about Aurora/Borealis, Magda Balazinska (now a professor at the University of Washington and then a student on the projects) and Stan Zdonik (a professor at Brown and one of Mike's closest collaborators) reflect on the breadth of ideas from the Borealis project, which considered how to re-engineer a data processing system that needs to process "streams": continuously arriving sequences of data that can be looked at only once. Like many of Mike's projects, Aurora eventually became StreamBase (Chapter 26), a successful startup (sold to TIBCO a few years ago) focused on this kind of real-time data, with applications in finance, Internet of Things, and other areas.

The "One Size Doesn't Fit All" Era

In the 2000s, the pace of Mike's research accelerated, and he founded companies at a breakneck pace, with five companies (Vertica, Goby, Paradigm4, VoltDB, Tamr) founded between 2005 and 2013. Two of these—Vertica and Goby—ended with acquisitions, and the other three are still active today. As inspiration for Vertica, Paradigm4, and VoltDB, Mike drew on his famous quip that "one size does not fit all," meaning that although it is technically possible to run massive-scale analytics, scientific data, and transaction processing workloads, respectively, in a conventional relational database, such a database will be especially good at none of these workloads. In contrast, by building specialized systems, order-of-magnitude speedups are possible. With these three companies and their accompanying research projects, Mike set out to prove this intuition correct.

In the case of C-Store, our idea was to show that analytics workloads (comprising read-intensive workloads that process many records at a time) are better served by a so-called "column-oriented" approach, where data from the same column is stored together (e.g., with each column in a separate file on disk). Such a design is suboptimal for workloads with lots of small reads or writes but has many advantages for read-intensive workloads including better I/O efficiency and compressibility. Daniel Abadi (now a professor at the University of Maryland) writes about his early experiences as a grad student on the project (Chapter 18) and how Mike helped shape his way of thinking about system design through the course of the project. C-Store was classic "systems research": no new deep theoretical ideas, but a lot of design that went into making the right decisions about which components to use and how to combine them to achieve a working prototype and system. Mike commercialized C-Store as the Vertica Analytic Database, which was quite successful and continues to be one of the more widely used commercial analytic database systems. Vertica was acquired by HP in 2011 and is now owned by Micro Focus, Int'l PLC.

In the H-Store project, the idea was to see how "one size does not fit all" could be applied to transaction processing systems. The key observation was that general-purpose database systems—which assume that data doesn't fit into memory and use standard data structures and recovery protocols designed to deal with this case—give up a great deal of efficiency compared to a system where data is assumed to be memory-resident (generally the case of transaction processing systems on modern large main-memory machines). We designed a new transaction processing system; Andy Pavlo (who writes about H-Store in Chapter 19) and several other graduate students built the prototype system, which pushed the boundaries of transaction processing several orders of magnitude beyond what general-purpose databases could achieve. The H-Store design became the blueprint for Mike's founding of VoltDB (Chapter 28), which is still in business today, focused on a variety of low-latency and real-time transaction processing use cases. More recently, Mike has extended the H-Store design with support for predictive load balancing ("P-Store" [Taft et al. 2018]) and reactive, or elastic, load balancing ("E-Store"; [Taft et al. 2014a].

After H-Store, a group of academics (including me) led by Mike and David DeWitt went after another community not well served by current "one size fits all" databases: scientists (Chapters 20 and 29). In particular, many biologists and physicists have array-structured data that, although it can be stored in databases as relations, is not naturally structured as such. The academic project, called SciDB, looked at a number of problems, including data models and query languages for array data, how to build storage systems that are good for sparse and dense arrays, and how to construct systems with built-in versioning of data appropriate for scientific applications. In prototypical fashion, Mike quickly started a company, Paradigm4, in this area as well. In an entertaining essay, Paul Brown, the chief architect of the product, describes the journey from conception to today in terms of a mountain-climbing expedition, replete with all of the thrills and exhaustion that both expeditions and entrepreneurship entail (Chapter 20). The development of the SciDB codeline is described in Chapter 29.

The 2010s and Beyond

After his sequence of one-size-does-not-fit-all projects in the 2000s, around the turn of the decade, Mike turned to a new area of research: data integration, or the problem of combining multiple related data sets from different organizations together into a single unified data set. First, in the Data Tamer project, Mike and a group of researchers set out to work on the problem of record deduplication and schema integration—that is, how to take a collection of datasets describing the same types of information (e.g., employees in two divisions of a company) and create a single, unified, duplicate-free dataset with a consistent schema. This is a classic database problem, traditionally solved through significant manual effort. In the Data Tamer project, the idea was to automate this process as much as possible, in a practical and functional way. This led to the creation of Tamr Inc. in 2013. In his write-up (Chapter 21), Ihab Ilyas talks about his experience on the academic project and as a co-founder of the company and relates some of the valuable lessons he learned while building a real system with Mike. The development of the Tamr codeline is described in Chapter 30.

This section concludes with a discussion of one of Mike’s most recent projects, which centers on the idea of polystores: “middleware” that allows users to query across multiple existing databases without requiring the data to be ingested into a single system called BigDAWG, which we built as a part of the Intel Science and Technology Center for Big Data (Chapter 22). The chapter is written by Timothy Mattson, Intel Technical Fellow and a co-developer of one of the first polystore systems, with co-authors Jennie Rogers (now a professor at Northwestern) and Aaron Elmore (now a professor at the University of Chicago), both of whom were postdocs working on the BigDAWG project. BigDAWG is available as open source code (Chapter 31).

In a related project, Mourad Ouzzani, Nan Tang, and Raul Castro Fernandez talk about working with Mike on data integration problems that go beyond schema integration toward a complete system for finding, discovering, and merging related datasets, possibly collected from diverse organizations or sub-groups (Chapter 23). For example, a data scientist at a pharmaceutical company may wish to relate his or her local drug database to a publically available database of compounds, which requires finding similar columns and data, eliminating noisy or missing data, and merging datasets together. This project, called Data Civilizer, is a key part of a research collaboration with the Qatar Center for Research and Innovation (QCRI), and is noteworthy for the fact that it has moved Mike away from some of the typical “systems” problems he has worked on toward much more algorithmic work focused on solving a number of nitty-gritty problems around approximate duplicate detection, data cleaning, approximate search, and more. The development of Aurum, a component of the yet-to-be-developed Data Civilizer, is described in Chapter 33.

In summary, if you make purchases with a credit card, check your bank balance online, use online navigation data from your car or use data to make decisions on the job, you’re likely touching technologies that originated with Mike Stonebraker and were perfected by him with his many students and collaborators over the last 40-plus years. We hope that you enjoy the following “under the hood” looks at the many innovations that made databases work for us all.

PART VII.A

Research Contributions by System

15

The Later Ingres Years

Michael J. Carey

This chapter is an attempt, albeit by a fairly latecomer to the Ingres party, to chronicle the era when it all started for Mike Stonebraker and databases: namely, the Ingres years! Although my own time at UC Berkeley was relatively short (1980–1983), I will attempt to chronicle many of the main activities and results from the period 1971–1984 or thereabouts.

How I Ended Up at the Ingres Party

The path that led me to Berkeley, and specifically to Mike Stonebraker's database classes and research doorstep, was entirely fortuitous. As an electrical engineering (EE) math undergraduate at Carnegie Mellon University (CMU) trying to approximate an undergraduate computer engineering or computer science major before they existed at CMU, I took a number of CS classes from Professor Anita Jones. At the time Anita was building one of the early distributed operating systems (StarOS) for a 50-node NUMA multiprocessor (Cm*). She also co-supervised my master's degree thesis (for which I wrote a power system simulator that ran on Cm*). I decided to pursue a Ph.D. in CS, and I sought Anita's sage advice when it came time to select a program and potential advisors to target. I was inclined to stay closer to home (East Coast), but Anita "made me" go to the best school that admitted me, which was Berkeley. When I asked her for advice on potential systems-oriented research advisors—because I wanted to work on "systems stuff," preferably parallel or distributed systems—Anita suggested checking out this guy named Mike Stonebraker and his work on database systems (Ingres and Distributed Ingres). CMU's CS courses didn't cover databases and, with parents who worked in business and accounting, I thought that working on databases sounded boring: possibly the CS equivalent of accounting. But I mentally filed away her advice nonetheless and headed off to Berkeley.

Once at Berkeley, it was time to explore the course landscape in CS to prepare for the first layer of Ph.D. exams as well as to explore areas to which I had not yet been exposed. Somewhat grudgingly following Anita's advice, I signed up for an upper-division database systems class taught by Mike. By that time (academic year 1980–81), Mike and his faculty colleague Professor Eugene Wong had built and delivered to the world the Ingres relational DBMS—more on that shortly—and it was the basis for the hands-on assignments in my first database class.

That first class completely changed my view of things. It turned out to be really interesting, and Mike (despite what he will say about himself as a teacher) did a great job of teaching it and making it interesting, thereby luring me down a path toward databases. I went on to take Mike's graduate database class next. By then I was hooked: very cool stuff. I learned that this newly emerging database systems field was really a vertical slice of all of CS—including aspects of languages, theory, perating systems, distributed systems, AI (rule systems), and so on—but with an emphasis on "doing it for data" and in a declarative way.

Databases weren't so boring after all. In fact, they were interesting and now on my list of possible Ph.D. subareas. By the time I passed the preliminary exams and it was time to seek an advisor, I had three or four possibilities on my list: computer architecture (Dave Patterson), operating systems/distributed systems (John Ousterhout or perhaps Mike Powell), and of course databases (Mike). I talked to each one about the road to a Ph.D. in their view, and for the most part what I heard was a five-year hike up the Ph.D. mountain. But not from Mike! He had various potential topics in mind, convinced me that transaction management was like operating systems (it is), and told me he could have me out in three years. Sold to the lowest bidder! It also didn't hurt that, at the time, Mike had created a fun and inviting database group culture, insisting that interested students join the Ingres team at La Val's Pizza up the street from Cory Hall (a.k.a. Ingres HQ) for periodic outings for pizza and pitchers of beer (Where, I seem to recall, Mike always had adverse reactions to empty glasses).

Ingres: Realizing (and Sharing!) a Relational DBMS

Mike's database story begins with Ingres [Held and Stonebraker 1975], short for INteractive Graphics REtrieval System. It was one of the world's two high-impact early relational DBMSs; the other was System R from IBM, a concurrent project at the IBM San Jose Research Center. The Ingres system was co-developed by Mike and Gene (ten years his senior, and the person who talked Mike into reading Ted Codd's seminal paper and into converting to be a "database guy"). I missed the early years of Ingres, as I arrived in 1980 and started hanging out in Ingres territory only in 1981, by which time the initial Ingres project was over and Distributed Ingres was also mostly winding down (Chapter 5). A number of the early Ingres system heroes had already come and gone—e.g., Jerry Held, Karel Youssefi, Dan Ries, Bob Epstein—and the single-system version of Ingres had been distributed to over 1,000 sites (prehistoric open source!). The Distributed Ingres project was also largely "done": there was a prototype that ran distributed queries over geographically distributed data, but it was never hardened or shared like Ingres. I overlapped for a year with team member Dale Skeen, who worked on distributed commit protocols and who mentored me (thanks, Dale!) and then left for academia before starting several notable companies, including TIBCO, the birthplace of pub/sub (publish/subscribe messaging). I did get to meet Eric Allman; he was the backbone (staff member) for the Ingres group.

In my day as a graduate student and in my "first career" as an academic at the University of Wisconsin-Madison, Ingres was tremendously well known and highly regarded (Chapter 6). I fear that the same is not true today, with the commercial success of IBM's DB2 system—the main competitor for Ingres—which has made System R much more visible in today's rear-view mirror for students. In reality, Ingres—Mike and Gene's first gift to the database world—was truly a remarkable achievement that helped shape the field today in ways that are well worth reiterating here.

Ingres made a number of contributions, both technically and socially. In my view, it is still Mike's finest database achievement, which is obviously saying a lot given the many things that Mike has done over the course of his career.

So, what was the big deal about Ingres? Let's have a look at what Ingres showed us . . .

1. One of the foremost contributions of Ingres—likewise for System R—was showing that Ted Codd wasn't out of his mind when he introduced the relational model in 1970, i.e., that it was indeed possible to build a relational DBMS. Ingres was a complete system: It had a query language, a query execution system, persistent data storage, indexing, concurrency control, and recovery—and even a C language embedding (EQUEL [Allman et al. 1976])—all assembled by a team of faculty, students, and a few staff at a university [Stonebraker 1976b]. This in and of itself is a truly remarkable achievement.

```
range of s is supply
range of p is parts
retrieve (s.snum) where s.pnum=p.pnum and p.pname=''central processor''
\g
```

Figure 15.1 A simple Quel join query, by Mike Stonebraker circa 1975. Source: [Stonebraker 1975].

2. Ingres showed how one could design a clean, declarative query language—Quel—based on Codd's declarative mathematical language ideas. Quel was the language used in the database class that converted me. It was a very clean language and easy to learn—so easy in fact that I taught my mother a little bit of Quel during one of my visits home from Berkeley and she "got it." Quel's design was based on the tuple relational calculus—adding aggregates and grouping—and (in my opinion) it should have won the query language wars in the 1980s. SQL is less concise and an odd mix of the relational calculus (FROM) and algebra (JOIN, UNION). In my view, it won largely because Larry Ellison read the System R papers (one of which Oracle actually cites in its new-employee training!) and joined IBM in deciding to commercialize and popularize SQL.[1] As an example for the historical record, the following (see Figure 15.1) is a simple Quel join query from the original user manual that Mike himself wrote (as an Electronics Research Lab (ERL) memo [Stonebraker 1975]).

3. Ingres showed that one could create an efficient implementation of a declarative query language. Ingres included a query optimizer that (a) reordered joins based on their input sizes and connecting predicates and (b) accessed relations by picking from among the access paths supported by the available indexes. Interestingly, the Ingres optimizer worked together with the query executor incrementally—alternately picking off the next "best" part of the query, running it, seeing the result size, and then proceeding to the next step [Wong and Youssefi 1976]. System R took a more static, pre-compilation-based approach. System R is (rightly) famous for teaching the world how to use statistics and cost functions and dynamic programming to compile a query. However, today there is renewed interest in more runtime-oriented approaches—not unlike the early Ingres approach—that do not rely as heavily on a priori statistics and costing.

[1] It's a little-known fact that Teradata, the big gorilla system in high-performance relational data warehousing, was initially based on Quel before converting to SQL.

4. Ingres showed that one could build a full storage manager, including heap files and a variety of indexes (ISAM (Indexed Sequential Access Method) and hashing) that an optimizer could see and use, and with support for concurrent transactions and crash recovery. Being a smaller, university-based project, Ingres took a simpler approach to transactions (e.g., relation-level locks and sorting by relation name before locking to avoid deadlocks). But the Ingres system was nevertheless complete, delivered, and used! As a historical sidebar, one of my favorite forgotten Mike papers was a Communications of the Association for Computing Machinery (CACM) article ("B-Trees Re-examined" [Held and Stonebraker 1978]) in which Mike and Jerry Held explained why B+ trees might never work, due to then-unsolved challenges of dealing with concurrency control and recovery for dynamic index structures, and why static indexes like those used in Ingres were preferable.

5. Ingres was built for AT&T Unix and led to a full-featured system that was shared and used (again, think "prehistoric open source") by many folks at other universities and labs to actually store and query their data. As an example, when I landed in Wisconsin after leaving Berkeley, I discovered that University of Wisconsin-Madison economics professor Martin David and several of his colleagues were using Ingres to store and query U.S. government Survey of Income and Program Participation (SIPP) data [Flory et al. 1988]. As another example as a graduate student at CMU Rick Snodgrass used Ingres to store and query software events coming from parallel Cm* program executions for debugging [Snodgrass 1982], which is what inspired Rick's later temporal database career (see Chapter 12 for more on Mike and his open-source impact(s)).

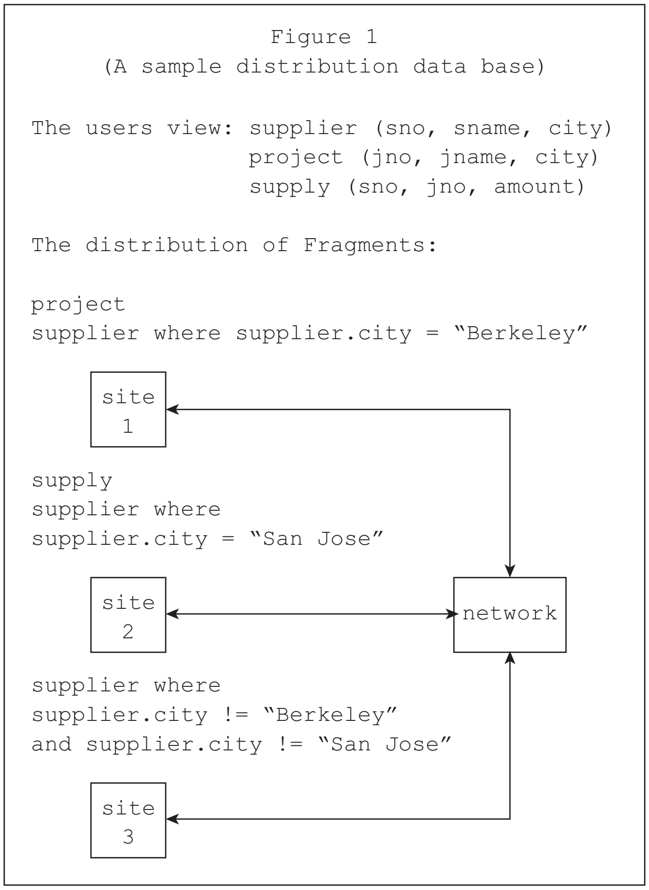
6. Ingres was used for teaching. Many students in the 1980s were able to get their hands on relational database technology solely because the Ingres system was so widely distributed and used in introductory database system classes. It helped to create a whole new generation of relational-database-savvy computer scientists!

7. Ingres was a sufficiently complete and solid software system that it became the basis for a very successful commercial RDBMS of its day, namely RTI (Relational Technology, Inc.) Ingres. See Chapter 24 by Paul Butterworth and Fred Carter for a very interesting discussion of the birth and subsequent development of Commercial Ingres, including how it began, what was changed from the university version, and how it evolved over time (and remained informed by university research).

Distributed Ingres: One Was Good, So More Must be Better

As I alluded to earlier, the next step for Mike, after Ingres, was to launch the Distributed Ingres project [Stonebraker and Neuhold 1977]. The distributed DBMS vision circa 1980—a vision shared by essentially the entire DB research community at that time—was of a single-image relational DBMS that could manage geographically distributed data (e.g., a database with sites in Berkeley, San Jose, and San Francisco) while making it look as though all the data was stored in a single, local relational DBMS. While the Distributed Ingres effort didn't lead to another widely shared "open source" system, it did produce a very interesting prototype as well as many technical results related to distributed query processing and transaction management. Major competing projects at the time were R* at IBM (i.e., distributed System R), SDD-1 at CCA (in Cambridge, MA), and a small handful of European efforts (e.g., SIRIUS). Among the contributions of the Distributed Ingres effort were the following.

1. Results by Bob Epstein, working with Mike (and also Gene), on distributed data storage and querying [Epstein et al. 1978]. Relations could be distributed in whole or as fragments with associated predicates. (See Figure 15.2, borrowed directly from Stonebraker and Neuhold [1977] in all of its late 1970s graphical glory.) The supply relation might be in San Jose, which might also store a fragment of supplier where supplier.city = "San Jose," with other supplier fragments being stored in Berkeley and elsewhere.) The project produced some of the first results, implemented in the prototype, on query processing in a geographically distributed setting. Distributed Ingres generalized the iterative runtime-oriented approach of Ingres and considered how and when to move data in order to run distributed queries involving multiple relations and fragments thereof.
2. Results by Dan Ries on concurrency control—particularly locking—in such an environment, based on simulations informed by the Distributed Ingres prototype [Ries and Stonebraker 1977a]. This was among the earliest work on distributed locking, considering centralized vs. distributed approaches to lock management in a distributed DBMS; studying the impact of lock granularity on such a system; and investigating alternative strategies for handling deadlocks (including methods based on prevention, preemption, and detection).
3. The birth of "shared nothing!" As the Distributed Ingres prototype effort was winding down, Mike threatened to create a new and different Ingres-



```
                    Figure 1
        (A sample distribution data base)

The users view: supplier (sno, sname, city)
                project (jno, jname, city)
                supply (sno, jno, amount)

The distribution of Fragments:

project
supplier where supplier.city = "Berkeley"
```

```
supply
supplier where
supplier.city = "San Jose"
```

```
supplier where
supplier.city != "Berkeley"
and supplier.city != "San Jose"
```

Figure 15.2 A simple example of an early (1977) distributed database. Source: [Stonebraker and Neuhold 1977].

based distributed DBMS prototype, which he named "MUFFIN" [erl-m79-28]. In this visionary 1979 ERL memo (which was never published as a paper nor realized as a system after all), Mike proposed to harness the combined power of a set of Ingres' ("D-CELLs") in parallel by organizing them in a shared-nothing fashion, much as was done later in the Teradata, Gamma, and GRACE database machines [DeWitt and Gray 1992].

4. Theoretical results by Dale Skeen on commit and recovery protocols for distributed DBMSs, again inspired by the Distributed Ingres context [Skeen and Stonebraker 1983]. This work provided a formal model for analyzing such

distributed protocols in the face of failures and applied the model to analyze existing protocols and to synthesize new ones as well. Both centralized and decentralized (e.g., quorum-based) schemes were considered.

5. Experimental, mostly simulation-based, results by yours truly, on concurrency control alternatives and their performance [Carey and Stonebraker 1984]. I undertook this work as a follow-on to what Dan Ries had started, since by the early 1980s a gazillion different approaches to concurrency control were starting to appear based on a diverse set of mechanisms (locks, timestamps, optimism followed by certification, and, orthogonally, versioning). This brilliant work provided some initial insight into the algorithmic design space's dimensions and their performance implications.

In spite of the database community's hopes and dreams, the homogeneous (single-site image) approach to distributed databases never took hold. It was just too early at that time. Nevertheless, the research results from those days have lived on in new contexts, such as parallel database systems and heterogeneous distributed databases. The Distributed Ingres project definitely bore a significant amount of (practical) fruit in the orchard of distributed database management.

Ingres: Moving Beyond Business Data

In the early Ingres era that I walked into, Mike was thinking about how to take Ingres to new levels of data support. This was something of a "between systems" study time in Ingres-land. Mike was looking around at a variety of domains: geographical information systems (one of the early motivating areas for Ingres), office automation systems, VLSI CAD (Very Large-Scale Integration Computer-Aided Design) systems (Berkeley was a hotbed in that field at the time), and so on. At the time, a number of folks in the DB research community were in a "Business data solved! What's next?" mindset (e.g., see the collection of papers in Katz [1982]). Noting that all these domains had serious data management needs, Mike and Ingres came to the rescue! To address these needs, Mike began to orchestrate his graduate students, many of them master's students, to tackle different facets of the looming "next-generation data management" problem. In usual Mike fashion, these projects were not simply paper designs, but designs accompanied by working prototypes based on experimentally changing the Ingres code base in different ways. Many interesting ideas and papers resulted, including the following.

1. A notion of "hypothetical relations" for Ingres [Stonebraker and Keller 1980]. The idea was to provide the ability to create a hypothetical branch of a rela-

tion: a differential version of a relation that could be updated and explored without impacting the source relation. The aim was to provide out-of-the-box database support for “what if” database use cases.

2. The addition of features to Ingres to enable it to be a viable platform for document data management [Stonebraker 1983a]. Features added in this line of work included support for variable-length strings (which Ingres didn’t initially have), a notion of ordered relations, various new substring operators for Quel, a new break operator for decomposing string fields, and a generalized concatenate operator (an aggregate function) for doing the reverse.

3. The addition of features to Ingres in support of CAD data management [Guttman and Stonebraker 1982]. The best-known result of that effort, by far, was due to my now famous (but soft-spoken) officemate Toni Guttman: the invention of the R-Tree index structure [Stonebraker and Guttman 1984]. Toni was studying the problem of storing VLSI CAD data from the emerging Caltech Mead-Conway VLSI design era. To simplify the problem of designing very large-scale integrated circuit chips, a new approach had emerged—thus filling the world with millions of rectangles to be managed! In order to index the 2-D geometry of a VLSI design, Toni generalized the ideas from Bayer’s B+ Tree structure, and R-Trees were thus born. Note that “R” stood for rectangle, and the first use case was for *actually* indexing rectangles. Of course, R-Trees are also widely used today to index other spatial objects by indexing their bounding boxes.

4. The foundation for what would later become a major feature of object-relational databases: support for user-defined extensions to Ingres based on Abstract Data Types (ADTs) [Ong et al. 1984], inspired by requirements for potentially adding spatial data support to Ingres. (For other views on Mike’s object-relational contributions, see Chapter 3 by Phil Bernstein, Chapter 12 by Mike Olson, Chapter 16 by Joe Hellerstein, and Chapter 6 by David DeWitt.) Developed by a pair of master’s students [Fogg 1982, Ong 1982], the ADT-Ingres prototype—which never became part of the public university release of Ingres—allowed Ingres users to declare new types whose instances could then be stored as attribute values in relational attributes declared to be “of” those types. Each such ADT definition needed to specify the name of the ADT, its upper bound sizes in binary and serialized string form, and string-to-binary (input) and binary-to-string (output) function names and their implementation file. New operators could also be defined, in order to operate on ADT instances, and they could involve either known primitive

```
DEFINE ADT (TYPENAME   IS “complex”,
            BYTESIN    IS 16,
            BYTESOUT   IS 27,
            INPUTFUNC  IS “tointernal”,
            OUTPUTFUNC IS “toexternal”,
            FILENAME   IS “/ja/guest/fogg/complex”)
```

```
CREATE ComplexNums (field1 = ADT:complex, field2 = f4
```

```
DEFINE ADTOP (OPNAME   IS “Magnitude”,
              FUNCNAME IS “magnitude”,
              FILENAME IS “/ja/guest/fogg/complex”,
              RESULT   IS f8,
              ARG      IS ADT:complex)
```

```
RANGE OF C IS ComplexNums,
RETRIEVE (C.field1)
        WHERE Magnitude C,field1 > Magnitude “3,4”
```

Figure 15.3 ADT-Ingres support for ADTs: Example of adding a data type to handle complex number data. Source: Ong et al. [1984].

types or (this or other) ADTs in their signatures. Based on lessons from that work, Mike himself developed a design for indexing on ADT values, and his paper on that topic [Stonebraker 1986b] eventually earned him a Test-of-Time award from the IEEE International Conference on Data Engineering. Again, for the historical record, the following sequence of Quel statements (see Figure 15.3) taken directly from [Ong et al. 1984] illustrates the nature of the ADT-Ingres support for ADTs using an example of adding a data type to handle complex number data.

5. Several Ingres-oriented takes on “AI and databases” or “expert database systems” (e.g., Stonebraker et al. [1982a], Stonebraker et al. [1983c], and Stonebraker [1985b]). This came in the form of several rule system designs—triggers and beyond—that Mike proposed as machinery that could be added to a relational DBMS for use in adding knowledge in addition to data for

advanced applications. Mike actually spent quite a few years looking in part at rules in databases from different angles.

6. A “Mike special” response to the database community’s desire to extend relational databases to support the storage and retrieval of complex objects. Instead of extending Ingres in the direction of supporting objects and identity—“pointer spaghetti!” in Mike’s opinion—he proposed adding a capability to “simply” add Quel queries to the list of things that one could store in an attribute of a relation in Ingres [Stonebraker 1984]. To refer to an object, one could write a query to fetch it, and could then store that query as a value of type Quel in an attribute of the referring object. (Scary stuff, in my view at the time: essentially “query spaghetti!”) Mike also proposed allowing such queries to be defined but customized per tuple by specifying the Quel query at the schema level but letting it be parameterized for a given tuple using values drawn from the tuple’s other attributes. (Better!)

Out of this last phase of “proof-of-concept efforts”—now informed by these experiences and inspired by their relative successes and failures—rose the Next Big Project for Mike: namely, Postgres [Stonebraker and Rowe 1986]. As I was living through the final phase of my thesis work, the “Postgres guys” started to arrive. Soon thereafter, this time with Mike working with Larry Rowe, another big Berkeley database system adventure (described in the next chapter!) got under way . . .

Looking back now, I realize that I was incredibly fortunate: An essentially random walk led me to Mike’s doorstep and his door was open when I got there. (That’ll teach you, Mike! Never, ever work with your door open . . .) The Ingres system was my educational on ramp to databases, and I got to be at Berkeley at a time when Mike and others around me were laying many of the foundations that would stand the test of time and be recognized for their significance now as part of the much-deserved presentation of the 2014 Turing Award to Mike.

Being in Ingres-land and being advised by Mike at that time has influenced my own career tremendously. I have since always sought to impart the same message about the database systems field in my teaching (database systems boring? not!!). I was also incurably infected with the “systems building bug” (due to double exposure to Mike and later to my Wisconsin colleague and mentor David DeWitt), which persists to this day.

16

Looking Back at Postgres

Joseph M. Hellerstein

Postgres was Michael Stonebraker's most ambitious project—his grand effort to build a one-size-fits-all database system. A decade long, it generated more papers, Ph.Ds., professors, and companies than anything else he did. It also covered more technical ground than any other single system he built. Despite the risk inherent in taking on that scope, Postgres also became the most successful software artifact to come out of Stonebraker's research groups, and his main contribution to open source. As of the time of writing, the open-source PostgreSQL system is the most popular, independent open-source database system in the world, and the fourth most popular database system in the world. Meanwhile, companies built from a Postgres base have generated a sum total of over $2.6 billion in acquisitions. By any measure, Stonebraker's Postgres vision resulted in enormous and ongoing impact.

Context

Stonebraker had enormous success in his early career with the Ingres research project at Berkeley (see Chapter 15), and the subsequent startup he founded with Larry Rowe and Eugene Wong: Relational Technology, Inc. (RTI).

As RTI was developing in the early 1980s, Stonebraker began working on database support for data types beyond the traditional rows and columns of Codd's original relational model. A motivating example current at the time was to provide database support for CAD tools for the microelectronics industry. In a 1983 paper, Stonebraker and students Brad Rubenstein and Antonin Guttman explained how that industry-needed support for "new data types such as polygons, rectangles, text strings, etc.," "efficient spatial searching," "complex integrity constraints," and "design hierarchies and multiple representations" of the same physical constructions [Stonebraker 1983a]. Based on motivations such as these, the group started work on indexing (including Guttman's influential R-trees for spatial indexing; [Guttman 1984]), and on adding Abstract Data Types (ADTs) to a relational database system. ADTs were a popular new programming language construct at the time, pioneered by subsequent Turing Award winner Barbara Liskov and explored in database application programming by Stonebraker's new collaborator, Larry Rowe. In a paper in SIGMOD Record in 1984 [Ong et al. 1984], Stonebraker and students James Ong and Dennis Fogg describe an exploration of this idea as an extension to Ingres called ADT-Ingres, which included many of the representational ideas that were explored more deeply—and with more system support—in Postgres.

Postgres: An Overview

As indicated by the name, Postgres was "Post-Ingres": a system designed to take what Ingres could do and go beyond. The signature theme of Postgres was the introduction of what he eventually called *Object-Relational* database features: support for object-oriented programming ideas within the data model and declarative query language of a database system. But Stonebraker also decided to pursue a number of other technical challenges in Postgres that were independent of object-oriented support, including active database rules, versioned data, tertiary storage, and parallelism.

Two papers were written on the design of Postgres: an early design in SIGMOD 1986 [Stonebraker and Rowe 1986] and a "mid-flight" design description in CACM 1991 [Stonebraker and Kemnitz 1991]. The Postgres research project ramped down in 1992 with the founding of Stonebraker's Illustra startup, which involved Stonebraker, key Ph.D. student Wei Hong, and then-chief programmer Jeff Meredith (see Chapter 25). Below, the features mentioned in the 1986 paper are marked with an asterisk (*); those from the 1991 paper that *were not* in the 1986 paper are marked with a plus sign ($^{+}$). Other goals listed below were tackled in the system and the research literature, but not in either design paper:

1. Supporting ADTs in a Database System
   (a) Complex Objects (i.e., nested or non-first-normal form data)*
   (b) User-Defined Abstract Data Types and Functions*
   (c) Extensible Access Methods for New Data Types*
   (d) Optimizer Handling of Queries with Expensive User-Defined Functions
2. Active Databases and Rules Systems (Triggers, Alerts)*
   (a) Rules implemented as query rewrites^{+}
   (b) Rules implemented as record-level triggers^{+}

3. Log-centric Storage and Recovery
   (a) Reduced-complexity recovery code by treating the log as data,* using non-volatile memory for commit status^{+}
   (b) No-overwrite storage and time travel queries+
4. Support for querying data on new deep storage technologies, notably optical disks*
5. Support for multiprocessors or custom processors*
6. Support for a variety of language models
   (a) Minimal changes to the relational model and support for declarative queries*
   (b) Exposure of "fast path" access to internal APIs, bypassing the query language^{+}
   (c) Multi-lingual support^{+}

Many of these topics were addressed in Postgres well before they were studied or reinvented by others; in many cases, Postgres was too far ahead of its time and the ideas caught fire later, with a contemporary twist.

We briefly discuss each of these Postgres contributions, and connections to subsequent work in computing.

Supporting ADTs in a Database System

The signature goal of Postgres was to support new Object-Relational features: the extension of database technology to support a combination of the benefits of relational query processing and object-oriented programming. Over time the Object-Relational ideas pioneered in Postgres have become standard features in most modern database systems.

A. Complex Objects

It is quite common for data to be represented in the form of nested bundles or "objects." A classic example is a purchase order, which has a nested set of products, quantities, and prices in the order. Relational modeling religion dictated that such data should be restructured and stored in an unnested format, using multiple flat entity tables (orders, products) with flat relationship tables (product_in_order) connecting them. The classic reason for this flattening is that it reduces duplication of data (a product being described redundantly in many purchase orders), which in turn avoids complexity or errors in updating all redundant copies. But in some cases, you want to store the nested representation, because it is natural for the application (say, a circuit layout engine in a CAD tool), and updates are rare. This data modeling debate is at least as old as the relational model.

A key aspect of Postgres was to “have your cake and eat it too” from a data modeling perspective: Postgres retained tables as its “outermost” data type but allowed columns to have “complex” types including nested tuples or tables. One of its more esoteric implementations, first explored in the ADT-Ingres prototype, was to allow a table-typed column to be specified declaratively as a query definition: “Quel as a data type” [Stonebraker et al. 1984a].

The “post-relational” theme of supporting both declarative queries and nested data has recurred over the years—often as an outcome of arguments about which is better. At the time of Postgres in the 1980s and 1990s, some of the object-oriented database groups picked up the idea and pursued it to a standard language called OQL, which has since fallen from use.

Around the turn of the millennium, declarative queries over nested objects became a research obsession for a segment of the database community in the guise of XML databases; the resulting XQuery language (headed by Don Chamberlin of SQL fame) owes a debt to the complex object support in Postgres’ PostQuel language. XQuery had broad adoption and implementation in industry, but never caught on with users. The ideas are being revisited yet again today in query language designs for the JSON data model popular in browser-based applications. Like OQL, these languages are in many cases an afterthought in groups that originally rejected declarative queries in favor of developer-centric programming (the “NoSQL” movement), only to want to add queries back to the systems post hoc. In the meantime, as Postgres has grown over the years (and shifted syntax from PostQuel to versions of SQL that reflect many of these goals), it has incorporated support for nested data like XML and JSON into a general-purpose DBMS without requiring any significant rearchitecting. The battle swings back and forth, but the Postgres approach of extending the relational framework with extensions for nested data has shown time and again to be a natural end-state for all parties after the arguments subside.

B. User-defined Abstract Data Types and Functions

In addition to offering nested types, Postgres pioneered the idea of having opaque, extensible Abstract Data Types (ADTs), which are stored in the database but not interpreted by the core database system. In principle, this was always part of Codd’s relational model: integers and strings were traditional, but really any atomic data types with predicates can be captured in the relational model. The challenge was to provide that mathematical flexibility in software. To enable queries that interpret and manipulate these objects, an application programmer needs to be able to register User-Defined Functions (UDFs) for these types with the system and be able to invoke those UDFs in queries. User-Defined Aggregate (UDA) functions are also desirable to summarize collections of these objects in queries. Postgres was the pioneering database system supporting these features in a comprehensive way.

Why put this functionality into the DBMS, rather than the applications above? The classic answer was the significant performance benefit of "pushing code to data," rather than "pulling data to code." Postgres showed that this is quite natural within a relational framework: It involved modest changes to a relational metadata catalog, and mechanisms to invoke foreign code, but the query syntax, semantics, and system architecture all worked out simply and elegantly.

Postgres was a bit ahead of its time in exploring this feature. In particular, the security implications of uploading unsafe code to a server were not an active concern in the database research community at the time. This became problematic when the technology started to get noticed in industry. Stonebraker commercialized Postgres in his Illustra startup, which was acquired by Informix in large part for its ability to support extensible "DataBlades" (extension packages) including UDFs. Informix's Postgres-based technology, combined with its strong parallel database offering, made Informix a significant threat to Oracle. Oracle invested heavily in negative marketing about the risks of Informix's ability to run "unprotected" user-defined C code. Some trace the demise of Informix to this campaign, although Informix's financial shenanigans (and subsequent federal indictment of its then-CEO) were certainly more problematic. Now, decades later, all the major database vendors support the execution of user-defined functions in one or more languages, using newer technologies to protect against server crashes or data corruption.

Meanwhile, the Big Data stacks of the 2000s—including the MapReduce phenomenon that gave Stonebraker and DeWitt such heartburn [DeWitt and Stonebraker 2008]–are a re-realization of the Postgres idea of user-defined code hosted in a query framework. MapReduce looks very much like a combination of software engineering ideas from Postgres combined with parallelism ideas from systems like Gamma and Teradata, with some minor innovation around mid-query restart for extreme-scalability workloads. Postgres-based start-ups Greenplum and Aster showed around 2007 that parallelizing Postgres could result in something much higher function and practical than MapReduce for most customers, but the market still wasn't ready for any of this technology in 2008. By now, in 2018, nearly every

Big Data stack primarily serves a workload of parallel SQL with UDFs—very much like the design Stonebraker and team pioneered in Postgres.

C. Extensible Access Methods for New Data Types

Relational databases evolved around the same time as B-trees in the early 1970s, and B-trees helped fuel Codd's dream of "physical data independence": B-tree indexes provide a level of indirection that adaptively reorganizes physical storage without requiring applications to change. The main limitation of B-trees and related structures was that they only support equality lookups and one-dimensional range queries. What if you have two-dimensional range queries of the kind typical in mapping and CAD applications? This problem was *au courant* at the time of Postgres, and the R-tree developed by Antonin Guttman in Stonebraker's group was one of the most successful new indexes developed to solve this problem in practice. Still, the invention of an index structure does not solve the end-to-end systems problem of DBMS support for multi-dimensional range queries. Many questions arise. Can you add an access method like R-trees to your DBMS easily? Can you teach your optimizer that said access method will be useful for certain queries? Can you get concurrency and recovery correct?

This was a very ambitious aspect of the Postgres agenda: a software architecture problem affecting most of a database engine, from the optimizer to the storage layer and the logging and recovery system. R-trees became a powerful driver and the main example of the elegant extensibility of Postgres' access method layer and its integration into the query optimizer. Postgres demonstrated—in an opaque ADT style—how to register an abstractly described access method (the R-tree, in this case), and how a query optimizer could recognize an abstract selection predicate (a range selection in this case) and match it to that abstractly described access method. Questions of concurrency control were less of a focus in the original effort: The lack of a unidimensional ordering on keys made B-tree-style locking inapplicable.[1]

PostgreSQL today leverages both the original software architecture of extensible access methods (it has B-tree, GiST, SP-GiST, and Gin indexes) and the extensibility and high concurrency of the Generalized Search Tree (GiST) interface as well.

[1] The Postgres challenge of extensible access methods inspired one of my first research projects at the end of graduate school: the Generalized Search Trees (GiST) [Hellerstein et al. 1995] and subsequent notion of Indexability theory [Hellerstein et al. 2002]. I implemented GiST in Postgres during a postdoc semester, which made it even easier to add new indexing logic in Postgres. Marcel Kornacker's thesis at Berkeley solved the difficult concurrency and recovery problems raised by extensible indexing in GiST in a templated way [Kornacker et al. 1997].

GiST indexes power the popular PostgreSQL-based PostGIS geographic information system; Gin indexes power PostgreSQL's internal text indexing support.

D. Optimizer Handling of Queries with Expensive UDFs

In traditional query optimization, the challenge was generally to minimize the amount of tuple-flow (and hence I/O) you generate in processing a query. This meant that operators that filter tuples (selections) are good to do early in the query plan, while operators that can generate new tuples (join) should be done later. As a result, query optimizers would "push" selections below joins and order them arbitrarily, focusing instead on cleverly optimizing joins and disk accesses. UDFs changed this: if you have expensive UDFs in your selections, the order of executing UDFs can be critical to optimizing performance. Moreover, if a UDF in a selection is really time consuming, it's possible that it should happen *after* joins (i.e., selection "pullup"). Doing this optimally complicated the optimizer space.

I took on this problem as my first challenge in graduate school and it ended up being the subject of both my M.S. with Stonebraker at Berkeley and my Ph.D. at Wisconsin under Jeff Naughton, with ongoing input from Stonebraker. Postgres was the first DBMS to capture the costs and selectivities of UDFs in the database catalog. We approached the optimization problem by coming up with an optimal ordering of selections, and then an optimal interleaving of the selections along the branches of each join tree considered during plan search. This allowed for an optimizer that maintained the textbook dynamic programming architecture of System R, with a small additional sorting cost to get the expensive selections ordered properly.[2]

The expensive function optimization feature was disabled in the PostgreSQL source trees early on, in large part because there weren't compelling use cases at that time for expensive user-defined functions.[3] The examples we used revolved around image processing and are finally becoming mainstream data processing tasks in 2018. Of course, today in the era of Big Data and machine learning workloads, expensive functions have become quite common, and I expect this problem to return to the fore. Once again, Postgres was well ahead of its time.

[2] When I started grad school, this was one of three topics that Stonebraker wrote on the board in his office as options for me to think about for a Ph.D. topic. I think the second was function indexing, but I cannot remember the third.

[3] Ironically, my code from grad school was fully deleted from the PostgreSQL source tree by a young open-source hacker named Neil Conway, who some years later started a Ph.D. with me at UC Berkeley and is now one of Stonebraker's Ph.D. grandchildren.

Active Databases and Rule Systems

The Postgres project began at the tail end of the AI community's interest in rule-based programming as a way to represent knowledge in "expert systems." That line of thinking was not successful; many say it led to the much discussed "AI winter" that persisted through the 1990s.

However, rule programming persisted in the database community in two forms. The first was theoretical work around declarative logic programming using Datalog. This was a bugbear of Stonebraker's; he really seemed to hate the topic and famously criticized it in multiple "community" reports over the years.[4] The second database rules agenda was pragmatic work on what was eventually dubbed Active Databases and Database Triggers, which evolved to be a standard feature of relational databases. Stonebraker characteristically voted with his feet to work on the more pragmatic variant.

Stonebraker's work on database rules began with Eric Hanson's Ph.D., which initially targeted Ingres but quickly transitioned to the new Postgres project. It expanded to the Ph.D. work of Spyros Potamianos on PRS2: Postgres Rules System 2. A theme in both implementations was the potential to implement rules in two different ways. One option was to treat rules as query rewrites, reminiscent of the work on rewriting views that Stonebraker pioneered in Ingres. In this scenario, a rule logic of "on condition then action" is recast as "on query then rewrite to a modified query and execute it instead." For example, a query like "append a new row to Mike's list of awards" might be rewritten as "raise Mike's salary by 10%." The other option was to implement a more physical "on condition then action," checking conditions at a row level by using locks inside the database. When such locks were encountered, the result was not to wait (as in traditional concurrency control), but to execute the associated action.[5]

[4] Datalog survived as a mathematical foundation for declarative languages and has found application over time in multiple areas of computing including software-defined networks and compilers. Datalog is declarative querying "on steroids" as a fully expressive programming model. I was eventually drawn into it as a natural design choice and have pursued it in a variety of applied settings outside of traditional database systems.

[5] The code for row-level rules in PRS2 was notoriously tricky. A bit of searching in the Berkeley Postgres archives unearthed the following source code comment—probably from Spyros Potamianos—in Postgres version 3.1, circa 1991:

```
* DESCRIPTION:
* Take a deeeeeeep breath & read. If you can avoid hacking the code
* below (i.e. if you have not been ‘‘volunteered'' by the boss to do this
* dirty job) avoid it at all costs. Try to do something less dangerous
* for your (mental) health. Go home and watch horror movies on~TV.
```

In the end, neither the query rewriting scheme nor the row-level locking scheme was declared a “winner” for implementing rules in Postgres—both were kept in the released system. Eventually all of the rules code was scrapped and rewritten in PostgreSQL, but the current source still retains both the notions of per-statement and per-row triggers.

The Postgres rules systems were very influential in their day and went “head to head” with research from IBM’s Starburst project and MCC’s HiPac project. Today, “triggers” are part of the SQL standard and implemented in many of the major database engines. They are used somewhat sparingly, however. One problem is that this body of work never overcame the issues that led to AI winter: The interactions within a pile of rules can become untenably confusing as the rule set grows even modestly. In addition, triggers still tend to be relatively time consuming in practice, so database installations that have to run fast tend to avoid the use of triggers. But there has been a cottage industry in related areas like materialized view maintenance, Complex Event Processing, and stream queries, all of which are in some way extensions of ideas explored in the Postgres rules systems.

Log-centric Storage and Recovery

Stonebraker described his design for the Postgres storage system this way:

> “When considering the POSTGRES storage system, we were guided by a missionary zeal to do something different. All current commercial systems use a storage manager with a write-ahead log (WAL), and we felt that this technology was well understood. Moreover, the original Ingres prototype from the 1970s used a similar storage manager, and we had no desire to do another implementation.” [Stonebraker and Kemnitz 1991]

While this is cast as pure intellectual restlessness, there were technical motivations for the work as well. Over the years, Stonebraker repeatedly expressed distaste for the complex write-ahead logging schemes pioneered at IBM and Tandem for database recovery. One of his core objections was based on a software engineering intuition that nobody should rely upon something that complicated—especially for functionality that would only be exercised in rare, critical scenarios after a crash.

```
* Read some Lovecraft. Join the Army. Go and spend a few nights in
* people's park. Commit suicide ...
* Hm, you keep reading, eh? Oh, well, then you deserve what you~get.
* Welcome to the gloomy labyrinth of the tuple level rule system, my
* poor hacker...
```

The Postgres storage engine unified the notion of primary storage and historical logging into a single, simple disk-based representation. At base, the idea was to keep each record in the database in a linked list of versions stamped with transaction IDs—in some sense, this is "the log as data" or "the data as a log," depending on your point of view. The only additional metadata required is a list of committed transaction IDs and wall-clock times. This approach simplifies recovery enormously since there's no "translating" from a log representation back to a primary representation. It also enables "time travel" queries: You can run queries "as of" some wall-clock time and access the versions of the data that were committed at that time. The original design of the Postgres storage system—which reads very much as if Stonebraker wrote it in one creative session of brainstorming—contemplated a number of efficiency problems and optimizations to this basic scheme, along with some wet-finger analyses of how performance might play out [Stonebraker 1987]. The resulting implementation in Postgres was somewhat simpler.

Stonebraker's idea of "radical simplicity" for transactional storage was deeply countercultural at the time when the database vendors were differentiating themselves by investing heavily in the machinery of high-performance transaction processing. Benchmark winners at the time achieved high performance and recoverability via highly optimized, complex write-ahead logging systems. Once they had write-ahead logs working well, the vendors also began to innovate on follow-on ideas such as transactional replication based on log shipping, which would be difficult in the Postgres scheme. In the end, the Postgres storage system never excelled on performance; versioning and time travel were removed from PostgreSQL over time and replaced by write-ahead logging.[6] But the time-travel functionality was interesting and remained unique. Moreover, Stonebraker's ethos regarding

[6] Unfortunately, PostgreSQL still isn't particularly fast for transaction processing: Its embrace of write-ahead logging is somewhat half-hearted. Oddly, the PostgreSQL team kept much of the storage overhead of Postgres tuples to provide multiversion *concurrency control*, something that was never a goal of the Berkeley Postgres project. The result is a storage system that can emulate Oracle's snapshot isolation with a fair bit of extra I/O overhead, but one that does not support Stonebraker's original idea of time travel or simple recovery.

Mike Olson notes that his original intention was to replace the Postgres B-tree implementation with his own B-tree implementation from the BerkeleyDB project, which developed at Berkeley during the Postgres era. But Olson never found the time. When Berkeley DB got transactional support years later at Sleepycat Corp., Olson tried to persuade the (then-) PostgreSQL community to adopt it for recovery, in place of no-overwrite. They declined; there was a hacker on the project who desperately wanted to build an multi-version currency control system, and as that hacker was willing to do the work, he won the argument.

simple software engineering for recovery has echoes today both in the context of NoSQL systems (which choose replication rather than write-ahead logging) and main-memory databases (which often use multi-versioning and compressed commit logs). The idea of versioned relational databases and time-travel queries are still relegated to esoterica today, popping up in occasional research prototypes and minor open-source projects. It is an idea that is ripe for a comeback in our era of cheap storage and continuously streaming data.

Queries over New Deep Storage Technologies

In the middle of the Postgres project, Stonebraker signed on as a co-principal investigator on a large grant for digital earth science called Project Sequoia. Part of the grant proposal was to handle unprecedented volumes of digital satellite imagery requiring up to 100 terabytes of storage, far more data than could be reasonably stored on magnetic disks at the time. The center of the proposed solution was to explore the idea of a DBMS (namely Postgres) facilitating access to near-line "tertiary" storage provided by robotic "jukeboxes" for managing libraries of optical disks or tapes.

A couple different research efforts came out of this. One was the Inversion file system: an effort to provide a UNIX filesystem abstraction *above* an RDBMS. In an overview paper for Sequoia, Stonebraker described this in his usual cavalier style as "a straightforward exercise" [Stonebraker 1995]. In practice, this kept Stonebraker student (and subsequent Cloudera founder) Mike Olson busy for a couple years, and the final result was not exactly straightforward [Olson 1993], nor did it survive in practice.[7]

The other main research thrust on this front was the incorporation of tertiary storage into a more typical relational database stack, which was the subject of Sunita Sarawagi's Ph.D. thesis. The main theme was to change the scale at which you think about managing space (i.e., data in storage and the memory hierarchy) and time (coordinating query and cache scheduling to minimize undesirable I/Os). One of the key issues in that work was to store and retrieve large multidimensional

Although the PostgreSQL storage engine is slow, that is not intrinsic to the system. The Greenplum fork of PostgreSQL integrated an interesting alternative high-performance compressed storage engine. It was designed by Matt McCline—a veteran of Jim Gray's team at Tandem. It also did not support time travel.

[7] Some years after Inversion, Bill Gates tilted against this same windmill with WinFS, an effort to rebuild the most widely used filesystem in the world over a relational database backend. WinFS was delivered in developer releases of Windows but never made it to market. Gates later called this his greatest disappointment at Microsoft.

arrays in tertiary storage—echoing work in multidimensional indexing, the basic ideas included breaking up the array into chunks and storing chunks together that are fetched together—including replicating chunks to enable multiple physical "neighbors" for a given chunk of data. A second issue was to think about how disk becomes a cache for tertiary storage. Finally, query optimization and scheduling had to take into account the long load times of tertiary storage and the importance of "hits" in the disk cache—this affects both the plan chosen by a query optimizer, and the time at which that plan is scheduled for execution.

Tape and optical disk robots are not widely used at present. But the issues of tertiary storage are very prevalent in the cloud, which has deep storage hierarchies in 2018: from attached solid-state disks to reliable disk-like storage services (e.g., AWS EBS) to archival storage (e.g., AWS S3) to deep storage (e.g., AWS Glacier). It is still the case today that these storage tiers are relatively detached, and there is little database support for reasoning about storage across these tiers. I would not be surprised if the issues explored on this front in Postgres are revisited in the near term.

Support for Multiprocessors: XPRS

Stonebraker never architected a large parallel database system, but he led many of the motivating discussions in the field. His "Case for Shared Nothing" paper [Stonebraker 1986d] documented the coarse-grained architectural choices in the area; it popularized the terminology used by the industry and threw support behind shared-nothing architectures like those of Gamma and Teradata, which were rediscovered by the Big Data crowd in the 2000s.

Ironically, Stonebraker's most substantive contribution to the area of parallel databases was a "shared memory" architecture called XPRS, which stood for eXtended Postgres on RAID and Sprite. XPRS was the "Justice League" of Berkeley systems in the early 1990s: a brief combination of Stonebraker's Postgres system, John Ousterhout's Sprite distributed OS, and Dave Patterson's and Randy Katz's RAID storage architectures. Like many multi-faculty efforts, the execution of XPRS was actually determined by the grad students who worked on it. The primary contributor ended up being Wei Hong, who wrote his Ph.D. thesis on parallel query optimization in XPRS. Hence, the main contribution of XPRS to the literature and industry was parallel query optimization, with no real consideration of issues involving RAID or Sprite.[8]

[8] Of the three projects, Postgres and RAID both had enormous impact. Sprite is best remembered for Mendel Rosenblum's Ph.D. thesis on Log Structured File Systems (LFS), which had nothing of

In principle, parallelism "blows up" the plan space for a query optimizer by making it multiply the traditional choices made during query optimization (data access, join algorithms, join orders) against all possible ways of parallelizing each choice. The basic idea of what Stonebraker called "The Wei Hong Optimizer" was to cut the problem in two: Run a traditional single-node query optimizer in the style of System R, and then "parallelize" the resulting single-node query plan by scheduling the degree of parallelism and placement of each operator based on data layouts and system configuration. This approach is heuristic, but it makes parallelism an additive cost to traditional query optimization, rather than a multiplicative cost.

Although "The Wei Hong Optimizer" was designed in the context of Postgres, it became the standard approach for many of the parallel query optimizers in industry.

Support for a Variety of Language Models

One of Stonebraker's recurring interests since the days of Ingres was the programmer API to a database system. In his Readings in Database Systems series, he frequently included work like Carlo Zaniolo's GEM language as important topics for database system aficionados to understand. This interest in language undoubtedly led him to partner up with Larry Rowe on Postgres, which in turn deeply influenced the design of the Postgres data model and its Object-Relational approach. Their work focused largely on data-centric applications they saw in the commercial realm, including both business processing and emerging applications like CAD/CAM computer-aided design (and manufacturing) and Geographic Information System (GIS).

One issue that was forced upon Stonebraker at the time was the idea of "hiding" the boundary between programming language constructs and database storage. Various competing research projects and companies exploring Object-Oriented Databases (OODBs) were targeting the so-called "impedance mismatch" between imperative object-oriented programming languages like Smalltalk, C++, and Java, and the declarative relational model. The OODB idea was to make programming language objects be optionally marked "persistent," and handled automatically by

note to do with distributed operating systems. All three projects involved new ideas for disk storage beyond mutating single copies in place. LFS and the Postgres storage manager are rather similar, both rethinking logs as primary storage, and requiring expensive background reorganization. I once gently probed Stonebraker about rivalries or academic scoops between LFS and Postgres, but I never got any good stories out of him. Maybe it was something in the water in Berkeley at the time.

an embedded DBMS. Postgres supported storing nested objects and ADTs, but its relational-style declarative query interface meant that each round trip to the database was unnatural for the programmer (requiring a shift to declarative queries) and expensive to execute (requiring query parsing and optimization). To compete with the OODB vendors, Postgres exposed a so-called "Fast Path" interface: basically, a C/C++ API to the storage internals of the database. This enabled Postgres to be moderately performant in academic OODB benchmarks, but never really addressed the challenge of allowing programmers in multiple languages to avoid the impedance mismatch problem. Instead, Stonebraker branded the Postgres model as "Object-Relational" and simply sidestepped the OODB workloads as a "zero-billion-dollar" market. Today, essentially all commercial relational database systems are "Object-Relational" database systems.

This proved to be a sensible decision. Today, none of the OODB products exist in their envisioned form, and the idea of "persistent objects" in programming languages has largely been discarded. By contrast, there is widespread usage of object-relational mapping layers (fueled by early efforts like Java Hibernate and Ruby on Rails) that allow declarative databases to be tucked under nearly any imperative object-oriented programming language as a library, in a relatively seamless way. This application-level approach is different than both OODBs and Stonebraker's definition of Object-Relational DBs. In addition, lightweight persistent key-value stores have succeeded as well, in both non-transactional and transactional forms. These were pioneered by Stonebraker's Ph.D. student Margo Seltzer, who wrote BerkeleyDB as part of her Ph.D. thesis at the same time as the Postgres group, which presaged the rise of distributed "NoSQL" key-value stores like Dynamo, MongoDB, and Cassandra.

Software Impact

Open Source

Postgres was always an open-source project with steady releases, but in its first many years it was targeted at usage in research, not in production.

As the Postgres research project was winding down, two students in Stonebraker's group—Andrew Yu and Jolly Chen—modified the system's parser to accept an extensible variant of SQL rather than the original PostQuel language. The first Postgres release supporting SQL was Postgres95; the next was dubbed PostgreSQL.

A set of open-source developers became interested in PostgreSQL and "adopted" it even as the rest of the Berkeley team was moving on to other interests. Over time, the core developers for PostgreSQL have remained fairly stable, and the open-source project has matured enormously. Early efforts focused on code stability and user-facing features, but over time the open-source community made significant modifications and improvements to the core of the system as well, from the optimizer to the access methods and the core transaction and storage system. Since the mid-1990s, very few of the PostgreSQL internals came out of the academic group at Berkeley—the last contribution may have been my GiST implementation in the latter half of the 1990s—but even that was rewritten and cleaned up substantially by open-source volunteers (from Russia, in that case). The open source community around PostgreSQL deserves enormous credit for running a disciplined process that has soldiered on over decades to produce a remarkably high-impact and long-running project.

While many things have changed in 25 years, the basic architecture of PostgreSQL remains quite similar to the university releases of Postgres in the early 1990s, and developers familiar with the current PostgreSQL source code would have little trouble wandering through the Postgres3.1 source code (c. 1991). Everything from source code directory structures to process structures to data structures remain remarkably similar. The code from the Berkeley Postgres team had excellent bones.

PostgreSQL today is without question the most high-function open-source DBMS, supporting features that are often missing from commercial products. It is also (according to one influential rankings site) the most popular widely used independent open-source database in the world[9] and its impact continues to grow: In 2017 it was the fastest-growing database system in the world in popularity.[10] PostgreSQL is used across a wide variety of industries and applications, which is perhaps not surprising given its ambition of broad functionality; the PostgreSQL website catalogs some of the uses at http://www.postgresql.org/about/users/. (Last accessed January 22, 2018.)

Heroku is a cloud SaaS provider that is now part of Salesforce. Postgres was adopted by Heroku in 2010 as the default database for its platform. Heroku chose

[9] According to DB Engines (http://db-engines.com/en/ranking. Last accessed January 22, 2018), PostgreSQL today is the fourth most popular DBMS in the world, after Oracle, MySQL and MS SQL Server, all of which are corporate offerings (MySQL was acquired by Oracle many years ago). See http://db-engines.com/en/ranking_definition (Last accessed January 22, 2018) for a discussion of the rules for this ranking.

[10] "PostgreSQL is the DBMS of the Year 2017," *DB Engines* blog, January 2, 2018. http://db-engines.com/en/blog_post/76. Last accessed January 18, 2018.

Postgres because of its operational reliability. With Heroku's support, more major application frameworks such as Ruby on Rails and Python for Django began to recommend Postgres as their default database.

PostgreSQL today supports an extension framework that makes it easy to add additional functionality to the system via UDFs and related modifications. There is now an ecosystem of PostgreSQL extensions—akin to the Illustra vision of DataBlades, but in open source. Some of the more interesting extensions include the Apache MADlib library for machine learning in SQL, and the Citus library for parallel query execution.

One of the most interesting open-source applications built over Postgres is the PostGIS Geographic Information System, which takes advantage of many of the features in Postgres that originally inspired Stonebraker to start the project.

Commercial Adaptations

PostgreSQL has long been an attractive starting point for building commercial database systems, given its permissive ope- source license, its robust codebase, its flexibility, and breadth of functionality. Summing the acquisition prices listed below, Postgres has led to over $2.6 billion in acquisitions.[11] Many of the commercial efforts that built on PostgreSQL have addressed what is probably its key limitation: the ability to scale out to a parallel, shared-nothing architecture.[12]

1. Illustra was Stonebraker's second major start-up company, founded in 1992, seeking to commercialize Postgres as RTI had commercialized Ingres.[13] The

[11] Note that this is a measure in real transaction dollars and is much more substantial than the values often thrown around in high tech. Numbers in the billions are often used to describe estimated value of stock holdings but are often inflated by 10× or more against contemporary value in hopes of future value. The transaction dollars of an acquisition measure the actual market value of the company at the time of acquisition. It is fair to say that Postgres has generated more than $2.6 billion of real commercial value.

[12] Parallelizing PostgreSQL requires a fair bit of work, but is eminently doable by a small, experienced team. Today, industry-managed open-source forks of PostgreSQL such as Greenplum and CitusDB offer this functionality. It is a shame that PostgreSQL wasn't parallelized in a true open-source way much earlier. If PostgreSQL had been extended with shared-nothing features in open source in the early 2000s, it is quite possible that the open-source Big Data movement would have evolved quite differently and more effectively.

[13] Illustra was actually the third name proposed for the company. Following the painterly theme established by Ingres, Illustra was originally called Miró. For trademark reasons the name was changed to Montage, but that also ran into trademark problems.

founding team included some of the core Postgres team including recent Ph.D. alumnus Wei Hong and then-chief programmer Jeff Meredith, along with Ingres alumni Paula Hawthorn and Michael Ubell. Postgres M.S. student Mike Olson joined shortly after the founding, and I worked on the Illustra handling of optimizing expensive functions as part of my Ph.D. work. There were three main efforts in Illustra: to extend SQL92 to support user-defined types and functions as in PostQuel, to make the Postgres code base robust enough for commercial use, and to foster the market for extensible database servers via examples of "DataBlades," domain-specific plug-in components of data types and functions (see Chapter 25). Illustra was acquired by Informix in 1997 for an estimated $400M,[14] and its DataBlade architecture was integrated into a more mature Informix query processing codebase as Informix Universal Server.

2. Netezza was a startup founded in 1999, which forked the PostgreSQL codebase to build a high-performance parallel query processing engine on custom field-programmable-gate-array-based hardware. Netezza was quite successful as an independent company and had its IPO in 2007. It was eventually acquired by IBM, with a value of $1.7B.[15]

3. Greenplum was the first effort to offer a shared-nothing parallel, scale-out version of PostgreSQL. Founded in 2003, Greenplum forked from the public PostgreSQL distribution, but maintained the APIs of PostgreSQL to a large degree, including the APIs for user-defined functions. In addition to parallelization, Greenplum extended PostgreSQL with an alternative high-performance compressed columnar storage engine and a parallelized rule-driven query optimizer called Orca. Greenplum was acquired by EMC in 2010 for an estimated $300M; in 2012, EMC consolidated Greenplum into its subsidiary, Pivotal. In 2015, Pivotal chose to release Greenplum and Orca back into open source. One of the efforts at Greenplum that leveraged its Postgres API was the MADlib library for machine learning in SQL; MADlib runs single-threaded in PostgreSQL and in parallel over Greenplum. MADlib lives on today as an Apache project. Another interesting open-source project based

[14] "Informix acquires Illustra for complex data management," *Federal Computer Week*, January 7, 1996. http://fcw.com/Articles/1996/01/07/Informix-acquires-Illustra-for-complex-data-management.aspx. Last accessed January 22, 2018.

[15] http://en.wikipedia.org/wiki/Netezza. Last accessed January 22, 2018.

on Greenplum is Apache HAWQ, a Pivotal design that runs the "top half" of Greenplum (i.e., the parallelized PostgreSQL query processor and extensibility APIs) in a decoupled fashion over Big Data stores such as the Hadoop File System.

4. EnterpriseDB was founded in 2004 as an open-source-based business, selling PostgreSQL in both a vanilla and enhanced edition with related services for enterprise customers. A key feature of the enhanced EnterpriseDB Advanced Server is a set of database compatibility features with Oracle to allow application migration off of Oracle.
5. Aster Data was founded in 2005 by two Stanford students to build a parallel engine for analytics. Its core single-node engine was based on PostgreSQL. Aster focused on queries for graphs and on analytics packages based on UDFs that could be programmed with either SQL or MapReduce interfaces. Aster Data was acquired by Teradata in 2011 for $263M.[16] While Teradata never integrated Aster into its core parallel database engine, it still maintains Aster as a standalone product for use cases outside the core of Teradata's warehousing market.
6. ParAccel was founded in 2006, selling a shared-nothing parallel version of PostgreSQL with column-oriented, shared-nothing storage. ParAccel enhanced the Postgres optimizer with new heuristics for queries with many joins. In 2011, Amazon invested in ParAccel, and in 2012 announced AWS Redshift, a hosted data warehouse as a service in the public cloud based on ParAccel technology. In 2013, ParAccel was acquired by Actian (which also had acquired Ingres) for an undisclosed amount—meaning it was not a material expense for Actian. Meanwhile, AWS Redshift has been an enormous success for Amazon—for many years it was the fastest-growing service on AWS, and many believe it is poised to put long-time data warehousing products like Teradata and Oracle Exadata out of business. In this sense, Postgres may achieve its ultimate dominance in the cloud.
7. CitusDB was founded in 2010 to offer a shared-nothing parallel implementation of PostgreSQL. While it started as a fork of PostgreSQL, as of 2016 CitusDB is implemented via public PostgreSQL extension APIs and can be

[16] "Big Pay Day For Big Data. Teradata Buys Aster Data For $263 Million," *TechCrunch*, May 3, 2011. (http://techcrunch.com/2011/03/03/teradata-buys-aster-data-263-million/. Last accessed January 22, 2018.

installed into a vanilla PostgreSQL installation. Also, as of 2016, the CitusDB extensions are available in open source.

Lessons

You can draw a host of lessons from the success of Postgres, a number of them defiant of conventional wisdom.

The highest-order lesson I draw comes from the fact that Postgres defied Fred Brooks' "Second System Effect." Brooks argued that designers often follow up on a successful first system with a second system that fails due to being overburdened with features and ideas. Postgres was Stonebraker's second system, and it was certainly chock full of features and ideas. Yet the system succeeded in prototyping many of the ideas while delivering a software infrastructure that carried a number of the ideas to a successful conclusion. This was not an accident—at base, Postgres was *designed for extensibility*, and that design was sound. With extensibility as an architectural core, it is possible to be creative and stop worrying so much about discipline: You can try many extensions and let the strong succeed. Done well, the "second system" is not doomed; it benefits from the confidence, pet projects, and ambitions developed during the first system. This is an early architectural lesson from the more "server-oriented" database school of software engineering, which defies conventional wisdom from the "component oriented" operating systems school of software engineering.

Another lesson is that a broad focus—"one size fits many"—can be a winning approach for both research and practice. To coin some names, "MIT Stonebraker" made a lot of noise in the database world in the early 2000s that "one size doesn't fit all." Under this banner he launched a flotilla of influential projects and startups, but none took on the scope of Postgres. It seems that "Berkeley Stonebraker" defies the later wisdom of "MIT Stonebraker," and I have no issue with that.[17] Of course there's wisdom in the "one size doesn't fit all" motto (it's always possible to find modest markets for custom designs!), but the success of "Berkeley Stonebraker's" signature system—well beyond its original intents—demonstrates that a broad majority of database problems can be solved well with a good general-purpose architecture. Moreover, the design of that architecture is a technical challenge and accomplishment in its own right. In the end—as in most science and engineering debates—there isn't only one good way to do things. Both Stonebrakers have

[17] As Emerson said, "a foolish consistency is the hobgoblin of little minds."

lessons to teach us. But at the base, I'm still a fan of the broader agenda that "Berkeley Stonebraker" embraced.

A final lesson I take from Postgres is the unpredictable potential that can come from open-sourcing your research. In his Turing talk, Stonebraker speaks about the "serendipity" of PostgreSQL succeeding in open source, largely via people outside Stonebraker's own sphere. It's a wonderfully modest quote:

> [A] pick-up team of volunteers, none of whom have anything to do with me or Berkeley, have been shepherding that open-source system ever since 1995. The system that you get off the web for Postgres comes from this pick-up team. It is open source at its best and I want to just mention that I have nothing to do with that and that collection of folks we all owe a huge debt of gratitude to.[18]

I'm sure all of us who have written open source would love for that kind of "serendipity" to come our way. But it's not all serendipity—the roots of that good luck were undoubtedly in the ambition, breadth, and vision that Stonebraker had for the project, and the team he mentored to build the Postgres prototype. If there's a lesson there, it might be to "do something important and set it free." It seems to me (to use a Stonebrakerism) that you can't skip either part of that lesson.

Acknowledgments

I'm indebted to my old Postgres buddies Wei Hong, Jeff Meredith, and Mike Olson for their remembrances and input, and to Craig Kerstiens for his input on modern-day PostgreSQL.

[18] Transcript by Jolly Chen, http://www.postgresql.org/message-id/A4BA155B-E762-4022-B7D1-6F4791014851@chenfamily.com. Last accessed January 22, 2018.

17

Databases Meet the Stream Processing Era

Magdalena Balazinska, Stan Zdonik

Origins of the Aurora and Borealis Projects

In the early 2000s, sensors and sensor networks became an important focus in the systems, networking, and database communities. The decreasing cost of hardware was creating a technology push, while the excitement about pervasive computing, exemplified by projects such as MIT's "Project Oxygen,"[1] was in part responsible for an application pull. In most areas, dramatic improvements in software were needed to support emerging applications built on top of sensor networks, and this need was stimulating research in all these three fields.

In the database community, as many observed, traditional database management systems (DBMSs) were ill-suited for supporting this new type of *stream-processing application*. Traditional DBMSs were designed for business data, which is stored on disk and modified by transaction-processing applications. In the new stream processing world, however, data sources such as sensors or network monitors were instead pushing data to the database. Applications[2] that needed to process data streams wanted to receive alerts when interesting events occurred. This switch from active users querying a passive database to passive users receiving alerts from an active database [Abadi et al. 2003a] was a fundamental paradigm

[1] http://oxygen.csail.mit.edu. Last accessed May 16, 2018.

[2] "It's largely forgotten now but RFID (radio frequency identification) tagging was then a huge area of investment. Walmart and the USAF had purchased a lot of RFID hardware and software to manage their supply chains in real time, initially at the palette level and planned eventually at the item level. Those streams of RFID telemetry data cried out for a good application development platform like StreamBase's.—John Partridge, StreamBase Co-Founder and VP Business Development

shift in the database community. The other paradigm shift was that data no longer resided on disk but was being continuously pushed by applications at high, and often variable, rates.

At the time when the above technology and application changes were occurring, Mike Stonebraker was moving from the West Coast to the East Coast. He left Berkeley in 2000 and joined MIT in 2001. At that time, MIT had no database faculty and Mike found himself collaborating with systems and networking groups. Some of the closest database groups were at Brown University and Brandeis University, and Mike would go on to create a successful and long-term collaboration across the three institutions. The collaboration would span multiple research trends and projects (starting with stream processing and the Aurora and Borealis projects) and, following Mike's model (see Chapter 7), would generate several purpose-built DBMS-engine startup companies (starting with StreamBase Systems).

Mike and his team, together with others in the database community, identified the following key limitations of traditional DBMSs with regard to stream processing applications.

- *Insufficient data ingest rates*: When data streams from multiple sources continuously, traditional DBMSs struggle to write the data to disk before making it available for processing.
- *Disk orientation*: In traditional DBMSs, data is stored on disk and only cached in memory as a result of query activity. Stream processing applications need to process data fast. They need the data to stay in memory as it arrives and be directly processed by applications.
- *Scalability and performance limitations of triggers*: It is possible to create alerts in a classical relational DBMS by using what is called a trigger. Triggers monitor tables and take actions in response to events that make changes to these tables. Triggers, however, were added to DBMSs as an after thought and were never designed to scale to the needs of streaming applications.
- *Difficulty of accessing past data*: Unlike relational queries, which focus on the current state of the database, stream processing queries are time-series-focused. They need to easily access past data, and at least a window of recent data.
- *Missing language constructs*: To support streaming applications, SQL must be extended with language constructs such as different types of window-based operations (windowed-aggregation and windowed-joins).

- *Precise query answers*: In traditional business applications, queries process data stored on disk and return precise answers. In a stream processing world, input data can get delayed, dropped, or reordered. Stream processing engines must capture these inaccuracies with clear language constructs and semantics. They need a way to ensure computation progress in the face of delayed data, clear semantics in the case of reordered data, and a way to handle data arriving after the corresponding windows of computation have closed.
- *Near real-time requirements*: Finally, stream processing applications require near real-time query results. In particular, queries that generate alerts based on data in streams cannot fall arbitrarily behind even when load conditions vary or data sources generate data at different rates.

These requirements[3]—which were initially motivated by military application scenarios where soldiers, equipment, and missiles are continuously tracked—formed the foundation of the new class of stream-processing engines that would emerge in the database community over the coming years.

The Aurora and Borealis Stream-Processing Systems[4]

The MIT-Brown-Brandeis researchers were on the “bleeding edge” with their Aurora and Borealis projects.

Soon after Mike moved from California to New Hampshire, he attended a National Science Foundation (NSF) meeting. While there, he gathered other participants who also were database researchers in New England computer science departments. Mike had an idea to set databases on their ear by moving from the traditional passive data/active query model to an active data/passive query model that defined stream processing. As was Mike’s style, he had already had discussions with his many industrial connections and determined their common “pain points.” The stream processing model he envisioned could soothe many of them,

[3] “One insight not listed here is that Aurora offered a workflow-based diagrammatic “language” that was much better suited to real-time application development than a traditional declarative language like SQL. Basically, it exposed the query planner that usually is hidden (or mostly hidden) from application developers and turned it into an expressive and powerful language in its own right. Developers liked it because it made them more productive—it was the right tool for the job. It gave them direct control over the processing of each tuple, rather than leaving them to worry what a capricious query optimizer might decide to do.”—John Partridge

[4] See Chapter 26.

but he needed collaborators and an army of students to build a workable prototype of the system he had in mind.

Under Mike's leadership, the three-institution research group embarked on an ambitious project to build not one but two stream-processing systems. The first system, called Aurora [Balakrishnan et al. 2004, Abadi et al. 2003b, Abadi et al. 2003a, Zdonik et al. 2003, Carney et al. 2002], was a single-node system that focused on the fundamental aspects of data model, query language, and query execution for stream processing. The second system, called Borealis [Abadi et al. 2005, Cherniack et al. 2003, Zdonik et al. 2003], was a distributed system that focused on aspects of efficient stream processing across local and wide-area networks including distribution, load balance, and fault-tolerance challenges. Borealis was built on top of the single-node Aurora system.

Both systems were widely successful and released as open-source projects. They led to many publications on various aspects of streaming from the fundamental data model and architecture to issues of load shedding, revision processing, high availability, and fault tolerance, load distribution, and operator scheduling [Tatbul and Zdonik 2006, Ryvkina et al. 2006, Xing et al. 2005, Abadi et al. 2005, Cherniack et al. 2003, Tatbul et al. 2007, Hwang et al. 2005, Balakrishnan et al. 2004, Carney et al. 2003, Abadi et al. 2003b, Tatbul et al. 2003, Abadi et al. 2003a, Zdonik et al. 2003, Carney et al. 2002, Balazinska et al. 2004a, Balazinska et al. 2005]. A large, five-year, multi-institution NSF grant awarded in 2003 allowed the team to scale and aggressively pursue such a broad and ambitious research agenda.

Aurora was an innovative and technically deep system. The fundamental data model remained the relational model with some extensions, namely that relations were unbounded in terms of size (i.e., continuously growing), pushed by remote data sources (and thus append-only), and included a timestamp attribute that the system added to each input tuple in a stream. Queries took the form of boxes and arrows diagrams, where operators were connected by streams into direct acyclic query execution graphs. The operators themselves were also relational but extended with windowing constructs to ensure non-blocking processing in the face of unbounded inputs.

Aurora's programming model and language was based on boxes-and-arrows. The boxes were built-in operators (e.g., SELECT, JOIN, MAP), and the arrows were data flows that would trigger downstream operators. A program was constructed by literally wiring up a boxes-and-arrows diagram using a GUI. For simple problems, this was a compelling way to visualize the data-flow logic. For more difficult problems, the boxes-and-arrows approach became hard to manage. Customers complained that there was no standard. Mike convened a group consisting of Jennifer

Widom from Stanford, the designer of STREAM [Motwani et al. 2003]; a few engineers from Oracle who had decided to use the stream language in Oracle's middleware product; and some designers from StreamBase (the product based on Aurora) to come up with a single textual language that the group could salute. Mike said that this should be easy since both the boxes-and-arrows language and STREAM supported features like Windows, and it should just be an exercise in merging the two. This would solve both problems. Upon further investigation, the committee decided that the underlying processing model was significantly different, and a merge would be too complex to be practical. The details are in the Proceedings of the VLDB Endowment (PVLDB) article [Jain et al 2008].

Interestingly, from its inception, Aurora's design included constructs to connect with persistent storage by using what we called *connection points*. A connection point could buffer a stream on disk. As such, it could serve as a location where new queries could be added and could re-process recent data that had accumulated in the collection point. A connection point could also represent a static relation on disk and serve to help join that relation with streaming data. The connection point design, while interesting, was somewhat ahead of its time and for many years, most focus was purely on processing data streams without connection points. In modern stream-processing systems, as we discuss below, the ability to persist and reprocess streams is an important function.[5]

Aurora included additional innovative features such as constructs and semantics for out-of-order or late data and corrections on streams, and novel methods for query scheduling, quality of service, load shedding and fault-tolerance [Tatbul and Zdonik 2006, Tatbul et al. 2007, Tatbul et al. 2003, Balakrishnan et al. 2004, Abadi et al. 2003b, Abadi et al. 2003a, Carney et al. 2003, Carney et al. 2002].

Following Aurora, the Borealis system tackled the distributed nature and requirements of streaming applications. In Borealis, operators in a query plan could be distributed across multiple machines in a cluster or even over wide area networks, as illustrated in Figure 17.1. Distribution is important for applications where sources send data at high rates from remote locations, such as network monitoring applications and sensor-based applications, for example. Borealis provided fundamental abstractions for distributed stream processing and included load

[5] "This insight is an example of Mike's foresight and it was a good thing he thought of it. Conversations with our early potential customers (trading desks at investment banks and hedge funds) quickly revealed that they wanted a clean way of integrating their real-time processing with querying historical data (sometimes years old, sometimes hours or minutes old). Connection points functionality matured quickly as a result."—John Partridge

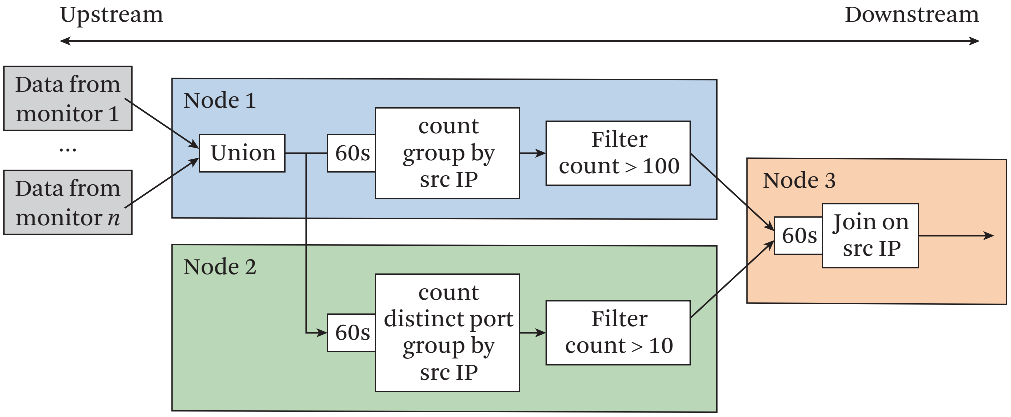


Figure 17.1 Example of a distributed stream-processing application in Borealis. This application labels as a potential network intruder any source IP that establishes more than 100 connections and connects over 10 different ports within a 1-min time window. Source: [Balazinska et al. 2004a]

management [Xing et al. 2005, Balazinska et al. 2004a] and different types of high-availability and fault-tolerance features [Hwang et al. 2005, Balazinska et al. 2005].

Incredibly, Mike was involved in the design of all system components in both systems. He seemed to possess infinite amounts of time to read through long design documents and provide detailed comments. He would attend meetings and listen to everyone's ideas and discussions. Importantly, he had infinite patience to coordinate all Ph.D. students working on the system and ensure regular publications at SIGMOD and VLDB and highly visible system demonstrations at those conferences [Ahmad et al. 2005, Abadi et al. 2003a].

By working with Mike, the team learned several important lessons. First, it is important to examine all problems from either 10,000 or 100,000 feet. Second, all problems should be captured in a quad chart and the top right corner always wins, for example, see the Grassy Brook quad chart in Figure 17.2. Third, new benchmarks shall be crafted in a way that makes the old technology look terrible. Fourth, database people deeply worry about employees in New York and employees in Paris and the differences in how their salaries are computed. (While this example never made it into any paper, it was the first example that Mike would draw on a board when explaining databases to systems people.) Fifth, stream processing was the killer technology that can stop people stealing overhead projectors.

Finally—and most importantly—the team learned that one could take graduate students and faculty from different institutions, with different backgrounds, with-

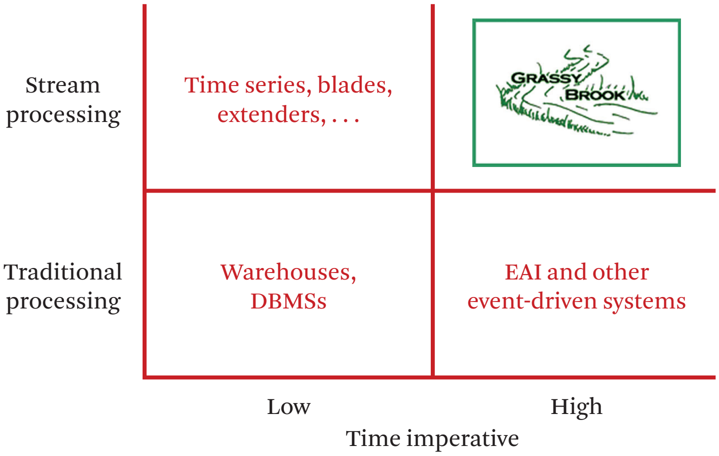


Figure 17.2 High-performance stream processing. Spirce: StreamBase (Grassy Brook) Pitch Deck (2003)

out any history of collaboration, put them in one room, and get them to build a great system together!

Concurrent Stream-Processing Efforts

At the same time as Mike's team was building the Aurora and Borealis systems, other groups were also building stream-processing prototypes. Most prominent projects included the STREAM [Motwani et al. 2003] processing system from Stanford, the TelegraphCQ [Chandrasekaran et al. 2003] project from Berkeley, NiagaraCQ [Chen et al. 2000] from Wisconsin, and the Gigascope [Cranor et al. 2003] project from AT&T. Other projects were also under development and many papers started appearing at SIGMOD and VLDB conferences on various aspects of streaming technologies. There was intense friendly competition, which moved the research forward quickly.

Overall, the community was focused on fundamental issues behind effectively processing data streams in database management systems. These issues included the development of new data models and query languages with constructs for basic data stream processing but also for processing data revisions and out-of-order data and to perform time-travel operations on streams. The community also developed new query optimization techniques including methods to dynamically change query plans without stopping execution. Finally, several papers contributed techniques for operator scheduling, quality of service and load shedding, and fault-tolerant distributed and parallel stream processing.

Aurora and Borealis were among the leaders in almost all aspects of stream processing at that time.

Founding StreamBase Systems[6]

The Aurora academic prototype was demonstrated at SIGMOD 2003 [Abadi et al. 2003a], and we had Aurora/Borealis ball caps made for the team members in attendance. (They are now collectors' items.) Not long after that, Mike decided that it was time to commercialize the system by founding a company with Stan Zdonik, Hari Balakrishnan, Ugur Cetintemel, Mitch Cherniack, Richard Tibbetts, Jon Salz, Don Carney, Eddie Galvez, and John Partridge. The initial name for the company was Grassy Brook, Inc.,[7] after Grassy Pond Road, the location of Mike's house on Lake Winnipesaukee in New Hampshire. We prepared a slide deck with the business plan that was to be presented to Boston-area venture capitalists. The slide deck included a famous Stonebraker Quad chart (see Figure 17.2) that argued that the StreamBase (then Grassy Brook) sweet spot was high-performance stream processing—something that no conventional data management platforms could accommodate and at the time coming into high demand by customers. We got halfway through the presentation when it was clear that the VCs wanted to invest. It helped that Mike was a "serial entrepreneur," and thus a relatively safe bet.

The company's first office was in Wellesley, Massachusetts, in the space of one of our investors, Bessemer Venture Partners. Later, Highland Capital, another investor, gave Grassy Brook some space in their Lexington, Massachusetts, facility (see Chapter 9). Soon the name was changed to StreamBase, which required buying the name from a previous owner. Mike and John Partridge developed a pitch deck for the VCs, and then Mike did most of the work and all of the presenting. Once they closed on the small initial financing, Mike and John met with potential early customers, mainly financial services people. When they had enough market feedback to substantiate their claims about customer demand, they went back to Bessemer and Highland to get the larger investment.

With fresh funding in hand, we hired a management team including Barry Morris as CEO, Bobbi Heath to run Engineering, and Bill Hobbib to run Marketing. Mike served as CTO. The venture capitalists helped enormously in connecting us with potential customers. We were able to get an audience with many CTOs

[6] See Chapter 26.

[7] The initial company name would have been "Grassy Pond" except that the domain name was taken at the time. So I bought grassybrook.com.—John Partridge

from Wall Street investment banks and hedge funds, a first target market for the StreamBase engine: Their need for extremely low latency seemed like a perfect match. This need led to many train trips to New York City, opening an opportunity to tune StreamBase for a demanding and concrete application. The company engaged in many proof-of-concept (POC) applications that, while an expensive way to make sales, helped sharpen the StreamBase engineers.[8]

The company tried to open a new market with government systems. In particular, it hired a Washington-based sales team that tried to penetrate the intelligence agencies. StreamBase made a big effort to sell to "three letter agencies" and In-Q-Tel (the CIA's venture capital arm) later invested in StreamBase. One partnership was with a Washington, D.C., Value Added Reseller (VAR) that built applications on top of StreamBase for government customers. StreamBase had a sales representative and a sales engineer in D.C. to sell directly to the government, but that never became a meaningful part of the business.

After some years, StreamBase was acquired by TIBCO Software, Inc. TIBCO is still in operation today in Waltham, Massachusetts., and sells TIBCO StreamBase®.

Stream Processing Today

In recent years, industry has been transformed by the wave of "Big Data" and "Data Science," where business decisions and product enhancements are increasingly based on results of the analysis of massive amounts of data. This data includes search logs, clickstreams, and other "data exhaust" from planetary-scale Web 2.0 applications. In many of these applications, the data of interest is generated in a continuous fashion and users increasingly seek to analyze the data live as it is generated. As a result, stream processing has emerged as a critical aspect of data processing in industry and many systems are being actively developed. Modern stream-processing systems include Apache Kafka, Heron, Trill, Microsoft StreamInsight, Spark Streaming, Apache Beam, and Apache Flink.

Interestingly, the stream-processing systems being developed in industry today are fundamentally the same as the ones as we and others in the database community built all those years ago. The goal is to process unbounded streams of tuples. Tuples are structured records and include timestamps. Processing involves grouping data into windows for aggregation and ensuring high availability and fault tolerance. Recent systems, however, do have a somewhat different emphasis than

[8] Following Mike's by now well-established pattern for starting a company, described in Chapter 7.

our original work from the database community. In particular, they seek a single programming model for batch data processing and stream processing. They focus significantly more on parallel, shared-nothing stream processing, and they seek to provide powerful APIs in Python and Java as well as seamless support for user-defined functions.

All first-generation streaming systems ignored the need for transactions. The MIT-Brown-Brandeis research continues in a project called S-Store [Çetintemel et al. 2014],[9] which attempts to integrate transactions into a streaming engine [Meehan et al. 2015a, Meehan et al. 2015b].

It is a testament to Mike's vision and leadership that our original Aurora/Borealis work stood the test of time and that today's streaming engines so naturally build on those past ideas. One of our papers, on fault tolerance in Borealis at SIGMOD'05 [Balazinska et al. 2005], won a "test of time" award at SIGMOD'17.

Acknowledgments

Thank you to John Partridge, StreamBase Systems Co-Founder and VP Business Development, for sharing his memories and stories for this chapter.

[9] http://sstore.cs.brown.edu/ (Last accessed March 28, 2018.)

18

C-Store: Through the Eyes of a Ph.D. Student

Daniel J. Abadi

I first met Mike Stonebraker when I was an undergraduate. Before I met Mike, I had no intention of going into a career in computer science, and certainly not into a career of computer science research.

I always knew that I wanted to go into a career that made an impact on people's lives and I had come to the conclusion that becoming a doctor would be the optimal way to achieve this career goal. At the time I met Mike, I was three quarters of the way through the pre-med undergraduate course requirements and was working in John Lisman's lab researching the biological causes of memory loss. However, to earn some extra income, I was also working in Mitch Cherniack's lab on query optimization. Mitch included me in the early research meetings for the Aurora project (see Chapter 17), through which I met Mike.

How I Became a Computer Scientist

At the time I met Mike, my only experience with computer science research involved writing tools that used automated theorem provers to validate the correctness of query rewrite rules during query optimization in database systems. I found the project to be intellectually stimulating and technically deep, but I was conscious of the fact that in order for my research to have an impact, the following chain of events would have to occur.

1. I would have to write a research paper that described the automated correctness proofs that we were working on.
2. This paper would have to be accepted for publication in a visible publication venue for database system research.

3. Somebody who was building a real system would have to read my paper and decide that the techniques introduced by the paper were a better way to ensure correctness of rewrite rules than alternative options, and therefore integrate our techniques into their system.
4. The real system would then have to be deployed by real people for a real application.
5. That real application would have to submit a query to the system for which a generic database system would have produced the wrong answer, but since they were using a system that integrated our techniques, the correct answer was produced.
6. The difference between the wrong answer and the correct answer had to be large enough that it would have led to an incorrect decision to be made in the real world.
7. This incorrect decision needed to have real-world consequences.

If any link in this chain failed to come to fruition, the impact of the research would be severely limited. Therefore, my belief at the time was that computer science research was mostly mathematical and theoretical, and that real-world impact was possible but had long-shot probability.

My early interactions with Mike very quickly disabused me of this notion. Any attempt to include math or theory in a meeting with Mike would either be brushed aside with disdain, or (more commonly) after completing the process of writing down the idea on the white board, we would glance at Mike and notice that he had lost consciousness, fast asleep with his face facing the ceiling. Any idea that we would introduce would be responded to with questions about feasibility, practicality, and how were we going to test the idea on real-world datasets and workloads. I learned the following rules of thumb from Mike regarding achieving real-world impact.

1. Complexity must be avoided at all costs. The most impactful ideas are simple ideas for the following reasons.
   (a) Complex ideas require more effort for somebody to read and understand. If you want people to read the papers that you write, you should minimize the effort you ask them to go through in reading your paper.
   (b) Complex ideas are hard to communicate. Impact spreads not only through your own communication of your ideas via papers and pre-

sentations, but also through other people summarizing and referring to your ideas in their papers and presentations. The simpler the idea, the more likely someone else will be able to describe it to a third party.

   (c) Complex ideas are hard to implement. To get an idea published, you generally have to implement it in order to run experiments on it. The harder it is to implement, the longer it takes to build, which reduces the overall productivity of a research group.

   (d) Complex ideas are hard to reproduce. One way of achieving impact is for other people to take your ideas and implement them in their system. But if that process is complicated, they are less likely to do so.

   (e) Complex ideas are hard to commercialize. At the end of the day, commercialization of the idea requires communication (often to non-technical people) and rapid implementation. Therefore, the communication and implementation barriers of the complex ideas mentioned above also serve as barriers to commercialization.

2. It is better to build a complete system than it is to focus your research on just a single part of a system. There are three main reasons for this.

   (a) Narrow ideas on isolated parts of the system risk being irrelevant because a different part of the system may be the bottleneck in real-world deployments. Mike would always be asking about "high poles in the tent"—making sure that our research efforts were on real bottlenecks of the system.

   (b) System components interact with each other in interesting ways. If research focuses on just a single part of the system, the research will not observe these important interactions across components.

   (c) It is generally easier to commercialize an entire system than just a single component. The commercialization of individual components requires deep partnership with existing software vendors, which a young, fledging startup usually struggles to achieve. A complete system can be built from scratch without relying on third parties during the implementation effort.

3. Impact can be accelerated via commercialization. There have been many great ideas that have been published in computer science venues that have made important impact via somebody else reading the paper and implementing the idea. However, the vast majority of ideas that are published in

> academic venues never see the light of day in the real world. The ones that do make it to the real world are almost never immediate—in some cases there is a delay of a decade or two before the technology is transferred and applied. The best way to increase both the probability and speed of transferring an idea to the real world is to go to the effort of raising money to form a company around the idea, build a prototype that includes at least the minimal set of features (including the novel features that form the central thesis of the research project) that enable the prototype to be used in production for real applications, and release it to potential customers and partners.
>
> A second advantage to direct commercialization of research technology is exposure to real-world ramifications of a particular technology. This experience can often be fed back into a research lab for successful future research projects.

In short, Mike taught me that computer science research could be far more direct than I had realized. Furthermore, the impact is much more scalable than the localized professions I had been considering. In the end, I decided to apply to graduate school and pursue a career in research.

The Idea, Evolution, and Impact of C-Store

At the time I applied to MIT (for admission in the fall of 2003), there were no database system faculty at MIT aside from Mike (who had recently joined MIT as an adjunct professor). Nonetheless, Mike helped to ensure that my application to MIT would be accepted, even though he had no intention of taking on the role of advising Ph.D. students in his early years at MIT. Mike matched me up with Hari Balakrishnan as my temporary official advisor, while he continued to advise me unofficially. Shortly thereafter, Sam Madden joined MIT, and Sam, Mike, and I, along with teams from Brown, UMass, and Brandeis, began to work on the C-Store project [Stonebraker et al. 2005a] in 2004.

The early days of the research on the C-Store project have formed my approach to performing research ever since that point. C-Store was never about innovating just for the sake of innovating. C-Store started with Mike taking his experience and connections in industry and saying, "There's a major pain point here. None of the 'Big Three' database systems—Oracle, IBM's DB2, and Microsoft's SQL Server—scale queries to the degree that the upcoming 'Big Data' era will require, and other existing solutions are wildly inefficient. Let's build a system that will scale and process queries efficiently."

We can see from this example two key things that Mike did that are instructive about making impact.

1. *He found a source of existing pain.* If you want to make impact, you have to do research in areas where there is so much existing pain that people will pay the switching costs to adopt your solution if it becomes available to them.
2. *He identified a trend that would magnify the pain before it took off* . "Big Data" was only just emerging as an industry term in 2004 when the project started.

The C-Store project involved creating a scalable database system that is optimized for read-mostly workloads, such as those found in data warehousing environments (i.e., workloads that are almost entirely read queries, with occasional batch appends of new records and rare updates of previously inserted records). The project included two components to the storage layer: a read-only component (where most of the data was stored) and a writable component. Updates to existing records were handled by deleting from the read-only component and inserting a new record into the writable component. By virtue of being read-only, the read-only component was able to make several optimizations, including dense-packing the data and indexes, keeping data in strictly sorted order (including redundantly storing different materialized views or "projections" in different sort orders), compressing the data, reading data in large blocks from disk, and using vastly simplified concurrency control and recovery protocols.

In contrast to the read-only component, the writable component was generally stored in-memory and optimized for inserting new records. These inserts can happen slowly over a period of time (they were known as "trickle updates") or they can happen in batch (e.g., when a previous day's log of data is written to the data warehouse overnight). All queries would include data from both the read-only and writable component (data from the two components would be dynamically merged on the fly at query time). However, it was important that the writable component fit entirely in main memory. Therefore, a "tuple mover" would move data from the writable component to the read-only component in batches as a background process.

C-Store was most famous for storing data in "columns." In general, a database table is a two-dimensional object and needs to be serialized to a one-dimensional storage interface when the data is written to storage. Most (but not all) database systems at the time performed this serialization process data row by row. First, they would store the first row, and then the second, and so on. In contrast, column-oriented systems such as C-Store stored data column by column. Storing columns separately helped to optimize the system for read-only queries that scan through many rows, since by storing columns separately, the system only needs to expend I/O time reading from storage the specific columns necessary to answer the query. For queries that accessed a small percentage of the columns in a table, the performance benefits would be large. However, inserting a new tuple can be slow since the different attributes in the tuple have to written to separate locations. This is why it was critical to have a separate writable component that was in-memory. Indeed, in some of the early designs of C-Store, only the read-only component stored data in columns, while the writable component used a traditional row-oriented design.

Building C-Store with Mike

My job within the C-Store project was to collaborate on the design and write code to implement both the core read-optimized storage later and the query execution engine. I therefore had the opportunity to spend many hours with Mike and Sam at the whiteboard, discussing the tradeoffs of some of the different design decisions of these parts of the system. When I look back at my time in graduate school, I think of this time very fondly: the excitement surrounding preparing for one of these meetings, the back and forth during these meetings, and then the process of replaying the meeting in my head afterward, reviewing the highlights and lowlights and planning for what I would do differently next time to try to do a better job convincing Mike of an idea that I had tried to present during the meeting.

In general, Mike has a tremendous instinct for making decisions on the design of a system. However, as a result, any idea that runs counter to his instinct has almost no chance of seeing the light of day without somebody actually going to the effort of building the idea and generating incontrovertible proof (which, given that fact that the idea runs counter to Mike's instinct, is unlikely to be possible).

One example of this is a discussion around the compression methods that should be used in the storage later. Column-stores present a tremendous opportunity for revisiting database compression under a new lens: Not only do column-stores, by virtue of storing data from the same attribute domain contiguously, observe much smaller data entropy (and therefore are more amenable to compression in general), but also, they make it possible to compress each column of a table using a different compression scheme. Mike had created a quad chart (see Figure 18.1) for what compression scheme should be used for each column. The two dimensions of the quad chart were: (1) is the column sorted and (2) how high is the cardinality of the column (the number of unique values).

	Low cardinality	High cardinality
Sorted	RLE	Delta
Unsorted	Bit-vector	No compression

Figure 18.1 A Stonebraker quad chart for choosing compression algorithms in C-Store.

I had made a proposal that instead of using Bit-vector encoding for unsorted, low-cardinality columns, we should instead use arithmetic encoding. This led to the following email conversation that I found in my email history from graduate school:

Mike wrote to me:

> I don't see the advantage of arithmetic coding. At best it would break even (space-wise) compared to what we have and adds decoding cost. Moreover, it gets in the way of passing columns between operators as bit maps. Also, coding scheme would have to "learn" alphabet, which would have to be "per column." Adding new values would be a problem.

I wrote back:

> The anticipated advantage would be the space savings (and thus i/o cost). I'm not sure why you're saying we will break even at best. We're seeing about 4× compression for the [unsorted, low cardinality columns] we have now, which is about what dictionary would get. Depending on the data probabilities, arithmetic would get about an additional 2× beyond that.

To which he responded:

> I don't believe you. Run length encoding (sic. RLE) of the bit map should do the same or better than arithmetic. However, this question must get settled by real numbers.
>
> Also, arithmetic has a big problem with new values. You can't change the code book on the fly, without recoding every value you have seen since time zero . . . .

This conversation was typical of the "prove it" requirement of convincing Mike of anything. I went ahead and spent time carefully integrating arithmetic encoding into the system. In the end, Mike was right: Arithmetic coding was not a good idea for database systems. We observed good compression ratios, but the decompression speed was too slow for viability in a performance-oriented system.[1]

A few more lessons from the C-Store project about systems research in general.

1. None of the ideas of the paper were new. Column-stores had already been in existence (Sybase IQ was the first widely deployed commercial implementation of a column-store). Shared-nothing database systems had already been in existence. C-Store's main contribution was the combination of several different ideas in a way that made the whole more than the sum of its parts.
2. Although the idea of column-stores had been around for over two decades before the C-Store paper, they were held back by several limiting design decisions. Sometimes, innovations within an existing idea can turn it from not really widely practical, to being so practical that the idea becomes ubiquitous. Once column-stores caught on, all major DBMSs developed column-store extensions, either by storing data column by column within a page (e.g., Oracle) or in some cases with real column-store storage managers (such as IBM Blu and Microsoft SQL Server Apollo).

Founding Vertica Systems

As mentioned above, a great way to accelerate research impact is to start a company that commercializes the research.[2] Mike's career is perhaps the quintessential example of this approach, with Mike repeatedly transferring technology from his lab into commercialization efforts. In the case of C-Store, the transition from research project to startup (Vertica) was extremely rapid. Vertica's first CEO and VP of Engineering were already in place in March 2005, the same month in which we submitted the original C-Store VLDB paper [Stonebraker et al. 2005a] to be reviewed by the VLDB program committee.

I was privileged to have the opportunity to work closely with Mike and Vertica's first CEO (Andy Palmer) in the technology transfer process.[3] The typical process

[1] I did, however, eventually convince Mike that we should move away from bit-vector compression and use dictionary compression for unsorted, low cardinality columns.

[2] For the story of the development of the Vertica product, see Chapter 27.

[3] For more on the founding of Vertica, read Chapter 8.

for any technology startup of this kind is to begin the commercialization effort in search of use cases. Although we were confident that the C-Store technology was widely applicable across many application spaces, different domains will have different existing solutions to handle their data and query workload. In the case of some domains, the existing solution is good enough, or close enough to being good enough that they are not in significant pain. In other domains, the current solutions are so insufficient, they would be willing to risk trying out a new and unproven technology from a tiny startup. Obviously, it is those domains that the first versions of a technology startup should focus on. Therefore, we spent many hours on the phone (sometimes together, sometimes independently) with CIOs, CTOs, and other employees at companies from different domains that were in charge of the data infrastructure and data warehousing solutions, trying to gauge how much pain they were currently in, and how well our technology could alleviate that pain.

Indeed, my longest continuous time with Mike was a road trip we took in the early days of Vertica. I took a train to Exeter, New Hampshire, and spent a night in Andy Palmer's house. The next morning, we picked up Mike in Manchester, and the three of us drove to central Connecticut to meet with a potential alpha customer, a large online travel company. The two things about this meeting that I remember:

1. Being in awe as Mike was able to predict the various pain points that the data infrastructure team was in before the CIO mentioned them. Interestingly, there were some pain points that were so prevalent in the industry that the CIO did not even realize that he was in pain.
2. C-Store/Vertica's main initial focus was on the storage layer and had a very basic first version of the optimizer that worked only with star schemas.[4] I remember an amusing exchange between Mike and a member of their team in which Mike tried to convince him that even though they didn't actually have a star schema, if they would squint and look at their schema from 10,000 ft, it was indeed a star schema.

In the end, Vertica was a huge success. It was acquired by Hewlett-Packard in 2011 for a large sum and remained successful through the acquisition. Today, Vertica has many thousands of paying customers, and many more using the free community edition of the software.

[4] For more about the significance of star schemas, read Chapter 14.

More importantly, the core design of C-Store's query execution engine, with direct operation on compressed, column-oriented data, has become prevalent in the industry. Every major database vendor now has a column-oriented option, with proven performance improvements for read-mostly workloads. I feel very fortunate. I will forever be grateful to Mike Stonebraker for believing in me and giving me the opportunity to collaborate with him on C-Store, and for supporting my career as a database system researcher ever since.

19

In-Memory, Horizontal, and Transactional: The H-Store OLTP DBMS Project

Andy Pavlo

I remember the first time that I heard the name Mike Stonebraker. After I finished my undergraduate degree, I was hired as a systems programmer at the University of Wisconsin in 2005 to work on the *HTCondor*, a high-throughput job execution system project, for Miron Livny. My colleague (Greg Thain) in the office next to me was responsible for porting a version of HTCondor called *CondorDB*[1] from David DeWitt's research group. The gist of CondorDB was that it used Postgres as its backing data store instead of its custom internal data files. Although my friend was never a student of Mike's, he regaled me with stories about all of Mike's research accomplishments (Ingres, Postgres, Mariposa).

I left Wisconsin in 2007 and enrolled at Brown University for graduate school. My original intention was to work with another systems professor at Brown. The first couple of weeks I dabbled in a couple of project ideas with that professor, but I was not able to find good traction with anything. Then the most fortuitous thing in my career happened after about the second week of classes. I was in Stan Zdonik's database class when suddenly he asked me if my name was Andy Pavlo. I said "yes." Stan then said that he had had a phone call with Mike Stonebraker and David DeWitt the previous night about ramping up development for the *H-Store*

[1] http://dl.acm.org/citation.cfm?id=1453856.1453865 (Last accessed March 26, 2018.)

project and that DeWitt recommended me as somebody that Stan should recruit to join the team.

At first, I was hesitant to do this. After I decided to leave Wisconsin to start graduate school at Brown, I had a parting conversation with DeWitt. The last thing that he said to me was that I should not work with Stan Zdonik. He never told me why. I later learned it was because Stan was traveling a lot to visit companies on behalf of Vertica with Mike. At the time, however, I did not know this, and thus I was not sure about switching to have Stan as my adviser. But I was curious to see what all the fuss was about regarding Stonebraker, so I agreed to at least attend the first kick-off meeting. That was when I met Mike.

As I now describe, there were several incarnations of the H-Store system as an academic project and eventually the VoltDB commercial product (see Chapter 28).

System Architecture Overview

The H-Store project was at the forefront of a new movement in DBMS architectures called *NewSQL* [Pavlo and Aslett 2016]. During the late 2000s, the hottest trend in DBMSs were the so-called *NoSQL* systems that forego the ACID (Atomicity, Consistency, Isolation, Durability) guarantees of traditional DBMSs (e.g., Oracle, DB2, Postgres) in exchange for better scalability and availability. NoSQL supporters argued that SQL and transactions were limitations to achieving the high performance needed in modern operational, on line transaction processing (OLTP) applications. What made H-Store different was that it sought to achieve the improved performance of NoSQL systems without giving up the transactional guarantees of traditional DBMSs.

Mike's key observation was that existing DBMSs at that time were based on the original system architectures from the 1970s that were too heavyweight for these workloads [Harizopoulos et al. 2008]. Such OLTP applications are characterized as comprising many transactions that (1) are short-lived (i.e., no user stalls), (2) touch a small subset of data using index lookups (i.e., no full table scans or large distributed joins), and (3) are repetitive (i.e., executing the same queries with different inputs).

H-Store is a parallel, row-storage relational DBMS that runs on a cluster of shared-nothing, main memory executor nodes. Most OLTP applications are small enough to fit entirely in memory. This allowed the DBMS to use architectural components that were more lightweight because they did not assume that the system would ever have to stall to read data from disk. The database is partitioned into disjoint subsets that are assigned to a single-threaded execution engine that is assigned to one and only one core on a node. Each engine has exclusive access to all the data in its partition. Because it is single-threaded, only one transaction at a time can access the data stored at its partition. Thus, there are no logical locks or low-level latches in the system, and no transaction will stall waiting for another transaction once it is started. This also means that all transactions had to execute as stored procedures to avoid delays due to network round trips between the DBMS and the application.

Another idea in the original design of H-Store was classifying transactions into different groups based on what data they accessed during execution. A *single-sited* transaction (later relabeled as *single-partition*) was one that only accessed data at a single partition. This was the ideal scenario for a transaction under the H-Store model as it did not require any coordination between partitions. It requires that the application's database be partitioned in such a way that all the data that is used together in a transaction reside in the same partition (called a *constrained tree schema* in the original H-Store paper [Kallman et al. 2008]). Another transaction type, called *one shot*, is where a transaction is decomposed into multiple single-partition transactions that do not need to coordinate with each other. The last type, known as a *general* transaction, is when the transaction accesses an arbitrary number of partitions. These transactions can contain either a single query that accesses multiple partitions or multiple queries that each access disparate partitions. General transactions are the worst-case scenario for the H-Store model.

First Prototype (2006)

The first version of the H-Store system was a prototype built by Daniel Abadi and Stavros Harizopoulos with Sam Madden and Stonebraker at MIT for their VLDB 2007 paper "The End of an Architectural Era: (It's Time for a Complete Rewrite" [Stonebraker et al. 2007b]. This early system was built to only execute a hardcoded version of TPC-C over arrays of data. It used a simple B+Tree for indexes. It did not support any logging or SQL.

The MIT H-Store prototype was able to achieve 82× better throughput than an Oracle installation tuned by a professional. This was evidence that H-Store's design of eschewing the legacy architecture components of disk-oriented systems was a promising approach for OLTP workloads.

Second Prototype (2007–2008)

Given the success of the first prototype, Mike decided to continue with the H-Store idea and build a new, full-featured system as part of a collaboration among MIT, Brown, and Yale (which Dan had since joined as a new faculty member). This was when I had just started graduate school and gotten involved in the project. The first H-Store meeting was held at MIT in November 2007.

My recollection from that first meeting was that Mike "held court" in what I now understand is his typical fashion. He leaned back in his chair with his long legs stretched out and his hands placed behind his head. He then laid out his entire vision for the system: what components that we (i.e., the students) needed to build and how we should build them. As a new graduate student, I thought that this was gospel and proceeded to write down everything that he said.

One thing that still stands out for me even to this day about this meeting was how Mike referred to DBMS components by the names of their inventors or paper authors. For example, Mike said that we should build a "Selinger-style" query optimizer and that we should avoid using a "Mohan-style" recovery scheme. This was intimidating: I knew of these concepts from my undergraduate database course but had never read the original papers and thus did not know who the concepts' inventors were. I made sure after that meeting that I read all the papers that he mentioned.

Our team of Brown and MIT students started building this second version of the system in late 2007. There was nothing that we could reuse from the original prototype since it was a proof-of-concept (i.e., it was hardcoded to only execute TPC-C transactions and store all tuples in long arrays), so we had to write the entire system from scratch. I was tasked with building the in-memory storage manager and execution engine. Hideaki Kimura was am M.S. student at Brown (later a Ph.D. student) who was helping me. Evan Jones was also a new Ph.D. student at MIT who joined the team to implement the system's networking layer.

There were several meetings between the Brown and MIT contingents in these early months of the project. I remember that there were multiple times when the students and other professors would squabble about certain design details of the system. Mike would remain silent during these discussions. Then, after some time, he would interrupt with a long speech that started with the phrase "seems to me . . . ." He would proceed to clarify everything and get the meeting back on track. Mike has this great ability to cut through complex problems and come up with a pithy solution. And he was correct almost every time.

To help reduce the amount of code that we had to write, Mike and the other professors suggested that we try to borrow components from other open-source DBMSs. Hideaki and I looked at SQLite, MySQL, and Postgres. For reasons that I do not remember, we were leaning toward using MySQL. I remember that I talked to somebody at the inaugural New England Database Day at MIT about my plans to do this. This person then later wrote an impassioned blog article where he pleaded for us not to use MySQL due to certain issues with its design. Given this, we then decided to borrow pieces from Postgres. Our plan was to use Postgres to parse the SQL queries and then extract the query plans. We would then execute those plans in our engine (this was before Postgres' Foreign Data Wrappers). Dan Abadi had an M.S. student at Yale implement the ability to dump out a Postgres plan to XML. As I describe below, we would later abandon the idea of using Postgres code in H-Store.

In March 2008, Evan and I wrote the H-Store VLDB demo paper. We still did not have a fully functioning system at that point. The single screenshot in that paper was a mock-up of a control panel for the system that we never ended up implementing. Around this time, John Hugg was hired at Vertica to start building the commercial version of H-Store (see Chapter 28). This was originally called *Horizontica*. John was building a Java-based front-end layer. He modified the HSQLDB (Hyper SQL Database) DBMS to emit XML query plans, and then he defined the stored procedure API. He did not have an execution engine.

The VLDB demo paper got accepted in late Spring 2008 [Kallman et al. 2008]. At this point we still only had separate components that were not integrated. Mike said something to the effect that given that the paper was going to be published, we had better build the damn thing. Thus, it was decided that the H-Store academic team would join forces with the Horizontica team (which at this point was just John Hugg and Bobbi Heath). We mashed together John's Java layer in Horizontica with our H-Store C++ execution engine. We ended up not using Evan's C++ networking layer code.

There were several people working on the system during Summer 2008 to prepare for the VLDB demo in New Zealand at the end of August. John, Hideaki, and I worked on the core system. Hideaki and I were hired as Vertica interns to deal with IP issues, but we still worked out of our graduate student offices at Brown. Evan worked with an undergrad at MIT to write the TPC-C benchmark implementation. Bobbi Heath was brought in as an engineering manager for the team; she was a former employee at Mike's previous startup (StreamBase). Bobbi eventually hired Ariel Weisberg and Ryan Betts[2] to help with development, but that was later in the year.

By the end of the summer, we had a functioning DBMS that supported SQL and stored procedures using a heuristic-based query optimizer (i.e., not the "Selinger-style" optimizer that Mike wanted!). John Hugg and I were selected to attend the conference to demonstrate the H-Store system. Although I do not remember

[2] Ryan later went on to become to the CTO of VoltDB after Mike stepped down in 2012.

whether the system at this time could support larger cluster sizes, our demo had only two nodes because we had to fly with the laptops to New Zealand. The conference organizers sent an email about a month before the conference confirming that each demo attendee would be provided a large screen. Before this, we had not decided what we were going to show in the demo. John and I quickly built a simple visualization tool that would show a real-time speedometer of how many transactions per second the system could execute.

I remember that John and I were debugging the DBMS the night before the demo in his hotel room. We got such a thrill when we were finally able to get the system to run without crashing for an extended period. I believe that our peak throughput was around 6,000 TPC-C transactions per second. This certainly does not sound like a lot by today's standards, but back then MySQL and Oracle could do about 300 and 800 transactions per second, respectively. This was fast enough that the laptops would run out of memory in about a minute because TPC-C inserts a lot of new records into the database. John had to write a special transaction that would periodically go through and delete old tuples.

VoltDB (2009–Present)

After the successful VLDB demo, we had a celebration dinner in September 2008. It was here that Mike announced that they were going to form a new company to commercialize H-Store. John Hugg and the company engineers forked the H-Store code and set about removing the various hacks that we had for the VLDB demo. I remember that they had a long discussion about what to name the new system. They hired a marketing firm. I think the first name they came up with was "The Sequel". Supposedly everyone hated it except for Mike. He thought that it would be a good punny jab at Oracle. Then they hired another marketing firm that came up with VoltDB ("Vertica On-Line Transaction Database"). Everyone liked this name.

My most noteworthy memory of the early VoltDB days was when I visited PayPal in San Jose with Mike, Evan, John, and Bobbi in October 2009 after the High Performance Transaction Systems conference. PayPal was interested in VoltDB because they were reaching the limits of their monolithic Oracle installation. Evan and I were there just as observers. Our host at PayPal was an ardent Mike supporter. Before the large meeting with the other engineering directors, this person went on for several minutes about how he had read all of Mike's papers and how much he loved every word in them. Mike did not seem concerned by this in the least. I now realize that this is probably how I reacted the first time I met Mike.

H-Store/VoltDB Split (2010–2016)

After a brief hiatus to work on a MapReduce evaluation paper with Mike and DeWitt in 2009 [Stonebraker et al. 2010], I went back to work on H-Store. My original plan was to use VoltDB as the target platform for my research for the rest of my time in graduate school, as VoltDB had several people working on the system by then and it was open-source.

The enhancements to the original H-Store codebase added by the VoltDB team were merged back into the H-Store repository in the summer of 2010. Over time, various components of VoltDB have been removed and rewritten in H-Store to meet my research needs. But I ended up having to rewrite a lot of the VoltDB code because it did not do the things that I needed. Most notable was that it did not support arbitrary multi-partition transactions.

Mike was always pushing me to port my work to VoltDB during this period, but it just was not feasible for me to do this. But in 2012, Mike came back to want to work on H-Store for the anti-caching project [Harizopoulos et al. 2008]. We then extended the H-Store code for the elastic version of the system (E-Store [Taft et al. 2014a]) and the streaming version (S-Store [Çetintemel et al. 2014]).

Conclusion

The H-Store project ended in 2016 after ten years of development. Compared to most academic projects, its impact on the research community and the careers of young people was immense. There were many students (three undergraduates, nine master's, nine Ph.Ds.) who contributed to the project from multiple universities (MIT, Brown, CMU, Yale, University of California Santa Barbara). Some of the Ph.D. students went off to get faculty positions at top universities (CMU, Yale, Northwestern, University of Chicago). It also had collaborators from research labs as well (Intel Labs, QCRI). During this time Mike won the Turing Award.

After working on the system for a decade, I am happy to say that Mike was (almost) correct about everything he envisioned for the system's design. More important is that his prediction that SQL and transactions are an important part of operational DBMSs was correct. When the H-Store project started, the trend was to use a NoSQL system that did not support SQL or transactions. But now almost all the NoSQL systems have switched over to SQL and/or added basic support for SQL.

20

Scaling Mountains: SciDB and Scientific Data Management

Paul Brown

Because it's there.

—George Mallory

"Serial summiteer" isn't an achievement many people aspire to.

The idea is simple enough. Pick a set of mountain peaks all sharing some defining characteristic—the tallest mountains on each of the seven continents, the 14,000-ft mountains of Colorado, the 282 "Munros" in Scotland, or even just the highest peaks in New England—and climb them, one after the other, until there are none left to climb. "Serial entrepreneur," by contrast, is a label at the opposite end of the aspirational spectrum. Any individual responsible for founding a series of companies, each with its own technical and business goals, is worth at least a book. Somewhere between the relative popularity of "serial summiteer" and "serial entrepreneur" we might find "successful academic": a career measured in terms of research projects, high-achieving graduates, and published papers. What all of these endeavors share is the need for planning, careful preparation, considerable patience, and stamina, but above all, stoic determination.

Mike Stonebraker has lived all of them. There are 48 mountains in New Hampshire over 4,000 ft in height and he's climbed them all. He's founded many start-up companies. And he won a Turing Award. In this chapter, we tell the story of one "journey up a mountain": a project I had the privilege to be a member of. Our expedition was conceived as a research project investigating scientific data management but was obliged to transform itself into a commercial start-up. The climb's not over.

No one knows what the climbers will find as they continue their ascent. Yet today, the company Paradigm4 continues to build and sell the SciDB Array DBMS with considerable success.

Selecting Your Mountain[1]

What are men to rocks and mountains?

—Jane Austen

What makes a mountain interesting? This is a surprisingly difficult question. And to judge by the sheer number of technology startups that set off annually, each trudging to its own carefully selected base camp, opinions about what problems are worth solving vary tremendously. For some climbers, it's the technical challenge. For others, it's the satisfaction of exploring new terrain, or the thrill of standing on the peak. And, of course, because any expedition needs funding, for some members it's the prospect that a glint of light on some remote mountaintop is actually a rich vein of gold.

No expedition ever really fails. Or, at least, they're rarely a failure for everyone. Sometimes the technical challenge proves too difficult—in which case the expedition falls short of its goals—or too easy—in which case it arrives to find a summit crowded with sprawled picnickers enjoying sandwiches and sherry. Sometimes, the new terrain reveals itself to be dull, barren, and featureless—in which case the expedition finds itself in possession of something of no interest or intrinsic value. Other times, the glint of gold turns out to be just a flash of sun on snow, and even though every technical challenge has been met and in spite of the climbers' heroism, the expedition descends dead broke.

Therefore, a successful serial entrepreneur or career academic must have very good taste in mountains. It is interesting, when reflecting on Mike's choices over the years, just how good his taste has been. With a regularity that has become a standing joke in the data management industry, every few years a fresh face comes along with a "great and novel" idea that will upend the industry. But on examination, the "great" bits aren't novel and the "novel" bits don't turn out to be all that great. Mike's career is a testament to picking the right problems. Given a choice between bolting a relational super-structure on a hierarchical or network storage layer or confronting the challenge of building something entirely new, we

[1] Chapter 7 by Michael Stonebraker is a must-read companion to this chapter, either before or after.

got InGReS. When the brave new world of object-oriented languages challenged SQL with talk of an "impedance mismatch" and promised performance magic with "pointer swizzling," Mike went instead with adapting the basic relational model to serve new requirements by developing on the relational model's concept of a *domain* to encompass new types (user-defined types—UDTs), user-defined functions (UDFs), and aggregates (UDAs). With Hadoop vendors all well along the path of abandoning their HDFS and MapReduce roots in favor of parallelism, column stores, and SQL, Mike's vocal aversion to MapReduce now seems justified.

The interesting thing about the mountain represented by SciDB was the management of "scientific" data. As far back at the early 1990s, when Mike served as co-PI of the Sequoia 2000 Project [Stonebraker 1995], it was clear that the challenges of storing, organizing, and querying data generated by things like atmospheric computer models, satellites, and networks of sensors indicated that there were profound gaps between the data management technology on offer and what scientific analysts and end users wanted. Many science databases were degenerate "write once" tape repositories. Users were on their own when it came to data access. Data objects were assigned semantically meaningless labels, forcing users to struggle through catalog files on FTP servers to identify what they wanted. Since the 1990s, Mike's friend, the late Jim Gray, had also devoted considerable portions of his considerable energy to the problem of scientific data management and had met with success in the design and implementation of the Sloan Digital Sky Survey (SDSS) [Szalay 2008]. SDSS had considerable impact in terms of how astronomy data was modeled and managed and indeed, on how the science of astronomy was conducted.

We stand today on the threshold of vast changes in the character of the data we are being called on to manage and the nature of what we do with it. Digital signal data—the kind of machine-generated or sensor-driven data commonly associated with the Internet of Things (IoT)—is utterly different from the human-centered, event-driven business data that drove the last great waves of DBMS technology. A modern precision medicine application brings together genomics data, biomedical imaging data, wearables data (the Medical IoT), medical monitoring data from MRIs and EKGs, and free text annotations alongside more regularly structured patient clinical data, and demographics details. But the bulk of the data and the analytic methods applied to it, the aspects of the application that utterly dominate the workload, can all be thought of as "scientific." Modern data analysis involves generating hypotheses, modeling the data to test them, deriving actionable insights from these models, and then starting the process over again with fresh ideas. Indeed, this change has yielded an entirely new class of computer users: the "data scientist," who employs methods that would be familiar to any working scientist to tease out insight from what the machines record.

Managing and analyzing scientific data was "interesting." It had immediate application to scientific research. And it constituted a set of technical challenges and requirements with future commercial uses. It was a mountain worthy of an expedition.

Planning the Climb

When you're doing mountain rescue, you don't take a doctorate in mountain rescue; you look for someone who knows the terrain.

—Rory Stewart

You never know exactly what dangers and surprises you're going to find on the mountain. So, it really helps to learn everything you can about it before you set out.

To understand the problems of scientific data management, the people to ask are working scientists. By the late 2000s, several ideas were coming together.

First, the coalface of science had shifted from being lab-coats-and-test-tubes-driven to becoming utterly dependent on computers for modeling, data management, and analysis. Scientific projects of any size always included a cohort of programmers who served as Sherpas.

Second, as a consequence of developments in sensor technology, the scale of data produced by the next generation of scientific projects could reasonably be characterized as an explosion. At CERN—currently the world's largest generator of raw scientific data—the detectors generate about 100 terabytes a day and store tens of petabytes of recent history. But just one of the telescopes proposed for deployment in the next decade—the Large Synoptic Survey Telescope—will collect about 20 terabytes of raw data per night and must store every bit of it for ten years, eventually accumulating about 100 petabytes.

Third, the success of the SDSS project—which relied heavily on Microsoft SQL Server—seemed to suggest that it was possible to use general-purpose data management tools and technologies to support specialized scientific tasks. Given a choice between building something from scratch or reusing someone else's components or even entire systems, the "lived preference" of working scientists had always been to roll their own—an expensive and risk-prone approach. Sponsors of large-scale public science projects were interested in new thinking.

All of these ideas were being explored at new conferences such as the Extremely Large Data Bases (XLDB) conference and workshop: a gathering of "real" scientists and computer scientists held at the Stanford Linear Accelerator (SLAC) facility. Through a mix of formal survey methods—interviews, presentations, panels—and informal interrogations—talking late into the night over beer and wine in the best mountaineering tradition—a group of academics and practitioners arrived at a list of what scientists required from data management technology [Stonebraker et al. 2009]. In summary, these requirements were as follows:

- a data model and query language organized around arrays and based on the methods of linear algebra;
- an extensible system capable of integrating new operators, data types, functions, and other algorithms;
- bindings with new languages like Python, "R," and MATLAB, rather than the traditional languages of business data processing;
- no-overwrite storage to retain the full history (lineage or provenance) of the data as it evolved through multiple, time-stamped versions;
- open source to encourage community contributions and to broaden the range of contributions as widely as possible;
- massively parallel or cluster approach to storage and computation;
- automatic, n-dimensional block data storage (rather than hashing or range partitioning);
- access to in-situ data (rather than requiring all data be loaded before query);
- integrated pipeline processing with storage and analytics; and
- first-class support for uncertainty and statistical error.

In more detailed design terms, we planned to implement SciDB using the following techniques.

- We decided to build SciDB in C/C++ using a suite of Linux open-source tools and made the code freely available. We did this to optimize for runtime performance, to make it easy to integrate with a number of freely available binary libraries that performed the mathematical "heavy lifting," and because Linux had become the operating system of choice in scientific data management.
- We adopted a shared-nothing storage and compute model. This is a common approach for systems that have high scalability and fault-tolerant requirements. Each SciDB node (we refer to them as instances) is a pure peer within a

cluster of computers. Data in the SciDB arrays, and the computational workload applied to the data, are distributed in a balanced fashion across the nodes [Stonebraker 1986b].

- Each instance implements a multi-threaded parsing, planning, and execution engine and a multi-version concurrency control (MVCC) approach to transactions similar to the one employed in Postgres [Stonebraker 1987].
- We opted for a conventional, ACID-compliant distributed transaction model. Read and write operations are all globally atomic, consistent, isolated, and durable.
- We decided to adopt a column-store approach to organizing records (each cell in a SciDB array can hold a multi-attribute record) and a pipelined or vectorized executor along the same lines as C-Store and MonetDB [Stonebraker et al. 2005a, Idreos et al. 2012].
- To achieve distributed consensus for the transactions and to support the metadata catalog we relied initially on a single (or replicated for failover) installation of the PostgreSQL DBMS. Eventually we planned to use a Paxos [Lamport 2001] distributed consensus algorithm to allow us to eliminate this single point of failure.
- Because the kinds of operations we needed to support included matrix operations such as matrix/matrix and matrix/vector product, as well as singular value decomposition, we adopted and built on the ScaLAPACK distributed linear algebra package [Blackford et al. 2017].
- We decided to support Postgres style user-defined types, functions and aggregates [Rowe and Stonebraker 1987]. In addition, we decided to implement a novel mode of parallel operator extensibility that was closer to how High Performance Computing dealt with such problems.

Our mountaintop goal was to build a new kind of DBMS: one tailored to the requirements of scientific users. We talked to a lot of scientists in an effort to understand how to get there. We took advice on what methods worked and heeded warnings about what wouldn't. And we decided to explore some new terrain and develop new techniques. While I was working at IBM, some even grayer beards had mentioned in passing that Ted Codd had fiddled around with a matrix algebra as the basis of his abstract data model before settling on the simpler, set theoretic, Relational Model. But while this fact served as evidence as to the wisdom of our thinking, it wasn't at all clear whether it was evidence for, or against, our strategic approach.

Expedition Logistics

Men[sic] Wanted: For hazardous journey. Small wages, bitter cold, long months of complete darkness, constant danger, safe return doubtful.

—Ernest Shackleton

By the end of 2008, the SciDB development team had a clear view of the mountain. We had agreed on our plans. Immediately before us was a straightforward, ACID transactional storage layer unified within an orthodox massively parallel compute framework. Superimposed upon this, instead of SQL tables, joins, and aggregates, we were thinking in terms of arrays, dot products, cross products, and convolutions.

Still, the path ahead wasn't completely obvious. We deferred important decisions for later. What query language? What client APIs? What about the vast number of numerical analytic methods our customers might want? Once established on the climb, we reasoned, we would have more information on which to base our decisions. But there remained one glaring problem. Money.

Scientific data processing was notoriously characterized—and the attribution is vague; some say Jim Gray, others Michael Stonebraker—as a "zero-billion-dollar problem." A successful pitch to venture capitalists must include the lure of lucre. What they want to hear is, "When we build it, users will come—carrying checkbooks." SciDB, unique among Mike's companies, seemed to be about something else. The pitch we made to potential funding partners emphasized the social and scientific importance of the expedition. Unlocking the mysteries of the universe, curing cancer, or understanding climate change: All of these noble efforts would be greatly accelerated with a tool like SciDB! It was pure research. A new data model! New kinds of queries! New kinds of requirements! Academic gold! Disappointingly, the research agencies who usually provide financial support to this kind of thing took one look at our ideas and disagreed with us about their potential merits.

Something else was afoot. In late 2008 a series of high-profile failures in the finance sector, combined with rapidly deteriorating economic conditions, caused many potential SciDB funders to pull in their horns. Early on, parties interested in backing an open-source DBMS that focused on array processing included some heavy hitters in e-commerce, bioinformatics, and finance. But once Bear Stearns and AIG died of hubris-induced cerebral hypoxia on their own respective mountains, enthusiasm for risk grew . . . thin. So, as we hunkered down to sit out the 12-month storm, the SciDB expedition was obliged to turn to the usual collection of “land sharks” [Stonebraker 2016].

By 2010 the clouds had cleared and SciDB had secured seed funding. But the finances were so strained that for the first few years of its life, the company consisted of Mike as CTO, an architect/chief plumber (the author), a CEO shouldering immense responsibilities, a couple of U.S.-based developers, a pick-up team of four Russians, and two or three part-time Ph.D. candidates. Our advisory board consisted of the very great and the very good—a dozen of them. Yet as the discerning and knowledgeable reader will have noted, our tiny expedition was embarking with a vast train of bag and baggage. To a first approximation, every one of the bullet points in our “techniques” list above implies about 30,000 lines of code: 20,000 to implement the functionality, another 5,000 for interfaces, and another 5,000 for testing. Such an undertaking implies about 2,000,000 lines of code. To be written by six people. In a year.

It’s one thing to be ambitious. Mountains are there to be climbed, even acknowledging that the overly optimistic risk disaster. Yet in the end . . . if prudence governed every human decision? There would be no adventures.

Base Camp

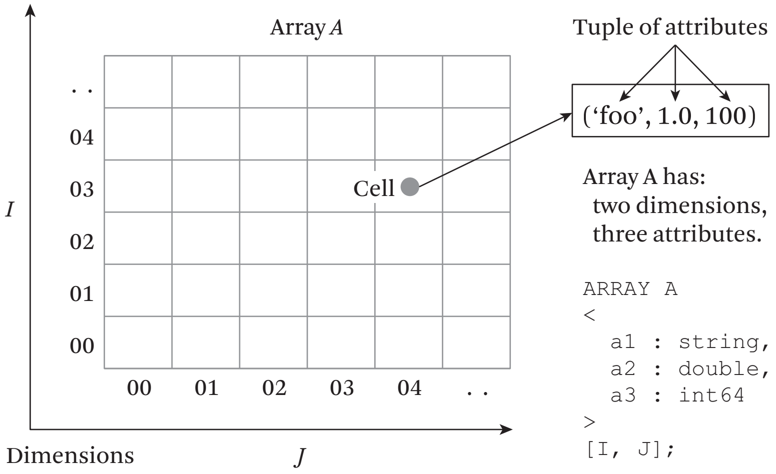
One may walk over the highest mountain one step at a time.

—Barbara Walters

So, we began. How the world has changed since the early days of Unix (or “open systems”) development! Where once the first order of a start-up’s business was to find a convenient office to co-locate programmers, their computers, management, and sales staff, the engineers working on SciDB were scattered across Moscow suburbs; Waltham, Massachusetts; and a New Delhi tower block. The virtual space we all shared involved check-in privileges to a source code trunk and a ticketing system, both running on a server in Texas.

Among our first tasks as we began the climb was to come up with the kind of data model that gave our working scientists what they said they wanted: something based around arrays, appropriate for the application of numerical methods like linear algebra, image filtering, and fast region selection and aggregation.

All data models begin with some notion of logical structure. We sketch the basics of the SciDB array in Figure 20.1. In SciDB, each n dimensional array organizes



```
ARRAY A
<
  a1 : string,
  a2 : double,
  a3 : int64
>
[I, J];
```

Figure 20.1 Structural outline of SciDB array data model.

data into a space defined by the array's n dimensions. In Figure 20.1, the array A has two dimensions, I and J. An array's dimensions each consist of an (ordered) list of integer index values. In addition, an array's dimensions have a precedence order. For example, if an array B is declared with dimensions [I, J, K], the shape of B is determined by the order of its dimension. So, if another array C uses the same dimensions but in a different order—for instance, C [K, I, J]—then we say the shape of B differs from C.

From this definition you can address each cell in an array by using an array's name, and a list (vector) consisting of one index value per dimension. For example, the labeled cell in Figure 20.1 can be addressed as A [I=3, J=4] (or more tersely, A [3,4] as the association between the index values and dimensions is inferred from the order of the array's dimensions). A cell in the three-dimensional array B would be addressed as B [I=5, J=5, K=5] (or B [5,5,5]). You can specify sub-regions of an array—and any region of an array is itself an array—by enumerating ranges along each dimension: A [I= 3 to 6, J= 4 to 6].

SciDB arrays have constraint rules. Some are implicit to the model. The combination of an array name and dimension values uniquely determines a cell, and that cell cannot be addressed in any other way. Some constraints can be made explicit as part of an array's definition. For example, the list of index values in array's dimensions can be limited to some range. If an application requires an array to hold 1024 × 1024-sized images as they changed over time, a user might limit two of the

```
regrid (
  filter (
    between ( A, 1, 1, 10, 10 ),
    not regex ( A.a1, ‘(.*)maybe(.*)' )
  ),
  2, 2,
  AVG ( R.a3 ) AS Mean_A3
);
```

Figure 20.2 Example functional language SciDB query.

array's dimensions to values between 1 and 1024—suggestively naming these dimensions X and Y—and leave the last dimension—named T—as unbound. SciDB would then reject any attempt to insert a cell into this array at a location outside its allowed area: say at X=1025, Y=1025, for any T. We envisioned other kinds of constraint rules: two arrays sharing a dimension, or a rule to say that an array must be dense, which would reject data where a cell within the array's dimensions space was "missing."

And, as with the relational model, the SciDB array data model defines a closed algebra of operators. Simple, unary operators filter an array by dimension index range or cell attribute values. Grouping operators would break an array up into (sometimes overlapping) sub-arrays and compute some aggregate per group. Binary operators would combine two arrays to produce an output with a new shape and with new data. Output of one operator can become input to another, allowing endless, flexible combinations. In Figure 20.2, we illustrate what a simple SciDB query combining multiple operators looks like. The inner filter() and between() operators specify what cells of the input array are to be passed on as output based on their logical location and the values in the cell's attributes. The regrid() partitions the filtered data—now a sparse array—into 2-by-2 sized regions and computes an aggregate per region.

The list of these array operators is extensive. The most recent SciDB community version—the code for which is made available under the Affero GPL license—ships with about 100 of them built in, while Paradigm4's professional services group have created another 30–40 that plug into the basic framework—although they are not open source because they sometimes use proprietary third-party libraries. In fact, several SciDB users have even created their own customized and application specific extensions. Operators run the gamut from providing simple selection, projection, and grouping functionality to matrix multiplication and singular value decomposition to image processing operations such as Connected Component Labeling [Oloso et al. 2016]. Every operator is designed to exploit the shared-nothing architecture to function at massive scale. And as with relational operators, it is possible to explore a number of ways to reorganize a tree of operators to find logically equivalent but computationally more efficient access paths to answer compound queries.

It's perhaps worth emphasizing just how novel this territory was. Since Codd's original Relational Model from 1971, DBMS data models have tended to be rather ad hoc. They would start with a programming language idea—such as object-oriented programming's notions of class, class hierarchy, inter-class references, and "messages" between classes—and would bolt on features such as transactional storage or data change. XML databases, to point to another example, started with a notion of a hierarchical markup language and then came up with syntax for specifying search expressions over the hierarchy—Xpath and XQuery. SciDB shared with Codd's Relational Model the idea of starting with an abstract mathematical framework—vectors, matrices, and the linear algebra—and fleshed out a data model by building from these first principles.

In the same way that Postgres solved the problem of non-standard data by exploiting the neglected notion of relational domains through the provision of user-defined types, functions, and aggregates, SciDB set about solving the very practical problems of scientific data processing by revisiting the mathematical fundamentals. New mountains can necessitate new climbing methods.

Plans, Mountains, and Altitude Sickness

No plan survives contact . . .

—Helmuth von Moltke the Elder

After about nine months of mania we had a code base we could compile into an executable, an executable we could install as a DBMS, and a DBMS we could load data into and run queries against. Time for our first users!

We decided to focus our energies on the bioinformatics market because of the vast increase in data volumes generated by new sequencing technologies, the scientific requirement to integrate multiple lines of evidence to validate more complex systems models, and the need to provide systems software with analytic capabilities that scaled beyond the desktop tools popular with individual scientists. Plus, Cambridge, Massachusetts, is home to two of the world's great universities, to labs operated by several of the world's largest pharmaceutical makers and research institutions, and any number of startups, all home to scientists and researchers seeking to understand how our genes and our environment combine and interact to make us sick or well. The underlying methods these researchers used appeared well-suited to SciDB's data model and methods. A chromosome is an ordered list of nucleotides. Teasing out causal relationships between genetic variants and patient outcomes involves methods like extremely large-scale statistical calculations. Even minor features like SciDB's no-overwrite storage were interesting in bioinformatics because of the need to guarantee the reproducibility of analytic query results.

SciDB had also attracted interest from a number of more "pure" science projects. We worked with a couple of National Labs on projects as diverse as spotting weakly interacting massive particles in the bottom of a mine in South Dakota, tracking stormy weather patterns in RADAR data over the USA with a NASA team, and looking for evidence of climate change in satellite images of the Brazilian rainforest. Interestingly, a pattern that emerged from these early engagements saw SciDB eagerly embraced by smaller, more poorly funded project teams. Given abundant resources scientists still preferred "rolling their own." SciDB proved useful when resources were constrained.

It didn't take long, working with these early adopters, to realize that once we got above base camp, the mountain we were on didn't match the one we'd seen while approaching over distant plains.

First, an implicit assumption almost universal to the scientists we talked to held that all array data was dense. In their view, scientific data consisted of collections of rectilinear images captured at regular intervals. While it might be ragged in the sense that the right-angled corners of a satellite's camera don't mold exactly to the contours of a spherical planet, image data itself doesn't have "holes." What we found out quite quickly was that the biomedical, and even much of the "pure" science data we were handed, was actually sparse. It was characterized by gaps in space and time and position rather than continuity. Of course, the matrices derived from the sparse data and used to perform the mathematical and numerical analysis were dense, as was much of the image and sensor data.

So, we had to rethink aspects of our approach. Instead of simply breaking the data up into blocks, serializing each block and writing serialized blocks to disk, we were obliged to come up with a method for associating each serialized value with its logical position in the array. We accomplished this with a kind of bit-mask. Given a logical array, we would first generate a list of the logical positions for each cell that occurred in the data, and then compress and encode this list. Thus, for dense data, the Run Length Encoding (RLE) representation of "all cells are present" could be very terse: 48 bytes of metadata per megabyte data chunk. Alternatively, for very sparse data, the same method yielded a terse entry per valid cell. And the most common case, long runs of values with the occasional run of empty cells, also compressed well.

Making this change required inflicting considerable violence on our storage layer, executor, and operator logic. But it made SciDB equally proficient at both dense and sparse arrays. Our query language and, therefore, SciDB's users, did not need to know which kind of array data they were dealing with.

Second, where we expected to get data from external applications already organized into matrix rows and columns, we found that the more usual organization was highly "relational." Instead of a file that listed all values in the first row, then all values in the second, and so on, the more usual presentation was a { row #, column #, values . . . } file. This meant that we had to reorganize data on load into the SciDB array form. Not a conceptually difficult task. But it meant that, in addition to keeping up with our planned development schedule, we had to implement a highly efficient, massively parallel sort operation. Without additional staff.

Third, we learned that the kinds of data encoding methods familiar to anyone who has worked on column-store DBMSs were a very poor substitute for more specialized encoding methods used for digital audio and video. Several of SciDB's initial customers wanted to combine video data with sensor data from wearables. They wanted to know, for example, what physical movement in a video could be temporally associated with accelerometer changes. But when we tried to pull video data into SciDB using column-store encodings, we found ourselves bloating data volumes. Video and audio data is best managed using specialized formats that, unlike techniques such as Adaptive Huffman or RLE, pay attention to regions of data rather than a simple serialization of values. To accommodate audio and video, we were obliged to retrofit these specialized methods into SciDB by storing blocks of audio and video data separately from the array data.

These early surprises were the consequence of not talking to enough people before designing SciDB. No bioinformatics researchers were included on the list of scientists we consulted.

Another set of requirements we had not fully appreciated revolved around the questions of elasticity and high availability. In several SciDB applications, we found the data and workload scale required that we run across dozens of physical compute nodes. This, combined with the fact that many of the linear algebra routines we were being asked to run were quadratic in their computational complexity, suggested that we needed to confront the prospect of dynamically expanding compute resources and the certainty of hardware faults and failures in our design.

Adding (and subtracting) computers from a running cluster without shutting it down involves some pretty sophisticated software engineering. But over time, we were able to support users running read queries while the overall system had some physical nodes faulted out, and we were even able to add new physical nodes to a running cluster without requiring that the overall system be shut down or even quiesced. With the emergence of cloud computing, we expect such functionality will become what Mike Stonebraker refers to as "table stakes."

Expeditions also measure progress by the thing they leave behind. We had learned about new requirements on the climb. But our aspirations exceeded our staffing. So, as we began to get feedback about what our priorities ought to be by talking to Paradigm4's early SciDB customers, we trimmed our ambition and deferred features that were not required immediately.

First to go was lineage support. Although simple enough in principle—the combination of our no-overwrite, versioning storage layer and the way we maintained a comprehensive log of every user query gave us all the information we needed—it was difficult to nail down the precise list of requirements. For some users it would have been enough to record data loading operations and the files that served as data sources. For others there was a need to track where every cell's value came from. Others mentioned that they would like to track the precise versions of data used in every read query to ensure that any interesting result could be reproduced. Yet provenance support was never anyone's highest priority. Performance or stability or some analytic feature always took precedence.

We also never managed to embed probabilistic reasoning or to make managing uncertainty a first-class feature of the data model. For one user we went so far as to implement a full probability distribution function as a basic data type applied in conjunction with a very large-scale Monte Carlo calculation: 128 gigabytes of input data, a 2-gigabyte static distribution, and 10,000 simulation runs. But this was accomplished through user-defined type and function extensibility. As with provenance, supporting uncertainty was no one's highest priority.

Our SQL-like query language was another casualty at this time. We had focused considerable effort on language bindings to client languages like "R" and Python—as these were the languages preferred by bioinformaticians, quants, and data scientists. In neither of these languages was the kind of Connection/Query/Result/Row-at-a-Time procedural interface designed for languages like COBOL, C/C++, or Java appropriate. Instead, the idea was to embed data management operations at a higher level of abstraction directly within the language. For example, "R" has a notion of a data frame, which is logically similar to a SciDB array. So, in addition to mechanisms to pull data out of a SciDB array to place it in a client side "R" program, we tried to design interfaces that would obscure the physical location of the data frame data. A user might interact with an object that behaved exactly as an "R" data frame behaved. But under the covers, the client interface passed the user action through to SciDB where a logically equivalent operation was performed—in parallel, at scale, and subject to transactional quality of service guarantees—before handing control back to the end user. A SQL-like language was therefore superfluous. So, we prioritized the Array Functional Language (AFL).

Sometimes it's the things you say "no" to that end up being critical to your success. Our engineering constraints led us to defer a large list of line-item level features: non-integer dimensions, additional data encodings, orthodox memory management, and a fully featured optimizer, among others. Given such limitations, the climb ahead looked daunting. Nevertheless, we persisted.

On Peaks

"The top of one mountain is always the bottom of another."

—Marianne Williamson

"What a deepity!"

—Daniel Dennett

"It is not the mountains that we conquer, but ourselves."

—Edmund Hillary

"That's better."

—Daniel Dennett

Six years into the project, we can look back over the ground we've covered with considerable satisfaction.

As of late 2017 SciDB is used in production in well over a dozen "industrial" applications, mostly in bioinformatics, but with a mixed bag of other use cases for good measure. SciDB is the engine behind public data sites like the National Institutes of Health's 1000 Genomes Browser and the Stanford Medical School's Global Biobank Engine. As we mentioned earlier, many of these production systems are characterized by the way SciDB proved itself the best, least expensive option—in terms of application development time and capital investment—among alternatives. What we've learned from our users is that there are significant hidden costs in developing applications when your "database" is really just a "big swamp of files." SciDB applications run the gamut from very large databases containing the sequenced genomes of thousands of human beings, to applications that use video and motion sensor data to analyze fine-grained details of how different kinds of human body respond in different circumstances, to financial applications with very high data throughput and sophisticated analytic workloads.

But perhaps the most exciting thing, for a platform built with the intention of furthering scientific study, is the number of pure science research projects that have come to rely on SciDB as their data management tool and analytic platform. The SciDB solutions team are collaborating with researchers from institutions such as Harvard Medical School, Purdue University, NASA, and Brazil's INPE, as well as continuing historical collaborations with national laboratories. SciDB's unprecedented extensibility and flexibility has allowed researchers to perfect new techniques and methods and then make them available to a broader academic community [Gerhardt et al. 2015].

Mike Stonebraker was there at the beginning. He recognized the importance of the work, and the fundamental interest of the climb. Without his influence and reputation our expedition might never have even begun: Through some of the most difficult economic times of the last 50 years he managed to move us ahead until we were fortunate enough to secure the funding to start our work. Throughout the climb he has been a steady and congenial companion, an opinionated but invaluable guide, a wise and pragmatic Sherpa, and a constant reminder of the importance of just putting one foot in front of the other.

Acknowledgments

The author would like to acknowledge the invaluable and generous assistance of Marilyn Matz and Alex Poliakov, the CEO and Solutions Group Manager at Paradigm4, respectively, who read a draft of this chapter and provided memories and suggestions.

21

Data Unification at Scale: Data Tamer

Ihab Ilyas

In this chapter, I describe Mike Stonebraker's latest start-up, Tamr, which he and I co-founded in 2013 with Andy Palmer, George Beskales, Daniel Bruckner, and Alexander Pagan. Tamr is the commercial realization of the academic prototype "Data Tamer" [Stonebraker et al. 2013b]. I describe how we started the academic project in 2012, why we did it, and how it evolved into one of the main commercial solution providers in data integration and unification at the time of writing this chapter.

Mike's unique and bold vision targeted a problem that many academics had considered "solved" and still provides leadership in this area through Tamr.

How I Got Involved

In early 2012, Mike and I, with three graduate students (Mike's students, Daniel Bruckner and Alex Pagan, and my student, George Beskales), started the *data tamer* project to tackle the infamous data-integration and unification problems, mainly record deduplication and schema mapping. At the time, I was on leave from the University of Waterloo, leading the data analytics group at the Qatar Computing Research Institute, and collaborating with Mike at MIT on another joint research project.

Encouraged by industry analysts, technology vendors, and the media, "big data" fever was reaching its peak. Enterprises were getting much better at ingesting massive amounts of data, with an urgent need to query and analyze more diverse datasets, and do it faster. However, these heterogeneous datasets were often accumulated in low-value "data lakes" with loads of dirty and disconnected datasets.

Somewhat lost in the fever was the fact that analyzing "bad" or "dirty" data (always a problem) was often worse than not analyzing data at all—a problem now multiplied by the variety of data that enterprises wanted to analyze. Traditional data-integration methods, such as ETL (extract, transform load), were too manual and too slow, requiring lots of domain experts (people who knew the data and could make good integration decisions). As a result, enterprises were spending an estimated 80% of their time *preparing* to analyze data, and only 20% actually analyzing it. We really wanted to flip this ratio.

At the time, I was working on multiple data quality problems, including data repair and expressive quality constraints [Beskales et al. 2013, Chu et al. 2013a, Chu et al. 2013b, Dallachiesa et al. 2013]. Mike proposed the two fundamental unsolved data-integration problems: record linkage (which often refers to linking records across multiple sources that refer to the same real-world entity) and schema mapping (mapping columns and attributes of different datasets). I still remember asking Mike: "Why deduplication and schema mapping?" Mike's answer: "None of the papers have been applied in practice. . . . We need to build it right." Mike wanted to solve a real customer problem: integrating diverse datasets with higher accuracy and in a fraction of the time. As Mike describes in Chapter 7, this was the "Good Idea" that we needed! We were able to obtain and use data from Goby, a consumer web site that aggregated and integrated about 80,000 URLs, collecting information on "things to do" and events.

We later acquired two other real-life "use cases": for schema integration (from pharmaceutical company Novartis, which shared its data structures with us) and for entity consolidation (from Verisk Health, which was integrating insurance claim data from 30-plus sources).

Data Tamer: The Idea and Prototype

At this point we had validated our good idea, and we were ready to move to Step Two: assembling the team and building the prototype. Mike had one constraint: "Whatever we do, it better scale!" In the next three months, we worked on integrating two solutions: (1) scalable schema mapping, led by Mike, Daniel, and Alex, and (2) record deduplication, led by George and me. Building the prototype was a lot of fun and we continuously tested against the real datasets. I will briefly describe these two problems and highlight the main challenges we tackled.

Schema Mapping. Different data sources might describe the same entities (e.g., customers, parts, places, studies, transactions, or events) in different ways and using different vocabularies and schemas (a schema of a dataset is basically a

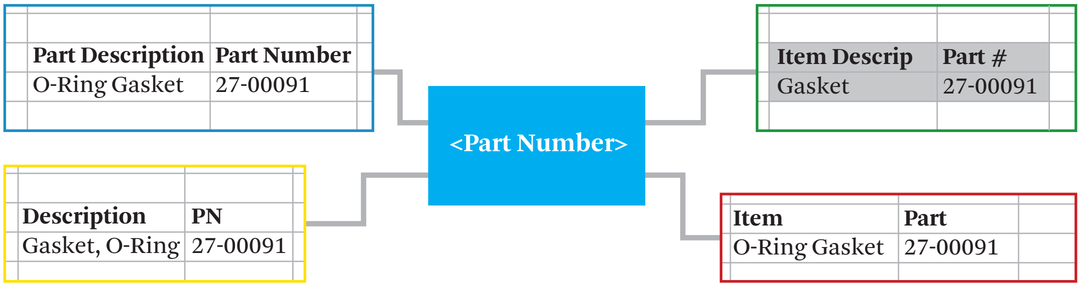


Figure 21.1 Schema mapping is needed to link all different columns describing the part number. (Source: Tamr Inc.)

formal description of the main attributes and the type of values they can take). For example: While one source might refer to a *part of a product* as two attributes (*Part Description* and *Part Number*), a second source might use the terms *Item Descrip* and *Part #*, and a third might use *Desc*. and *PN* to describe the same thing (cf. Figure 21.1). Establishing a mapping among these attributes is the main activity in schema mapping. In the general case, the problem can be more challenging and often involves different conceptualizations, for example when relationships in one source are represented as entities in another, but we will not go through these here.

Most commercial schema mapping solutions (usually part of an ETL suite) traditionally focused on mapping a small number of these schemas (usually fewer than ten), and on providing users with suggested mappings taking into account similarity among columns' names and their contents. However, as the big data stack has matured, enterprises can now easily acquire a large number of data sources and have applications that can ingest data sources as they are generated.

A perfect example is clinical studies in the pharmaceutical industry, where tens of thousands of studies/assays are conducted by scientists across the globe, often using different terminologies and a mix of standards and local schemas. Standardizing and cross-mapping collected data is essential to the companies' businesses, and is often mandated by laws and regulations. This changed the main assumption of most schema mapping solutions: suggestions curated by users in a primarily manual process. Our main challenges were: (1) how to provide an automated solution that required reasonable interaction with the user, while being able to map thousands of schemas; and (2) how to design matching algorithms robust enough to accommodate different languages, formats, reference master data, and data units and granularity.

Fname	Lname	Occupation	Institution	Number Students
Michael	Stonebraker	Professor	UC Berkeley	5
Mike	Stonebraker	PI	MIT-CSAIL	4+2 postdocs
M	Stonebreaker	Adjunct Professor	MIT	16
Mike Stonebraker		Faculty	Massachusetts Institute of Technology	n/a

Figure 21.2 Many representations for the same Mike!

Record Deduplication. Record linkage, entity resolution, and record deduplication are a few terms that describe the need to unify multiple mentions or database records that describe the same real-world entity. For example, "Michael Stonebraker" information can be represented in different ways. Consider the example in Figure 21.2 (which shows a single schema for simplicity). It's easy to see that the four records are about Mike, but they look very different. In fact, except for the typo in Mike's name in the fourth record, all these values are correct or were correct at some point in time. While it's easy for humans to judge if such a cluster refers to the same entity, it's hard for a machine. Therefore, we needed to devise more robust algorithms that could find such matches in the presence of errors, different presentations, and mismatches of granularity and time references.

The problem is an old one. Over the last few decades, the research community came up with many similarity functions, supervised classifiers to distinguish matches from non-matches, and clustering algorithms to collect matching pairs in the same group. Similar to schema mapping, current algorithms can deal with a few thousands of records (or millions of records but partitioned in disjointed groups of thousands of records!) However, given the massive amount of dirty data collected—and in the presence of the aforementioned schema-mapping problem—we now faced multiple challenges, including:

1. how to scale the quadratic problem (we have to compare every record to all other records, so computational complexity is quadratic in the number of records);
2. how to train and build machine learning classifiers that handle the subtle similarities as in Figure 21.2;

3. how to involve humans and domain experts in providing training data, given that matches are often rare; and
4. how to leverage all domain knowledge and previously developed rules and matchers in one integrated tool.

Mike, Daniel, and Alex had started the project focusing on schema mapping, while George and I had focused on the deduplication problem. But it was easy to see how similar and correlated these two problems were. In terms of similarity, both problems are after finding matching pairs (attributes in the case of schema mapping, records in the case of deduplication).

We quickly discovered that most building blocks we created could be reused and leveraged for both problems. In terms of correlation, most record matchers depend on some known schema for the two records they compare (in order to compare apples to apples); however, unifying schemas requires some sort of schema mapping, even if not complete.

For this and many other reasons, Data Tamer was born as our vision for consolidating these activities and devising core matching and clustering building blocks for data unifications that could: (1) be leveraged for different unification activities (to avoid piecemeal solutions); (2) scale to a massive number of sources and data; and (3) have human in the loop as a driver to guide the machine in building classifiers and applying the unification at large scale, in a trusted and explainable way.

Meanwhile, Stan Zdonik (from Brown University) and Mitch Cherniack (from Brandeis University) were simultaneously working with Alex Pagan on *expert sourcing*: crowdsourcing, but applied inside the enterprise and assuming levels of expertise. The idea was to use a human in the loop to resolve ambiguities when the algorithm's confidence on a match falls below a threshold. They agreed to apply their model to the Goby data to unify entertainment events for tourists.

Our academic prototype worked better than the Goby handcrafted code and equaled the results from a professional service on Verisk Health data. And it appeared to offer a promising approach to curate and unify the Novartis data (as mentioned in Chapter 7).

The vision, prototype, and results were described in the paper "Data Curation at Scale: The Data Tamer System," presented at CIDR 2013, the Sixth Biennial Conference on Innovative Data Systems Research in California [Stonebraker 2013].

The Company: Tamr Inc.

Given Mike's history with system-building and starting companies, it wasn't hard to see where he was going with Data Tamer. While we were building the prototype, he clearly indicated that the only way to test “this” was to take it to market and to start a VC-backed company to do so. And Mike knew exactly who would run it as CEO: his long-term friend and business partner, Andy Palmer, who has been involved with multiple Stonebraker start-ups (see Chapter 8). Their most recent collaboration at the time was the database engine start-up Vertica (acquired in 2011 by Hewlett-Packard (HP) and now part of Micro Focus).

Tamr was founded in 2013 in Harvard Square in Cambridge, Massachusetts, with Andy and the original Data Tamer research team as co-founders. The year 2013 was also when I finished my leave and went back to the University of Waterloo and George moved to Boston to start as the first full-time software developer to build the commercial Tamr product, with Daniel and Alex leaving grad school to join Tamr[1] as full-time employees.

Over the years, I have been involved in few start-ups. I witnessed all the hard work and the amount of anxiety and stress sometimes associated with raising the seed money. But things were different at Tamr: The credibility of the two veterans, Mike and Andy, played a fundamental role in a fast, solid start, securing strong backing from Google Ventures and New Enterprise Associates (NEA). Hiring a world-class team to build the Tamr product was already under way.

True to Mike’s model described in his chapter on how to build start-ups, our first customer soon followed. The problem Tamr tackled, data unification, was a real pain point for many large organizations, with most IT departments spending months trying to solve it for any given project. However, a fundamental problem with data integration and data quality is the non-trivial effort required to show return on investment in starting these large-scale projects, like Tamr. With Tamr living much further upstream (close to the silo-ed data sources scattered all over the enterprise), we worked hard to show the real benefit of unifying all the data on an enterprise’s final product or main line of business—unless the final product is the curated data itself, as in the case of one of Tamr’s early adopters, Thomson Reuters, which played a key role in the early stages of Tamr creation.

Thomson Reuters (TR), a company in which curated and high-quality business data *is* the business, was thus a natural early adopter of Tamr. The first deployment of Tamr software in TR focused on deduplicating records in multiple key datasets that drive multiple businesses. Compared to the customer’s in-house, rule-based record matchers, Tamr’s machine learning-based approach (which judiciously involves TR experts in labeling and verifying results) proved far superior. The quality of

[1] The commercial name was changed from Data Tamer to Tamr, as Data Tamer had already been taken.

results matched those of human curators on a scale that would have taken humans literally years to finish [Collins 2016].

With the success of the first deployment, the first product release was shaping up nicely. Tamr officially launched in May 2014 with around 20 full-time employees (mostly engineers, of course), and a lineup of proofs of concepts for multiple organizations.

As Mike describes in Chapter 8, with TR as the "Lighthouse Customer," Andy Palmer the "adult supervisor," and the strong support of Google Ventures and NEA, Steps 3, 4, and 5 of creating Tamr the company were complete.

More enterprises soon realized that they faced the same problem—and business opportunity—with their data as TR. As I write this, Tamr customers include GE, HP, Novartis, Merck, Toyota Motor Europe, Amgen, and Roche. Some customers—including GE, HP, Massachusetts Mutual Insurance, and TR—went on to invest in our company through their venture-capital arms, further validating the significance of our software for many different industries.

In February 2017, the United States Patent and Trademark Office issued Tamr a patent (US9,542,412) [Tamr 2017] covering the principles underlying its enterprise-scale data unification platform. The patent, entitled "Method and System for Large Scale Data Curation," describes a comprehensive approach for integrating a large number of data sources by normalizing, cleaning, integrating, and deduplicating them using machine learning techniques supplemented by human expertise. Tamr's patent describes several features and advantages implemented in the software, including:

- the techniques used to obtain training data for the machine learning algorithms;
- a unified methodology for linking attributes and database records in a holistic fashion;
- multiple methods for pruning the large space of candidate matches for scalability and high data volume considerations; and
- novel ways to generate highly relevant questions for experts across all stages of the data curation lifecycle.

With our technology, our brand-name customers, our management team, our investors, and our culture, we've been able to attract top talent from industry and universities. In November 2015, our company was named the #1 small company to work for by *The Boston Globe*.

Mike's Influence: Three Lessons Learned.

I learned a lot from Mike over the last five years collaborating with him. Here are three important lessons that I learned, which summarize his impact on me and are indicative of how his influence and leadership have shaped Tamr's success.

Lesson 1: Solve Real Problems with Systems

A distinctive difference of Tamr (as compared to Mike's other start-ups) is how old and well-studied the problem was. This is still the biggest lesson I learned from Mike: It doesn't really matter how much we think the problem is solved, how many papers were published on the subject, or how "old" the subject is, if real-world applications cannot effectively and seamlessly *use a system* that solves the problem, it is *the* problem to work on. In fact, it is Mike's favorite type of problem. Indeed, we're proud that, by focusing on the challenge of scale and creating reusable building blocks, we were able to leverage and transfer the collective effort of the research community over the last few decades, for practical adoption by industry—including a large number of mega enterprises.

Lesson 2: Focus, Relentlessly

Mike's influence on the type of challenges Tamr will solve (and won't) was strong from Day One. In the early days of Tamr, a typical discussion often went as follows.

Team: "Mike, we have this great idea on how to enable Feature X using this clever algorithm Y."

Mike (often impatiently): "Too complicated . . . Make it simpler . . . Great for Version 10 . . . Can we get back to scale?"

I have often measured our progress in transferring ideas to product by the version number Mike assigns to an idea for implementation! (Lower being better, of course). His impressive skill in judging the practicality and the probability of customer adoption is one of Mike's strongest skills in guiding the construction of adoptable and truly useful products.

Lesson 3: Don't Invent Problems. Ever

Mike simply hates inventing problems. If it isn't somebody's pain point, it is not important. This can be a controversial premise for many of us, especially in academia. Far too often in academia, the argument is about innovation and solutions to fundamental theoretical challenges that can open the door for new practical problems, and so on.

In identifying problems, my lesson from Mike was not to be convinced one way or another. Instead, simply take an extreme position and make the biggest tangible impact with it. Mike spends a lot of his time listening to customers, industry practitioners, field engineers, and product managers. These are Mike's sources of challenges, and his little secret is to always look to deliver the biggest bang for the buck. As easy as it sounds, talking to this diverse set of talents, roles, and personalities is an art, requiring a good mix of experience and "soft" skills.

Watching Mike has greatly influenced the way I harvest, judge, and approach research problems, not only at Tamr but also in my research group at Waterloo. These lessons also explain the long list of his own contributions to both academia and industry to deserve computing's highest honor.

22 The BigDAWG Polystore System

Tim Mattson, Jennie Rogers, Aaron J. Elmore

The BigDAWG polystore system is for many of us the crowning achievement of our collaboration with Mike Stonebraker during the years of the Intel Science and Technology Center (ISTC) for Big Data. Perhaps the best way to explain this statement is to break it down into its constituent components.

Big Data ISTC

The Intel Science and Technology Center (ISTC) for Big Data was a multi-university collaboration funded over five years (2012–2017) by Intel. The idea was that certain problems are so big and so complex that they need a focused investigation free from the constraints of industry product cycles or academic NSF grant-chasing. When faced with such problems, Intel steps in and funds a group of professors over a three- to five-year period to address those problems. The research is open-IP or, in the language of industry, pre-competitive research designed to further the state of the art in a field rather than create specific products.

Big Data, whatever that pair of words means to you, clearly falls into this category of problem. In 2012, Intel worked with Sam Madden and Mike Stonebraker of MIT to launch the ISTC for Big Data. This center included research groups at MIT, the University of Washington, Brown University, Portland State University, UC Santa Barbara, and the University of Tennessee. Over time the cast of characters changed. We lost the UC Santa Barbara team and added research groups at Carnegie Mellon, Northwestern University, and the University of Chicago.

The authors of this chapter came together as part of this ISTC: one of us (Tim Mattson) as the Intel Principal Investigator (PI) for the ISTC, and the others (Jennie Rogers and Aaron Elmore) as postdocs at MIT. By the end of the project, Tim was still at Intel, but Jennie and Aaron, in part due to the success of our work in the ISTC, were assistant professors at Northwestern and the University of Chicago, respectively.

The Origins of BigDAWG

BigDAWG started as a concept in the mind of the Intel PI for the Big Data ISTC. The center was one year old when the Intel PI was drafted into that role. It was an awkward role since his background was in high-performance computing (HPC) and computational physics. Data was something other people worried about. The I/O systems on supercomputers were generally so bad that in HPC you went out of your way to pick problems that didn't depend on lots of data. Hence, HPC people, almost by design, know little of data management.

Collaborations and how to make them work, however, is something anyone well into a research career learns. Getting professors to work together toward a common goal is an unnatural act. It happens only with deliberate focus, and to create that focus we needed a common target for everyone to rally around. It wasn't called BigDAWG yet, but this initial seed—common Big Data solution stack into which many projects would connect—was there. In Figure 22.1, we reproduce the earliest PowerPoint slide representing what later became BigDAWG. At the top level were the visualization and applications-oriented projects. Underneath were various data stores ranging from pure storage engines to full-fledged database management systems. Underneath those data stores were math libraries tightly coupled to them to support big data analytics. And in the middle, a "narrow waist" that would create a common middleware to tie everything together.

That narrow waist was a simple API everyone could use to connect their systems together. It would, at least in the mind of the HPC guy on the team, be a simple software layer to create. We just needed a messaging-based API so the different packages would know how to connect to each other. It would take a few graduate students under the direction of a wise professor a quarter or two to pull it together.

Mike quickly picked up on how naïve the *HPC guy* was. Because of Mike's experience building real systems over the years, he immediately recognized that the real contribution of this work was that "narrow waist." Getting that right would be a project of massive scale. Hence, with each successive meeting and each successive version of this grand solution PowerPoint stack, the narrow waist grew and the other parts of the figure shrank. We eventually ended up with the picture in Figure 22.2, where the "narrow waist" had now become the API, islands, shims, and casts that dominate the system (and which will be explained later).

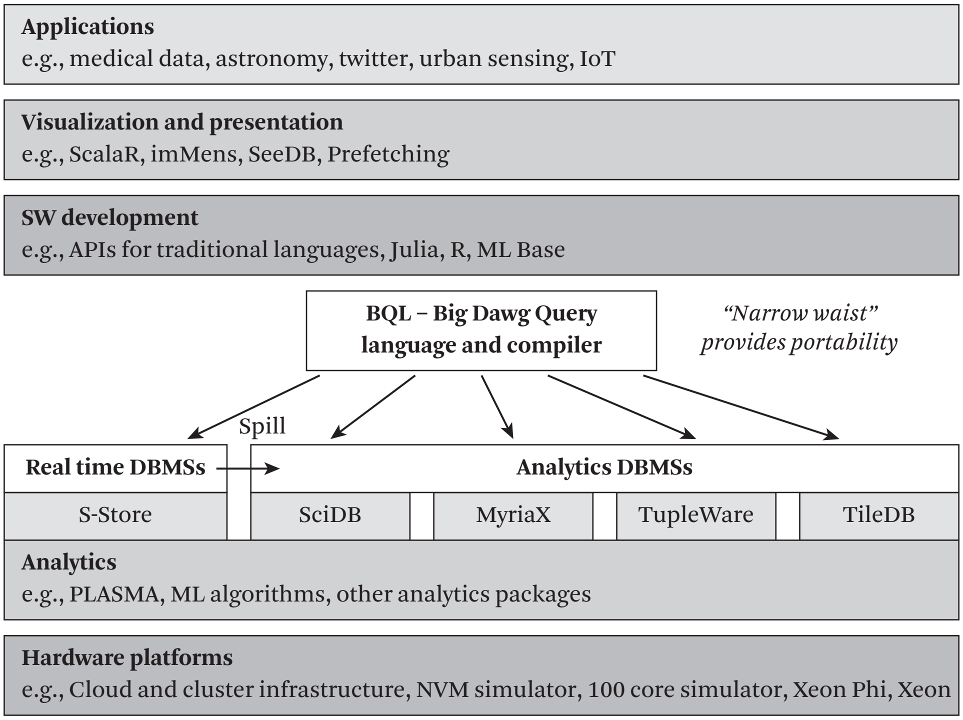

Applications
e.g., medical data, astronomy, twitter, urban sensing, IoT
Visualization and presentation
e.g., ScalaR, imMens, SeeDB, Prefetching
SW development
e.g., APIs for traditional languages, Julia, R, ML Base
BQL – Big Dawg Query language and compiler
"Narrow waist" provides portability
Spill
Real time DBMSs
Analytics DBMSs
S-Store
SciDB
MyriaX
TupleWare
TileDB
Analytics
e.g., PLASMA, ML algorithms, other analytics packages
Hardware platforms
e.g., Cloud and cluster infrastructure, NVM simulator, 100 core simulator, Xeon Phi, Xeon

Figure 22.1 The original BigDAWG concept.

Getting professors to work so closely together is akin to "herding cats." It's difficult if the glue holding them together is a nebulous "solution stack." You need a name that people can connect to, and Mike came up with that name. Sam Madden, Mike, and the humble Intel PI were sitting in Mike's office at MIT. We were well aware of the attention the Big Data group from UC Berkeley was getting from its BDAS system (pronounced "bad ass"). Mike said something to the effect "they may be bad asses, but we're the big dog on the street." This was offered in a tone of a friendly rivalry since most of us have long and ongoing connections to the data systems research groups at UC Berkeley. The name, however, had the right elements. It was light-hearted and laid down the challenge of what we hoped to do: ultimately create the system that would leap past current state of the art (well-represented by BDAS).

It took a while for the name to stick between the three of us. Names, however, take on a life of their own once they appear in PowerPoint. Hence, within a few ISTC meetings and a recurring PowerPoint presentation or two, the name stuck. We were

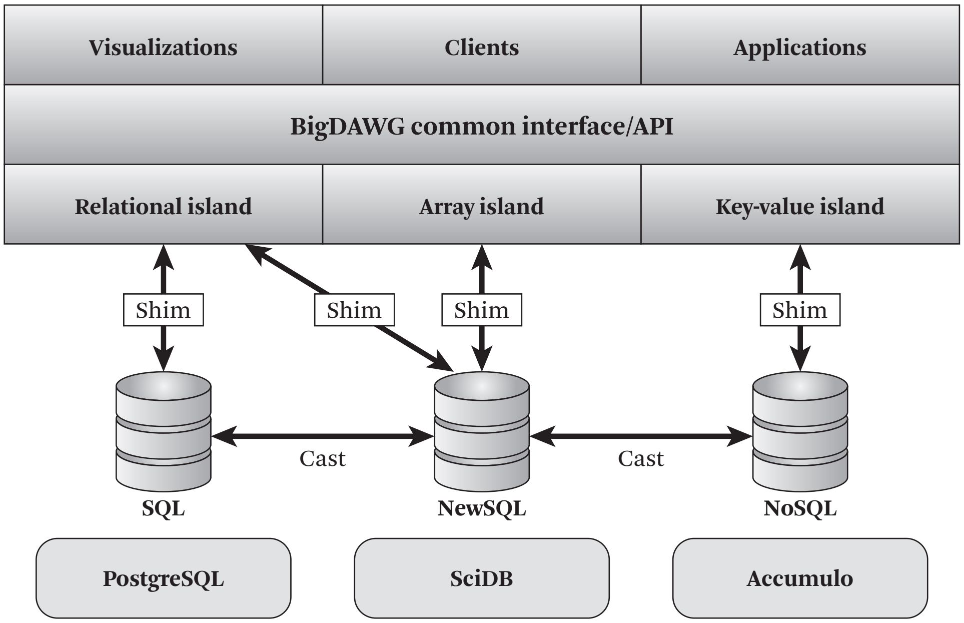


Figure 22.2 The final BigDAWG concept.

creating the BigDAWG solution stack for Big Data. What exactly that meant in terms of a long-term contribution to computer science, however, wasn't clear to anyone. That required yet another term that Mike coined: polystore.

One Size Does Not Fit All and the Quest for Polystore Systems

"One size does not fit all" embodies the idea that the structure and organization of data is so diverse that you cannot efficiently address the needs of data with a single data store. This famous slogan emerged from Mike's career of building specialized systems optimized for a particular use case. He first explored this topic with Ugur Çetintemel in an ICDE 2005 paper [Stonebraker et al. 2005b], which later won the Test of Time Award (2015). Mike further validated this bold assertion with benchmarking results in CIDR 2007 [Stonebraker et al. 2007a].

The design of these specialized "Stonebraker Stores" favored simplicity and elegance. Mike would use the heuristic that it was not worth building a specialized store unless he thought he could get at least an order of magnitude of performance improvement over "the elephants."[1]

[1] Mike's term for the dominant RDBMS products.

A decade later, we have seen an explosion in specialized data management systems, each with its own performance strengths and weaknesses. Further complicating this situation, many of these systems introduced their own data models, such as array stores, graph stores, and streaming engines. Each data model had its own semantics and most had at least one domain-specific language.

At least two problems arose from this plethora of data stores. First, this encouraged organizations to scatter their data across multiple data silos with distinct capabilities and performance profiles. Second, programmers were deluged with new languages and data models to learn.

How could the "narrow waist" make these many disparate systems work together (relatively) seamlessly? How could we take advantage of the performance strengths of each data store in an ecosystem? How could we meet the data where it was and meet the programmers where they were? In an age where new data management systems are becoming available all the time, is this attempt at unifying them all a fool's errand?

These questions dominated our work on BigDAWG, with the dream of bringing together many data management systems behind a common API. Mike coined the name "polystore" to describe what we hoped to build. At the simplest level, a polystore is a data management system that exposes multiple data stores behind a single API. We believe, however, that to fully realize the direction implied by "one size does not fit all," we need to take the definition further: to distinguish a polystore from related concepts such as data federation. A data federation system unifies multiple database engines behind a single data model, most commonly the relational model. In the classic idea of a data federation system, however, the database engines contained in the system are completely independent and autonomous. The data federation layer basically creates a virtual single system without fundamentally changing anything inside the individual engines or how data is mapped to them.

When we use the term polystore, we refer to single systems composed of data stores that are tightly integrated. A data store may be a fully featured DBMS or a specialized storage engine (such as the TileDB array-based storage engine). Data can be moved between data stores and therefore transformed between data models attached to each engine. Queries provide location independence through an "island" in other words, a virtual data model that spans data stores. Alternatively, when a specific feature of one of the data stores is needed, a query can be directed to a particular engine. Hence, we see that the fundamental challenge of a polystore system is to balance location independence and specificity.

While we first heard the word "polystore" from Mike—who coined the term in the first place—this term is descriptive of systems people have been building for a while (as recently surveyed in [Tan et al. 2017]). Early polystore systems, such as Polybase [DeWitt et al. 2013] and Miso [LeFevre et al. 2014], focused on mixing big data systems and relational databases to accelerate analytic queries. In addition, IBM's Garlic project [Carey et al. 1995] investigated support for multiple data models in a single federated system. The Myria project [Halperin et al. 2014] at the University of Washington is a polystore system that emphasizes location independence by making all of the data stores available through an extended relational model.

Mike took a more pragmatic point and stressed that he believed there is no "Query Esperanto." The very reason to go with a polystore system is to expose the special features of the underlying data stores instead of limiting functionality based on the common interface. Hence, we built a query interface for BigDAWG that encapsulated the distinct query languages of each of the underlying data stores. This approach offered the union of the semantics of the underlying data stores and enabled clients to issue queries in the languages of their choice.

Putting it All Together

BigDAWG embraced the "one size does not fit all" mantra so people could benefit from the features of specialized storage engines. Choosing the right storage engines and physically integrating the systems is extremely complex. Our goal in designing BigDAWG was to simplify the lives of users and administrators without limiting the expressiveness or functionality of the data stores within BigDAWG. We introduced this novel architecture in SIGMOD Record [Duggan et al. 2015a].

We started by defining how we'd manage the abstractions specific to each class of storage engines. We did so by defining the concept of an *island*. An island is an abstract data model and a set of operators with which clients may query a polystore. We implemented *shims* to translate "island statements" into the language supported by each system. In a BigDAWG query, the user denotes the island he or she are invoking by specifying a *scope*. For example, a client invokes the relational scope in the following query:

RELATIONAL(SELECT avg(temperature) FROM sensor)

Scopes are composable, so one might combine the relational scope with an array scope such as the following:

ARRAY(multiply(A, RELATIONAL(SELECT avg(temperature) FROM sensor))

An island offers location independence; in other words, a single query using the island's model and operators returns the same answer for a given query/data pair regardless of which data store connected to the island holds the data. This may require data to move between data stores, which is accomplished through cast operators that map the storage layer of one store to that of another.

Query Modeling and Optimization

Polystore systems execute workloads comprising diverse queries that span multiple islands. If we can find and exploit "sweet spots" in a workload, where execution is significantly faster if specialized to a particular storage engine, we can realize potentially dramatic performance benefits. In other words, we needed to build into BigDAWG a capability to model the performance characteristics of distinct storage engines and capture the strengths and weaknesses of their query processing systems.

To match queries with storage engines, we took in a user's BigDAWG workload and observed the performance of its queries when they execute in different systems. Here our goal was to establish a set of query classes each of which would have an expected performance profile across engines. Hence, when new queries arrived, we would identify the class to which they belonged and be able to make better decisions about where they would run.

To learn a query's class, we execute the queries in an expansive mode, a process akin to the training phase of machine learning applications. This expansive mode executes the query on all of the associated storage engines that match the query and BigDAWG records the performance in each case. Expansive execution may be done all at once—when the query is initially submitted by the use—or opportunistically when slack resources arise in individual databases. The statistics collected during training are paired with a signature summarizing the query's structure and the data accessed. These results are compared to other queries the system has monitored to maintain an up-to-date, dynamic representation of the performance of the system for a given query class.

Armed with these query models, BigDAWG enumerates query plans over the disparate storage engines to identify those that will deliver the highest performance. Like traditional federated databases, BigDAWG's query optimizer represents its query plans using a directed acyclic graph. Planning polystore queries, however, is more complicated than for data federation systems since, depending on the pattern of shims and casts, BigDAWG supports engines with overlapping capabilities.

When the BigDAWG optimizer receives a query, it first parses the query to extract its signature. The planner then compares the signature to ones it has seen before and assigns it to a predicted performance profile. BigDAWG then uses this profile paired with the presently available hardware resources on each database to assign the query to one or more data stores. As BigDAWG executes a workload, it accumulates measurements about the query's performance. This allows the BigDAWG optimizer to incrementally refine its signature classification and performance estimates. This feature of the system was particularly complicated to implement since BigDAWG supports such diverse data models with a distributed execution plan.

Data Movement

To effectively query among multiple storage engines, a polystore system must be able to transform and migrate data between its systems. This data movement may be temporary to accelerate some portion of a query or to leverage functionality required by the user via a cast operator. Alternatively, the move may be permanent to account for load balancing or other workload-driven optimizations. Regardless of the reason, efficient just-in-time data migration is critical for BigDAWG.

To address data migration, BigDAWG includes a data migration framework to transform data between all member storage engines. The shims and cast operators between engines and islands of information provide the migration framework and logical transformations required to change data models. BigDAWG's migration framework is responsible for doing efficient extraction, transformation, movement, and loading of data, which is an example of differentiating component from federated systems. All storage engines in a BigDAWG system have a local migration agent running that listens to the query controller for when to move data. This information includes the destination engine, logical transformation rules, and required metadata about how the data is locally stored.

As most storage engines support a naïve CSV export and import functionality, the initial prototype utilized this common text-based representation to move data. However, this format requires a great deal of work to parse and convert data into the destination binary format. We explored having each engine support the ability to directly generate the destination binary format of other engines, which we found to be as much as 400% faster than the CSV-based approach. However, for a BigDAWG system that supports N database engines, writing custom connectors means writing N2 connectors for the systems, which results in significant code maintenance requirements. Instead, the migrator settled on a concise binary intermediate representation that is still 300% faster than CSV-based migration and only requires N connectors to be maintained. In a related ISTC project, researchers from the University of Washington developed a system that used program-synthesis to automatically generate connectors by examining source code for CSV importers and exporters [Haynes et al. 2016].

Significant effort went into optimizing the migration framework for efficient data transfer. We explored parallelization, SIMD (single instruction multiple data) based data transformation, lightweight compression, and adaptive ingestion when multiple methods exist for getting data into the destination system.

BigDAWG Releases and Demos[2]

Early in the project Mike realized that we needed a public demo as a "forcing function" to get the teams distributed across the ISTC to make progress quickly on our BigDAWG system. Many ISTC participants worked together for a demonstration at VLDB 2015 that coupled relational, array, and text-based databases for a series of workflows using a medical-based dataset (the MIMIC II dataset[3]).

Mike was instrumental in managing this ambitious project with a distributed and diverse team. The team outlined the initial prototype to support the demo, and a roadmap was constructed to force the march forward. Mike played a leadership role to help balance ambition with reality and made sure that unnecessary scope creep did not hinder the chances of assembling a system that was greater than its individual parts. Mike and Vijay Gadepally from MIT Lincoln Laboratory organized regular hackathons where team members from Seattle, Chicago, Providence, and Cambridge assembled at MIT to help glue the team and code together.

The experimental results from the demo system (summarized in Figure 22.3) was presented as a demo at VLDB [Elmore et al. 2015] and later published in a paper at the IEEE High Performance Extreme Computing Conference [Gadepally et al. 2016a]. This showed that leaving array data in an array-based DBMS (SciDB) and relational data in a relational DBMS (MyriaX) resulted in better performance compared to moving all the data into one DBMS or the other. This result demonstrated the benefits of the polystore concept.

For our second demo, we wanted data that was free from the privacy considerations connected to medical data. We settled on data from the Chisholm Laboratory at MIT (http://chisholmlab.mit.edu). This group works with metagenomics data collected from the ocean to understand the biology of Prochlorococcus, a tiny marine cyanobacteria responsible for 15–20% of all oxygen in earth's atmosphere. This

[2] For a description of the sequence of demonstrations, see Chapter 31.

[3] A medical dataset available at https://physionet.org/mimic2.

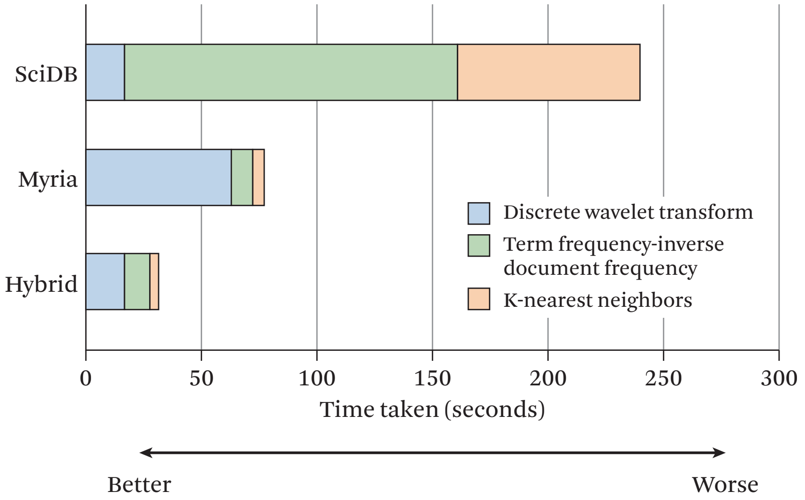


Figure 22.3 Performance of a complex analytic workflow over the MIMIC II dataset showing the benefit of matching different parts of the workflow to the data store best suited to the operation/data.

data was more heterogeneous than the medical datasets in our first demo and included a new streaming data island, S-Store [Meehan et al. 2015b], to represent real-time data from a moving data collection platform. This demo was presented at the Conference on Innovative Data Systems Research (CIDR) in 2017 [Mattson et al. 2017].

With two completed demo milestones, Mike and the ISTC PI drove toward another open-source "feather in Mike's cap." In 2017, the ISTC released BigDAWG 0.10 to the general public [Gadepally et al. 2017] (see Chapter 31). Vijay Gadepally and Kyle O'Brien of MIT Lincoln Laboratory played a major role in integrating the components that make up BigDAWG into a coherent package (see Chapter 31). Our job moving forward is to build a community around BigDAWG and hopefully participate in its growth as researchers download the software and build on it.

Closing Thoughts

We realized several important research goals with BigDAWG. It came at a time when the storage engine landscape was extremely fractured and offered a way to unify many disparate systems in a simple-to-use interface. Mike led us to explore how to get the most out of these diverse data management offerings. BigDAWG itself also served as a common project that the members of the ISTC could rally around to integrate their work into a larger system. The two demos showed that polystores make their underlying storage engines greater than the sum of their parts. The bigger question, however, is whether BigDAWG and the polystore concept will have a long-term impact that lives beyond the ISTC, and on that count it is too early to say. We are hopeful that an open-source community will emerge around the system. It is used at Lincoln Laboratory and we hope to attract other users. To help grow the polystore community, a group of us (Vijay Gadepally, Tim Mattson, and Mike Stonebraker) have joined forces to organize a workshop series on polystore systems at the IEEE Big Data Conference.

One of the biggest problems unaddressed by BigDAWG was the quest for a Query Esperanto. We believe such a quest is worthwhile and important. The current BigDAWG query language requires that the user specify the islands for each component of a query and that the query contents match the data model for an island. This gives us maximum flexibility to take advantage of the full features of any single island. It sacrifices “location independence” and the convenience of writing queries that work regardless of how data is distributed between storage engines. We avoided this quest since we choose to build a working system first. The quest, however, is important and could have a profound impact on the usability of polystore systems.

We already mentioned the work by the Myria team at the University of Washington. They have built a working system that uses an extended relational algebra to join multiple storage engine data models. A closely related project at Lincoln Laboratory and the University of Washington is exploring how linear algebra can be used to define a unifying high-level abstraction that can unify SQL, NoSQL, and NewSQL databases [Kepner et al. 2016]. It’s too early to say if this approach will lead to useful systems, but early theoretical results are promising and point to a bright future for polystores exposed behind a common query language.

23

Data Civilizer: End-to-End Support for Data Discovery, Integration, and Cleaning

Mourad Ouzzani, Nan Tang, Raul Castro Fernandez

Mike Stonebraker was intrigued by the problem of how to ease the pain data scientists face getting their data ready for advanced data analytics: namely, finding, preparing, integrating, and cleaning datasets from thousands of disparate sources. At the time, Mark Schreiber was a director of information architecture at Merck Research Laboratories, a research lab for a large pharmaceutical company where he oversaw approximately 100 data scientists. Mark told Mike that the data scientists spend 98% of their time on grunt work preparing datasets of interest and only one hour per week on useful work for running their analyses. This is well beyond the 60–80% usually reported in the literature [Brodie 2015]. In 2015, Laura Haas, who then led IBM's Accelerated Discovery Lab, described how they addressed this problem when building specialized solutions for different customers. In addition, Mike had heard numerous similar war stories from customers of Tamr, his startup that provides solutions for curating data at scale (see Chapters 21 and 30).

This chapter describes our journey with Mike in building Data Civilizer, an end-to-end platform to support the data integration needs of data scientists and enterprise applications with components for data discovery, data cleaning, data transformation, schema integration, and entity consolidation, together with an advanced workflow system that allows data scientists to author, execute, and retrofit these components in a user-defined order. Our journey explored different scenarios, addressed different challenges, and generated cool ideas to clean, transform, and otherwise prepare data for serious data analytics, which resulted in Data Civilizer.

We Need to Civilize the Data

For some time, data in enterprise data repositories, databases, data warehouses, and data lakes have been rapidly turning into data swamps, a collection of unstructured, ungoverned, and out-of-control datasets where data is hard to find, hard to use, and may be consumed out of context.[1] Enterprise and public data repositories (e.g., data.gov) are rapidly becoming Big Data swamps. For example, data swamps have many materialized views with no lineage information (i.e., multiple redundant copies without their method of generation) and are used as a place to dump data with a vague intent to do something with them in the future. In this context, the future never comes. Data owners quickly lose track of the data that goes into these swamps, causing nightmares for anyone needing to extract valuable insights from them. To convert data swamps into well-governed data repositories so that valuable insights could be discovered, we decided to build Data Civilizer, in order to *civilize the data*. Before exploring the requirements of such a system, we illustrate a common problem using an example representative of many that we encountered during the project.

The Day-to-Day Life of an Analyst

To determine if a particular variable is correlated with an activity of interest, you decide to use the Pearson Correlation Coefficient (PCC) to calculate that correlation. At this point, the task seems pretty obvious: just get the data for the variable and evidence of the activity and run a small PCC program on the data. In fact, you can use an algorithm in many existing libraries with no need to write your own implementation. It is not even 10 AM and you are already wondering how you will spend the rest of your day. After all, as soon as you get this PCC, you just need to write a couple of paragraphs justifying whether there is indeed a correlation or not, based on the result of the PCC analysis. It's an easy task. Of course, at this point reality hits you. Where is the data? Where can you find the data about the variable and the activity? In principle, this seemed obvious, but now, how do you know in which of the multiple databases, lakes, and spreadsheets you can find the necessary data? You decide to get to it ASAP and call Dave, an employee who has been working with similar data in the past. Dave, who is visibly upset by your unsolicited visit, does

[1] http://www.nvisia.com/insights/data-swamp. Last accessed March 22, 2018.

not know where the data is, but fortunately he points you to Lisa, who according to him, has been doing “some kind of analysis” with the indicator. In your next interaction, you decide to avoid a bad look and instead pick up the phone to call Lisa. Lisa points you to exactly the place where you can find the data. After wasting a few more minutes with different passwords, you find the right credentials and access the database. After a few more minutes needed to edit the right SQL query and *voilà!* You found the indicator data, so you are almost done. Right? No. That was wishful thinking.

The data is incomplete. As it stands, you cannot make any statistically significant conclusions. You need to join this data with another table, but joining the two tables is not obvious. It would be obvious if you knew that the column “verse_id” is an ID that you can use in a mapping table that exists somewhere else and gives you the mapping to the “indicator_id”. This is the column you must use to join the tables. Of course, the problems do not end here. The formats of those indicators—which were dumped to the database by different people at different times and for different purposes—are different. So before using the data, one solution is to transform them to a common representation. It’s a simple transformation, but it becomes a painful process in which you must first make sure you deal with any missing values in the column, or otherwise your PCC Python program will complain. But, oh well, it is dinner time, so at this point you decide to deal with this tomorrow.

Welcome to the unsexy, unrewarding, and complex problem of data discovery and preparation: how to find, prepare, stitch (i.e., join and integrate different datasets), and clean your data so that your simple PCC analysis can be done quickly and efficiently. It turns out that these tasks take most of the time analysts routinely spend in their day-to-day jobs. Of course, there are point solutions for some of the tasks. For example, you can definitely find outliers in a column, clean some data with custom transformations helped by software, and so on. However, the problem is that all of these stages are interconnected, and there is no tool that assists you end to end through the process and helps you understand what needs to be done next. Of course, the quality of many of the tools may be unsatisfactory and not meet your specific needs.

The Data Civilizer team, led by Mike, is looking into new ways of attacking these problems. In particular, Data Civilizer [Deng et al. 2017a, Fernandez et al. 2017a] is being developed to:

- profile datasets to discover both syntactic and semantic linkage between columns and uncover data lineage between the datasets;

- discover datasets relevant to the task at hand—the indicator and the activity in the example above (see Chapter 33);
- obtain access to these datasets;
- unite datasets, put duplicate data records in clusters, and create golden records out of the clusters;
- stitch together datasets through join paths;
- clean datasets with a limited budget;
- query datasets that live across different systems; and
- use a workflow engine to compose the above components in arbitrary ways.

Each of these tasks has received considerable attention on its own resulting in point solutions. Point solutions would help a data scientist with the data preparation task; however, an end-to-end system could better support the data scientist in solving each problem in context and benefit from synergies and optimization across the functions.

Designing an End-to-End System

If you have the sense that the example above is insurmountable, let us just say it is vastly oversimplified. How would you attack the problem? Well, as computer scientists, you would start thinking of how to slice the large problem into smaller chunks. By defining smaller problems very well, you can come up with technically sound solutions, and even write an evaluation and publish a paper! The problem is that successfully solving the smaller problems does not necessarily lead to an effective solution to the larger problem. Mike's approach is to attack the problem vertically—just find an end-to-end example—and at the same time, "keep it simple." End-to-end examples with simple prototypes have been a key guiding principle. Mike started with this proposal, which then became our roadmap for the project.

To design a system, or a quick end-to-end prototype, one needs to understand the requirements first, and that requires use cases. Mike would never accept testing ideas on synthetic data because "it's not realistic." So, Mike's principles include: contextualize your ideas and understand the real problems and the scope of your contributions. Based on that, our plan was to address the needs of real users, in other words, a "user's pain," as Mike puts it. So, Mike brought in several real-world use cases. One example came from Mark Schreiber of Merck and Faith Hill of the MIT data warehouse team. These individuals are faced with the challenge of sifting through a very large number of datasets (often in the thousands) to find data relevant to a particular task. He or she must find the data of interest and then curate it, i.e., generate a coherent output dataset from this massive pool of potentially relevant source data by putting it through a curation process that involves combining related datasets, removing outliers, finding duplicates, normalizing values, and so on. Another example was of a professional IT person, typified by Nabil Hachim of Novartis. He has an enterprise-wide integration task to perform. He must continuously put a collection of known datasets through a similar curation pipeline to generate a final collection of datasets for Novartis' data scientists.

The approach of seeking external users with real problems to give feedback on working prototypes is what eventually shaped the Data Civilizer project into its current direction. Only after numerous interactions and hours of collaboration does one get to learn what is the "highest pole of the tent" and design the system so as to avoid surprises when the "rubber hits the road" (both popular Mike phrases).

Data Civilizer has to be designed and built so that it meets the needs of such people and to "ease their pain." These are some of the modules that we have built so far:

1. a module to build an enterprise knowledge graph that summarizes and indexes the data as well as uncovers all possible syntactic and semantic relationships, via available ontologies;
2. a flexible data discovery system with several queries to find relevant data and possible ways to join them;
3. various ways to transform and clean the data through automatically discovering abbreviations and disguised, i.e., default value, missing values; and
4. a user-guided module to consolidate records found to be duplicates into one canonical representation or golden record.

Data Civilizer Challenges

In this section, we discuss in some detail certain challenges we have been working on to build Data Civilizer.

The Data Transformation Challenge

When integrating data from multiple sources there is often a need to perform different kinds of transformations. These transformations entail converting a data element from one representation to another, e.g., unit, currency, and date format conversions, and generating a semantically different but related value, e.g., airport code to city name, and ISBN to book title. While some transformations can be computed via a formula, such as pounds to kilograms, others require looking up in a dictionary or other data sources. For such semantic transformations, we could not find an adequate automatic system or tool. It is clear that semantic transformations cannot be computed solely by looking at the input values, for example, and applying a formula or a string operation. Rather, the required transformations are often found in a mapping table that is either explicitly available to the application (e.g., as a dimension table in a data warehouse) or is hidden behind a transformation service or a Web form.

So, the challenge was to find sources of data that are readily available and that could help to automate this process. So, Mike said: “Why don’t you try Web tables?” Indeed, many Web tables, such as airport code to city, SWIFT code to bank, and symbol to company, may contain just the transformations we are after, either entirely or partially. So, we started working on ways to automatically discover transformations given some input and output examples.

Mike also suggested looking at how to automatically exploit Web forms, as many of them, such as currency converters, can help in the transformation task. We further extended the work to exploit knowledge bases for covering more transformations, mostly prominent head-topic transformations, such as soccer player to birthplace, soccer player to birth date, or country to head of state. Another extension was to find non-functional transformations such as books to authors and teams to players. To evaluate our tool, Mike’s idea was to simply collect transformation tasks, mostly from engineers at Tamr, and then see how much coverage we could achieve using the different sources. As it turned out, the coverage was quite high. We were able to cover 101 transformation tasks out of 120. Preliminary ideas of our tool were first described in a vision paper in CIDR 2015 [Abedjan et al. 2015b]. We then presented a full demo in SIGMOD 2015 [Morcos et al. 2015] and a full paper at ICDE 2016 [Abedjan et al. 2016b]. Most importantly, we won a best demo award in SIGMOD 2015!

The Data Cleaning Challenge

Mike had introduced us to Recorded Future, an intelligence company that monitors more than 700,000 Web sources looking for threats and other intelligence. The idea was to get some of its event data and see how clean or dirty it was and whether existing or to-be-discovered approaches could help in detecting and repairing errors in this kind of data. While Mike was convinced that the data was dirty and that something ought to be done to clean it, he was skeptical about being able to “automatically" clean the data. In fact, he threw the following challenge at us: "If you can automatically clean 50% of the data, I will license the technology for my company," and "I believe you can clean 2% of the data." The challenge was daunting.

Mike helped secure a three-month snapshot of data extracted by Recorded Future: about 188M JSON documents with a total size of about 3.9 TB. Each JSON document contained extracted events defined over entities and their attributes. An entity can be an instance of a person, a location, a company, and so on. Events also have attributes. In total, there were 150M unique event instances.

Looking at the data using different profilers and through eyeballing, it was clear that it contained many errors. One major observation was that some of the reported events did not fit well together when putting them on a time scale. For example, we saw that within less than an hour Barack Obama was in Italy and in South Africa. We discovered several similar cases for people traveling around as well for other events such as insider transactions and employment changes. To capture this kind of error, we introduced a new type of temporal dependency, namely Temporal Functional Dependencies. The key challenges in discovering such rules stem from the very nature of Web data: extracted facts are (1) sparse over time, (2) reported with delays, and (3) often reported with errors over the values because of inaccurate sources or non-robust extractors.

Details of the actual techniques can be found in our PVLDB 2016 paper [Abedjan et al. 2015a]. More importantly, our experimental results turned out to be quite positive; we showed that temporal rules improve the quality of the data with an increase of the average precision in the cleaning process from 0.37 to 0.84, and a 40% relative increase in the average F-measure.

Continuing with the data cleaning challenge, Mike wanted to see what would really happen when the rubber hit the road with the many existing data cleaning techniques and systems, such as rule-based detection algorithms [Abedjan et al. 2015a, Chu et al. 2013a, Wang and Tang 2014, Fan et al. 2012, Dallachiesa et al. 2013, Khayyat et al 2015]; pattern enforcement and transformation tools such as OpenRefine, Data Wrangler [Kandel et al. 2011], and its commercial descendant Trifacta, Katara [Chu et al. 2015], and DataXFormer [Abedjan et al. 2015b]; quantitative error detection algorithms [Dasu and Loh 2012, Wu and Madden 2013, Vartak et al 2015, Abedjan et al. 2015, Prokoshyna et al 2015]; and record linkage and deduplication algorithms for detecting duplicate data records, such as the Data Tamer system [Stonebraker et al. 2013b] and its commercial descendant, Tamr.

So, do these techniques and systems really work when run on data from the real world? One key observation was that there was no established benchmarking for these techniques and systems using real data. So, Mike assembled a team of scientists, Ph.D. students, and postdocs from MIT, QCRI, and University of Waterloo, with each site tasked to work on one or more datasets and run one or more data cleaning tools on them. One reason for such a setting was not only for the division of labor but also because some of the datasets could not be moved from one site to another due to restrictions imposed by their owners. We had several meetings and it was imperative that the different experiments be performed in a way that results were comparable. Mike played a great role in coordinating all of these efforts and making sure that we stayed focused to meet the deadline for the last submission for VLDB 2016. One important ingredient was a "marching order" statement from Mike at the end of each meeting on the specific tasks that needed to be accomplished within a well-defined timeframe.

A key conclusion was that there is no single dominant tool. In essence, various tools worked well on different datasets. Obviously, a holistic "composite" strategy must be used in any practical environment. This is not surprising since each tool has been designed to detect errors of a certain type. The details and results can be found in our PVLDB 2016 paper [Abedjan et al. 2016a].

The Data Discovery Challenge

We say an analyst has a data discovery problem when he or she spends more time finding relevant data than solving the actual problem at hand. As it turns out, most analysts in data-rich organizations—that's almost everybody—suffer this problem to varying degrees. The challenge is daunting for several reasons.

1. Analysts may be interested in all kinds of data relevant to their goal. Relevant data may be waiting for them in a single relation in an often-used database, but it may also be in a CSV file copied from a siloed RDBMS and stored in a lake, or it may become apparent only after joining two other tables.
2. Analysts may have a strong intuition about the data they need, but not always a complete knowledge of what it contains. If I want to answer: "What's the gender gap distribution per department in my company?" I have a strong intuition of the schema I'd like to see in front of me, but I may have no clue on where to find such data, in what database, with what schema, etc.
3. The amount of data in organizations is humongous, heterogeneous, always growing, and continuously changing. We have a use case that has on the order of 4,000 RDBMSs, another one that has a lake with 2.5 PB of data plus a few SQL Server instances, and a use case where the organization does not know exactly the amount of data it has because "it's just split into multiple

systems, but it's a whole lot of data we have in here." Of course, this data is always changing.

4. Different analysts will have very different discovery needs. While an analyst in the sales department may be interested in having fresh access to all complaints made by customers, an analyst in the marketing department may want wide access to any data that could be potentially useful to assemble the features needed to build a prediction model. As part of their day-to-day jobs, analysts will have very different data discovery needs, and that will be changing continuously and naturally.

The good news is that we have built a prototype that helps in all of the above four points. In general, the reasoning is to recognize that discovery needs will change over time, and what is relevant today will not be relevant tomorrow. However, the observation is that for X to be relevant to Y, there must be a "relationship" between X and Y. So the idea is to extract all possible relationships from the data. Relationships may include aspects such as similarity of columns, similarity of schemas, functional dependencies (such as PK/FK; primary key/foreign key), and even semantic relationships. All relationships are then materialized in what we have named an "enterprise knowledge graph (EKG)," which is a graph structure that represents the relationships between data sources within an organization. This EKG is central to approach the data discovery challenges, as we explain next, but it comes with its own challenge: It must be built first!

Aurum is a system for building, maintaining, and querying the EKG. It builds it by applying a lot of techniques from systems, sketching and profiling, so as to avoid the huge scalability bottleneck one would find otherwise when trying to read and compute complex relationships among thousands of data sources. It maintains the EKG by understanding when the underlying data changes and updating the EKG accordingly, so as to ensure it always has fresh data. Last, we built a collection of discovery primitives, which can be composed arbitrarily to write complex data discovery queries. This is in turn the interface analysts will have to query the EKG and to find relevant data.

Data discovery is not a solved problem, but the approach sketched above has led us to explore more in depth the open problems—and is helping a handful of collaborators with their own discovery needs. Following Mike's vertical approach (get an end-to-end system working early, then figure the "highest pole in the tent," and keep it simple), has helped us refine the biggest issues yet to solve in the larger Data Civilizer project. Having a way of discovering data with Aurum has helped us understand the challenges of joining across repositories; it has made us think hard about what to do when the data quality of two seemingly equivalent sources is different, and it has helped us to understand the importance of data transformation, which ultimately enables more opportunities for finding more relevant data.

We are just at the beginning of the road on this research, but we have built a reasonable car and are going full speed—hopefully—in the right direction!

Concluding Remarks

It is very different to comment on Mike's contributions on well-established projects that have already had clear impacts, such as Ingres and Vertica, rather than on an ongoing project, such as Data Civilizer. However, from our five-year-old collaboration with Mike, we can distill what we think are the constant characteristics of such collaboration, the ones that have largely shaped the project.

1. What was seemingly a too-big problem for a reasonably small group of collaborators ended up being manageable after finding an attack strategy. The attack was to have one or two clear, precise, end-to-end use cases early on.
2. If you want to make the research relevant and work on problems that matter beyond academia, then base that end-to-end use case on real problems people suffer from in the wild, and only then design a quick prototype.
3. Try out the prototype in the wild and understand what fails so that you can move forward from there.

PART VII.B

Contributions from Building Systems

24

The Commercial Ingres Codeline

Paul Butterworth, Fred Carter

Mike Stonebraker's earliest success in the software business was Relational Technology, Inc. (RTI), later renamed Ingres Corporation, which was formed by Mike, Gene Wong, and Larry Rowe in 1980 to commercialize the Ingres research prototype.

As context, we (Paul and Fred) had no involvement with the Ingres academic research projects. Paul was brought on specifically to manage the day-to-day effort of creating the commercial Ingres product starting in 1980, and Fred joined RTI in 1982 to boost RTI's industrial expertise in network and distributed computing.

The Ingres project produced one the first relational databases, functioning as both a prototype/proof of concept and an operational system. The Ingres system was based on a declarative query language (QUEL) with an optimizer that was independent of the query statement itself. This was a first, the only such optimizer for relational databases around—for quite a while. Based on the working research codeline (which had been provided to and used by various research partners), commercial Ingres delivered a relational database system that became the cornerstone of many customers' businesses.

In his chapter "Where Good Ideas Come from and How to Exploit Them" (Chapter 10), Mike states: "Ingres made an impact mostly because we persevered and got a real system to work." This working system was tremendously important to the commercial Ingres success. (We'll explore this a bit more in the next section.)

Moreover, the research project continued. From this, although a number of features were added to the commercial Ingres product. These include, but are not limited to, distributed Ingres (Ingres Star), user-defined types (see below), and an improved optimizer.

Mike's continuing work, both on the research system and his work as part of RTI, allowed us to move forward quickly. We were very fortunate to have that knowledge and vision as we forged ahead.

The following sections look into some of this work in more detail.

Research to Commercial

The first commercial effort was taking the Ingres research DBMS code and converting it into a commercial product. This activity involved converting the research prototype from Unix on PDP-11s to VAX/VMS on the VAX. This conversion was done by Paul and Derek Frankforth, a truly gifted systems programmer, producing a set of VAX/VMS adaptations on which the Ingres prototype was hosted. The prototype code—bound to Unix and designed to make a large system run on PDP-11s—had to be reworked or eliminated. Since we had no Unix/PDP-11, we had no running version of the Ingres code available, making this a very interesting forensic exercise. In some cases, it was unclear what the intent of various modules was and what correct results should look like. Many times, we would get something running and then simply run queries through the system to try to figure out if what we thought the module should do is really what it did. Backtracking was not unusual! Having worked on the conversion effort, we give a huge amount of credit to the university research team because the core database code was solid. Thus, we didn't have to worry about maintaining the correctness of the DBMS semantics.

It is interesting to note that the conversion effort and subsequent development activities were performed without direct communication with the Ingres research team, and it's an interesting coincidence that none of the members of the research team joined RTI. The commercial Ingres effort significantly predated the contemporary notion of open source (see Chapter 12) and started with the publicly available code developed at the University of California, Berkeley. Without a known process and in an effort to make sure we were not putting anyone at the university in an awkward position, we worked without the benefit of their much deeper knowledge of the code base. It is also interesting to note the original research prototype was the only code developed by the Ingres research team used in commercial Ingres, as the two codelines quickly diverged.

Other major changes in the initial release of commercial Ingres were removing the code that drove communication in the multi-process version of Ingres that ran on the PDP-11 (since that was just extra overhead on the VAX); hardening many of the system components so that errors were dealt with in a more graceful fashion; and adding a few tools to make Ingres a more complete commercial offering.

Although not working closely with the Ingres research teams, commercial Ingres engineering benefited from continuous updates on Mike's, Larry's, and Gene's research efforts, and many of those efforts were quickly implemented in commercial Ingres. Examples include: Distributed Ingres, which was introduced in the commercial product Ingres Star [Wallace 1986]; abstract datatypes,[1] investigated in later Ingres research activities and Postgres, which were introduced into commercial Ingres as Universal Data Types (discussed in detail below); a simplified form of the Postgres rule system; and the development of a comprehensive suite of developer and end-user tools that made commercial Ingres attractive to the business community. Since the code bases diverged immediately, these features and others were implemented without any knowledge of the research implementations. The next few paragraphs discuss some of the synergies and surprises we encountered when incorporating research ideas into the commercial product.

Producing a Product

Once the system was running on VAX/VMS, the next problem was how to increase its commercial attractiveness. This required improving robustness, scalability, and performance, and providing the comprehensive set of tools required to make the system accessible to corporate development teams. When we first brought commercial Ingres up, we measured performance in seconds per query rather than queries per second!

Improving performance involved basic engineering work to make the system more efficient as well as applying ongoing research from various Ingres research efforts to the problem. Much improvement came from improving code efficiency throughout the system, caching frequently used metadata as well as the database pages, and improving the efficiency with which operating system resources were used. Within two years, the efficiency of the system increased by a factor of 300 due to such engineering improvements. A number of these improvements were suggested by Mike and Larry as thoughts they had previously considered as performance improvements in the research prototype but could not justify as research investments.

In addition, we leveraged ongoing research efforts to both good and bad effects. At one point the Ingres research team came up with a new version of the Ingres dynamic optimizer. We implemented the algorithm within commercial Ingres only to find that the Ingres query processing system was now so much faster that the additional overhead from the more sophisticated optimizer actually *reduced* the

[1] This major contribution of Mike's is discussed in Chapters 1, 3, 12, and 15).

performance of many queries rather than improving it. This effort was abandoned before being released. Soon thereafter we took on the effort of converting to query optimization based on other Ingres research (not from Berkeley) into statistics-based optimization. In fact, Mike helped us recruit the author of the research, Bob Kooi, to RTI to implement a commercial version of this work. This effort was very successful as Ingres was acknowledged to have the best complex query optimization throughout the life of the company, as shown by a number of benchmarks.

In contrast to DBMS performance work at the university, performance work on commercial Ingres was rarely driven by formal performance models. Much of the work involved just measuring the performance of the system, identifying the highest cost modules, and then improving the code or eliminating the use of high cost resources.

Lesson. From this work, we can distill a few lessons. The fact that we started with a complete and working code base made our lives much easier. As noted, we did not have to spend time worrying about the database semantics, as those were correct in the research system. Instead, we could focus on building a commercial system. Error handling, internal operations, etc., are critical. Turning the project into a commercial system involves a lot of work to ensure that recovery is always complete and consistent. As part of building any prototype, this is an important consideration. We were very fortunate to have someone with Mike's knowledge and vision as we moved forward. Mike drove the technical agenda for Ingres, and, consequently, Ingres was recognized for a long time as the technical leader and visionary database.

Storage Structures

As Michael Carey has noted (see Chapter 15), one of the Ingres contributions was that a full storage manager could be integrated with the optimizer to improve the performance of the database manager. As we moved Ingres into the commercial world, one of the issues that we encountered involved this storage manager.

Specifically, the index structures (HASH and ISAM [indexed sequential access method]) in Ingres were static. The index structures (be they number of hash buckets or ISAM key structure) were fixed to the data (specifically key) set at the time the relation was indexed. A HASH structure had a fixed number of buckets (based on data at the time of indexing), and an ISAM table had a fixed key tree (again, based on the key set at the time of indexing). QUEL had a command to remedy this, *modify* relation *to ISAM*, but this reorganized the entire table, locking the entire table with the corresponding impact on concurrent access.

This was acceptable in many scientific uses, but for highly concurrent and 24-hr use cases, this type of maintenance activity became increasingly difficult for customers to manage. Moreover, to newer users of the system, the static nature of these structures was not obvious. I cannot count the number of times when a user would complain about performance issues; our first question was "Did you modify to ISAM and then load the data?" (This would have created a 0-level index, with a lot of overflow pages.) The answer was often *yes*, and the solution was to *modify* the table again (that is, recreate the index). While happy to have a solution, customers were often somewhat disappointed that things were that easy. What seemed like a hard problem had a trivial fix. Of course, not all performance issues were this simple.

To address this problem, we added a BTREE storage structure. This structure was a B+ Tree, incorporating various additions from concurrent research (R-trees, etc.). BTREEs, of course, provided a dynamic key organization that allowed users' data to shrink and grow with appropriate search performance. This was, of course, a big improvement for many types and uses of data.

That said, we did find that keeping the static data structures around was of value. For some datasets, the ability to fix the key structure was perfectly acceptable. Indeed, in some cases, these provided better performance—partly due to the nature of the data, and partly due to the lower concurrency costs because the index need not (indeed, could not) be updated.

The "Ingres Years" produced a system that functioned and had the notion of a flexible storage manager. Indices could be created for specific purposes, and the underlying keying structure made this very efficient. HASH-structured tables provided excellent query performance when the complete key was known at query time. ISAM and, later, BTREEs, did so with range queries. This flexibility provided by the variety of storage structures, extended from the research system to the commercial one, served our customers well.

Lesson. Moving from a research project (used primarily by research consumers) to a commercial system that often had very near 24-hr access requirements required us to approach software development differently. The need for the BTREE structure was a manifestation of this, requiring a product feature set change to make the Ingres database viable in some environments. As our customers increased their usage of Ingres throughout their organizations, RTI had to step up to the 24-7 use cases.

User-Defined Types

Later, we began to see that a number of customers had a need to perform queries on non-traditional data types. The reasons are well documented in the literature, but primary among them is the ability for the optimizer and/or query processors to use domain-specific information. This may take the form of data storage capabilities or domain-specific functions.

In any case, we set out to provide this capability. Again, as Michael Carey notes, ADT-Ingres had approached this problem. We looked at what they had done and incorporated similar functionality (see Chapter 15).

By this time, of course, the commercial Ingres code base had a vastly different structure from the prototype code—the code base had diverged, and the optimizer had been replaced. We needed to fully support these user-defined data types in the optimizer to fully enable performant query processing. As one might imagine, this added a certain complexity.

As with ADT-Ingres and Postgres, each user-defined type (UDT) had a name, conversions to/from external forms (typically strings of some form), and comparison functions. To provide for more generic input and error processing, there was a set of functions to aid in parsing, as well as various functions to aid in key construction and optimization.

While we needed to be able to convey information about hash functions and optimizer statistics to the Ingres runtime code, requiring them before anything would work made it a daunting task for our customers. (If our memory serves, somewhere in the neighborhood of 18 C-language functions were required to fully implement a UDT.) To support incremental development, we added the ability to restrict the functionality: UDTs could be defined that precluded keying or statistics generation for the optimizer.

Eventually, these capabilities became known as the Object Management Extension.

A good deal of work was done here to succinctly express the needs of the optimizer and storage manager into relatively simple things that our users (albeit the more sophisticated ones) could do without an in-depth knowledge of the intricacies of the statistics-based optimizer. The requirement here was to continue to support the high-performance, high-concurrency operations in which the commercial Ingres system was used while still providing the capabilities that our customer base required.

The model used in commercial Ingres was similar to that used in ADT-Ingres and Postgres. We definitely tried to build on the concepts, although our code and customer bases were different. It was most definitely a help to build on some of the thinking from that closely related research community.

Lesson. As Ingres moved from research to business-critical use cases, the need for reliable data access increased. When allowing Ingres users (albeit system programmers, not end users) to alter the Ingres server code, this presented a number of challenges, specifically with respect to data integrity. The C language, in which Ingres is implemented, did not provide much protection here. We looked at building the UDT system in a separate process, but that really wasn't feasible for performance reasons. Consequently, we checked and protected all data access as much as possible and placed very close to red flashing warning lights in the documentation. Given the state of the software at the time, that was the best that could be done. In the intervening years, there have been numerous advancements in language and system design, so we might do things differently today. But writing "user extensions" is always a tradeoff. These tradeoffs must be carefully considered by anyone providing these in a commercial environment.

Conclusions

The shared beginning in separate code bases allowed us to make direct use of the ongoing research. There were always differences for many reasons, of course, but we were able to make good use of the work and adapt it to the commercial system.

Mike's knowledge and vision helped drive the commercial effort, resulting in Ingres' being recognized as the visionary leader. Mike's leadership, in the form of the ongoing research efforts and his insight into the product direction, were tremendously important to the Ingres product and company and to us personally.

We were and are very grateful for our ongoing relationship.

Open Source Ingres

Today, Ingres is one of the world's most widely downloaded and used open-source DBMSs. There are multiple versions (different codelines) of open-source Ingres. The Ingres Database (see below) is the original open-source Ingres, based on a software donation by Computer Associates (CA). Ingres Corporation (formerly RTI) was acquired by ASK, which was subsequently acquired by Computer Associates. After a time, CA spun out a separate enterprise for Ingres, and that enterprise was involved in a number of acquisitions and name changes including VectorWise BV, Versant, Pervasive, and ParAccel (see Wikipedia). That enterprise is now Actian. Actian released an open-source version of Ingres, called Ingres Database 10.

The following was copied from the open source web page: Ingres Database, BlackDuck OpenHub (http://openhub.net/p/ingres) on March 14, 2018. The data was subsequently deleted.

> Project Summary
>
> Ingres Database is the open source database management system that can reduce IT costs and time to value while providing the strength and features expected from an enterprise class database. Ingres Database is a leader in supporting business critical applications and helping manage the most demanding enterprise applications of Fortune 500 companies. Focused on reliability, security, scalability, and ease of use, Ingres contains features demanded by the enterprise while providing the flexibility of open source. Core Ingres technology forms the foundation, not only of Ingres Database, but numerous other industry-leading RDBMS systems.
>
> In a Nutshell, Ingres Database . . .
>
> - has had 3,978 commits made by 74 contributors representing 3,761,557 lines of code;
> - is mostly written in C with a very well-commented source code;
> - has a well-established, mature codebase maintained by one developer with decreasing Y-O-Y commits; and
> - took an estimated 1,110 years of effort (COCOMO model) starting with its first commit in March, 2008 ending with its most recent commit about 2 years ago.

25

The Postgres and Illustra Codelines

Wei Hong

I worked on Postgres from 1989–1992, on Illustra from 1992–1997, and then on offshoots of Postgres on and off for several years after that. Postgres was such a big part of my life that I named my cats after nice-sounding names in it: Febe (Frontend-Backend, pronounced Phoebe) and Ami (Access Method Interface, pronounced Amy). I first learned RDBMS at Tsinghua University in China with the Ingres codebase in 1985. At the time, open-source software was not allowed to be released to China. Yet, my advisor and I stumbled across a boxful of line-printer printouts of the entire Ingres codebase. We painstakingly re-entered the source code into a computer and managed to make it work, which eventually turned into my master's thesis. Most of the basic data structures in Postgres evolved from Ingres. I felt at home with Postgres code from the beginning. The impact of open-source Ingres and Postgres actually went well beyond the political barriers around the world for that era.

Postgres: The Academic Prototype

I joined Michael Stonebraker's research group in the summer of 1989, the summer after his now-famous cross-America coast-to-coast bike trip.[1] At the time, the group's entire focus was on eliminating Lisp from the codebase. The group had been "seduced by the promise of AI," as Mike puts it, and opted to implement Postgres in a combination of Lisp and C. The result was a horrendously slow system

[1] This is the story behind the Turing Award lecture "The land sharks are on the squawk box."

suffering from massive memory leaks around the language boundaries[2] and unpredictable performance due to untimely garbage collection. The team was drawn to Lisp partially because of its nice development environment. However, the lack of any symbolic debugging below the Lisp/C interface (as the C object files were dynamically loaded into the running image of a stripped commercial Lisp binary) forced the team to debug with the primitive "advanced" debugger (adb) with only function names from the stack and raw memory/machine code! So, we spent the whole summer converting Lisp to C and achieving performance gains by an order of magnitude. I can still vividly remember the cheers in Evans Hall Room 608-3 on the UC Berkeley campus and how pleased Mike was to see the simplest PostQuel statement "retrieve (1)" work end to end in our new Lisp-free system. It was a big milestone.

One secret to Stonebraker's success in open-source software was that he *always hired a full-time chief programmer for each project.* Like Ingres, Postgres was developed by groups of undergraduate part-time programmers and graduate students who ultimately want to publish papers on their work on the systems. The chief programmers were the key to hold the whole system together and to support the user base around the world through mailing lists. Postgres had groups of very talented programmers, both undergraduate and graduate students. However, they came and went and lacked consistency. Their motivations were mostly around playing with Mike's latest and greatest workstations, hanging around cool people/projects, and/or prototyping ideas for publications.[3] The chief programmers had a huge challenge on their hands. Even though the system was great for demos and academic experiments, it was nowhere near robust, reliable, or easy to use.

In 1991, in came the young chief programmer Jeff Meredith ("Quiet" in Mike's Turing Award lecture). For whatever reasons at the time, most of the undergraduate programmers disappeared and most of Mike's graduate students either graduated or were working on unrelated topics. The system Postgres v3.1 was not in good shape, with lots of glaring bugs. Jeff was tasked to produce v4.0 to make it much

[2] "Not to mention the lack of any symbolic debugging below the Lisp/C interface, as the C object files were dynamically loaded into the running image of a stripped commercial Lisp binary. Debugging with adb, with only the function names from the stack and raw memory/machine code—good times!"—Paul Aoki, Postgres team member.

[3] "I doubt many undergrads were primarily motivated by money since the hourly pay for an undergrad programmer was exactly the same as the people who worked at the dining hall or shelved books at the library. For a long time, the draw was access to better hardware (your own Unix workstation!) and working with cool people on something cool. But except for the hackers on the "six-year plan," they'd all churn in a year or so . . . ."—Paul Aoki

more usable and reliable. I was recruited to help along with fellow Stonebraker students Joe Hellerstein and Mike Olson (“Triple Rock” in Mike’s Turing lecture).

We spent many long days and nights cleaning up and rewriting many sections of flaky code, with only occasional breaks to play a primitive online game of Hearts together. The most memorable area that I fixed is in the buffer manager. At the time, Postgres suffered from major buffer page leaks because many parts of the code were careless in releasing buffers. It was a painstaking process to fix all the leaks. In the end, I was so sure that I got all the leaks plugged that I put in an error message telling people to contact me if a leak were ever detected again. I also put comments all over the code to make sure that people would follow my convention of releasing buffers or else! I don’t dare to search for my name in the Postgres codebase today. I certainly hope that no one ever saw the error message and those comments containing my name! Postgres v4.0 was finally released with much improved query semantics and overall reliability and robustness.

Stonebraker pointed out at the end of his Turing lecture that all the successful systems he created had “a collection of superstar research programmers.” I think that one of the key ingredients to Mike’s successful career is his ability to attract and retain such talent. Despite the messy codebase, mounting number of bugs, and high pressure from Mike’s usual “3 lines of code” estimates for our schedules, we managed to have a lot of fun together as post_hackers.[4] Mike always took us out for beer and pizza whenever there was a milestone to celebrate or an external visitor in town. He would hang out with us for a while and then leave behind a fistful of $20 dollar bills for us to continue “exchanging research ideas.” We fondly called these sessions “study groups.” Some of us still continue this tradition around Berkeley. The Postgres experience built lifelong friendships among us.

Illustra: “Doing It for Dollars”

By Spring 1992, Postgres was already a successful open-source[5] project, with a few hundred downloads and a few thousand users around the world. We were always

[4] post_hackers@cs.berkeley.edu was the mailing list for everyone actively developing Postgres at UC Berkeley.

[5] “Open source factoid: Every port of the Postgres from Sun/SPARC to something else (Solaris/x86, Linux/x86, HP-UX/PRISM, AIM/power) other than Ultrix/mips and OSF1/Alpha, was first done by external users. This wasn’t just doing an ‘autoconf’; to do so, most had to hand-implement mutex (spinlock) primitives in assembly language, because these were the days before stable thread libraries. They also had to figure out how to do dynamic loading of object files on their OS, since that was needed for the extensibility (function manager, DataBlade) features.”—Paul Aoki

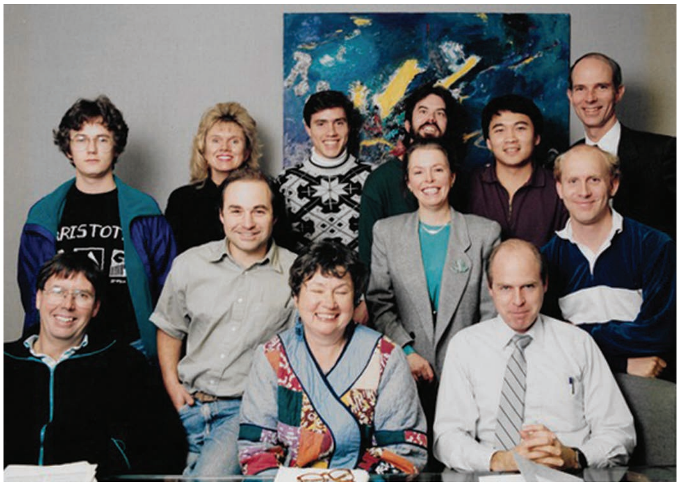


Figure 25.1 The Miro team. Back row, left to right: Cimarron Taylor, Donna Carnes, Jeff Meredith ("Quiet"), Jim Shankland, Wei Hong ("EMP1"), Gary Morgenthaler ("Tall Shark"); middle row: Mike Ubell ("Short One"), Ari Bowes, Jeff Anton; front row: Mike Stonebraker, Paula Hawthorn ("Mom"), and Richard Emberson.

curious about what some of the users from Russia were doing with Postgres. There was also a user of the rule systems with rules involving Tomahawk missiles.[6] We certainly hoped that it was not real!

Developing Postgres aside, I also managed to make good progress on my research on parallel query processing and was getting ready to graduate. Mike was wrapping up Postgres as a research project and contemplating its commercialization. When he first approached me and Jeff Meredith to join him in the commercialization effort, I managed to beat Jeff to Mike's office by half an hour and became employee #1 of Illustra the company. Hence my nickname of "EMP1" in Mike's Turing Award lecture. When faced with design decisions, Mike often lectured us on what "people who do it for dollars" would do. . . . We were so excited to finally get to do it for dollars ourselves!

Illustra Information Technologies, Inc., (then called Miro Systems, Inc., [Stonebraker 1993c]) was up and running in the summer of 1992.

Our main mission at the beginning of Illustra was to (1) make Postgres production-ready, (2) replace PostQuel with SQL (see Chapter 35), and (3) figure out a go-to-market plan.

[6] See http://www.paulaoki.com/.admin/pgapps.html for more details.

1. **Productionize Postgres**. Despite our best efforts as post_hackers, Postgres was nowhere near production-ready. Luckily Mike was able to recruit a few former graduate students—Paula Hawthorn and Mike Ubell (“Mom” and “Short One,” respectively, in Mike’s Turing lecture)—and chief programmer Jeff Anton from the Ingres days, who had all grown into accomplished industry veterans by then. Somehow the two generations of Stonebraker students and chief programmers were able to bond as a team immediately. The veterans played a huge role in the productization of Postgres. They instilled in us that *a production database system must never, ever corrupt or lose customers’ data*. They brought in tools like Purify to help eradicate memory corruptions. They fixed or rewrote critical but often neglected commands like vacuum which could easily cause data corruption or permanent loss. They also taught us how to patch corrupted disk pages with all but dissertation when dealing with elusive heisenbugs[7] that were impossible to catch. They also taught us the importance of testing. Before Illustra, Postgres was only tested by running demos or benchmarks for paper writing. When we started the first regression test suite, someone brought a chicken mask to the office. We made whoever broke the regression test wear the chicken mask for a day. This test suite became known as the “Chicken Test.” Eventually we built a substantial quality assurance team with tests ranging from synthetic coverage tests and SQL92 compliance tests to standard benchmark tests, customer-specific tests, and more. Finally, we were ready to sell Illustra as a production-worthy DBMS!
2. **SQLize Postgres**. Stonebraker famously called SQL the “Intergalactic Data Speak. “It was clear from the beginning that we had to replace PostQuel with SQL. This primarily means to extend SQL92 with Postgres’ extensible type system and the support for composite types. It was not hard to extend SQL. What took us the most time was to catch up on the vanilla SQL92 features which Postgres lacked. For example, integrity constraints and views were implemented by the Postgres rule system, which was not compliant with SQL92. We had to rewrite them completely from scratch. The Postgres rule systems were incredibly complex and buggy anyway. So, we “pushed them off a cliff,” as Mike would say. Other SQL features that took us a long time to complete included nested subqueries, localized strings, and

[7] A heisenbug is a software bug that seems to disappear or alter its behavior when one attempts to study it.

date/time/decimal/numeric types. Finally, we were able to pass all SQL92 entry-level tests, which was a huge milestone.

3. **Go-to-Market**. As Joe Hellerstein points out in Chapter 16, Postgres was jam-packed with great research ideas, most of which were far ahead of their time. In addition to the technical challenges, we had a huge challenge on our hands to figure out a go-to-market plan. We went over each unique feature in Postgres and discussed how to market it "for dollars":
    (a) *ADTs*: a great extension to the relational model, easy for the market to digest, but sounded way too "abstract"! As researchers, we would proudly market how "abstract," "complex," and "extensible" our type system was, but customers needed something concrete and simple to relate to.
    (b) *Rules, Active/Deductive Databases*: It was an easy decision to push both Postgres rule systems "off a cliff" because this was an "AI Winter" at the time, as Joe puts it.
    (c) *Time Travel*. The market couldn't care less that Postgres' crash recovery code was much simpler thanks to Stonebraker's no-overwrite storage system. The additional benefit of the Time Travel feature was also too esoteric for the mass-market customers. We kept the no-overwrite storage system, but no customers ever time-traveled to anywhere but NOW.
    (d) *Parallel DBMS*: This was near and dear to my heart because it was my dissertation. Unfortunately, the market for any form of shared something or nothing parallel DBMS was too small for a start-up to bet on at the time.
    (e) *Fast Path*: This was our secret weapon to combat the performance claims by OODBMS (Object-Oriented DBMS) vendors on benchmarks by bypassing the SQL overhead. Ultimately this was too low level an interface for customers to adopt. It just remained as our secret weapon to win benchmarking battles.

It became clear to us that we must capitalize on Postgres' ADT system in our go-to-market strategy. Stonebraker quickly applied his legendary quad chart analysis (see Chapter 6, Figure 6.2) to map out the market, which put us dominantly at the upper-right quadrant (always!) as the new breed of DBMS: Object-Relational DBMS (ORDBMS), the best of both worlds of OO and Relational DBMS. It was still difficult for people to wrap their heads around such a generic system without concrete applications. We used the razor-and-blade analogy and coined the term "DataBlade," which is a collection of data types, methods, and access methods. We knew that we must build some commercially compelling datablades to jump-start the market. So, in 1995 the company was reorganized into three business units: Financial, targeting Wall Street with TimeSeries DataBlade; Multimedia, targeting media companies with text and image search DataBlade; and Web, targeting the then-emerging World Wide Web. Even though there were three business units, most of the codeline development was concentrated on supporting the TimeSeries DataBlade because the company expected most of the revenues from the Financial business unit, and TimeSeries was an incredibly challenging DataBlade to implement. Doing time series well can be an entire company of its own. Wall Street customers are notoriously savvy and demanding. Even though we made inroads to most of the major firms on Wall Street, each pilot deal was hard fought by our top engineers. I did my time on Wall Street trying to optimize our system so that our performance would come close to their proprietary time series systems. I remember having to stay up all night to rewrite our external sorting module to meet some customer's performance requirement. Despite all our heroics on Wall Street, the company had very little revenue to show for from the efforts.

In the meantime, the Multimedia business unit limped along by partnering with text and image search vendors while the Web business unit—with only some simple data types for handling web pages and few engineers—made huge traction marketing-wise. We became the "Database for Cyberspace" and eventually were acquired by Informix.

All active development on the Illustra codeline essentially stopped after the Informix acquisition. Illustra engineers were teamed up with Informix counterparts to start a new codeline called Informix Universal Server. It was based on the Informix Dynamic Server line with extensive changes throughout the codebase to support Object-Relational features. Aside from the organizational challenges in merging teams with completely different cultures, the main technical challenge for the Illustra engineers was to go from Illustra's multi-process single-thread environment to Informix's homegrown, non-preemptive multithreaded environment. Despite Informix's management and financial woes, the combined team ultimately succeeded. That codeline still lives today powering IBM's Informix Universal Server product.

PostgreSQL and Beyond

While we were busy building and selling DataBlades at Illustra, back on the Berkeley campus, Stonebraker students Jolly Chen and Andrew Yu (Happy and Serious, respectively, in Mike's Turing lecture) decided that they had had enough of PostQuel and did their own SQLization project on the open-source codebase. They released it as Postgres95. This turned out to be the tipping point for open-source Postgres to really take off.[8] Even though Postgres was always open source,[9] its core was developed exclusively by UC Berkeley students and staff. In April 1996, Jolly Chen sent out a call for volunteers to the public.[10] An outsider, unrelated to Berkeley, Marc Fournier, stepped up to maintain the Concurrent Version Systems (CVS) repository and run the mailing lists. Then a "pickup team of volunteers" magically formed around it on its own to take over Postgres from Berkeley students to this randomly distributed team around the world. Postgres became PostgreSQL and the rest is history. . . .

Open Source PostgreSQL

PostgreSQL is one of the world's most widely downloaded and used open-source DBMSs. It is impossible to estimate how many copies are in use since PostgreSQL ships with almost every distribution of Linux. The Linux Counter estimates that over 165,000 machines currently run Linux with over 600,000 users (last accessed March 14, 2018).

The following is from its open-source web page: PostgreSQL Database Server, BlackDuck OpenHub. Last accessed March 7, 2018.

"Project Summary

PostgreSQL is a powerful, open source relational database system. It has more than 15 years of active development and a proven architecture that has earned it a strong reputation for reliability, data integrity, and correctness. It runs on all major operating systems, including Linux, UNIX (AIX, BSD, HP-UX, SGI IRIX, Mac OS X, Solaris, Tru64), and Windows.

In a nutshell, PostgreSQL Database Server . . .

[8] "Postgres has always been open source in the sense that the source code is available but up to this point, the contributors are all UC Berkeley students. The biggest impact that contributed to the longevity of PostgreSQL is transitioning the development to an open source community beyond Berkeley."—Andrew Yu, Postgres team member

[9] "Mike supported its general release as well as releasing it under BSD. Postgres95 would have taken a very different historical path if it was GPL or dual-licensed (a la MySQL)."—Jolly Chen, Postgres team member

[10] "I sent out the email in April 1996. The thread is still archived by PostgreSQL historians. Often open-source projects die when initial contributors lose interest and stop working on it."—Jolly Chen

- has had 44,391 commits made by 64 contributors representing 936,916 lines of code;
- is mostly written in C with a well-commented source code;
- has a well-established, mature codebase maintained by a large development team with stable Y-O-Y commits; and
- took an estimated 262 years of effort (COCOMO model) starting with its first commit in July, 1996 ending with its most recent commit 4 days ago" http://www.postgresql.org. Last accessed April 12, 2018.

Final Thoughts

Stonebraker spent a decade "struggling to make Postgres real" [Stonebraker 2016]. I have been lucky enough to be a large part of this "struggle." As Postgres/Illustra developers, we had a general rule of not letting Mike anywhere near the code repositories. However, the codelines throughout were heavily influenced by his vision, ideas, and relentless push to dominate in the "upper-right quadrant" in the market sector. In his Turing lecture, Mike gave a lot of credit to the collection of superstar programmers "on whose shoulders I've ridden" [Stonebraker 2016]. On the flip side, so many of these programmers have also ridden on Mike's shoulders and grown into new generations of leaders in both industry and academia.

26

The Aurora/Borealis/ StreamBase Codelines: A Tale of Three Systems

Nesime Tatbul

StreamBase Systems, Inc. (now TIBCO StreamBase) was the first startup company that Michael Stonebraker co-founded after moving from UC Berkeley to MIT in the early 2000s. Like Mike's other start-ups, StreamBase originated as an academic research project, called Aurora.[1] One of the first data stream processing systems built, Aurora was the result of close collaboration among the database research groups of three Boston-area universities: MIT, Brown, and Brandeis. It also marked the beginning of a highly productive, long-lasting period of partnership among these groups that has continued to this day, with several successful joint research projects and startup companies (see Figure 26.1; Chapters 27 and 28).

As Aurora was transferred to the commercial domain, the academic research continued full steam ahead with the Borealis[2] distributed stream processing system. Based on a merger of the Aurora codebase (providing core stream processing functionality) with the Medusa[3] codebase from MIT's networking research group (providing distribution functionality), Borealis drove five more years of strong collaboration under Mike's leadership. Years later, around the same time as StreamBase was being acquired by TIBCO Software, the usual suspects would team up

[1] http://cs.brown.edu/research/aurora/. Last accessed May 14, 2018.

[2] http://cs.brown.edu/research/borealis/. Last accessed May 14, 2018.

[3] http://nms.csail.mit.edu/projects/medusa/. Last accessed May 14, 2018.

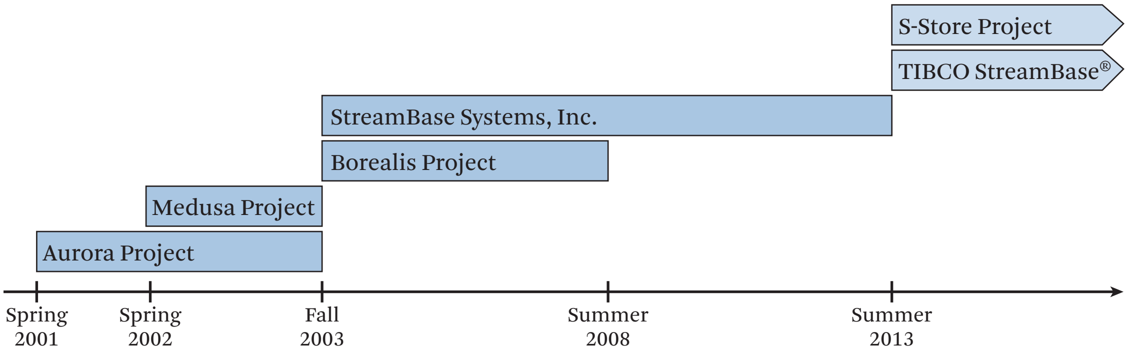


Figure 26.1 Michael Stonebraker's Streaming Systems Timeline.

again to build a novel system for transactional stream storage and processing, S-Store.[4]

This chapter provides a collection of stories from members of the Aurora/Borealis and StreamBase teams, who were first-hand witnesses to the research and development behind Stonebraker's streaming systems and his invaluable contributions.

Aurora/Borealis: The Dawn of Stream Processing Systems

Developing under Debian Linux, XEmacs, CVS, Java, and C++: free
Gas for a round-trip, two-hour commute from New Hampshire to Rhode Island: $15
Funding 4 professors, 14 graduate students, and 4 undergrads for six months: $250,000
Getting a data stream processing system to process 100 gigabytes in six months: priceless!

—Brown University [2002]

This opening quote from a Brown departmental newsletter article about the Aurora project captures in a nutshell how it all got started. Shortly after Mike had arrived on the East Coast, started living in New Hampshire, and working at MIT, he and Stan Zdonik of Brown got in touch to discover their joint research interests. This was way before everything from phones to refrigerators got so smart and connected, but

[4] http://sstore.cs.brown.edu/. Last accessed May 14, 2018.

futurists had already been talking about things like pervasive computing, sensor networks, and push-based data. Mike and Stan realized that traditional database systems would not be able to scale to the needs of an emerging class of applications that would require low-latency monitoring capabilities over fast and unpredictable streams of data. In no time, they set up a project team that included two young database professors, Ugur Çetintemel and Mitch Cherniack, as well as several grad students from Brown and Brandeis. I was a second-year Ph.D. student in the Brown Database Group at the time, just finishing off my coursework, in search of a good thesis topic to work on, and not quite realizing yet that I was one lucky grad student who happened to be in the right place at the right time.

The first brainstorming meeting at Brown kicked off a series of others for designing and prototyping a new data management system from scratch and tackling novel research issues along the way. One of the fundamental things to figure out was the data and query model. What was a data stream and how would one express queries over it? Everyone had a slightly different idea. After many hot debates, we finally converged on SQuAl (the Aurora [S]tream [Qu]ery [Al]gebra). SQuAl essentially consisted of streaming analogs of relational operators and several stream-specific constructs (e.g., sliding windows) and operators (e.g., resample), and supported extensibility via user-defined operators. In Aurora terminology, operators were represented with "boxes," which were connected with "arrows" representing queues of tuples (like database rows), together making up "query networks." We were ready to start implementing our first continuous queries. Well, almost . . .

As systems students, we were all confident coders and highly motivated for the project, but building a whole new database system from the ground up looked way more complex than anything we had ever done before. Where does one even start? Mike knew. The first thing to do was to implement the catalog that would hold the metadata needed by systems components, using BerkeleyDB. We then implemented the data structures for basic primitives such as streams, tuple queues, and windows as well as the boxes for SQuAl operators. The scheduler to execute the boxes was implemented next. These core systems components were implemented in C++. Initially, system settings and workloads to run (i.e., query networks) were simply specified by means of an XML-based, textual configuration file. Later we added a Java-based GUI for making it easier for users to construct query networks and manage their execution using a drag-and-drop boxes-and-arrows interface. The graphical specification would also be converted into the textual one under the covers. Our procedural approach to visually defining dataflows was what set Aurora apart from other systems that had SQL-based, declarative front-ends. Other visual tools were added over time to facilitate system monitoring and demonstration of

Figure 26.2 Aurora research meeting in the Brown Computer Science Library Room, Fall 2002. Left to right, top row: Adam Singer, Alex Rasin, Matt Hatoun, Anurag Maskey, Eddie Galvez, Jeong-Hyon Hwang, and Ying Xing; bottom row: Christina Erwin, Christian Convey, Michael Stonebraker, Robin Yan, Stan Zdonik, Don Carney, and Nesime Tatbul.

various advanced features, (e.g., Quality of Service [QoS] specification and tracking, monitoring tuple queues and system load).

During the first year, we were both building the initial system prototype and trying to identify the key research issues. The highest priority was to build a functional system and publish our first design paper. At first, the grad students did not know which research problems each would be working on. Everyone focused on engineering the first working version of the system. This was well worth the effort, since we learned a great deal along the way about what it took to build a real data management system, how to figure out where the key systems problems lay, as well as creating a platform for experimental research. After about six months into the project, I remember a brainstorming session at Brown where major research topics were listed on the board and grad students were asked about their interests. After that day, I focused on load shedding as my research topic, which eventually became the topic of my Ph.D. dissertation.

By the end of year one, our first research paper got into VLDB'02 [Carney et al. 2002]. I remember an enthusiastic email from one of the professors saying something like, "Congratulations folks, we are on the map!" This paper was later selected to appear in a special issue of the VLDB Journal with best papers from that year's VLDB [Abadi et al. 2003b]. According to a subsequent publication shortly thereafter, Aurora was already an "operational system with 75K lines of C++ code and 30K lines of Java code" by this time [Zdonik et al. 2003]. By the end of the second year, grad students started publishing the first papers on their individual research topics (e.g., load shedding, scheduling), we presented a demo of the system at SIGMOD’03,[5] and did our first public code release. At some point, probably shortly before the release, a full-time software engineer was hired to the project to help us manage the growing codebase, organize and clean up the code, create proper documentation, etc.

As Aurora moved to the commercial space, the university team switched attention to its distributed version, Borealis. We explored new research topics such as high availability, fault tolerance, and dynamic load balancing, and published a series of SIGMOD/VLDB/ICDE papers [Ahmad et al. 2005, Balazinska et al. 2004b, 2005, Hwang et al. 2005, Xing et al. 2005]. Within a couple of years, we built a comprehensive system prototype for Borealis, which won the best demo award at SIGMOD’05[6] (this was the same year as Mike’s IEEE John von Neumann Medal celebration event). Further details on the research work behind Aurora/Borealis can be found in Chapters 17 and 26.

We were not the only team working on streaming research in those days. There were several other leading groups, including the STREAM Team at Stanford (led by Jennifer Widom), the Telegraph Team at UC Berkeley (led by Mike Franklin and Joe Hellerstein), and the Punctuated Streams Team at Oregon Graduate Institute (led by Dave Maier). We interacted with these teams very closely, competing with one another on friendly terms but also getting together at joint events to exchange updates and ideas on the general direction of the field. I remember one such event that was broadly attended by most of the teams: the Stream Winter Meeting (SWiM) held at Stanford in 2003, right after the CIDR’03 Conference in Asilomar, CA.[7] Such interactions helped form a larger community around streaming as well as raised the impact of our collective research.

Aurora/Borealis was a major team effort that involved a large number of student developers and researchers with different skills, goals, and levels of involvement in the project across multiple universities. Under the vision and leadership of Mike and the other professors, these projects represent unique examples of how large

[5] Photos from the Aurora SIGMOD’03 demo are available at: http://cs.brown.edu/research/aurora/Sigmod2003.html. Last accessed May 14, 2018.

[6] Photos from the Borealis SIGMOD’05 demo are available at: http://cs.brown.edu/research/db/photos/
BorealisDemo/index.html. Last accessed May 14, 2018.

[7] See the “Stream Dream Team” page maintained at http://infolab.stanford.edu/sdt/. (Last accessed May 14, 2018) and detailed meeting notes from the SWiM 2003 Meeting at http://telegraph.cs.berkeley.edu/swim/. (Last accessed May 14, 2018).

systems research teams can work together and productively create a whole that is much bigger than the sum of its parts. Mike's energy and dedication was a great source of inspiration for all of us on the team. He continuously challenged us toward building novel but also practical solutions, never letting us lose sight of the real world.

By the final public release of Borealis in summer 2008, all Aurora/Borealis Ph.D. students had graduated, with seven of them undertaking faculty positions, Mitch and Ugur had been promoted to tenured professors, and StreamBase had already closed its Series C funding and started generating revenue. Mike and Stan? They had long been working on their next big adventure (see Chapters 18 and 27).

From 100K+ Lines of University Code to a Commercial Product

Richard Tibbetts, one of Mike's first grad students at MIT, witnessed the complete StreamBase period from its inception to the TIBCO acquisition. After finishing his M.Eng. thesis with Mike on developing the Linear Road Stream Data Management Benchmark [Arasu et al. 2004], Richard got involved in the commercialization of Aurora, first as one of the four engineer co-founders, and later on as the CTO of StreamBase. He reminisces here about the early days of the company:

> Initially, we called the company 'DBstream,' but then the trademark attorneys complained. Then inspired by the pond near Mike's vacation house in NH where we did an offsite, we wanted to call it 'Grassy Pond.' Oops, there was a computer company in Arkansas with that name! Could we pick some other body of water? We were going to change it later anyway . . . And so became the first official name of the company: 'Grassy Brook.' Mike's daughter [Lesley] created the logo. Later in 2004, Bill Hobbib [VP of marketing] would rename it to 'StreamBase,' a stronger name which he and others would build into a very recognizable brand.
>
> We put together very rough pitch decks and presented them to a handful of Boston-area VCs. We had hardly anything in there about commercialization, really much more a solution in search of a problem, but we did have a UI (user interface) we could demonstrate. The visual tool really won people over. It took me years to really be a fan of the visual programming environment. But it was the case that graph-structured applications were a more correct way to look at what we were doing: a natural, "whiteboard" way of representing application logic. Eventually, most of the competitors in the space copied what we were doing, adding visual languages to their SQL-like approaches. Even in tools that didn't have a visual environment, APIs became graph-topology-oriented. StreamBase invested deeply in visual programming, adding modularity, diff/merge, debugging, and other capabilities in ways that were always visual native. Users, sometimes skep-

tical at first, fell in love with the visual environment. Years later, after we had sold the company, integrated the tech, and I had left, that visual environment lived on as a major capability and selling point for StreamBase.

For the first four months, from September 2003 to January 2004, we were in the same codebase as academia, making things better and building around. But at some point, needs diverged and it was appropriate to fork the code. We BSD-licensed a dump of the code and published it at MIT, then downloaded a copy at the company, and went from there with our own code repository.

One of the first things we did was to build a test harness. Regressions and right answers matter a lot to industry—even more than performance and architecture. Most of the academic code wouldn't survive in the long term. It would get replaced incrementally, 'Ship of Theseus' style. Some of the architecture and data structures would remain. We had to make sure we liked them, because they would get harder to change over time. Ph.D. theses typically represented areas of code where complexity exceeded what was strictly necessary, so we started deleting them. Also, anything with three to five distinct implementations of a component was highly suspect, due to the redundant level of flexibility and higher cost to maintain all of those implementations. We either picked the simplest one or the one that was a key differentiator. Deleting code while maintaining functionality was the best way to make the codebase more maintainable and the company more agile.

Encounters with StreamBase Customers

John Partridge was a co-founder of StreamBase and its initial VP of marketing. Having been on the business development and product marketing side of the company, John shares the following anecdote about Mike's interaction with StreamBase customers:

> We knew that investment banks and hedge funds cared a lot about processing real-time market data with minimal latency and so we targeted all the obvious big-name banks: Goldman Sachs, Morgan Stanley, etc. Those banks were always looking for competitive advantage and their appetite for new technology was widely known. What was also widely known was that the egos of the managing directors who managed the trading desks were enormous, but these were the people we were trying to meet. Getting four of them in the same room for a one-hour presentation took weeks, sometimes months, to schedule because they all wanted to be viewed as the scarcest, and hence most valuable, person to attend.
>
> Anyhow, at last we had four of the top people from a leading investment bank scheduled to hear Mike give the StreamBase pitch. Mike and I show up for the meeting and three of them are there; no one knows where the fourth guy is.

> They're all wearing the impeccable tailored suits, power ties, and ludicrously expensive analog watches. I'm wearing one of the two business suits I own and feeling completely outgunned. Mike is wearing his red fleece jacket, polo shirt underneath, and open-toed sandals. There's no small talk with these guys; they are all business. I do the customary five minutes of introductions, then Mike takes over with his pitch. The technical questions start and, coming from business people, the questions are pretty insightful. They're also pretty pointed, probably because Mike likes to frame alternative approaches, including this bank's, as things 'only an idiot would do.' So it's a real grilling they're giving Mike. Then, 20 minutes into it, the fourth managing director walks in, doesn't say hello, doesn't apologize for being late, doesn't even introduce himself; he just takes a seat and looks at the screen, ignoring the guy in the sandals. Mike, without missing a beat, turns on the late arrival and declaims, 'You're late.' There's something about the way a tenured professor can say those words that strikes terror in the heart of anyone who has ever graduated college. Not only did the late arrival flinch and reflexively break away from Mike's withering stare, but you could see the other three managing directors grinning through a wave of shared *schadenfreude*. At that moment, what had been a tough business meeting became a college lecture where the professor explained how the world worked and everyone else just listened.

"Over My Dead Body" Issues in StreamBase

Richard Tibbetts reminds us of the infamous phrase from Mike that whoever worked with him has probably heard him say at least once:

> Mike was fond of taking very strongly held positions, based on experience, and challenging people to overcome them. Occasionally these would elicit the familiar phrase 'Over My Dead Body (OMDB).' However, it turned out that Mike could be swayed by continued pressure and customer applications.

John Partridge remembers one such OMDB issue from the early days of StreamBase:

> The StreamBase stream processing engine was originally implemented as an interpreter. This was the standard way to build database query executors at the time, and it made it easier to add new operators and swap in alternative implementations of operators. I think we were about eight or nine months in when the lead engineers on the engine, Jon Salz and Richard Tibbetts, began to realize that the performance hit for running the interpreter was disastrous. The more they looked at the problem, the more they believed that using a just-in-time Java compiler would run much faster and still provide a mechanism for

> swapping out chunks of application logic on the fly. Mike would have none of this and it became an 'Over My Dead Body' issue for months. Finally, our CEO resolved the debate by giving Jon Salz and Richard Tibbetts two months to get an experimental version up and running. Mike viewed this as a complete waste of precious developer time, but agreed just to keep the peace. Jon and Richard finished it a week early and the performance advantages were overwhelming. Mike was hugely impressed with the results and, of course, with Jon and Richard. We switched over to the Java JIT compiler and never looked back.

Richard Tibbetts adds the following details from the same time period, and how taking up the challenge raised by Mike led the engineering team to a new and improved version of the StreamBase engine to be shipped to the customers:

> Professors spend a lot of time challenging graduate students to do impossible things, expecting them to be successful only a fraction of the time, and this yields a lot of scientific advancement. At StreamBase, the Architecture Committee was where Mike, some of the other professors, the engineering founders, and some of the other senior engineers came together to discuss what was being built and how it was being built. These meetings regularly yielded challenges to Engineering to build something impossible or to prove something.
>
> The first instance I recall came very early in the life of the company, when Jon Salz wanted to completely replace the user interface for editing graphical queries that had been developed in the university. Mike and Hari [Balakrishnan] thought it would be lower risk to begin modifying it and incrementally improve it over time. Jon asserted he could build a better UI that would be easier to improve going forward, and they challenged him to do it in two weeks. He delivered, and that became the basis for our commercial GUI, and also our move into Eclipse-based development environments, which enabled a vast array of capabilities in later versions. It was also the first Java code in the company.
>
> Another instance, possibly even more impactful, came as it became clear that the original architecture wasn't a fit for what the market needed. StreamBase 1.0, like Aurora, managed queries as collections of processing nodes and queues, with a scheduler optimizing multi-threaded execution by dispatching work based on queue sizes and available resources. The processing nodes were implemented in C++, making this system a sort of interpreter for queries.
>
> It turned out that execution performance was dominated by the cost of queuing and scheduling. Jon Salz proposed an alternative approach, codenamed "SB2" for being the second version of StreamBase. In Jon's proposal, queries would be compiled to Java byte-code and executed by the system without requiring any queuing or scheduling. This would also dramatically change the

> execution semantics of the graphical queries, making them much easier to reason about, since data would flow in the same path every time.
>
> Of course, it would also make the system completely different from the academic approach to high performance streaming. And Mike was appalled at the idea of building database tech in Java, which he believed was slow. Jon was confident that he could make it faster, much faster. So, Mike challenged him to prove it. In a couple of months Jon and I had a prototype, which was 3–10 times higher throughput than the existing implementation, and even more dramatically lower latency. Mike happily conceded the point, and StreamBase 3.0 shipped with the new faster engine, with a nearly seamless customer upgrade.

Richard Tibbetts recalls two additional OMDB situations, where customer requirements and hard work of engineers overcame oppositions from Mike: adding looping and nested data support to StreamBase.

> Within a processing graph, it made computational sense to have cycles: loops where messages might feed back on one another. However, this was at odds with the SQL model of processing, and made our system less declarative, as well as admitting the possibility of queries that would 'spin loop,' consuming lots of CPU and never terminating. There were discussions of ways to enable some of this while maintaining declarative semantics, but in the end, customer use cases (for example, handling a partially filled order by sending the unfilled part back to match again) demanded loops, and they became a core part of the system.
>
> For nested data, the relational database model said it was a bad thing. That XML had been a horrible idea as a data representation, just a retread of CODASYL (the Conference/Committee on Data Systems Languages) from the 1970s. This may be true for data at rest. But data in motion is more like self-contained messages. Many of the other systems and protocols StreamBase had to integrate with had nested structures in their messages. Over time, StreamBase handled more and more of these message structures, and at no point was Mike's health adversely affected . . .

An April Fool's Day Joke, or the Next Big Idea?

Both Eddie Galvez of TIBCO StreamBase (a former grad student from the Aurora Brandeis team and one of the four engineer co-founders of StreamBase) and Richard Tibbetts fondly remember how the StreamBase engineers tried to fool Mike several times with their April Fool's Day jokes about some technical topic around StreamBase. Richard retells one story that I have found especially interesting:

> StreamBase Engineering had a tradition of announcing major new product capabilities to the company, usually with an enthusiastic mass email, on April 1st.

> One year, Jon announced that he had rewritten the whole system in Haskell and it was another ten times faster. On more than one occasion, Mike responded to these in good faith. One occurrence in particular comes to mind:
>
> Very early versions of StreamBase had embedded a MySQL query engine to manage stored data. This software never shipped, and had many technical flaws. It also offended Mike, because MySQL was just not a very good database. However, it was designed to be embeddable (today SQLite would be the logical choice), and so we tried it out. Eventually, we switched to our own, more limited stored data management, in memory and on disk using Sleepycat. But MySQL did come up again.
>
> On Sunday, April 1, 2007, Hayden Schultz and I had worked through the weekend integrating with a MySQL-based system at Linden Lab, a prospective customer. I sent an email to the whole company, announcing that in addition to succeeding at the customer integration, we had had an epiphany about MySQL, enumerating its many wonderful capabilities. The email concluded, ‘The sum of all these benefits is that MySQL is the obvious choice for all StreamBase persistence. We should begin exploring a MySQL-based architecture for StreamBase 5.0, and also look at how we can bring this technology to bear in existing engagements.’
>
> Mike quickly responded saying that he had a superior proposal, based on a planned research system called ‘Horizontica,’ and we should table any decision until the next Architecture Committee meeting. I laughed out loud at the pun on Vertica. Mike had clearly gotten in on the April 1st activities. But then I opened the attachment, which was a 13-page preprint of a VLDB paper. In fact, this was an actually interesting alternative approach for StreamBase persistence, and some pretty cool research. That system would later become H-Store/VoltDB.

As an interesting note to add to Richard’s story, exactly six years later on Monday, April 1, 2013, I joined the Intel Science and Technology Center for Big Data based at MIT as a senior research scientist to work with Mike and the old gang again, on a new research project that we named “S-Store.” The idea behind S-Store was to extend the H-Store in-memory OLTP database engine with stream processing capabilities, creating a single, scalable system for processing stored and streaming data with transactional guarantees [Meehan et al. 2015b]. S-Store was publicly released in 2017.

Next time you make an April Fool’s Day joke to Mike, think twice!

Concluding Remarks

Stream processing has matured into an industrial-strength technology over the past two decades. Current trends and predictions in arenas such as the Internet

Figure 26.3 The Aurora/Borealis/StreamBase reunion on April 12, 2014 at MIT Stata Center for Mike's 70th Birthday Celebration (Festschrift). From left to right, front row: Barry Morris, Nesime Tatbul, Magda Balazinska, Stan Zdonik, Mitch Cherniack, Ugur Çetintemel; back row: John Partridge, Richard Tibbetts, and Mike Stonebraker. (Photo courtesy of Jacek Ambroziak and Sam Madden.)

of Things, real-time data ingestion, and data-driven decision-making indicate that the importance of this field will only continue to grow in the future. Stonebraker's streaming systems have been immensely influential in defining the field and setting its direction early on, all the way from university research to the software market. These systems have also been great examples of productive collaboration and teamwork at its best, not to mention the fact that they shaped many people's careers and lives. The codeline stories retold in this chapter provide only a glimpse of this exciting era of Mike's pioneering contributions.

Acknowledgments

Thanks to Eddie Galvez, Bobbi Heath, and Stan Zdonik for their helpful feedback.

27 The Vertica Codeline

Shilpa Lawande

The Vertica Analytic Database unequivocally established column-stores as the superior architecture for large-scale analytical workloads. Vertica's journey started as a research project called C-Store, a collaboration by professors at MIT, Brown, Brandeis, and UMass Boston. When Michael Stonebraker and his business partner Andy Palmer decided to commercialize it in 2005, C-Store existed in the form of a research paper that had been sent for publication to VLDB (but not yet accepted) and a C++ program that ran exactly seven simple queries from TPC-H out of the box—it has no SQL front-end or query optimizer, and in order to run additional queries, you had to code the query plan in C++ using low level operators! Six years later (2011), Vertica was acquired by Hewlett-Packard Enterprise (HPE). The Vertica Analytics Engine—its code and the engineers behind it—became the foundation of HPE's "big data" analytics solution.

What follows are some highlights from the amazing Vertica journey, as retold by members of its early engineering team. And some lessons we learned along the way.

Building a Database System from Scratch

My involvement with Vertica started in March 2005 when I came across a job ad on Monster.com that said Stonebraker Systems: "Building some interesting technology for data warehousing." As someone who was getting bored at Oracle and had studied Mike's Red Book[1] during my DB classes at University of Wisconsin-Madison, I was intrigued, for sure. My homework after the first interview was—you guessed it—read the C-Store paper [Stonebraker et al. 2005a] and be ready to discuss it with Mike (a practice we continued to follow, except eventually the paper was replaced with the C-Store Seven Years Later paper [Lamb et al. 2012], and the

[1] *Readings in Database Systems* http://www.redbook.io/.

interview conducted by one or more senior developers). I do not recall much of that first interview but came away inspired by Mike's pitch: "It doesn't matter whether we succeed or fail. You would have built an interesting system. How many people in the world get to build a database system from scratch?" And that's why I joined Vertica (see Chapter 18).

The early days were filled with the usual chaos that is the stuff of startups: hard stuff like getting the team to jell, easier stuff like writing code, more hard stuff like sorting through disagreements on whether to use push- or pull-based data-flow operators (and whether the building was too hot for the guys or too cold for me), writing some more code, and so on.

In the summer of 2005, we hired Chuck Bear, who at the time was living out of his last company's basement and working his way down the Appalachian Trail. After Chuck's interview, Mike barged into the engineering meeting saying, "We must do whatever it takes to hire this guy!" And since the team was fully staffed, Chuck got asked to do "performance testing." It did not take long for everyone to realize that Chuck's talents were underutilized as a "tester" (as Mike called quality assurance engineers). There was one occasion where Chuck couldn't convince one of the engineers that we could be way faster than C-Store, so, over the next few nights, while his tests were running, he wrote a bunch of code that ran 2× faster than what was checked in!

The first commercial version of Vertica was already several times faster than C-Store, and we were only just getting going, a fantastic feat of engineering! From here on, C-Store and Vertica evolved along separate paths. Vertica went on to build a full-fledged petabyte-scale distributed database system, but we did keep in close touch with the research team, sharing ideas, especially on query execution with Daniel Abadi and Sam Madden, on query optimization with Mitch Cherniack at Brandeis, and on automatic database design with Stan Zdonik and Alex Rasin at Brown. Vertica had to evolve many of the ideas in the C-Store paper from real-world experience, but the ideas in Daniel Abadi's Ph.D. thesis on compressed column stores still remained at the heart of Vertica's engine, and we should all be glad he chose computer science over medicine.

Lesson. In effective software engineering organizations, the best ideas win. Shared ownership of the code base is essential. And, if you can't resolve a disagreement with words, do it with code.

Code Meets Customers

The codeline journey of Vertica was a good example of what is called a "Lean Startup" these days—again Mike was ahead of his time (see Chapter 7). The first version “Alpha” was supposed to only do the seven C-Store queries, but with an SQL front-end, not C++ and run on a single node. To do this, the decision was to use a “brutalized Postgres” (see Chapter 16), throwing away everything except its parser and associated data structures (why reinvent the wheel?) and converting it from a multi-process model to a single-process multi-threaded model. Also left out by choice: a lot of things that you can’t imagine a database not being able to do!

Omer Trajman was one of the early engineers. He later went on to run the Field Engineering team (charged with helping deploy Vertica in customer sites). He recalls:

> One of these choices was pushing off the implementation of delete, a crazy limitation for a new high-performance database. In the first commercial versions of Vertica, if a user made a mistake loading data, the data couldn’t be changed, updated, or even deleted. The only command available to discard data was to drop the database and start over. As a workaround to having to reload data from flat files, the team later added INSERT/SELECT to order to create a copy of loaded data with some transformation applied, including removing rows. After adding the ability to rename and drop tables, the basic building blocks to automate deletes were in place. As it turns out, this was the right decision for Vertica’s target market.
>
> The Vertica team found that there were two types of ideal early customers: those whose data almost never changed, and those whose data changed all the time. For people with relatively static data, Vertica provided the fastest and most efficient response times for analytics. For people whose data changed all the time, Vertica was able to go from raw data to fast queries more quickly than any other solution in the market. To get significant value from Vertica, neither customer type needed to delete data beyond dropping tables. Customers with data that rarely changed were able to prepare it and make sure it was properly loaded. Customers with rapidly changing data did not have the time to make corrections. Mike and the team had a genuine insight that at the time seemed ludicrous: a commercial database that can’t delete data.

Lesson. Work with customers, early and often. Listen carefully. Don’t be constrained by conventional wisdom.

Don’t Reinvent the Wheel (Make It Better)

Discussions about what to build and what not weren’t without a share of haggling between the professors who wrote the academic C-Store paper [Stonebraker et al. 2005a] and engineers who were building the real world Vertica. Here’s Chuck Bear recounting those days.

Back in 2006, the professors used to drop by Vertica every week to make sure we (the engineers) were using good designs and otherwise building the system correctly. When we told Mike and Dave DeWitt[2] that we were mulling approaches to multiple users and transactions, maybe some sort of optimistic concurrency control or multi-versioning, they yelled at us and said, in so many words, "Just do locking! You don't understand locking! We'll get you a copy of our textbook chapter on locking!" Also, they told us to look into the Shore storage manager [Carey et al. 1994], thinking maybe we could reuse its locking implementation.

We read the photocopy of the chapter on locking that they provided us, and the following week we were prepared. First, we thanked the professors for their suggested reading material. But then we hit them with the hard questions . . . "How does locking work in a system like Vertica where writers don't write to the place where readers read? If you have a highly compressed table, won't a page-level lock on an RLE[3] column essentially lock the whole table?"

In the end, they accepted our compromise idea, that we'd "just do locking" for transaction support, but at the table level, and additionally readers could take snapshots so they didn't need any locks at all. The professors agreed that it was a reasonable design for the early versions, and in fact it remains this way over ten years later.

That's the way lots of things worked. If you could get a design that was both professor-approved and that the engineers figured they could build, you had a winner.

Lesson. This decision is a great case study for "Keep it simple, stupid," (aka KISS principle) and "Build for the common case," two crucial systems design principles that are perhaps taught in graduate school but can only be cemented through the school of hard knocks.

Architectural Decisions: Where Research Meets Real Life

The decision about locking was an example of something we learned over and over during Vertica's early years: that "professors aren't always right" and "the customer always wins."

The 2012 paper "The Vertica Analytic Database: C-Store 7 years later " [Lamb et al. 2012] provides a comprehensive retrospective on the academic proposals from the original C-Store paper that survived the test of real-world deployments—and others that turned out to be spectacularly wrong.

[2] Dave DeWitt (see Chapter 6), on Vertica's technical advisory board, often visited the Vertica team.

[3] Run Length Encoding

For instance, the idea of permutations[4] was a complete disaster. It slowed the system down to the point of being useless and was abandoned very early on. Late materialization of columns worked to an extent, for predicates and simple joins, but did not do so well once more complex joins were introduced. The original assumption that most data warehouse schemas [Kimball and Ross 2013] were "Star" or "Snowflake" served the system well in getting some early customers but soon had to be revisited. The optimizer was later adapted for "almost star" or "inverted snowflake" schemas and then was ultimately completely rewritten to be a general distributed query optimizer. Eventually, Vertica's optimizer and execution engine did some very clever tricks, including leveraging information on data segmentation during query optimization (vs. building a single node plan first and then parallelizing it, as most commercial optimizers tend to do); delaying optimizer decisions like type of join algorithm until runtime; and so on.

Another architectural decision that took several iterations and field experience to get right was the design of the Tuple Mover. Here's Dmitry Bochkov, the early lead engineer for this component, reminiscing about his interactions with Mike during this time.

> The evolution of the Tuple Mover design in the first versions of Vertica demonstrated to me Mike's ability to support switching from academic approach to "small matters of engineering" and back. What started as a simple implementation of an LSM (log-structured merge-tree) quickly degenerated into a complicated, low-performance component plagued by inefficient multiple rewrites of the same data and a locking system that competed with the Execution Engine and Storage Access Layer locking mechanisms.
>
> It took a few rounds of design sessions that looked more like thesis defense, and I will forever remember the first approving nod I received from Mike. What followed was that the moveout and mergeout algorithms ended up using "our own dog food." Our own Execution Engine was used for running the Tuple Mover operations to better handle transactions, resources planning, failover, and reconciliation among other tasks. And while it added significant pressure on other components, it allowed the Tuple Mover to become an integral part of Dr. Stonebraker's vision of a high-performance distributed database.

Anyone who has worked with Mike knows he is a man of few words, and if you listen carefully, you can learn a massive amount from his terseness. If you

[4] The idea that multiple projections in different sort orders could be combined at runtime to recreate the full table. Eventually, it was replaced by the notion of a super projection that contains all the columns.

worked at Vertica in the early days, you often heard Mike-isms, such as "buying a downstream farm" (referring to "engineering debt")[5] and the famous "over Mike's dead body" (OMDB). These phrases referred to all the "bells and whistles" that database systems are filled with that Vertica would never build, perfectly capturing the tension between "research" and "real-life" choices that Vertica faced repeatedly over its life.

Min Xiao,[6] founding engineer turned sales engineer, describes an OMDB encounter with Mike.

> One day in 2008, I came back to the office after visiting a global bank customer. I saw that Mike, wearing a red shirt, sat in a small corner conference room working on his laptop. I stepped in and told him that the bank needed the feature of disaster recovery (DR) from Vertica. In the past, Mike had always wanted me to let him know the product requests from the customers. For this customer, their primary Vertica instance was in Manhattan and they wanted a DR instance in New Jersey. They had used Oracle for the same project prior to Vertica and therefore also hoped to have a statement-by-statement-via-change-data-capture type of DR. Mike listened to me for a minute. Apparently, he had heard the request from someone else and didn't look surprised at all. He looked at me and calmly said "They don't need that type of DR solution. All they need is an active replication thru parallel loading." As always, the answer was concise as well as precise. While I took a moment to digest his answer, he noticed my hesitation and added "over my dead body." I went back to the customer and communicated with them about the proposal of having a replicated copy. The bank wasn't overly excited but didn't raise the DR request anymore. Meanwhile, one of our largest (non-bank) customers, who had never used Oracle, implemented exactly what Mike had proposed and was very happy with it. They loaded into two 115-node clusters in parallel and used them to recover from each other.

Lesson. Complexity is often the Achilles' heel of large-scale distributed systems, and as Daniel Abadi describes in vivid detail in Chapter 18, Mike hated complexity. With the liberally used phrase, OMDB, Mike forced us to think hard about every feature we added, to ensure it was truly required, a practice that served us well as our customer base grew. One of the reasons for Vertica's success was that we thought very hard about what NOT to add, even though there was a ton of pressure from

[5] A farm downstream along a river will always be flooded and may appear to be cheaper. This is an analogy for engineering debt, decisions made to save short-term coding work that required a ton of time and effort (i.e., cost) in the long run.

[6] Min Xiao followed Mike and Andy Palmer to join the founding team of Tamr, Inc.

customers. Sometimes we had to relent on some earlier decisions as the system evolved to serve different classes of customers, but we still always thought long and hard about taking on complexity.

Customers: The Most Important Members of the Dev Team

Just as we thought hard about what features to add, we also listened very carefully to what customers were *really* asking for. Sometimes customers would ask for a feature, but we would dig into what problem they faced instead and often find that several seemingly different requests could often be fulfilled with one "feature." Tight collaboration between engineering and customers became a key aspect of our culture from early on. Engineers thrived from hearing about the problems customers were having. Engineering, Customer Support, and Field Engineers all worked closely together to determine solutions to customer problems and the feedback often led to improvements, some incremental, but sometimes monumental.

The earliest example of such a collaboration was when one of the largest algorithm trading firms became a customer in 2008. Min Xiao recalls a day trip by the founders of this trading firm to our office in Billerica, Massachusetts, one Thursday afternoon.

> Their CTO was a big fan of Mike. After several hours of intense discussions with us, we politely asked if they needed transportation to the airport. (This was before the days of Uber.) Their CEO casually brushed aside our request. Only later we found out that they had no real schedule constraints because they had flown in their own corporate jet. Not only that, but once he found out that Mike played the banjo, the next day he brought his bass guitar to the Vertica office. Mike, Stan Zdonik (a professor in Brown University), and John "JR" Robinson (a founding engineer of Vertica) played bluegrass together for several hours. This wasn't an isolated "Mike fan": customers loved and respected Mike for his technical knowledge and straight talk. We often joked that he was our best salesperson ever. :-)
>
> Over time, this customer became a very close development partner to Vertica. They voluntarily helped us build Time-series Window functions, a feature-set that was originally on the "OMDB" list. Due to Vertica's compressed and sorted columnar data storage, many of the windowing functions, which often take a long time to execute in other databases, could run blazingly fast in Vertica.

I recall the thrill that engineers felt to see the fruits of their work in practice.

It was a day of great celebration for the engineering team when this customer reached a milestone running sub-second queries on 10 trillion rows of historical trading data! These time-series functions later become one of the major performance differentiators for Vertica, and enabled very sophisticated log analytics to be expressed using rather simple SQL commands.

A big technical inflection point for Vertica came around 2009, when we started to land customers in the Web and social gaming areas. These companies really pushed Vertica's scale to being able to handle petabytes of data in production. It took many iterations to really get "trickle loads" to work, but in the end this customer had an architecture where every click from all their games went into the database, and yet they were able to update analytical models in "near real-time."

Another inflection point came when a very high profile social media customer decided to run Vertica on 300 nodes of very cheap and unreliable hardware. Imagine our shock when we got the first support case on a cluster of this size! This customer forced the team to really think about high availability and the idea that nodes could be down any time. As result, the entire system—from the catalog to recovery to cluster expansion—had to be reviewed for this use case. By this time, more and more customers wanted to run on the cloud, and all this work proved invaluable to support that use case.

Lesson. Keep engineers close to customers. Maybe make some music together. Listen carefully to their problems. Collaborate with them on solutions. Don't be afraid to iterate. There is no greater motivator for an engineer than to find out his or her code didn't work in the real world, nor greater reward than seeing their code make a difference to a customer's business!

Conclusion

Vertica's story is one of a lot of bold bets, some of which worked right from academic concept, and others that took a lot of hard engineering to get right. It is also a story of fruitful collaboration between professors and engineers. Most of all, it is a story of how a small startup, by working closely with customers, can change the prevailing standard of an industry, as Vertica did to the practices of data warehousing and big data analytics.

Acknowledgments

Thank you to Chuck Bear, Dmitry Bochkov, Omer Trajman, and Min Xiao of the early Vertica Engineering team for sharing their stories for this chapter.

28

The VoltDB Codeline

John Hugg

I was hired by Mike Stonebraker to commercialize the H-Store[1] research [Stonebraker et al. 2007b] in early 2008. For the first year, I collaborated with academic researchers building the prototype, with close oversight from Mike Stonebraker.[2] Andy Pavlo and I presented our early results at VLDB 2008 [Kallman et al. 2008] in August of that year. I then helped lead the efforts to commercialize VoltDB, ultimately spending the next ten years developing VoltDB with a team I was privileged to work with. In my time at VoltDB, Inc., Mike Stonebraker served as our CTO and then advisor, offering wisdom and direction for the team.

VoltDB was conceived after the success of Vertica[3]; if, Vertica, a system dedicated to analytical data, could beat a general-purpose system by an order of magnitude at analytical workloads, could a system dedicated to operational data do the same for operational workloads? This was the next step in Mike Stonebraker's crusade against the one-size-fits-all database.

VoltDB was to be a shared-nothing, distributed OLTP database. Rethinking assumptions about traditional systems, VoltDB threw out shared-memory concurrency, buffer pools and traditional disk persistence, and client-side transaction control. It assumed that high-volume OLTP workloads were mostly horizontally partitionable, and that analytics would migrate to special-purpose systems, keeping queries short.

The proposed system would dramatically reduce scaling issues, support native replication and high availability, and reduce costs for operational workloads without sacrificing transactions and strong consistency.

[1] For more on H-Store see Chapter 19: H-Store/VoltDB.

[2] See https://dl.acm.org/citation.cfm?id=1454211 for the list of collaborators.

[3] For more on Vertica see Chapters 18 and 27.

VoltDB 1.0 was originally released in April 2010, after nearly two years of internal development. Work on the H-Store academic project continued in parallel. Over the years, many ideas and experimental results were shared between the researchers and the VoltDB engineering team, but code diverged as the two systems had different purposes. VoltDB also hired a number of graduate students who worked on the H-Store project.

Compaction[4]

In the Fall of 2010, the very first customer, who was equal parts brave and foolish, was using VoltDB 1.x in production and was running into challenges with memory usage.

This customer was using the resident set size (RSS) for the VoltDB process as reported by the OS as the key metric. While memory usage monitoring is more complex than disk usage monitoring, this is a good metric to use in most cases.

The problem was that the RSS was increasing with use, even though the data was not growing. Yes, records were being updated, deleted, and added, but the total number of records and the size of the logical data they represented was not growing. However, eventually, VoltDB would use all of the memory on the machine. This early customer was forced to restart VoltDB on a periodic basis—not great for a system designed for uptime. Needless to say, this was unacceptable for an in-memory database focused on operational workloads.

The problem was quickly identified as allocator fragmentation. Under it all, VoltDB was using GNU LibC malloc, which allocated big slabs of virtual address space and doled out smaller chunks on request. Allocator fragmentation happens when a slab is logically only half used, but the "holes" that can be used to service new allocations are too small to be useful.

There are two main ways to deal with this problem. The most common approach is to use a custom allocator. The two most common alternatives are JEMalloc and TCMalloc. Both are substantially more sophisticated at avoiding fragmentation waste than the default GLibC malloc.

The VoltDB team tried these options first but ran into challenges because VoltDB mixed C++ and Java in the same process. Using these allocators with the in-process JVM was challenging at the time.

[4] Compaction, which is critical to running VoltDB for more than a few hours, didn't come up in the initial design or research because academics don't always run things the way one might in production. It ended up being critical to success.

The second approach, which is both more challenging and more effective, is do all the allocation yourself. You don't actually have to manage 100% of allocations. Short-lived allocations and permanent allocations tend not to contribute to allocator fragmentation. You primarily have to worry about data with unknown and variable life cycles, which is really critical for any in-memory database.

The team focused on three main types of memory usage that fit this profile.

- Tuple storage—a logical array of fixed size tuples per table.
- Blob storage—a set of variable-sized binary objects linked from tuples.
- Index storage—trees and hash tables that provide fast access to tuples by key.

Two teams set about implementing two different approaches to see which might work best.

The first team took on indexes and blob storage. The plan was to remake these data structures in such a way that they never had any "holes" at all. For indexes, all allocations for a specific index with a specific key width would be done sequentially into a linked list of memory-mapped slabs. Whenever a tree node or hash entry was deleted, the record at the very end of the set of allocations would be moved into the hole, and the pointers in the data structure would be reconfigured for the new address. Blob storage was managed similarly, but with pools for various size blobs.

There was a concern that the extra pointer fixups would impact performance, but measurements showed this was not significant. Now indexes and blobs *could not fragment.* This came at an engineering cost of several engineer-months, but without much performance impact to the product.

Tuple storage took a different approach. Tuples would be allocated into a linked list of memory-mapped slabs, much like index data, but holes from deletion would be tracked, rather than filled. Whenever the number of holes exceeded a threshold (e.g., 5%), a compaction process would be initiated that would rearrange tuples and merge blocks. This would bind fragmentation to a fixed amount, which met the requirements of VoltDB and the customer.

In the end, we didn't pick a winner; we used both schemes in different places. Both prototypes were sufficient and with an early product, there were many other things to improve. The anti-fragmentation work was a huge success and is considered a competitive advantage of VoltDB compared to other in-memory stores that often use memory less efficiently.[5] Without it, it would be hard to use VoltDB in any production workloads.

[5] The competition catch-up is a long story. Most systems can't do what VoltDB does because they use shared-memory multi-threading and even lock-free or wait-free data structures. These are

These kinds of problems can really illustrate the gulf between research and production.

It turns out compaction is critical to running VoltDB for more than a few hours, but this didn't come up because of the research results. We previously assumed that if a steady state workload worked for an hour, it would work forever, but this is absolutely not the case.

Lesson. Memory usage should closely track the actual data stored, and systems should be tested for much longer periods of time.

Latency

Version 1.0 of the VoltDB database, like the H-Store prototype it was based on, used a transaction ordering and consensus scheme that was based on the ideas described in the original H-Store paper [Stonebraker et al. 2007b], but with additional safety. Oversimplifying a bit, nodes would collect all candidate work in a 5 ms epoch and then exchange between all nodes the work inside the cluster for that 5 ms. This work would then be ordered based on a scheme similar to Twitter Snowflake.[6]

This scheme guaranteed a total, global pre-order for all submitted transactions. That is, before a transaction was run, its serializable order with respect to all other transactions was known.

Compared to contemporary transaction ordering schemes, VoltDB offered more fault tolerance than two-phase-commit and was dramatically simpler than using a schema like Paxos for ordering. It also supported significantly higher throughput than either.

Having a global pre-ordering of all transactions required less coordination between cluster nodes when the work itself was being done [Stonebraker et al. 2007b]. In theory, participants have broad leeway to re-order work, so it can be executed more efficiently, provided it produces results effectively equivalent to the specified order. This was all part of the original H-Store research [Stonebraker et al. 2007b].

So, what's the catch? This scheme used wall clocks to order transactions. That meant transactions must wait up to 5 ms for the epoch to close, plus network round trip time, plus any clock skew. In a single data center, Network Time Protocol (NTP)

much harder to compact. Other systems *can* use TCMalloc or JEMalloc because they don't embed the JVM.

[6] "Announcing Snowflake," the Twitter blog, June 1, 2010. https://blog.twitter.com/engineering/en_us/a/2010/announcing-snowflake.html. Last accessed March 29, 2018.

is capable of synchronizing clocks to about 1 ms, but that configuration isn't trivial to get right. Network skew is also typically low but can be affected by common things like background network copies or garbage collections.

To put it more clearly, on a single-node VoltDB instance, client operations would take at least 5 ms even if it did no actual work. That means a synchronous benchmark client could do 200 trivial transactions per second, substantially slower than MySQL for most workloads.

In a cluster, it was worse. Getting NTP set up well in order to evaluate VoltDB was a stumbling block, especially in the new world of the cloud. This meant the delay might be 10–20 ms. The original VoltDB paper assumes achieving clock synchronization is trivial, but we found that to be just false *enough* to cause problems. We didn't just need synced-clocks, we needed them to stay synced for days, months, or even years without issue.

None of this affected throughput. The VoltDB client was fully asynchronous by design and could processes responses in the order they arrived. A proper parallel workload could achieve millions of transactions per second on the right cluster, but asking prospective users to build fully asynchronous apps proved too much of a challenge. Users were not used to developing that way and changing user habits is difficult.

VoltDB needed to be faster than MySQL without application wizardry.

Many months of disagreement and thought from the engineering team culminated in a small meeting where a decision had to be made.

A rough plan was hashed out to replace VoltDB consensus with a post-order system that would slash latency to near zero while keeping throughput. The new system would limit some performance improvements to cross-partition transactions (which are typically rare for VoltDB use cases) and it would require several engineers working for almost a year, time that could be spent on more visible features.

Engineering came out of that meeting resolved to fix the latency issues. As part of the plan, the VoltDB 1.0 consensus scheme would be kept, but only to bootstrap a new system of elected partition leaders that serialized all per-partition work and a single, global cross-partition serializer that determined the order of cross-partition work.

This scheme was launched with version 3.0, and average cluster latency was reduced to nearly nothing now that we did not have to hold transactions for clock skew and the all-to-all exchange. Typical response latencies were less than a millisecond with a good network.

This directly led to VoltDB use in low-latency industries like ad-tech and personalization.

Lesson. Response time is as important as throughput.

Disk Persistence

When VoltDB launched, the high-availability story was 100% redundancy through clustering. There were periodic disk snapshots, so you would see data loss only if you lost multiple nodes, and then you might only lose minutes of recent data. The argument was that servers were more reliable, and per-machine UPSs (uninterruptive power supplies) were increasingly common, so multiple failures weren't a likely occurrence.

The argument didn't land.

VoltDB technical marketing and sales spent too much time countering the idea that VoltDB wouldn't keep your data safe. Competitors reinforced this narrative. In early 2011, it got to the point where lack of disk persistence was severely limiting customer growth.

VoltDB needed per-transaction disk persistence without compromising the performance it was known for. Part of the original H-Store/VoltDB thesis was that logging was one of the things holding traditional RDBMSs back when they moved to memory [Harizopoulos et al. 2008], so this posed quite a challenge.

To address this problem, Engineering added an inter-snapshot log to VoltDB but broke with the ARIES (Algorithms for Recovery and isolation Exploiting Semantics) style logs used by traditional RDBMSs. VoltDB already heavily relied on determinism and logical descriptions of operations to replicate between nodes. Engineering chose to leverage that work to write a logical log to disk that described procedure calls and SQL statements, rather than mutated data.

This approach had a huge technical advantage for VoltDB. As soon as transactions were ordered for a given partition (but before they were executed), they could be written to disk. This meant disk writes *and* the actual computation could be done *simultaneously*. As soon as both were completed, the transaction could be confirmed to the caller. Other systems performed operations and *then* wrote binary change-logs to disk. The logical approach and VoltDB implementation meant disk persistence didn't have substantial impact on throughput, and only minimal impact on latency.

Per-transaction disk-persistence was added in VoltDB 2.5 in Fall 2011 and almost immediately silenced persistence-based criticism of VoltDB. It's clear that without this feature, VoltDB would have seen much more limited use.

As an addendum, we have a lot more data today about how common complete cluster failure is with VoltDB. Cluster failures for well-run VoltDB instances are rare, but not always 100% unavoidable, and not all VoltDB clusters are well run. Disk persistence is a feature that not only cut off a line of criticism, but also gets exercised by users from time to time.

Lesson. People don't trust in-memory systems as system of record.

Latency Redux

In 2013, within a year of reducing average latency in VoltDB to nil, VoltDB was courted by a major telecommunications OEM (original equipment manufacturer) looking to replace Oracle across their stack. Oracle's pricing made it hard for them to compete with upstart Asian vendors who had built their stacks without Oracle, and Oracle's deployment model was poorly suited to virtualization and data-center orchestration.

Replacing Oracle would be a substantial boost to competitiveness.

During the OEM's VoltDB evaluation, latency quickly became an issue. While average latency met requirements, long tail latency did not. For a typical call authorization application, the service level agreement might dictate that any decision not made in 50 ms can't be billed to the customer, forcing the authorization provider to pay the call cost.

VoltDB created a new automated test to measure long tail latency. Rather than measure average latency or measure at the common 99th percentile or even the 99.999th percentile, Engineering set out to specifically count the number of transactions that took longer than 50 ms in a given window. The goal was to reduce that number to zero for a long-term run in our lab so the customer could support P99.999 latency under 50 ms in their deployments.

Once you start measuring the right things, the problem is mostly solved, but there was still code to write. We moved more of the statistics collection and health monitoring code out of blocking paths. We changed how objects were allocated and used to nearly eliminate the need for stop-the-world garbage collection events. We also tuned buffer sizes and Java virtual machine parameters to get everything running nice and "boring."

If there's one thing VoltDB Engineering learned over the course of ten years of development, it's that customers want their operational databases to be as boring and unsurprising as possible. This was the final piece of the puzzle that closed the first major telecommunications customer, with more coming right on their heels.

Today, a significant portion of the world's mobile calls and texts are authorized through a VoltDB-based system.

Lesson. P50 is a bad measure—P99 is better—P99.999 is best.

Conclusion

Of course, the incidents described here are just a tiny sliver of the challenges and adventures we encountered building VoltDB into the mature and trusted system it is today. Building a system from a research paper, to a prototype, to a 1.0, and to a robust platform deployed around the world is an unparalleled learning experience.

29

The SciDB Codeline: Crossing the Chasm

Kriti Sen Sharma, Alex Poliakov, Jason Kinchen

Mike's a database guy, so his academic innovations get spun out and branded as database products. But SciDB—in contrast to PostgreSQL, Vertica, and VoltDB—is a rather different kind of beast: one that blurs the line between a database and HPC. As a computational database for scientific applications, it's actually two products in one tightly architected package: a distributed database and a massively parallel processing (MPP), elastically scalable analytics engine. Along with its support for a new kind of efficiently stored and accessible n-dimensional data model, the hybrid design made development triply challenging. As a lean team under intense pressure to produce revenue early on, we[1] had to make many tough tradeoffs between long-term vision and short-term deliverables. Some worked out; some had to be ripped out. Here are a few tales from the trek up Mike's SciDB mountain, in keeping with Paul Brown's story line, "Scaling Mountains: SciDB and Scientific Data Management" (see Chapter 20).

Playing Well with Others

SciDB ships with an extensive set of native analytics capabilities including ScaLAPACK—scalable linear algebra neatly integrated into SciDB by James McQueston. But given the scope of SciDB's mission—enabling scientists and data scientists to run their advanced array data workflows at scale—it would be impossible to provide all the functionality to cover the myriad of potential use cases. One of the core design decisions allowed users to add user-defined types, functions, and

[1] The authors and their colleagues mentioned in this chapter are with Paradigm4, the company that develops and sells SciDB.

aggregates (UDTs, UDFs, and UDAs, respectively) in a fashion very similar to other databases. SciDB went one step further by supporting a much more powerful user-defined operator (UDO) abstraction. This decision proved to be successful as many user-defined extensions (UDXs) were developed by users of SciDB. Noteworthy examples include NASA's connected-component labeling of MERRA satellite image data spatially and temporally aligned with ground-based sensor data to identify storms [Oloso et al. 2016], and a GPU-accelerated convolution running in SciDB for solar flare detection [Marcin and Csillaghy 2016].

However, the limitations of relying solely on a UDX approach became apparent as many customers already had their analytics coded in R/Python/MATLAB (MaRPy languages). They turned to SciDB to scale up their work to run the same algorithm on larger datasets or to execute an expensive computation in a shorter amount of time using elastic computing resources. But they did not want to incur the development costs to re-implement or revalidate their algorithms as UDXs. Moreover, writing UDXs required sufficient understanding about details of the SciDB architecture (e.g., tile-mode, chunking strategy, SciDB array-API, etc.). Quite often, researchers we spoke to had a "favorite package" in mind, asking us to help run *exactly that* package on terabytes of data. We realized that while UDXs were a powerful way to customize SciDB, the market needed to develop, iterate, and deploy faster.

In the Spring of 2016, Bryan Lewis and Alex Poliakov decided to take this up. They were under intense time pressure as the demonstration date for a prospect was looming. Thus began a three-week long collaborative programming frenzy in which Bryan and Alex first sketched out the design, divvied up the work, and produced a working implementation.

The overall architecture of SciDB streaming is similar to Hadoop streaming or the Apache *RDD.pipe*. The implementation they chose was to use pipes: standard-in and standard-out for data transfer. The SciDB chunk—already a well-formed segment of data that was small enough to fit in memory—would be used as the unit of data transfer. Existing SciDB operators were used to move arrays between instances for "reduce" or "summarize" type workflows. The first implementation shipped with a custom R package offering an easy API to R specifically.

Customers liked this capability immediately. One could now connect SciDB with the vast universe of open-source libraries in MaRPy languages and C++. SciDB streaming spares the user from doing the plumbing by serving up the data to each instance and orchestrating the parallelism. Specialized Python support and integration with Apache Arrow was subsequently added by SciDB customer solutions architects Jonathan Rivers and Rares Vernica. SciDB streaming is now an important part of the SciDB user's toolkit.

As Mike likes to say, one database does not fit all. Similarly, one data distribution format does not fit all algorithms for high-performance computing. So, James McQueston is adding support to preserve data in alternate distributions—like "block-cyclic" for matrix operations and "replicated" for certain kinds of joins—to notch up performance by making broader sets of algorithms more highly efficient in a distributed system. This will boost both embarrassingly parallel (streaming) and non-embarrassingly parallel execution, such as large-scale linear algebra.

SciDB streaming is currently deployed successfully for a wearables project at a top 5 pharma company where 7 TB of data from 63 streams of data from 13 different devices are time-aligned and analyzed, and for the Stanford Global Biobank Engine to evaluate gene-level effect models on the UK Biobank genotype-phenotype association data.

You Can't Have Everything (at Once)

When implementing an elastic, scalable, multi-dimensional, multi-attribute array MPP distributed database, the Paradigm4 team quickly realized that this was very much uncharted territory, requiring us to build many components from the ground up. In such a scenario, it was important to pick out which features to focus on first, and which capabilities to defer until later in the product roadmap. While a multi-dimensional array database with ACID properties had been the development focus from very early on, it was decided that full elasticity support would be designed in from the start, but rolled out in phases.

The very earliest implementations of SciDB did not fully adhere to shared-nothing architecture[2] principles. User queries had to be sent to a special coordinator node rather than to any instance in a cluster. This was not ideal, and also introduced a single point of failure in the system. This initial compromise simplified the coding effort and was good enough for most of the early users who queried SciDB only via one instance.

Around the end of 2015, the implementation of true elasticity was completed. Now, all instances in a cluster could accept incoming queries—there was no single coordinator instance (even though the term "coordinator" continues to be commonly used among SciDB users). More importantly, instances could go online and

[2] A shared-nothing architecture prescribes that all participating nodes of a multi-node distributed computing architecture are self-sufficient. In SciDB parlance, the unit of participation in a cluster is the "instance," while the term "node" is reserved for one physical (or virtual) computer within a multi-computer cluster.

offline at any moment as SciDB could detect node failures and still keep functioning properly. This required significant architectural changes which were led by Igor Tarashansky and Paul Brown.

The new implementation removed some of the exposure to the single point of failure. That architectural change also supported another important capability: ERGs, or elastic resource groups. As customer installations grew in data volume or computational needs, elasticity and ERGs allow users to add more instances to the cluster either permanently (e.g., to handle more storage), or on an as-required basis (e.g., only while running an expensive computation routine).

In business as in life, one cannot have everything at once. But with time, persistence, and a clear focus, we are able to deliver many of the product goals that we had planned early on.

In Hard Numbers We Trust

Typical queries on SciDB involve both large and small data. For example, a user might want to retrieve the gene-expression values at certain loci for patients aged 50–65 from within a large clinical trial dataset. To slice the appropriate "sub-array" from the larger gene-expression array, one must infer necessary indices from at least three other metadata arrays (corresponding to sets of studies, patients, and genes). SciDB is remarkably fast at slicing data from multi-TB arrays—after all, this is what SciDB was designed and optimized for. However, it turned out that the initial design and implementation had not focused sufficiently on optimizing query latency on "smaller" arrays.

Initially, we thought this slower performance might be because small arrays that could fit on one machine were being treated the same as large arrays that had to be distributed across multiple machines. We hypothesized that an unnecessary redistribution step was slowing down the query. If that were the case, the engineering team thought a sizable rewrite of the codebase would be required to achieve a significant speedup. With immediate customer deliverables pressing, we deferred tackling this specific performance shortfall for a while.

In a separate project, James McQueston had spent time building powerful and very granular profiling tools for SciDB. His engineering mantra was that developers must look at hard numbers before devising any theory about why something goes as slow (or as fast) as it does. The profiling tool had previously been useful in certain other optimization scenarios. However, its true value shone when revealing the actual cause of the slowness on small-array queries. Dave Gosselin and James painstakingly replicated anonymized customer data and queries in the profiling environment. Armed with those tools, they discovered that the major bottleneck was not the redistribution step that everyone suspected. Instead, the delay was caused by inefficient querying of the system catalog, something that was implemented eons ago in the product's history but had never been revisited. This fix was relatively easy.

Our initial hypothesis had been incorrect; careful performance analysis pointed us to the correct root cause. Happily, we were able to achieve significant speedup for the small-array queries without a major rewrite of the codebase.

Language Matters

Scientists and data scientists prefer statistical analysis languages like R and Python, not SQL. We knew that writing intuitive interfaces to SciDB in these languages would be important if we wanted these folks to start using our system. Thus, we developed SciDB-R and SciDB-Py as R and Python connectors to SciDB, respectively. One could use these connectors to dispatch queries to the multi-TB data on SciDB while also utilizing the programming flexibility and vast open-source libraries of their favorite programming language. Today, almost all customers who use SciDB in production use one of these interfaces to work with SciDB.

The journey of these two connector libraries reflects our evolving understanding of the user experience. Initial implementations of these connectors (SciDB-R by Bryan Lewis and SciDB-Py by Jake VanderPlas and Chris Beaumont) shared the common philosophy that we should overload the target language to compose/compile SciDB queries. For example, R's *subset* functionality was overridden to call SciDB operators *between* and *filter* underneath. In the SciDB-Py package, Pythonic syntax like *transform* and *reshape* were implemented that actually invoked differently named SciDB operators underneath. We thought that this kind of R- / Python- level abstraction would be preferred by the users. However, when we trained customers, we found that the user experience was not ideal, especially for advanced users. Some corner cases could not be covered—the abstracted layer did not always produce the most optimal SciDB query.

A complete rewrite of SciDB-R (Bryan Lewis) and SciDB-Py (Rares Vernica) was carried out recently where the interface language mapped more closely to SciDB's internal array functional language (AFL). However, we concluded that one cannot resolve language issues in the abstraction or translation layer. Instead we are now investing the time and effort to improve the user experience by redesigning the SciDB AFL language itself. We have made significant progress on this front: Donghui Zhang unified three operators (*filter, between,* and *cross_between*) into a more user-friendly *filter* syntax; Mike Leibensperger introduced auto-chunking to spare the user from having to think about physical details. More language improvements are on the roadmap to improve the user experience and to align SciDB's language more closely with R and Python.

Security is an Ongoing Process

Our first-pass on security for SciDB involved separation of arrays via named domains or "namespaces" (e.g., users A and B might have access to arrays in namespace *NS_1*, but only A has access to arrays in *NS_2*). Access was authorized locally by SciDB.

This implementation of security was a first pass and we knew we would have to improve it. The opportunity came when one of our pharma customers required an industry-standard authorization protocol (e.g., LDAP [Lightweight Direct Access Protocol]), and more fine-grained access control such as different access control privileges for each user across hundreds of their clinical studies.

Two short-term security projects were spun off simultaneously. The implementation of study-level access control was implemented by Rares Vernica and Kriti Sen Sharma using SciDB's UDX capability. SciDB-R and SciDB-Py also required major plumbing changes. In tandem, integration with pluggable authentication modules (PAM, an industry standard for secure authorization control) enabled SciDB's utilization of LDAP. Mike Leibensperger rewrote the entire security implementation to achieve these new capabilities, with which we won the contract.

Despite "security is an ongoing process" being a cliché, we understand the truth of the statement. We also realize that there is more ground to cover for us in this realm and are embarked on continuous improvements.

Preparing for the (Genomic) Data Deluge

The deluge of genomic data has been long anticipated [Schatz and Langmead 2013]. With the advent of sub-$1,000 DNA sequencing, the drive toward personalized medicine, and the push for discovering and validating new therapies to deliver more effective outcomes, more and more people are getting their DNA sequenced via nationalized data collection efforts like the UK Biobank program, the 1 Million Veterans program, and the AllofUs program. While marketing and selling SciDB, we heard claims from competitors that they were already capable of handling data at these scales. Our engineering team decided to tackle these claims head on and to produce hard data about SciDB's ability to deliver that scalability today.

Using genomics data available in the public domain, our VP of Engineering Jason Kinchen, along with consulting engineer James McQueston, generated and loaded 100K, 250K, 500K, and 1M exomes (the coding regions of the genome) into SciDB. We showed that common genomics queries scaled linearly with the size of the data. For example, given a particular set of genomics coordinates, the search for variants overlapping the regions, including long insertions and deletions, took 1 sec on a 100K exome dataset and 5 sec on a 500K dataset. The growth was linear, thanks to SciDB's multidimensional indexing capabilities. Further, having arrived at the "500K point," adding more SciDB instances would speed the query up. Apart from greatly enhancing the marketing message with hard data proving SciDB's scalability, this benchmark work also pointed to opportunities for significant performance improvements, many of which have been tackled. More are planned.

By setting up test scripts with benchmarks from real-world use cases, our engineering team regression-tests and profiles each software release. This relentless focus on real-world performance and usability is fundamental to our development culture and keeps us focused improving our customers' experience and their use cases.

Crossing the Chasm: From Early Adopters to Early Majority

Paradigm4 develops and makes available both an open-source Community Edition of the core SciDB software (available at http://forum.paradigm4.com/ under an Affero GPL license) and an Enterprise Edition (Paradigm4 license) that adds support for faster math, high availability, replication, elasticity, system admin tools, and user access controls.

While Mike set a grand vision for a scientific computational database that would break new ground providing storage, integration, and computing for n-dimensional, diverse datasets, he left many of the engineering details as an exercise for the Paradigm4 development team. It has been an ambitious and challenging climb, especially given that we are creating a new product category, requiring both technical innovations and missionary selling. Mike remains actively engaged with SciDB and Paradigm4, providing incomparable and invaluable guidance along the trek. But the critical lessons for the product and the company—new features, performance enhancements, improving the user experience—have come in response to our experiences selling and solving real customer applications.

30

The Tamr Codeline

Nikolaus Bates-Haus

Volume, velocity, and variety: Among the three Vs of Big Data [Laney 2001], there are well-established patterns for handling volume and velocity, but not so for variety. The state of the art in dealing with data variety is manual data preparation, which is currently estimated to account for 80% of the effort of most data science projects [Press 2016]. Bringing this cost down would be a huge benefit to modern, data-driven organizations, so it is an active area of research and development. The Data Tamer project [Stonebraker et al. 2013b] introduced a pattern for handling data variety, but nobody had ever put such a system into production. We knew there would be challenges, but not what they would be, so we put together a team with experience, talent, drive, and the audacity to assume we would overcome whatever got in our way.

I joined Tamr in April 2013 as employee #3. My first tour of duty was to recruit the core engineering team and build the first release of our commercial product for data unification (see Chapter 21). My personal motivation to join Tamr was that tackling the challenge of data variety would be the culmination of a decades-long career working with data at large scale, at companies such as Thinking Machines (on the Darwin Data Mining platform), Torrent Systems (a parallel ETL [extract, transform, load] system) and Endeca (on the MDEX parallel analytic database). I've seen messy data foil far too many projects and, after talking to Tamr co-founders Mike Stonebraker and Andy Palmer about their aspirations, I concluded that Tamr would provide the opportunity to apply those decades of experience and thinking to fundamentally change the way we approach big data.

Because we were breaking new ground with Tamr, we faced a number of unusual challenges. The rest of this chapter looks at some of the most vexing challenges, how we overcame them, lessons learned, and some surprising opportunities that emerged as a result.

Neither Fish nor Fowl

From the start, the Tamr codeline has been a kind of chimera, combining intense technical demands with intense usability demands. The first board meeting I attended was about a week after I joined, and Andy announced that we were well on our way to building a Java replacement for the Python academic system. The system also included large amounts of PL-SQL used to do the back-end computation, and large amounts of JavaScript used for the user interface. The fact that we used SQL to express our backend computation reveals one of the earliest and most enduring principles of Tamr: We are *not* building a custom database. At times, this has made things exceptionally difficult: We needed to get deep into database-specific table and query optimization to get the system to perform. But it enabled us to keep our focus on expanding our functionality and to lean on other systems for things like data movement speed, high availability, and disaster recovery.

There is no list of functions and performance benchmarks for Data Unification. It doesn't have a Transaction Processing Performance Council (TPC) benchmark where you can run the queries from the benchmark and say you've got a working system. The closest domains are MDM (master data management), with a history of massive budget overruns and too little value delivered, and ETL, a generic data processing toolkit with a bottomless backlog and massive consulting support. We didn't know exactly what we wanted to be, but we knew it wasn't either of those. We believed that our codeline needed to enable data engineers to use the machine to combine the massive quantities of tabular data inside a large enterprise (see [Stonebraker et al. 2013b] Sec. 3.2). We also believed that the dogma of traditional MDM and ETL toolkits was firmly rooted in the deterministic role of the data engineer, and that highly contextual recommenders—subject matter experts (or SMEs)—would have to be directly engaged to enable a new level of productivity in data delivery.

This begged the question of who our users would be. Data is traditionally delivered by data engineers, and patterns and semantics of data are traditionally derived by data scientists. But both data engineering and data science organizations are already overloaded with work, so putting either of those on the path to delivery would entangle delivery of results from Tamr with the IT backlog and slow things down too much. It would also hold the subject matter experts one level removed, making it dramatically harder to achieve our goals for productivity in data delivery.

This set up a long-standing tension within the company. On the one hand, much of what needs to happen in a data unification project is pretty standard data engineering: join these tables; aggregate that column; parse this string into a date, etc. Mike, in particular, consistently advocated that we have a "boxes and arrows" interface to support data engineers in defining the data engineering workflows necessary for data unification. The argument is that these interfaces are ubiquitous and familiar from existing ETL tools, and there is no need for us to innovate there.

On the other hand, the very ubiquity of this kind of tool argues strongly against building another one. Rather than building and delivering what by all appearances is yet another way to move data from system A to system B, with transformations along the way, we should focus on our core innovations—schema mapping, record matching, and classification—and leave the boxes and arrows to some other tool.

We had seen in early deployments that enormous portions of data unification projects could be distilled to a few simple activities centered around a dashboard for managing assignment and review of expert feedback. This could be managed by a data curator, a non-technical user who is able to judge the quality of data and to oversee a project to improve its quality. To keep this simple case simple, we made it easy to deploy with this pre-canned workflow. However, many projects required more complex workflows, especially as teams started to incorporate the results into critical operations. To ensure that the needs of these deployments would also be met, we built good endpoints and APIs so the core capabilities could be readily integrated into other systems. As a result, many of our projects delivered initial results in under a month, with incremental, weekly deliveries after that. This built a demand for the results within the customer organization, helping to motivate integration into standard IT infrastructure. This integration became another project deliverable along the way rather than a barrier to delivering useful results.

Lesson. Building for the users who benefit from the results is essential when we can't just point to an existing standard and say, "Look, we're so much better!"

Taming the Beast of Algorithmic Complexity

In the summer of 2013, we were engaged with a large information services provider to match information on organizations—corporations, non-profits, government agencies, etc.—against a master list of 35 million organizations. A given input was expected to have 1–2 million listings, for a total of 70 trillion comparisons. This is the core N2 challenge of entity resolution that is as old as time for enterprise data professionals.

The broadly accepted technique to address this is *blocking*: identifying one or a few attributes that can be used to divide the data into non-overlapping blocks of records, then doing the N2 comparisons only within each block. Blocking requires insight into the data to know which attributes to use, and high-quality data to ensure that each record ends up in the correct block.

We had discussed at great length our desire to build a platform that would be able to address many entity types. We already had a data-independent method of breaking the problem down, but the academic implementation didn't work well enough at the scale of 35 million records: Queries would take days, and our customer wanted them in minutes for batch queries and in a few milliseconds for single-record queries. Even though the original system was far from being able to meet these targets—and we barely had a design that would let us meet them—we agreed to the goals and to the follow-up goal of being able to perform entity resolution at the scale of 70 million on parallel hardware within a year or so after delivering at the scale of 35 million. The customer's requirements were entirely reasonable from a business perspective and embodied goals that we believed would be appealing to other customers. So we set out to build something better than what had been attempted in the previous systems.

Tamr co-founder George Beskales, who did much of the pioneering work on deduplication in the academic prototype, came up with a promising approach that combined techniques from information retrieval, join optimization, and machine learning. When Ihab Ilyas, co-founder and technical advisor to the company for all things machine learning, reviewed the initial definition of this approach, he identified multiple challenges that would cause it to fail in realistic scenarios. This kicked off several weeks of intense design iteration on how to subdivide the N2 problem to get us to the performance required. We ultimately developed an approach that is data-independent and delivers excellent pruning for batch jobs, with which we have demonstrated scaling up to 200 million records with no sign of hitting any hard limits. It is also very amenable to indexed evaluation, which provided a foundation on which we built the low-latency, single-record match also desired by many customers.

In the winter of 2016, we were working with another large information services provider to perform author disambiguation on scientific papers. While our pairwise match worked as desired, we ran into real problems with clustering. The project was deduplicating the authors of articles published in scholarly journals, and many journals use only author first initial and last name, leading to massive connected subgraphs for authors' names such as "C. Chen." Since clustering is even worse than N2 in the number of edges in the connected component, even on distributed hardware we weren't able to complete clustering of a 4 million-node, 47 million-edge-connected component. Again, George Beskales and Ihab Ilyas, in conjunction with field engineers/data scientists Eliot Knudsen and Claire O'Connell, spent weeks iterating on designs to convert this massive problem into something tractable. By arranging the data to support good invariants and tuning the clustering algorithm to take advantage of those invariants, we were able to derive a practical approach to clustering that is predictable, stable, distributes well, and has complexity of approximately Nlog(N) in the number of edges in a connected component. This let us tackle the clustering challenges for the large information services provider, as well as similar problems that had arisen with clients who were attempting to integrate customer data and master supplier data that included many subsidiaries.

Lesson. Pushing to meet customers' seemingly unreasonable demands can lead to dramatic innovation breakthroughs.

Putting Users Front and Center

The original goal of the Data Tamer academic system was to learn whether machine learning could address the challenge of data variety—the third "V" in big data, after volume and velocity. The early work on the project showed that machine learning could help quite a bit, but the quality of results wasn't good enough that enterprises would be willing to put the results into production. When we engaged with one of the academic collaborators at the Novartis Institute for Biomedical Research (NIBR) on a schema mapping project, we had an opportunity to involve subject matter experts directly in the central machine learning cycle, and this was transformative for results quality. Having subject matter experts directly review the machine learning results, and having the platform learn directly from their feedback, got us into the high 90th percentile of results quality, which was good enough for enterprises to work with. Mark Schreiber (at NIBR at the time) was a key contributor to the academic effort as well as later commercial efforts to artfully and actively integrate human subject matter expertise into our machine learning models for schema mapping.

We knew from the beginning that the product would succeed or fail based on our ability to carefully integrate human expertise through thoughtful user experience (UX) design and implementation. Building thoughtful UX is not a natural skill set for a bunch of database system and machine-learning-algorithms folks, so we set out to hire great UX and design people and set up our product development practices to keep UX front and center.

Early versions of the platform did not incorporate active learning. Customers always want to know how much SME time they will need to invest before the system will deliver high-quality results. The answer can't be 10% coverage: In the example of mastering 35 million organizations, this would mean 3.5 million labels, which is orders of magnitude too many for humans to produce. SMEs are people with other jobs; they can't spend all their time labeling data, especially when the answers seem obvious. We thus incorporated active learning to dramatically reduce the amount of training data the system needs. With these changes, entity mastering projects are able to deliver high-quality results with a few days of subject matter experts' time to train an initial model, and very low ongoing subject matter expert engagement to keep the system tuned.

SMEs are very sensitive to having their time wasted. In addition to the system not asking questions with "obvious" answers, SMEs don't want to be asked the same question, or even very similar questions, multiple times. We also need to prioritize the questions we're asking of SMEs, and the most effective method is by "anticipated impact": How much impact will answering this question have on the system? To calculate this, we need both an estimate of how many "similar" questions the system will be able to answer automatically and a metric for impact of each question. Very early on, we incorporated a value metric, often labeled "spend," that the system can use to assess value. We use coarse clustering to estimate the overall impact of a question and can prioritize that way. The Tamr system provides a facility for SMEs to provide feedback on questions, and they have not been shy about using this facility to complain when they feel like the system is wasting their time. When we incorporated these changes, the rate of that kind of feedback plummeted.

Data curators need visibility into the SME labeling workflow, so we built dashboards showing who is involved, how engaged they are, what outstanding work they have, etc. Some of the early feedback we received is that SMEs will not use a system if they believe it can be used against them in a performance review, e.g., if it shows how they perform relative to some notion of "ground truth." To overcome this impediment to engagement, we worked with curators to identify ways to give them the insight they need without providing performance scores on SMEs. The result is that curators have the visibility and tools they need to keep a project on track, and SME engagement is consistently high.

Lesson. Building a system around direct interaction with non-engineers is very challenging, but it enables us to deliver on a short timeline not otherwise possible. Humans were going to be primary in our system and we needed to have human factors engineering and design on our core team.

Scaling with Respect to Variety

The academic system was built on Postgres (of course) (see Chapter 16), and the first versions of the commercial platform were also built on Postgres, taking advantage of some Postgres-specific features—like arbitrary length "text" data type—that make design relatively straightforward. But using Postgres as the backend had two disadvantages: first, most IT departments will not take responsibility for business continuity and disaster recovery (BCDR) or high availability (HA) for Postgres; and second, it limited us to the scale practical with single-core query evaluation. We knew we needed to take advantage of multi-core, and eventually of scale-out, architecture, but we also knew building a large distributed system would be hard, with many potential pitfalls and gotchas.

We therefore evaluated a few different options.

Option #1 was explicit parallelism: Since Postgres could parallelize at the connection level, we could rewrite our queries so that each partition of a parallel query was run in a separate connection. We would need to manage transactions, consistency, etc., ourselves. This would be tantamount to building a parallel RDBMS ourselves, an option that, in Mike's words, we should pursue "over my dead body."

Option #2 was to migrate to a platform that supported parallelism internal to a query, in theory making parallelism invisible to us at the SQL level. Systems such as Vertica and Oracle provide this. This option had the advantage that IT organizations would already be familiar with how to provide BCDR and HA for these platforms. But it also had multiple downsides: It would require customers to carry an expensive database license along with their Tamr license; it would require us to support many different databases and all their idiosyncrasies; and its longevity was questionable, as we had heard from many of our customers that they were moving away from traditional proprietary relational databases and embracing much less-expensive alternatives.

Option #3 was to embrace one of the less-expensive alternatives our customers were considering and rewrite the backend to run on a scale-out platform that didn't carry the burden of a separate license. Impala and Spark were serious candidates on this front. The disadvantage of this option was that IT organizations probably wouldn't know any time soon how to provide BCDR or HA for these systems, but many organizations were building data lake teams to do exactly that, so it seemed like this option would be riding a positive wave.

After a lot of intense debate, we decided to take option #2, and build plugability to support multiple backends, hopefully reducing the cost of eventually pursuing option #3. Our early customers already had Oracle licenses and DBAs in place, so we started there. Our initial estimate was that it would take about three months to port our backend code to run on Oracle. That estimate ended up about right for a functionally complete system, although the capabilities of the product ended up being different with an Oracle backend, and it took six more months to get the performance we expected. Once we had the Oracle port ironed out, we started prototyping a Vertica port for the backend. We quickly determined that, because the behavior of Vertica is even more different from Postgres than Oracle's, we would get very little leverage from the Oracle port and estimated another six to nine months for the Vertica port, which was exorbitantly expensive for a small, early-stage company.

The reason the ports were so difficult lies at the core of what the Tamr platform is and does. A simplified version is that it takes customer data in whatever form, uses schema mapping to align it to a unified schema of our customer's design, and then runs entity resolution on the data, delivering the results in that user-defined schema. To support this workflow, the backend needs to accommodate data in a large variety of schemas. For example, a customer doing clinical trial warehousing had 1,500 studies they wanted to reformat. Each study spans 30 domains, and the source for each domain averages 5 tables. To represent these in source format, two versions of SDTM (Study Data Tabulation Model—a regulatory data model for data on clinical trials of pharmaceuticals), and their own internal clinical trial format—results in $1{,}500 \times 30 \times 8 = 360{,}000$ tables. Another customer has 15,000 studies, for 3.6 million tables. This kind of data scale—scale in the number of tables—is not something that existing RDBMSs are designed to handle.

The academic Data Tamer system chose a particular approach to address this problem, and the early versions of the Tamr platform used the same approach. Rather than represent each logical table as a separate database table, all the data was loaded as entity, attribute, value (E, A, V) triples in a single-source table—actually, (Table, E, A, V) quads—with a second (E, A, V) table for the unified data of all tables. The platform could then define the logical tables as temporary views that first filtered the EAV table, then used a crosstab to convert from EAV to rectangular table. This way, the number of tables visible to the RDBMS was limited to the tables actually in use by running queries, keeping the total number of tables visible at any one time within the limits of what the RDBMS could handle.

The downfall of this approach was that all data processing workflows needed to load, update, and read from the same two tables, so, although it scales with respect to variety of input sources, it does not scale with respect to source activity. Although we could meet customer requirements for latency in propagation of changes in one workflow, meeting those requirements in multiple workflows required additional Tamr backends. This was antithetical to our goal of having a system that scales smoothly with data variety, or the number of input tables.

This motivated us to curtail our investment in supporting additional RDBMS backends and accelerate our pursuit of option #3, embracing a backend that does not have issues with hundreds of thousands or even millions of tables and that

Figure 30.1 Particularly when architecting and building a data unification software system, technology and business strategy must evolve together. Standing, from left, are consulting software engineer John "JR" Robinson; Tamr co-founders Andy Palmer, Michael Stonebraker, and George Beskales; me (technical lead Nik Bates-Haus); and solution developer Jason Liu. Technical co-founders Alex Pagan, Daniel Bruckner, and Ihab Ilyas appear below via Google Hangout (on the screen).

supports scale-out query evaluation. This platform is a combination of Spark for whole-data queries and HBase for indexed queries.

Lesson. Scaling a system or platform hits limits in unexpected places; in our case, limits in the number of tables a backend can handle. We are pushing the limits of what traditional data management systems are able to do. Technology and business strategy are entangled and need to evolve together.

Conclusion

The Data Tamer system ventured out of Mike's well-established domain of databases and into the domain of data integration. In keeping with Mike's assertion that commercialization is the only way to validate technology, Tamr was formed to commercialize the ideas developed in the Data Tamer system. In part, customers judge how well the Tamr system works using clear metrics, such as performance and scalability, and our customers themselves are the best guides to the metrics that matter, however unreasonable those metrics might seem to us. But much of

Figure 30.2 Tamr founders, employees, and their families enjoy the 2015 Tamr summer outing at Mike and Beth Stonebraker's lake house on Lake Winnipesaukee in New Hampshire.

Tamr's approach to data integration can only be assessed by more abstract measures, such as time to deliver data, or subject matter expert engagement. As we continue to work closely with Mike to realize his vision and guidance, the ultimate validation is in the vast savings our customers attribute to our projects[1] and the testimonials they give to their peers, describing how what they have long known to be impossible has suddenly become possible.[2]

[1] "$100+ millions of dollars of ROI that GE has already realized working with Tamr" https://www.tamr.com/case-study/tamrs-role-ges-digital-transformation-newest-investor/. Last accessed April 22, 2018.

[2] "GSK employed Tamr's probabilistic matching approach to combine data across the organization and across three different initial domains (assays, clinical trial data, and genetic data) into a single Hadoop-based data within 3 months—'an unheard-of objective using traditional data management approaches.'" https://www.tamr.com/forbes-tamr-helping-gsk-bite-data-management-bullet/. Last accessed April 22, 2018.

31

The BigDAWG Codeline

Vijay Gadepally

Introduction

For those involved in the Intel Science and Technology Center (ISTC) for Big Data,[1] releasing the prototype polystore, BigDAWG, was the culmination of many years of collaboration led by Mike Stonebraker. I joined the project as a researcher from MIT Lincoln Laboratory early in 2014 and have since helped lead the development of the BigDAWG codeline and continue to champion the concept of polystore systems.

For many involved in the development of BigDAWG, releasing the software as an open-source project in 2017 was a major step in their careers. The background behind BigDAWG—such as architecture, definitions and performance results—is given in Chapter 22. This chapter gives a behind-the-scenes look at the development of the BigDAWG codeline.

The concept of polystores and BigDAWG, in particular, has been an ambitious idea from the start. Mike's vision of the future, discussed in his ICDE paper [Stonebraker and Çetintemel 2005], involves multiple independent and heterogeneous data stores working together, each working on those parts of the data for which they are best suited. BigDAWG is an instantiation of Mike's vision.

Looking back at the timeline of Mike's numerous contributions to the world of database systems, BigDAWG is one of the more recent projects. Mike's vision and leadership were critical in all stages of the project. Mike's vision of a polystore system was one of the large drivers behind the creation of the ISTC. Mike's honest, straightforward communication and attitude kept the geographically distributed team moving towards a common goal. Mike's leadership, pragmatic experience,

[1] "MIT's 'Big Data' Proposal Wins National Competition to Be Newest Intel Science and Technology Center," May 30, 2012. http://newsroom.intel.com/news-releases/mits-big-data-proposal-wins-national-competition-to-be-newest-intel-science-and-technology-center/. Last accessed March 23, 2018.

and deep theoretical knowledge were invaluable as a group of researchers spread across MIT, University of Washington, Northwestern University, Brown University, University of Chicago, and Portland State University worked together to not only advance their own research, but also integrate their contributions into the larger BigDAWG codeline.

One of the greatest strengths of the BigDAWG project has been the contributions from a diverse set of contributors across some of the best database groups in the country. While this was helpful in developing the theory, one practical challenge was working around the geographic distance. Thus, from very early on in the project we realized that, instead of weekly telecons and Skype sessions, it would be most efficient to have major code integrations done during hackathons and sprints. To keep ourselves in line with cutting-edge research, we also made sure that these hackathons led to demonstrations and publications.

The process of development (borrowing terminology from Mike's Turing lecture[2]) was:

```
Until (software release) {

   1. Meeting (often at MIT) amongst PIs and students to decide target
      demonstration and scope of code development -- Mike ends
      meetings with ‘‘Make it happen'';

   2. Distributed code development towards individual goals;

   3. Regular meetings and teleconferences to discuss and overcome
      technical challenges;

   4. Hackathon to integrate various pieces;

   5. Demonstration and paper;

}
```

As you can see, BigDAWG was developed in parts, with each new version a closer representation of the Mike polystore vision than the previous. The development of the codeline was unique in many ways: (1) individual components were built by different research groups, each with their own research agenda; (2) hackathons were used to bring these individual contributions into a coherent system; and (3)

[2] Stonebraker, M., The land sharks are on the squawk box, ACM Turing Award Lecture (video), Federated Computing Research Conference, June 13, 2015.

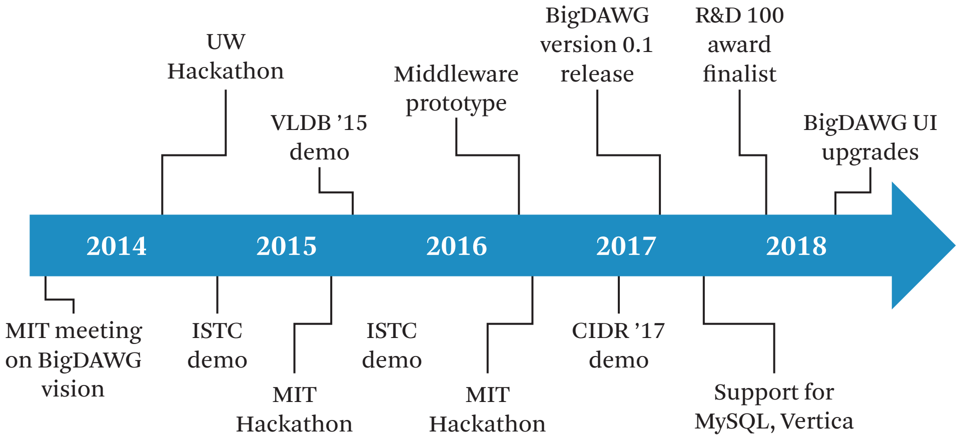


Figure 31.1 Timeline of BigDAWG milestones.

we worked closely with end users to create relevant demonstrations. BigDAWG today [Gadepally et al. 2017] is a software package that allows users to manage heterogeneous database management systems. The BigDAWG codeline[3] is made up of middleware, connectors (shims) to databases such as Postgres and SciDB, and software that simplifies getting started with BigDAWG such as an administrative interface and scripts to simplify data loading. The middleware enables distributed query planning, optimization and execution, data migration, and monitoring. The database connectors allow users to take data in existing databases and quickly register them with the middleware so that queries can be written through the BigDAWG middleware. We also have an API that can be used to issue queries, develop new islands, and integrate new database systems. The latest news and status on the BigDAWG project can be found at http://bigdawg.mit.edu.

A few major milestones in the project are shown in Figure 31.1.

Based on external observations, the BigDAWG project has been a major success on many fronts.

1. It has brought together some of the leading minds in database systems.
2. It has developed a codebase that serves as a prototype implementation of the “polystore” concept.

[3] http://github.com/bigdawg-istc/bigdawg. Last accessed March 23, 2018.

3. It has effectively created a new area of data management research. For example, Dr. Edmon Begoli (Chief Data Architect, Oak Ridge National Laboratory) says: “We recognized in the late 2000s, and through our work on large healthcare data problems, that the heterogeneity in data management and related data analysis is going to be a challenge for quite some time. After working with Mike, we learned about the polystore concept, and the BigDAWG work being led by his team. We got involved with early adoption, and it has become one of our major focus areas of database research.”

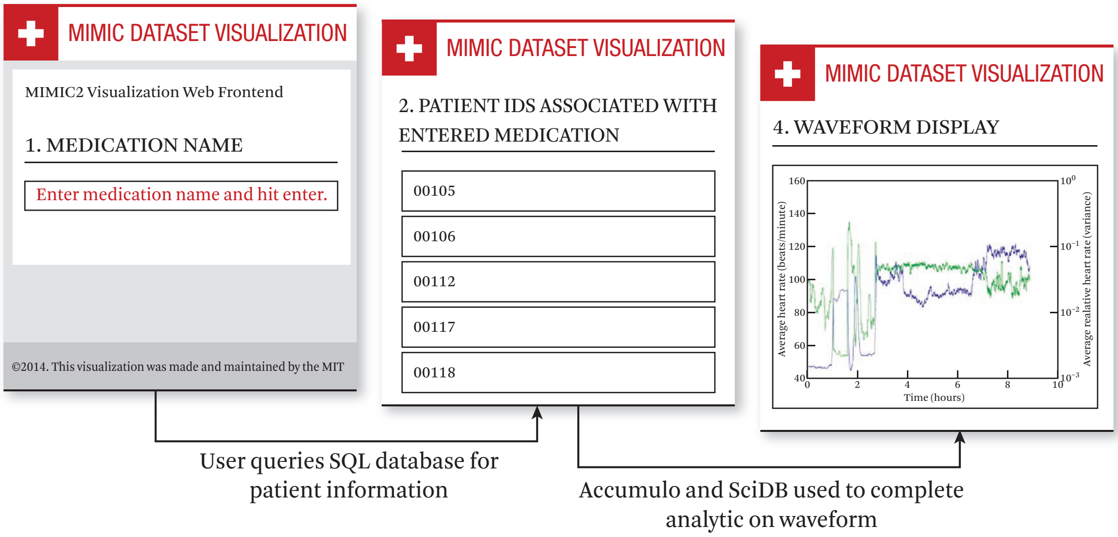
While still a relatively young codeline, we are very excited about where BigDAWG is headed.

BigDAWG Origins

The first “BigDAWG” proof of concept was demonstrated during the ISTC Retreat in Portland, Oregon, in August 2014. At this time, Mike postulated that a medical application would be the perfect use-case for a polystore system. Fortunately, fellow MIT researchers Peter Szolovits and Roger Mark had developed and released a rich medical dataset called MIMIC (short for Multiparameter Intelligent Monitoring in Intensive Care). You can find more information about the dataset at Johnson et al. [2016].

So, we had a dataset but no application in mind, no middleware, or anything, really. We promised to demonstrate “BigDAWG” well before we had any idea what it would be. As I look back, the ready-fire-aim approach to development seems to be a central theme to most of the major BigDAWG developments.

We put together the first prototype of the BigDAWG system at the University of Washington (UW) by drawing heavily upon their experience building the Myria system [Halperin et al. 2014] and our work developing D4M [Gadepally et al. 2015]. Over a two-day sprint, Andrew Whittaker, Bill Howe, and I (with remote support from Mike, Sam Madden, and Jeremy Kepner of MIT) were able to give a very simple demonstration that allowed us to perform a simple medical analysis—heart rate variability—using SciDB (see Chapter 20) and Postgres/Myria. In this demonstration, patient metadata such as medications administered was stored in Postgres/Myria and SciDB was used to compute the actual heart rate variability. While this was a bare-bones implementation, this demonstration gave us the confidence that it would be possible to build a much more robust system that achieved Mike’s grander polystore vision [Stonebraker 2015c]. Funny enough, after days of careful and successful testing, the actual demo at Intel failed due to someone closing the demonstration laptop right before the demo and subsequent complications recon-



MIMIC DATASET VISUALIZATION

MIMIC2 Visualization Web Frontend

1. MEDICATION NAME

Enter medication name and hit enter.

©2014. This visualization was made and maintained by the MIT

MIMIC DATASET VISUALIZATION

2. PATIENT IDS ASSOCIATED WITH ENTERED MEDICATION

00105

00106

00112

00117

00118

MIMIC DATASET VISUALIZATION

4. WAVEFORM DISPLAY

160
140
120
100
80
60
40
Average heart rate (beats/minute)
10^0
10^{-1}
10^{-2}
10^{-3}
Average relative heart rate (variance)
0
2
4
6
8
10
Time (hours)

User queries SQL database for patient information

Accumulo and SciDB used to complete analytic on waveform

Figure 31.2 Screenshots for MIMIC II demonstration using BigDAWG, presented for the Intel Retreat.

necting to Intel's Wi-Fi. However, we were able to get the demo running again and this initial demonstration, while hanging by a thread, was important in the development of polystore systems.

First Public BigDAWG Demonstration

The initial prototype of BigDAWG at the ISTC demonstration proved, to us and others, that it was possible for East and West Coast researchers to agree on a concept and work together and that Mike's polystore vision could be huge. However, we also realized that our initial prototype, while useful in showcasing a concept, was not really a true polystore system.

During a meeting at Intel's Santa Clara office in January 2015, we decided to push forward, use lessons learned from the first demonstration, and develop a polystore that adhered to the tenets laid out by Mike in his ACM SIGMOD blog post [Stonebraker 2015c]. Further, since Mike wanted an end application, we also decided to focus this development around the MIMIC dataset mentioned earlier. During this January 2015 meeting, BigDAWG researchers and Intel sponsors charted out what the prototype system would look like along with an outline for a demonstration to showcase the polystore in action (we eventually demonstrated BigDAWG for medical analytics at VLDB 2015 [Elmore et al. 2015], Intel, and many other

Figure 31.3 Hackathon pictures. (Top left) initial demonstration wireframes, (top right) initial BigDAWG architecture, (bottom left) hackathon in action, and (bottom right) demonstration in action at VLDB 2015.

venues). The proposed demonstration would integrate nearly 30 different technologies being developed by ISTC researchers. While individual researchers were developing their own research and code, we led the effort in pulling all these great technologies together. The goal was to allow ISTC researchers to push the boundaries of their own work while still contributing to the larger polystore vision.

Keeping track of the status of various projects was done via regular meetings and a very large spreadsheet. We also fixed the dataset and expressed to the various collaborators that a particular MIT cluster would be used for the software integration and demonstration. This helped us avoid some of the compatibility issues that can arise in large software integration efforts. In July 2017, we held a hackathon at MIT that brought together researchers from MIT, University of Washington, Brown, University of Chicago, Northwestern University, and Portland State University. Lots of long nights and pizza provided the fuel needed to develop the first BigDAWG codeline. By the end of this hackathon, we had our first BigDAWG codeline [Dziedzic et al. 2016], a snazzy demonstration, and a very long list of missing features.

Lesson. Integrating multiple parallel research projects can be a challenge; however, clear vision from the beginning and fixing certain parameters such as datasets and development environments can greatly simplify integration.

After a number of successful demonstrations, it was clear to a number of us that polystore systems would have their day in the sun. However, even with a successful demonstration, the underlying system still had no principled way of doing important tasks such as query optimization and data migration, and no clear query language. With the help of talented graduate students at MIT, Northwestern University, and University of Chicago (and further help from University of Washington and Brown University), over the next six months we developed these essential components.

Putting these pieces together was a complicated task that involved a number of very interesting technical challenges. One feature we wanted to include in the system was a monitoring system that could store information about queries, their plans, and related performance characteristics [Chen et al. 2016]. At the same time, Zuohao (Jack) She at Northwestern was working on a technique to develop query plans across multiple systems and determine semantic equivalences across heterogeneous systems [She et al. 2016]. Jack and Peinan Chen (MIT) worked together to develop a signature for each unique query, store the performance information of that query, and store these results for future use. Then, when a similar new query came in, they could leverage a pre-run query plan in order to execute the query across multiple systems (if a dissimilar query came in, the middleware would attempt to run as many query plans as possible to get a good understanding of performance characteristics that could be used for future queries). Another key feature was the ability to migrate data across multiple systems either explicitly or implicitly. Adam Dziedzic (University of Chicago) did a lot of the heavy lifting to make this capability a reality [She et al. 2016]. Ankush Gupta (MIT) also developed an execution engine that is skew-aware [Gupta et al. 2016]. These pieces formed the first real implementation of the BigDAWG middleware [Gadepally et al. 2016a].

Lesson. A layered approach to software development can be advantageous: The first steps prove an idea and subsequent steps improve the quality of the solution.

Refining BigDAWG

The VLDB and subsequent demonstrations exhibited to the wider community that the concept of a polystore was not only feasible, but also full of potential. While the medical dataset was a great use-case, it did not show the scale at which BigDAWG could work. Thus, we set about searching for an interesting use-case that showcased all the great developments since VLDB 2015 as well as a real-world large-scale problem. After months of searching for large, heterogeneous datasets without too many sharing caveats, we were introduced to a research group at MIT led by Sallie

Figure 31.4 Hackathon 3 at MIT Strata Center.

(Penny) Chisholm, Steve Biller, and Paul Berube. The Chisholm Lab specializes in microbial oceanography and biological analysis of organisms. During research cruises around the world, the Chisholm Lab collects samples of water with the goal of understanding the ocean's metabolism. These samples are then analyzed by a variety of means. Essentially, seawater is collected from various parts of the ocean, and then the microbes in each water sample are collected on a filter, frozen, and transported to MIT. Back in the lab, the scientists break open the cells and randomly sequence fragments of DNA from those organisms. The dataset contains billions of FASTQ-format [Cock et al. 2009] sequences along with associated metadata such as the location, date, depth, and chemical composition of the water samples. Each of these pieces is stored in disparate data sources (or flat files). This seemed like the perfect large-scale use-case for BigDAWG. Over the course of four months, we were able to refine BigDAWG and develop a set of dashboards that the Chisholm team could use to further their research. As before, the majority of the integration work was done in a hackathon hosted at MIT over the summer of 2016. With the help of Chisholm Lab researchers, we were able to use the BigDAWG system to efficiently process their large datasets. One of the largest challenges with this particular dataset was that, due to the volume and variety, very little work had been done in analyzing the full dataset. Putting our analytic hats on, and with significant help from Chisholm Lab researchers, we were able to develop a set of dashboards they could use to better analyze their data. By the end of this hackathon, we had integrated a number of new technologies such as S-Store [Meehan et al. 2015b], Macrobase [Bailis et al. 2017], and Tupleware [Crotty et al. 2015]. We also had a relatively stable BigDAWG codebase along with a shiny new demonstration! These results were presented at Intel and eventually formed the basis of a paper at CIDR 2017 [Mattson et al. 2017].

Lesson. Working closely with end users is invaluable. They provide domain expertise and can help navigate tricky issues that may come up along the way.

BigDAWG Official Release

By the end of the CIDR demonstration, there were now loud requests for a formal software release. This was certainly one of the larger challenges of the overall project. While developing demonstrations and code that was to be mainly used by insiders was challenging enough, we now had to develop an implementation that could be used by outsiders! This phase had a number of goals: (1) make code that is robust and usable by outsiders; (2) automate test/build processes for BigDAWG (until now, that was handled manually by graduate students), (3) develop unit tests and regression tests, and (4) documentation, documentation, and more documentation. Fortunately, we were able to leverage the experience of MIT Lincoln Laboratory researcher Kyle O'Brien, who was knowledgeable in developing software releases. He quickly took charge of the code and ensured that the geographically distributed developers would have to answer to him before making any code changes.

We ran into a number of technical and non-technical issues getting this release ready. Just to illustrate some of the complications, I recall a case where we spent many hours wondering why data would not migrate correctly between Postgres and SciDB. Going from system A to system B worked great, as did the reverse when independently done. Finally, we realized that SciDB represents dimensions as 64-bit signed integers and Postgres allows many different datatypes. Thus, when migrating data represented by int32 dimensions, SciDB would automatically cast them to int64 integers; migrating back would lose uniqueness of IDs. There were also many instances when we regretted our choice to use Docker as a tool to simplify test, build, and code distribution. We learned the hard way that Docker, while a great lightweight virtualization tool, has many known networking issues. Since we were using Docker to launch databases, middleware, and many other components, we definitely had a number of long telecons trying to debug where errors were coming up. These telecons were so frequent that Adam Dziedzic remembers a call where someone was looking up the conference line number and Mike just rattled the conference number and access code off the top of his head.

Beyond technical issues, licensing open-source software can be a nightmare. After tons of paperwork, we realized about a week before our code release that one of the libraries we were using at a very core level (computing tree edit distances in the query planner) had an incompatible license with the BSD license we intended to use. Thus, overnight, we had to rewrite this component, and test and retest everything before the code release! Finally, however, the code was released *almost* on schedule (with documentation).

Since this first release, we've had a number of contributors: Katherine Yu (MIT) developed connectors with MySQL and Vertica [Yu et al. 2017]; Matthew Mucklo (MIT) developed a new UI and is building a new federation island; and John Meehan and Jiang Du at Brown University have created a streaming island with support for the open-source streaming database S-Store [Meehan et al. 2017].

Lesson. There is often a big gap between research code and production code. It is very helpful to leverage the experience of seasoned developers in making this leap.

BigDAWG Future

Compared to many of Mike's projects, BigDAWG has a relatively young codeline. While it is currently difficult to judge the long-term impact of this project, in the short term, there are many encouraging signs. BigDAWG was selected as a finalist for the prestigious R&D 100 Award, and we have started to form a research community around the concept of polystore systems [Tan et al. 2017] as well as workshops and meetings. For example, we have organized polystore-themed workshops at IEEE BigData 2016 and 2017 and will be organizing a similar workshop (Poly'18) at VLDB 2018. We've used BigDAWG as the source for tutorials at conferences, and several groups are investigating BigDAWG for their work. Looking further into the future, it is difficult to predict where BigDAWG will go technically, but it is clear that it has helped inspire a new era in database systems.

PART VIII

PERSPECTIVES

32

IBM Relational Database Code Bases[1]

James Hamilton

Why Four Code Bases?

Few server manufacturers have the inclination and the resources needed to develop a relational database management system. Yet IBM has internally developed and continues to support four independent, full-featured relational database products. A production-quality RDBMS with a large customer base typically is well over a million lines of code and represents a multi-year effort of hundreds and, in some cases, thousands of engineers. These are massive undertakings requiring special skills, so I'm sometimes asked: How could IBM possibly end up with four different RDBMS systems that don't share components?

Mike Stonebraker often refers to the multiple code base problem as one of IBM's biggest mistakes in the database market, so it's worth looking at how it came to be, how the portable code base evolved at IBM, and why the portable version of DB2 wasn't ever a strong option to replace the other three.

At least while I was at IBM, there was frequent talk of developing a single RDBMS code base for all supported hardware and operating systems. The reasons this didn't happen are at least partly social and historical, but there are also many strong technical challenges that would have made it difficult to rewind the clock and use a single code base. The diversity of the IBM hardware and operating systems would slow this effort; the deep exploitation of unique underlying platform characteristics like the single-level store on the AS/400 or the Sysplex Data Sharing on System z would make it truly challenging; the implementation languages used by many of

[1] A version of this chapter was previously published in James Hamilton's Perspectives blog in December 2017. http://perspectives.mvdirona.com/2017/12/1187. Last accessed March 5, 2018.

the RDBMS code bases don't exist on all platforms; and differences in features and functionality across the four IBM database code bases make it even less feasible. After so many years of diverse evolution and unique optimizations, releasing a single code base to rule them all would almost certainly fail to be feature- and performance-compatible with prior releases. Consequently, IBM has four different relational database management system codelines, maintained by four different engineering teams.

DB2/MVS, now called Db2 for z/OS, is a great product optimized for the z/OS operating system, supporting unique System z features such as the Sysplex Coupling Facility. Many of IBM's most important customers still depend on this database system, and it would be truly challenging to port to another operating system such as Windows, System i, UNIX or Linux. It would be even more challenging to replace Db2 for z/OS with one of the other IBM relational code bases. Db2 for z/OS will live on for the life of the IBM mainframe and won't likely be ported to any other platform or ever be replaced by another RDBMS codeline from within IBM.

DB2/400, now called Db2 for i, is the IBM relational database for the AS/400. This hardware platform, originally called the System/38, was released way back in 1979 but continues to be an excellent example of many modern operating system features. Now called System i, this server hosts a very advanced operating system with a single-level store where memory and disk addresses are indistinguishable and objects can transparently move between disk and memory. It's a capability-based system where pointers, whether to disk or memory, include the security permissions needed to access the object referenced. The database on the System i exploits these system features, making Db2 for i another system-optimized and non-portable database. As with Db2 for z/OS, this code base will live on for the life of the platform and won't likely be ported to any other platform or ever be replaced by another RDBMS codeline.

There actually is a single DB2 code base for the VM/CMS and DOS/VSE operating systems. Originally called SQL/Data System or, more commonly, SQL/DS (now officially *Db2 for VSE & VM*), it is the productization of the original System R research code base. Some components such as the execution engine have changed fairly substantially from System R, but most parts of the system evolved directly from the original System R code base developed at the IBM San Jose Research Center (later to become IBM Almaden Research Center). This database is not written in a widely supported or portable programming language, and recently it hasn't had the deep engineering investment of the other IBM RDBMS code bases. But it does remain in production use and continues to be fully supported. It wouldn't be a good choice to port to other IBM platforms and it would be very difficult to replace while maintaining compatibility with the previous releases in production on VM/CMS and DOS/VSE.

The Portable Code Base Emerges

For the OS/2 system, IBM wrote yet another relational database system but this time it was written in a portable language and with fewer operating system and hardware dependencies. When IBM needed a fifth RDBMS for the RS/6000, many saw porting the *OS/2 DBM* code base as the quickest and most efficient option. As part of this plan, in early 1992 the development of OS/2 Database Manager (also called OS/2 DBM) was transferred from the OS/2 development team to the IBM Software Solutions development lab in Toronto. The Toronto mission was both to continue supporting and enhancing OS/2 DBM and to port the code base to AIX on the RS/6000. We also went on to deliver this code base on Linux, Windows, HP/UX, and Sun Solaris.

My involvement with this project started in January 1992 shortly after we began the transfer of the OS/2 DBM code base to the Toronto lab. It was an exciting time. Not only were we going to have a portable RDBMS code base and be able to support multiple platforms but, in what was really unusual for IBM at the time, we would also support non-IBM operating systems. This really felt to me like "being in the database business" rather than being in the systems business with a great database.

However, we soon discovered that our largest customers were really struggling with OS/2 DBM and were complaining to the most senior levels at IBM. I remember having to fly into Chicago to meet with an important customer who was very upset with OS/2 Database Manager stability. As I pulled up in front of their building, a helicopter landed on the lawn with the IBM executives who had flown in from headquarters for the meeting. I knew that this was going to be a long and difficult meeting, and it certainly was.

We knew we had to get this code stable fast, but we also had made commitments to the IBM Software Solutions leadership to be in production quickly on the RS/6000. The more we learned about the code base, the more difficult the challenge looked. The code base wasn't stable and didn't perform well, nor did it scale well in any dimension. It became clear we either had to choose a different code base or make big changes to this one quickly.

There was a lot to be done and very little time. The pressure was mounting and we were looking at other solutions from a variety of sources when the IBM Almaden database research team jumped in. They offered to put the entire Almaden database research team on the project, with the goal of replacing the OS/2 DBM optimizer and execution engine with Starburst research database components and helping to solve scaling and stability problems we were currently experiencing in the field. Accepting a research code base is a dangerous step for any development team, but this proposal was different in that the authors would accompany the code base. Pat Selinger of IBM Almaden Research essentially convinced us that we would have a world-class optimizer and execution engine and the full-time commitment of Pat, Bruce Lindsay, Guy Lohman, C. Mohan, Hamid Pirahesh, John McPherson, Don Chamberlin, the co-inventor of the Structured Query Language, and the rest of the IBM Almaden database research team. This entire team worked shoulder to shoulder with the Toronto team to make this product successful.

The decision was made to take this path. At around the same time we were making that decision, we had just brought the database up on the RS/6000 and discovered that it was capable of only six transactions per second (TPS) measured using TPC-B. The performance leader on that platform at the time, Informix, was able to deliver 69 TPS. This was incredibly difficult news in that the new Starburst optimizer, although vital for more complex relational workloads, would have virtually no impact on the simple transactional performance of the TPC-B benchmark.

I remember feeling like quitting as I thought through where this miserable performance would put us as we made a late entrance to the UNIX database market. I dragged myself up out of my chair and walked down the hall to Janet Perna's office. Janet was the leader of IBM Database at the time and responsible for all IBM database products on all platforms. I remember walking into Janet's office—more or less without noticing she was already meeting with someone—and blurting out, "We have a massive problem." She asked for the details. Janet, typical of her usual "just get it done" approach to all problems, said, "Well, we'll just have to get it fixed then. Bring together a team of the best from Toronto and Almaden and report weekly." Janet is an incredible leader and, without her confidence and support, I'm not sure we would have even started the project. Things just looked too bleak.

Instead of being a punishing or an unrewarding "long march," the performance improvement project was one of the best experiences of my career. Over the course of the next six months, the joint Toronto/Almaden team transformed the worst performing database management system to the best. When we published our audited TPC-B performance later that year, it was the best-performing database management system on the RISC System/6000 platform.

It was during this performance work that I really came to depend upon Bruce Lindsay. I used to joke that convincing Bruce to do anything was nearly impossible, but, once he believed it was the right thing to do, he could achieve as much by himself as any mid-sized engineering team. I've never seen a problem too big for

Bruce. He's saved my butt[SK1][MLB2] multiple times over the years and, although I've bought him a good many beers, I still probably owe him a few more.

The ad hoc Toronto/Almaden performance team did amazing work and that early effort not only saved the product in the market but also cemented the trust between the two engineering teams. Over subsequent years, many great features were delivered and much was achieved together.

Many of the OS/2 DBM quality and scaling problems were due to a process model where all connected users ran in the same database address space. We knew that needed to change. Matt Huras, Tim Vincent, and the teams they led completely replaced the database process model to one where each database connection had its own process and each could access a large shared buffer pool. This gave us the fault isolation needed to run reliably. The team also kept the ability to run in operating system threads, and put in support for greater than 4GB addressing even though all the operating systems we were using at the time were 32-bit systems. This work was a massive improvement in database performance and stability. And, it was a breath of fresh air to have the system stabilized at key customer sites so we could focus on moving the product forward and functionally improving it with a much lower customer support burden.

Another problem we faced with this young code base, originally written for OS/2, was that each database table was stored in its own file. There are some downsides to this model, but it can be made to work fairly well. What was absolutely unworkable was that no table could be more than 2GB. Even back then, a database system where a table could not exceed 2GB would have been close to doomed in the Unix database market.

At this point, we were getting close to our committed delivery date. The collective Toronto and Almaden teams had fixed all the major problems with the original OS/2 DBM code base and we had it running well on both the OS/2 and AIX platforms. We also could support other operating systems and platforms fairly easily. But the one problem we just hadn't found a way to address was the 2GB table size limit.

At the time I was lead architect for the product and felt very strongly that we needed to address the table size limitation of 2GB before we shipped. I was making that argument vociferously, but the excellent counter argument was we were simply out of time. Any reasonable redesign would have delayed us significantly from our committed product ship dates. Estimates ranged from 9–12 months, and many felt bigger slips were likely if we made changes of this magnitude to the storage engine.

I still couldn't live with the prospect of shipping a UNIX database product with this scaling limitation, so I ended up taking a long weekend and writing support for a primitive approach to supporting greater-than-2GB tables. It wasn't a beautiful solution, but the beautiful solutions had been investigated extensively and just couldn't be implemented quickly enough. What I did was implement a virtualization layer below the physical table manager that allowed a table to be implemented over multiple files. It wasn't the most elegant of solutions, but it certainly was the most expedient. It left most of the storage engine unchanged and, after the files were opened, it had close to no negative impact on performance. Having this code running and able to pass our full regression test suite swung the argument the other way and we decided to remove the 2GB table size limit before shipping.

When we released the product, we had the world's fastest database on AIX measured using TPC-B. We also had the basis for a very available system, and the customers that were previously threatening legal action became happy reference customers. Soon after, we shipped the new Starburst optimizer and query engine further strengthening the product.

Looking Forward

This database became quite successful and I enjoyed working on it for many releases. It remains one of the best engineering experiences of my working life. The combined Toronto and Almaden teams are among the most selfless and talented group of engineers with which I've ever worked. Janet Perna, who headed IBM Database at the time, was a unique leader who made us all better, had incredibly high standards, and yet never was that awful boss you sometimes hear about. Matt Huras, Tim Vincent, Al Comeau, Kathy McKnight, Richard Hedges, Dale Hagen, Berni Schieffer, and the rest of the excellent Toronto DB2 team weren't afraid of a challenge and knew how to deliver systems that worked reliably for customers. Pat Selinger is an amazing leader who helped rally the world-class IBM Almaden database research team and kept all of us on the product team believing. Bruce Lindsay, C. Mohan, Guy Lohman, John McPherson, Hamid Pirahesh, Don Chamberlin, and the rest of the Almaden database research team are all phenomenal database researchers who were always willing to roll up their sleeves and do the sometimes monotonous work that seems to be about 90% of what it takes to ship high-quality production systems. For example, Pat Selinger, an IBM Fellow and inventor of the relational database cost-based optimizer, spent vast amounts of her time writing the test plan and some of the tests used to get the system stable and ready to deploy into production with confidence.

IBM continues to earn billions annually from its database offerings, so it's hard to refer to these code bases as anything other than phenomenal successes. An ar-

Figure 32.1 Many leaders from DB2 Toronto. Standing, from left to right, are Jeff Goss, Mike Winer, Sam Lightstone, Tim Vincent, and Matt Hura. Sitting, from left to right, are Dale Hagen, Berni Schiefer, Ivan Lew, Herschel Harris, and Kelly Schlamb.

gument might be made that getting to a single code base could have allowed the engineering resources to be applied more efficiently. I suppose that is true, but market share is even more important than engineering efficiency. To grow market share faster, it would have been better to focus database engineering, marketing, and sales resources to selling DB2 on non-IBM platforms earlier and with more focus. It's certainly true that Windows has long been on the DB2-supported platforms list, but IBM has always been most effective selling on its own platforms. That's still true today. DB2 is available on the leading cloud computing platform but, again, most IBM sales and engineering resources are still invested in their own competitive cloud platform. IBM Platform success is always put ahead of IBM database success. With this model, IBM database success will always be tied to IBM server platform market share. Without massive platform success, there can't be database market share growth at IBM.

33

Aurum: A Story about Research Taste

Raul Castro Fernandez

Most chapters in this section, *Contributions from Building Systems*, describe systems that started in the research laboratory and became the foundation for successful companies. This chapter focuses on an earlier stage in the research lifecycle: the period of uncertainty when it is still unclear whether the research ideas will make it out of the laboratory into the real world. I use as an example Aurum, a data discovery system that is part of the Data Civilizer project (see Chapter 23). I do not give a technical overview of Aurum or explain the purpose of the system—only the minimum necessary to provide some context. Rather, this is a story about research taste in the context of systems. Concretely, it's a summary of what I have learned about research taste in the two-and-a-half-plus years that I have worked with Mike Stonebraker at MIT.

Of the many research directions one can take, I focus on what I call "new systems," that is, how to envision artifacts to solve ill-specified problems for which there is not a clear success metric. Aurum falls in this category. Within this category we can further divide the space of systems research. At one extreme, one can make up a problem, write an algorithm, try it with some synthetically generated data, and call it a system. I don't consider this to be an interesting research philosophy and, in my experience, neither does Mike (see Chapters 10 and 11); good luck to anyone who comes into Mike's office and suggests something along those lines. Let's say the minimum requirement of a "new system" is that the resulting artifact is interesting to someone other than the researchers who design the system or other academic researchers in the same research community.

Research on "new systems" starts by identifying an existing problem or user pain point. The next step is usually to identify why the problem exists, and come up with a hypothesis for how to solve it. The system should help test the hypothesis in a real scenario, in such a way that if the system works well, it should alleviate the identified problem. With Aurum, we were trying to test ideas for helping organizations discover relevant data in their databases, data lakes, and cloud repositories. It turns out that "data discovery" is a very common problem in many companies that store data across many different storage systems. This hurts the productivity of employees that need access to data for their daily tasks, e.g., filling in a report, checking metrics, or finding data necessary for populating the features of a machine learning model.

So the story of Aurum started with this "data discovery" problem. The first steps involved setting up meetings with different organizations to understand how they were thinking about their data discovery problem and what they were doing to solve or avoid it. This stage is "crucial" if one cares about actually helping in a real use-case. Often, you find research papers that claim some problem area and cite another research paper. This is perfectly fine; lots of research results are built directly on top of previous research. However, many times the claims are vague and dubious, e.g., "In the era of big data, organizations need systems that can operate underwater without electricity." Then, the researchers cite some existing paper. They probably have not talked to the people who would use the system that they are designing, but rather rely on the motivation of some previous paper to ground their contributions. It's easy to see how this quickly gets out of hand. Citing previous contributions is OK, citing previous results is OK. Citing previous motivations should raise eyebrows, but it often does not. In any case, it turns out that if you talk to real customers, they have a ton of problems. These problems may not be the ones you expect, but they are hard enough to motivate interesting research directions.

With an initial list of requirements motivated by the problem at hand, one moves on to design a system. What then follows is an aggressive back and forth of ideas and implementations that are quickly prototyped, built, and then discarded. This is because at the beginning the requirements are vague. One must always challenge the assumptions and adjust the prototype as new requirements emerge and existing ones become more and more defined. This is remarkably difficult.

Another challenge in this first stage is that the technical requirements are intermingled with business rules or idiosyncrasies of specific organizations. Often, it helps to distill the fundamental problems by talking to many different companies. Initially I wrongly assumed that as long as Aurum had read-only access to the original data sources, it would be possible to design a system that could repeatedly read that data. It turns out that reading data only once is a very desirable property of a system that is going to access many data sources within an organization—it reduces overhead on the end systems, it reduces costs in the cloud, etc. If you cannot read the same data more than once, the way you design your data structures and how you put together the system change fundamentally. As a result, the system would be completely different. Of course, this process is far from perfect, so you typically finish a first prototype of the system and find many necessary features are missing.

A lot of noise is introduced while designing the prototype as well, from simplifying assumptions to pieces that are not optimized and pose serious usability issues. On top of that, it is easy to prioritize the more interesting technical challenges instead of those that may have more of an impact for end users. This noisy process is the whole reason why it's so important to release prototypes soon, showing them to the people who have the real problem, and receiving feedback and impressions as early as possible. This is why it's so important to be open to challenging one's assumptions relentlessly. Getting continuous feedback is key to steering the system in the right direction.

What then is the right direction? One may argue, if you are in research, that the right direction in computer science in the 21st century is to optimize the least publishable unit (LPU) grain. In other words, define small problems, do thorough technical work, and write lots of papers that are necessary to progress in one's career. These days, this approach increases your chances of getting the paper published while minimizing the amount of effort that goes into the research. This, however, is generally at odds with doing impactful research. Focusing on real problems is a riskier path; just because one aims to build a system to solve a real problem does not mean the process will be successful, and it is definitely incompatible with the research community's expectation of publishing many papers. This brings to the table two different philosophies for systems research: make it easy or make it relevant.

The "right" style is a matter of research taste. My research taste is aligned with making research relevant. This is one of the main things you learn working with Mike. The major disadvantage of making research relevant is that it is a painful process. It involves doing a lot of work you know won't have a significant impact in the resulting research paper. It brings a handful of frustrations that have no connection with either the research or the system. It exposes you to uncountable sources of criticism, from your mentors, your collaborators, and the users of the system. When you show your prototype to the public, there are always many ruminating thoughts: Will the prototype be good enough? Will their interests be aligned or will they think this is irrelevant and disconnected? On top of those imagined frustrations and fears, there are real ones.

I still remember clearly a meeting with an analyst from a large company. We had been collaborating for a while and discussing different data discovery problems within the company. I had showed him multiple demos of Aurum, so I was confident that the interests were well aligned. After some back and forth we agreed to try Aurum in some of their internal databases. This is a painful process for industrial collaborators because they have to deal with internal challenges, such as obtaining the right permissions to the data, getting the legal team to set up appropriate agreements, and a myriad of other hurdles that I could not have imagined. This was the first real test for Aurum. When I arrived in the office, we made coffee—that is always the first step. I always have a long espresso, so that's what I brought to the meeting room. We sat in our desks and started the job right away; we wanted to minimize the deployment time to focus on what we would do next. I had pre-loaded two public databases, which I had access to, so the only remaining bit was to include the internal database. I started the instance, connected to the database with the credentials they gave me, and fired up the process. A couple of minutes into our meeting, Aurum was already reading data from the internal database at full speed. We started chatting about some ideas, discussing other interesting technologies while enjoying our coffee. I had barely taken a sip of my espresso when I looked at the screen and saw that something was obviously very wrong. A variable which should have been ranging in the single-digit millions was tens of millions and growing!

Previously, I had tested Aurum using large datasets that I found in the field, under different test scenarios and using more complex queries than I expected to find in the company. However, I had overlooked a basic parameter, the vocabulary size. I knew that the deployment was poised to break. The gist of the problem was that Aurum was building an internal vector proportional to the size of the vocabulary. As long as the vector fits in memory, there is no problem. Although I had tried Aurum with large datasets, I did not account for the vocabulary size. The database that we were processing had tens of millions of different domain-specific terms. Ten minutes into the meeting the process had failed. The analyst, very graciously, proposed to rerun the process, but knowing the internal issue, I said that it would not help. There was something fundamental that I would need to change. The feeling of having wasted everybody's time and resources was discouraging.

When you see a project fail in front of your eyes, a lot of questions come to mind: Did I waste this opportunity? Am I going to be able to try this again? Even if it had worked, would it have helped the research? The only way forward, though, is to get back to the office and power through the storm. Building research systems is arduous. There is a constant pressure to understand whether what you are working on is contributing to the research question you are trying to solve. There is a voice that keeps asking: Is somebody going to care about this anyway? It is a painful experience to demonstrate prototypes; naturally people focus on the things that do not work as expected—such things may not have been on your radar, but are the important points that should be handled. You receive all kinds of feedback, the type you are interested in and other types that may not be immediately relevant, but that people will deliver, trying to be helpful. On top of these frustrations, there are other kinds of external pressures that add to the mix. Today, if you are in research, building systems and taking the time to make them relevant is not necessarily aligned with an academic value system that tends to value higher paper counts (see Chapter 11). This makes you continually challenge your decisions, and wonder whether the pain is necessary.

Mike's unperturbed belief that "make it relevant" is the only way forward has helped me stay on track despite setbacks. The story above has a happy ending. We redeployed the system a few months later and loaded the database successfully. We learned a lot during the process and, more importantly, we made the research relevant to a real problem. Making research relevant is not easy, but the pain is worth it. Hearing a new customer echoing the problems you've been dealing with before, and noting how quickly you can empathize and help, is satisfying. Creating something with the potential of having an impact beyond the research lab is even more satisfying. Ultimately, it all boils down to research taste: make it easy or make it relevant. I've chosen my path.

34

Nice: Or What It Was Like to Be Mike's Student

Marti Hearst

There are three people who were pivotal to my success as a researcher, and Mike Stonebraker was the first of these—and also the tallest! I am very pleased to be able to share in this tribute to Mike, from my perspective as one of his former students.

Mike truly is a visionary. He not only led the way in the systems end of databases, but he was also always trying to bring other fields into DBMSs. He tried to get AI and databases to work together (OK, that wasn't the most successful effort, but it turned into the field of database triggers, which was enormously successful). He led early efforts to bring economic methods to DBMSs, and was a pioneer in the area of user interfaces and databases. I remember him lamenting back around 1993 that the database conferences would not accept papers on user interfaces when the work on Tioga [Stonebraker et al. 1993b, 1993c] was spurned by that community. That work eventually led to the Informix visualization interface, which again was ahead of its time. Shortly after that, Oracle had more than 100 people working in its visualization group.

Not only is Mike a visionary, but he is also inspirational to those around him. Back around 1989, when I was a graduate student who didn't find databases interesting enough for my dissertation and instead wanted to work on natural language processing (NLP), Mike said to me: "If you're going to work with text, think BIG text." I took that on as a challenge, despite the strange looks I got from my NLP colleagues, and as a result, was one of the first people to do computational linguistics work on large text corpora. That approach now dominates in the field, but was nearly unheard of at the time.

Another time, after I'd obtained significant research results with big text, Mike said something to the effect of, "What we need are keyboardless interfaces for text—why don't you solve that?" This led me to start thinking about visualization for interfaces for text search, which in turn led to several inventions for which I am now well known, and eventually to my writing the very first academic book on the topic, *Search User Interfaces* [Hearst 2009]. Again, it was Mike's vision and his special form of encouragement that led me down that path.

Mike also taught me that a world-famous professor wasn't too important to help a student with annoying logistical problems that were blocking research. In 1989, it was very difficult to get a large text collection online. I still remember Mike helping me download a few dozen *Sacramento Bee* articles from some archaic system in some musty room in the campus library, and paying the $200 to allow this to happen.

I first met Mike when I was a UC Berkeley undergraduate who wandered into his seminar on next-generation databases. I needed a senior honors project, and even though he had just met me and I hadn't taken the databases course, Mike immediately suggested I work with him on a research project. He was the first and only CS professor I'd encountered who simply assumed that I was smart and capable. In retrospect, I think that Mike's attitude toward me is what made it possible for me to believe that I could be a CS Ph.D. student. So even though I suspect that sometimes Mike comes across as brusque or intimidating to others, toward students, he is unfailingly supportive.

As further evidence of this, in terms of number of female Ph.D. students advised and graduated from the UC Berkeley CS department, in 1995 Mike was tied for first with eight female Ph.D.s graduated (and he'd have been first if I'd stuck with databases rather than switching to NLP). I don't think this is because Mike had an explicit desire to mentor female students, but rather that he simply supported people who were interested in databases, and helped them be the very best they could be, whoever they were and whatever skills they brought with them. As a result, he helped elevate many people to a level they would never have reached without him, and I speak from direct experience.

Mike's enormous generosity is reflected in other ways. I still remember that when he converted the research project Postgres into a company (Illustra Corporation), he made a big effort to ensure that every person who had contributed code to the Postgres project received some shares in the company before it went public, even though he had no obligation whatsoever to do that. Although a few of us who contributed small amounts of code were overlooked until almost the very end, he insisted that the paperwork be modified right before the IPO so that a few more

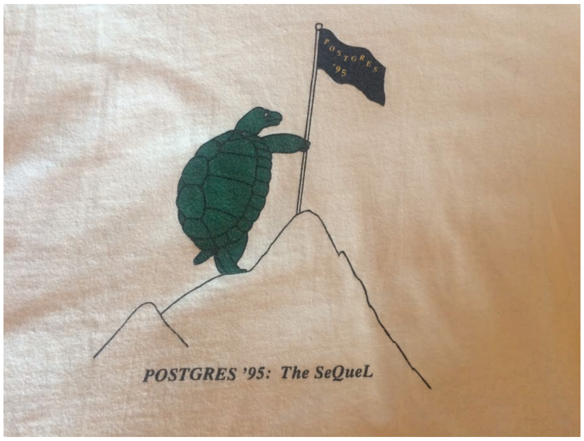


Figure 34.1 Logo from the Postgres'95 t-shirt.

people would get shares. I find it hard to imagine anyone else who would do that, but Mike is extraordinarily fair and generous.

Mike is also very generous with assigning research credit. The aforementioned undergraduate research thesis had to do with the Postgres rules system. After I joined his team as a graduate student in 1987, Mike wrote up a couple of thought pieces on the topic [Stonebraker and Hearst 1988, Stonebraker et al. 1989] and insisted on including my name on the papers even though I don't believe I added anything substantive.

Mike's projects were very much in the tradition of the UC Berkeley Computer Science Department's research teams, consisting of many graduate students, some postdoctoral researchers, and some programming staff. Mike fostered a feeling of community, with logos and T-shirts for each project (see Figure 34.1 for one example) and an annual party at his house in the Berkeley hills at which he gave out goofy gifts. Many of his former students and staff stay in touch to various degrees, and, as is common in graduate school, many romantic relationships blossomed into eventual marriages.

So that's Mike Stonebraker in a nutshell: visionary, inspirational, egalitarian, and generous. But surely you are thinking: "Hey, that can't be the whole story! Wasn't Mike kind of scary as a research advisor?" Well, OK, the answer is yes.

I still remember the time when I was waffling around, unable to decide on a thesis topic with my new advisor, and so I made an appointment to talk with my now former advisor Mike. For some reason, we'd scheduled it on a Saturday, and it was pouring outside. I still remember Mike appearing at the office and looking down at me from his enormous height and basically saying something like, "What's wrong with you? Just pick a topic and do it!" From that day on, I was just fine and had no problem doing research. I have found that for most Ph.D. students, there is one point in their program where they need this "just do it" speech; I'd otherwise never have had the guts to give it to students without seeing how well this speech worked on me.

I also remember the extreme stances Mike would take about ideas—mainly that they were terrible. For instance, I was there for the ODBMS wars. I remember Mike stating with great confidence that object-oriented was just not going to cut it with databases: that you needed this hybrid object-relational thing instead. He had a quad chart to prove it. Well, he hadn't been right about putting expert systems into databases, but he certainly ended up being right about this object-relational thing (see Chapter 6).

As with many great intellects, Mike very much wants people to push back on his ideas to help everyone arrive at the best understanding. I remember several occasions in which Mike would flatly state, "I was utterly and completely wrong about that." This is such a great lesson for graduate students. It shows them that they have the opportunity to be the one to change the views of the important professor, even if those views are strongly held. And that of course is a metaphor for being able to change the views of the entire research community, and by extension, the world (see Chapter 3).

As I mentioned, Mike is a man of few words, at least over email. This made it easy to tell when you'd done something really, truly great. Those of you who've worked with him know that treasured response that only the very best ideas or events can draw out of Mike. You'd send him an email and what you'd see back would be, on that very rare occasion, the ultimate compliment:

neat.
/mike

35

Michael Stonebraker: Competitor, Collaborator, Friend

Don Haderle

I became acquainted with Mike in the 1970s through his work on database technology and came to know him personally in the 1990s. This is the perspective of a competitor, a collaborator, and a friend. Mike is a rare individual who has made his mark equally as an academic and an entrepreneur. Moreover, he stands out as someone who's always curious, adventurous, and fun.

What follows are my recollections of Mike from the early days of the database management industry through today.

After IBM researcher Ted Codd proposed the relational model of data in 1969 [Codd 1970], several academic and industry research laboratories launched projects that created language and supporting technologies, including transaction management and analytics. IBM's System R [Astrahan et al. 1976] and UC Berkeley's Ingres [Held et al. 1975] emerged as the two most influential projects (see Chapter 13). By the mid-1970s, both efforts produced concrete prototypes. By the early 1980s, various relational database management systems had reached the commercial market.

In the early 1970s, I was an IBM product developer focusing on real-time operating systems and process control, file systems, point-of-sales systems, security systems, and more. In 1976, I joined a product development team in IBM that was exploring new database technology that responded to intense customer demands to dramatically speed up the time it took them to provide solutions for their fast-moving business requirements. This was my baptism in database and my first acquaintance with Mike. I devoured his writings on nascent database technology.

In the 1970s, Mike and the Ingres team developed seminal work in concurrency control [Stonebraker 1978, 1979b] indexing, security [Stonebraker and Rubinstein 1976], database language, and query optimization for the distributed relational database Ingres [Stonebraker et al. 1976b]. Ingres targeted DEC minicomputers, combining a set of those machines to address large database operations. By contrast, IBM System R targeted mainframes, which had adequate power for most enterprise work of the era; it was this research project that formed the basis of IBM's DB2 [Saracco and Haderle 2013] mainframe database.

In the early 1980s, Mike formed Relational Technology, Inc. with Larry Rowe and Gene Wong, and delivered a commercial version of Ingres to the market. This made Mike our indirect competitor (Ingres addressed a different market than DB2 and competed directly against Oracle). Mike remained unchanged. He shared what he learned and was a willing collaborator.

Mike impressed me not only as an academic but also as a commercial entrepreneur. The trio of Mike, Larry, and Gene had to learn how to create and operate a business while still maintaining their professorial positions at UC Berkeley. This was no small feat, struggling through financing, staffing, and day-to-day operations while still advancing technology and publishing at the university. They pioneered open source through Berkeley Software Distribution (BSD) licensing of the university Ingres code, which overcame the commercial restrictions of the university licensing arrangement of the time. They came to the conclusion that the value was in their experiences, not in the Ingres code itself. This enlightenment was novel at the time and paved the way for widespread use of open source in the industry (see Chapter 12).

The trio made some rookie missteps in their commercial endeavor. Mike and Gene had developed the QUEL language for relational operations as part of Ingres while IBM had developed SQL for System R [Chamberlin et al. 1976]. There were academic debates on which was better. In 1982, serious work began by the American National Standards Institute (ANSI) to standardize a relational database language. SQL was proposed and strongly supported by IBM and Oracle. Mike did not submit QUEL, rationalizing that putting it in the hands of a standards committee would limit the Ingres team's ability to innovate. While that was a reasonable academic decision, it was not a good commercial decision. By 1986, the industry standardized on SQL, making it a requirement for bidding on relational database contracts with most enterprises and governments around the world. As a result, Ingres had to quickly support SQL or lose out to Oracle, their primary competitor. At first the Ingres team emulated SQL atop the native QUEL database but with undesirable results. The true SQL version required major reengineering and debuted in the early 1990s. This misstep cost the team five-plus years in the market.

Beyond the database, the Ingres team developed fantastic tools for creating databases and applications using those databases (see Chapter 15). They recognized that a database needed an ecosystem to be successful in the market. Oracle created popular applications (business financials) for their database, responding to a changing market wherein customers wanted to buy generic business applications rather than build the applications themselves. Unfortunately, Mike and the team couldn't convince their investors to fund the development of applications for Ingres, and they had a difficult time convincing application vendors to support their nascent database, especially since the team did not support the standard interface of SQL. The trio had more to learn to succeed commercially.

Commerce rapidly adopted relational databases and all manner of information technology in the early 1980s, digitizing their businesses. With this came demand to include new types of data beyond the tabular bookkeeping data supported by the first-generation relational databases. In the mid-1980s, object-oriented databases appeared to take on this challenge. Researchers explored ways to extend the relational data model to manage and operate on new data types (e.g., time series, spatial, and multimedia data). Chief among such researchers was Mike, who launched the Postgres project at UC Berkeley to explore new and novel ways to extend Ingres to solve more problems. (IBM Research did similarly with the Starburst project [Haas et al. 1989]). Indeed, a presentation [Stonebraker 1986c] delivered by Larry Rowe and Mike in 1986 at an object-oriented conference in Asilomar, CA, inspired the rest of us in the relational community to step it up.

Mike led the way in defining the Object-Relational model, drove the early innovations, and brought it to market. In the early 1990s, Mike started his second company, Illustra Corporation, to commercialize Postgres object-relational. Illustra[1] offered superb tools with their "data blades" for creating data types, building functions on these types, and specifying the use of storage methods created by third parties for those objects that would yield great performance over and above the storage methods provided by the base Illustra server. Once again, Mike's company demonstrated that a good database needs a great ecosystem of tools and applications. This technology would extend the relational database to handle geophysical data, multimedia, and more.

[1] Illustra was acquired by Informix in 1997, which was in turn acquired by IBM.

When Mike created Postgres in 1986, QUEL was the database language to which he added extensibility operators (PostQuel). SQL was added to Postgres in the mid-1990s, creating PostgreSQL, which has since become one of the most popular databases on the planet.[2] The Illustra team had to re-engineer their database to provide native SQL support. At the same time, SQL was being extended in ANSI for object-relational. This would manifest in standard language extensions to SQL in 1999 (SQL3), covering object abstraction (data types and methods), and not covering any implementation extensions (e.g., storage methods). The Illustra team didn't participate in the standardization committees, which set them back a bit to reconcile the different models. Illustra had the best technology, database, and tools for this market, but skipped a couple of steps to conform to SQL.

The rest of us would lag behind another couple of years, focusing our energies on the evolving distributed computing complex and the insatiable demands for performance and availability for transaction and analytics (parallelism). In the relational arena, Tandem set the bar for highly available transaction processing, and Teradata set the bar for high-performance query processing. By the late 1980s, thanks to Moore's Law, the minicomputers of the 1970s were growing powerful enough to be combined to do the work of a mainframe, only increasingly cheaper. To compete on cost, the mainframe was reengineered to use CMOS, the technology underlying the minis, resulting in IBM's Parallel Sysplex [IBM 1997, Josten et al. 1997], a cluster of IBM mainframes acting together as a single-system image delivered to market in 1995.

The client-server architecture emerged on the back of the growing popularity and capabilities of personal computers and workstations in business environments. Sybase debuted in the late 1980s, pioneering client-server. With the evolution of Sun and other network computers, enterprise architectures evolved from single-tier to multi-tier computing. Distributed computing had arrived. We harvested the work of IBM's System R* [Lindsay 1987], Mike's Ingres, and David DeWitt's Gamma [DeWitt et al. 1990] to deliver distributed database technology for transaction processing and highly parallelized query processing. In the early 1990s, with a change in IBM leadership, we were funded to construct DB2 on open systems (UNIX, etc.) on popular hardware platforms (IBM, HP, Sun). The combination of open systems, distributed database, and massive parallelism would occupy most

[2] In my view, Mike's most significant achievement commercially was perhaps unintentional: the development of and open sourcing of Postgres. PostgreSQL is one of the most popular database management systems on the planet and paved the way for the open-source movement.

of our development energies through the mid-1990s along with mobile and embedded databases.

The volume of data and number of databases within enterprises grew rapidly through the 1980s across the spectrum of database vendors. Customers separated their transactional systems from their analytical systems to better manage performance. And they discovered that they needed to analyze data across multiple data sources. This gave rise to *data warehousing*, which extracted data from multiple sources, curated it, and stored it in a database designed and tuned for analytics. An alternative architecture, *federation*, was proposed, which allowed for analytics across disparate data sources without copying the data into a separate store. This architecture was well suited for huge data, where copying was a prohibitive cost, as well as near real-time requirements. IBM Research's Garlic project [Josifovski 2002] and Mike's Mariposa project [Stonebraker et al. 1994a] explored this architecture, spurring technology in semantic integration and performance in queries on disparate databases by taking advantage of the unique performance characteristics of the underlying stores. Mariposa became the basis for Cohera Corporation's[3] application system and was later incorporated in PeopleSoft in the early 1990s. Garlic was incorporated in DB2 as the Federated Server in 2002. Neither reached widespread popularity because of the complexity in managing heterogeneous, federated topologies. As of 2018, we're seeing the evolution of multi-model databases and Mike's polystore (see Chapter 22), which draw on the federated technology and the modeling capabilities of object-relational to integrate a set of data models (relational, graph, key value) while providing best-of-breed capability for the individual data model—a bit of a snap back to OSFA (One Size Fits All) [Stonebraker and Çetintemel 2005].

In the late 1990s, the executive team for database management within IBM viewed the company as too inward-focused. We needed external perspective on technology directions as well as business directions and to assess ourselves against the best industry practices. I asked Mike to present an external perspective on technology to the IBM database management product executives, led by Janet Perna. Although he was competing with IBM at the time, Mike agreed. And he did a stellar job. His message was clear: "You need to step it up and look beyond IBM platforms." And it had the intended effect. We stepped it up.

In 2005 I retired from IBM. I worked as little as possible, consulting with venture capitalists and startups. Mike started Vertica Systems, Inc., the column-store

[3] Founded in 1997 and acquired by PeopleSoft in 2001.

database based on C-Store [Stonebraker et al. 2005a]. He asked me to present the Vertica technology to prospective customers on the West Coast so he could focus more on the development teams on the East Coast and the customers in that geography. I was impressed with C-Store and Vertica for dramatically improving the performance of analytical systems (see Chapter 18). I agreed. And I worked as little as possible. Vertica was sold to HP in 2011.

In 2015 Mike received the Turing Award and spoke at IBM on analytics. Mike was working on SciDB (see Chapter 20) at the time and he was not enamored of Apache Spark, the analytical framework that IBM was pushing in the market. I was asked to attend the talk along with a few other retired IBM Fellows. Mike asked if one of us would defend Spark. He wanted a lively discussion and needed someone to provide counterpoint. I agreed. It was fun. That was Mike. He won. Then we went out for a drink.

Mike and I orbited the database universe on different paths. Mike was an innovator-academic who created commercial products, whereas I created commercial products and did some innovation [Mohan et al. 1992]. They sound alike, but they're not. We shared our knowledge of customer needs from our different perspectives and ideas on technology to serve them better. And, as I said, Mike was a competitor, a collaborator, and always a friend.

36

The Changing of the Database Guard

Michael L. Brodie

You can be right there at a historic moment and yet not see its significance for decades. I now recount one such time when the leadership of the database community and its core technology began to change fundamentally.

Dinner with the Database Cognoscenti

After spending the summer of 1972 with Ted Codd at IBM's San Jose Research Lab, Dennis Tsichritzis, a rising database star, returned to the University of Toronto to declare to Phil Bernstein and myself that we would start Ph.Ds. on relational databases under his supervision. What could possibly go wrong?

In May 1974, I went with Dennis to the ACM SIGFIDET (Special Interest Group on File Description and Translation) conference in Ann Arbor, Michigan, my first international conference, for the *Great Relational-CODASYL Debate* where Dennis would fight for the good guys. After the short drive from Toronto, we went to a "strategy session" dinner for the next day's debate. Dinner, at the Cracker Barrel Restaurant in the conference hotel, included the current and future database cognoscenti and me (a database know-nothing). It started inauspiciously with Cracker Barrel's signature, neon orange cheese dip with grissini ('scuse me, breadsticks).

I was quiet in the presence of the cognoscenti—Ted Codd, Chris Date of IBM UK Lab, and Dennis—and this tall, enigmatic, and wonderfully confident guy, Mike something, a new UC Berkeley assistant professor and recent University of Michigan Ph.D. According to him, he had just solved the database security problem with QUEL, his contrarian query language. During dinner, Mike sketched a few visionary ideas. This was further evidence for me to be quiet since I could barely spell datakbase [sic].

The *Great* Relational-CODASYL Debate

The cognoscenti who lined up for the debate were, on the relational side, Ted Codd, Dennis Tsichritzis, and Kevin Whitney, from General Motors, who had implemented RDMS [Kevin and Whitney 1974], one of the first RDBMSs. On the CODASYL side were Charlie Bachman, who was awarded the 1973 Turing Award "for his outstanding contributions to database technology"; J. R. Lucking, International Computers Limited, UK; and Ed Sibley, University of Maryland and National Bureau of Standards.

The much-ballyhooed debate was less than three years after Codd's landmark paper [Codd 1970]; one year after Charlie's Turing Award; one year into Mike's and Eugene Wong's pioneering Ingres project (see Chapter 15) at UC Berkeley; coincident with the beginning of the System R project (see Chapter 35) at IBM Research, San Jose; five years before the release of Oracle, the first commercial RDBMS, in 1979, followed in 1983 by IBM's DB2 (see Chapter 32); and almost a decade before Ted was awarded the 1981 Turing Award "for his fundamental and continuing contributions to the theory and practice of database management systems," specifically relational databases.

SIGFIDET 1974 focused largely on record-oriented hierarchical and network databases. Relational technology was just emerging. Most significantly, SEQUEL (now SQL) was introduced [Chamberlin and Boyce 1974]. Three papers discussed concepts and six[1] RDBMS implementations: IBM Research's XRM-An Extended (N-ary) Relational Memory, The Peterlee IS/1 System, and Rendezvous; Whitney's RDMS; and ADMINS and the MacAIMS Data Management System. Mike's paper [Stonebraker 1974b] on a core relational concept, like those of Codd, Date, and Whitney, showed a succinct and deep understanding of the new relational concepts, in contrast to the debate.

The much-anticipated debate was highly energized yet, in hindsight, pretty ho-hum, more like a tutorial as people grappled with new relational ideas that were so different from those prevalent at the time. The 23-page debate transcript [SIGFIDET 1974] should be fascinating to current database folks given the emergent state of database technology and the subsequent relational vs. CODASYL history. Ted, some IBMers, Whitney, Mike, and about five others were the only people in the crowded room that had any RDBMS implementation experience. Of that number, only Ted and Kevin Whitney spoke in the debate. Everyone else was living in a different world.

[1] Amazingly, approximately 10 RDBMSs were implemented or under way within three years of Ted's landmark paper.

From the transcript, Mike seemed curiously quiet.[2] Truth was he had his hand up the whole time but was never called upon.[3]

In hindsight, most questions/comments seem weird. "Why were ownerless sets better than navigating data?" "Why is the network model worse than the relational model as a target for English?" "I couldn't find many examples of the relational sublanguage compared to CODASYL subschemas." "I can think of many systems that I have had in which questions would come up so that it was almost impossible, and certainly impractical, to automate a way of coming up with the answer. To me, it boils down to a question of economics. Is it worth spending the money and taking the time to be able to provide this kind of availability to anybody?" In contrast, Ted's clear focus was on "applications programming, support of non-programmers, . . . and implementation" and on the logical and physical data independence that remain the cornerstones of the relational model [Codd 1970, Date and Codd 1975], emphasized succinctly by Mike [Stonebraker 1974b] and in sharp contrast to the network approach and most of what was said in the debate. The relational side was casting pearls [Matthew 7:6].

For all the fireworks projected for the debate, it was bland. So, my mind wandered to J.R. Lucking, who smoked a cigarette throughout. It was, after all, 1974. Why pay attention? It was distracting. Smoke never came out. We imagined that J.R. periodically left the room to empty an otherwise hollow leg of smoke and ash.

The debate had little impact outside the room. The real debate was resolved in the marketplace in the mid-1980s after the much-doubted adoption of Oracle and DB2[4] and as SQL became, as Mike called it, "intergalactic data speak." The elegance of Codd's model would never have succeeded had it not been for RDBMS performance due to Pat Selinger's query optimization, enabled by Ted's logical and physical data independence,[5] plus tens of thousands of development hours spent on query and performance optimization.

[2] Holding back like that didn't last long for Mike.

[3] The database cognoscenti who were running the debate may not have foreseen that in 40 years the tall, new guy with the unanswered hand would receive the Turing Award for the very issues being debated.

[4] DB2 was IBM's #2 DBMS product after its #1 DBMS product, IMS.

[5] Mike was already at the heart of the performance issue [Stonebraker 1974b] described so eloquently by Date and Codd [1975] in the same conference and missed by debate questioners. Mike has generalized this as the key requirement of any new data model and its data manager.

The debate and conference had a huge impact . . . on me. Ted Codd became a mentor and friend, calling me almost daily throughout the Falklands War to review the day's efforts of Britain's Royal Air Force (RAF).[6] Charlie, who lived up the street from me in Lexington, MA, later offered me a CTO job. I declined but gained sartorial knowledge about buttons that I didn't know I had. Ed Sibley, my first academic boss at the University of Maryland, got me appointed chair of the ANSI/SPARC (American National Standards Institute, Standards Planning and Requirements Committee) Relational Standards Committee, where I proposed, with other academics, to standardize the relational calculus and algebra, to allow multiple syntaxes, e.g., SQL, QUEL, and QBE. I lost that job to an IBMer who came with a 200-page SQL specification. (Who knew that standards were a business and not a technical thing? Nobody tells me anything.)

While the debate had little impact on the community at the time, it marked the changing of the guard from the leaders of the hierarchal and network period of database research and product development. In the debate, they had posed the odd questions presumably trying to understand the new ideas relative to what they knew best. The torch was being passed to those who would lead the relational period that is still going strong almost half a century later. As the 1981, 1998, and 2014 Turing Awards attest, the new leaders were Ted Codd, Jim Gray, and Michael Stonebraker. With more than ten relational DBMSs built at the time of the debate and the three most significant relational DBMSs in the works, the database technology shift to relational databases was under way.

Mike: More Memorable than the Debate, and Even the Cheese

Apart from the neon orange cheese, SQL, and being awed by database cognoscenti, there was little memorable about SIGFIDET 1974, except meeting Mike Stonebraker. Mike Something became a colleague and friend for life. Although a stranger and the most junior academic at the strategy dinner (I don't count), Mike was unforgettable, more so as time went on. Me: "Hey, Mike, remember that dinner before the Relational-CODASYL Debate?" Mike: "Sorry, I don't remember." Maybe it's like a fan meeting Paul McCartney: Only one of the two remembers. For this chapter, I asked Dennis Tsichritzis and other database cognoscenti for memorable moments at this event, to a uniform response of "not really." Don Chamberlain and Ray Boyce,

[6] In World War II, Ted trained as a pilot in Canada with the British Royal Air Force. I am Canadian, and my mother, an Englishwoman, had been in the British Women's Royal Air Force.

SQL inventors, were there [Chamberlin and Boyce 1974]. But most future relational cognoscenti had not even joined the database party. Bruce Lindsay and Jim Gray were at UC Berkeley and would move that year to the System R project at IBM. The instrumental Pat Selinger was at Harvard (Ph.D. 1975) and wouldn't join System R until after her Ph.D. SIGFIDET 1974 was a milestone that marked the end of the old guard and emergence of the relational era with most of the relational cognoscenti still wet behind the relational model, and Mike Stonebraker, unwittingly, taking the lead.

To this day, Mike is succinct in the extreme, intense, visionary, and superbly confident. Yet at the debate, he was strangely quiet (not called upon) especially as he was in the 1% who understood Ted's model and had implementation experience. Perhaps he was gaining his sea legs. He had been an assistant professor for about three years. Forty years later, at his Festschrift, Mike recalled those tenure-grinding years as the worst of his career due to the pressures of an academic life—teaching and tenure, in a new domain, developing one of the most significant database systems from scratch, while, as Don Haderle says in Chapter 35, having to learn "how to create and operate a business." At SIGFIDET he was new to databases, having learned what a database was two years earlier when, while Mike was wondering what to do at UC Berkeley, Gene Wong had suggested that he read Codd's paper [Codd 1970]. Mike's first Ph.D. student, Jerry Held, had already implemented a DBMS. By May 1974, Mike had already impressed the relational cognoscenti, the then-future of databases. Today at conferences, people universally wait to hear Mike's opinions. Or in his absence, as at VLDB 2017, Mike's opinions tend to be quoted in every keynote speech. On issues critical to him, he speaks out with succinct observations and questions that get right to the heart of the matter. For example, he might ask, "What use-case and workload do you envisage?" Answer: Rhubarb, rhubarb, rhubarb. Mike replies: "Interesting. VoltDB is in that space but in seven years has never encountered a single customer asking for those features."

A Decade Later: Friend or Foe?

At the First International Conference on Expert Database Systems, Kiawah Island, South Carolina [van de Riet 1986], I debated with Mike on the topic "Are Data Models Dead?" I do not recall the content nor the tone, which must have appeared confrontational because I do recall a look of utter surprise from Larry Kerschberg, the program committee chair, as Mike and I hugged off stage. Mike had arrived just before the debate, so we had not yet greeted each other. When it matters, Mike speaks his mind pithier than most. His directness and honesty may seem confrontational to some. I have never seen such an intent; rather, he is getting to the heart of the matter quickly. That enriches the discussion for some and can end it for others.

My first meeting with Mike over 40 years ago was memorable. There were others at the strategy dinner, but I do not recall them. Mike was quiet, calm, succinct, scary smart, and contrarian. He was a Turing laureate in the making. My impression was that he was the smartest man in the room. My impression, like data in Ingres, Postgres, and his many other DBMSs, persists.

PART IX

SEMINAL WORKS OF MICHAEL STONEBRAKER AND HIS COLLABORATORS

OLTP Through the Looking Glass, and What We Found There

Stavros Harizopoulos (HP Labs), **Daniel J. Abadi** (Yale University), **Samuel Madden** (MIT), **Michael Stonebraker** (MIT)

Abstract

Online Transaction Processing (OLTP) databases include a suite of features—disk-resident B-trees and heap files, locking-based concurrency control, support for multi-threading—that were optimized for computer technology of the late 1970's. Advances in modern processors, memories, and networks mean that today's computers are vastly different from those of 30 years ago, such that many OLTP databases will now fit in main memory, and most OLTP transactions can be processed in milliseconds or less. Yet database architecture has changed little.

Based on this observation, we look at some interesting variants of conventional database systems that one might build that exploit recent hardware trends, and speculate on their performance through a detailed instruction-level breakdown of the major components involved in a transaction processing database system

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

Original DOI: 10.1145/1376616.1376713

(Shore) running a subset of TPC-C. Rather than simply profiling Shore, we progressively modified it so that after every feature removal or optimization, we had a (faster) working system that fully ran our workload. Overall, we identify overheads and optimizations that explain a total difference of about a factor of 20x in raw performance. We also show that there is no single "high pole in the tent" in modern (memory resident) database systems, but that substantial time is spent in logging, latching, locking, B-tree, and buffer management operations.

Categories and Subject Descriptors

H.2.4 [Database Management]: Systems—*transaction processing; concurrency.*

General Terms

Measurement, Performance, Experimentation.

Keywords

Online Transaction Processing, OLTP, main memory transaction processing, DBMS architecture.

1 Introduction

Modern general purpose online transaction processing (OLTP) database systems include a standard suite of features: a collection of on-disk data structures for table storage, including heap files and B-trees, support for multiple concurrent queries via locking-based concurrency control, log-based recovery, and an efficient buffer manager. These features were developed to support transaction processing in the 1970's and 1980's, when an OLTP database was many times larger than the main memory, and when the computers that ran these databases cost hundreds of thousands to millions of dollars.

Today, the situation is quite different. First, modern processors are very fast, such that the computation time for many OLTP-style transactions is measured in microseconds. For a few thousand dollars, a system with gigabytes of main memory can be purchased. Furthermore, it is not uncommon for institutions to own networked clusters of many such workstations, with aggregate memory measured in hundreds of gigabytes—sufficient to keep many OLTP databases in RAM.

Second, the rise of the Internet, as well as the variety of data intensive applications in use in a number of domains, has led to a rising interest in database-like applications without the full suite of standard database features. Operating systems and networking conferences are now full of proposals for "database-like" storage systems with varying forms of consistency, reliability, concurrency, replication, and queryability [DG04, CDG+06, GBH+00, SMK+01].

This rising demand for database-like services, coupled with dramatic performance improvements and cost reduction in hardware, suggests a number of interesting alternative systems that one might build with a different set of features than those provided by standard OLTP engines.

1.1 Alternative DBMS Architectures

Obviously, optimizing OLTP systems for main memory is a good idea when a database fits in RAM. But a number of other database variants are possible; for example:

- **Logless databases.** A log-free database system might either not need recovery, or might perform recovery from other sites in a cluster (as was proposed in systems like Harp [LGG+91], Harbor [LM06], and C-Store [SAB+05]).
- **Single threaded databases.** Since multi-threading in OLTP databases was traditionally important for latency hiding in the face of slow disk writes, it is much less important in a memory resident system. A single-threaded implementation may be sufficient in some cases, particularly if it provides good performance. Though a way to take advantage of multiple processor cores on the same hardware is needed, recent advances in virtual machine technology provide a way to make these cores look like distinct processing nodes without imposing massive performance overheads [BDR97], which may make such designs feasible.
- **Transaction-less databases.** Transactional support is not needed in many systems. In particular, in distributed Internet applications, eventual consistency is often favored over transactional consistency [Bre00, DHJ+07]. In other cases, lightweight forms of transactions, for example, where all reads are required to be done before any writes, may be acceptable [AMS+07, SMA+07].

In fact, there have been several proposals from inside the database community to build database systems with some or all of the above characteristics [WSA97, SMA+07]. An open question, however, is how well these different configurations would perform if they were actually built. This is the central question of this paper.

1.2 Measuring the Overheads of OLTP

To understand this question, we took a modern open source database system (Shore—see http://www.cs.wisc.edu/shore/) and benchmarked it on a subset of the TPC-C benchmark. Our initial implementation—running on a modern desktop machine—ran about 640 transactions per second (TPS). We then modified it by removing different features from the engine one at a time, producing new benchmarks each step of the way, until we were left with a tiny kernel of query processing code that could process 12700 TPS. This kernel is a single-threaded, lock-free, main memory database system without recovery. During this decomposition, we identified four major components whose removal substantially improved the throughput of the system:

Logging. Assembling log records and tracking down all changes in database structures slows performance. Logging may not be necessary if recoverability is not a requirement or if recoverability is provided through other means (e.g., other sites on the network).

Locking. Traditional two-phase locking poses a sizeable overhead since all accesses to database structures are governed by a separate entity, the Lock Manager.

Latching. In a multi-threaded database, many data structures have to be latched before they can be accessed. Removing this feature and going to a single-threaded approach has a noticeable performance impact.

Buffer management. A main memory database system does not need to access pages through a buffer pool, eliminating a level of indirection on every record access.

1.3 Results

Figure 1 shows how each of these modifications affected the bottom line performance (in terms of CPU instructions per TPC-C New Order transaction) of Shore. We can see that each of these subsystems by itself accounts for between about 10% and 35% of the total runtime (1.73 million instructions, represented by the total height of the figure). Here, “hand coded optimizations” represents a collection of optimizations we made to the code, which primarily improved the performance of the B-tree package. The actual instructions to process the query, labelled “useful work” (measured through a minimal implementation we built on top of a hand-coded main-memory B-tree package) is only about 1/60th of that. The white box below “buffer manager” represents our version of Shore after we had removed everything from it—Shore still runs the transactions, but it uses about 1/15th of the

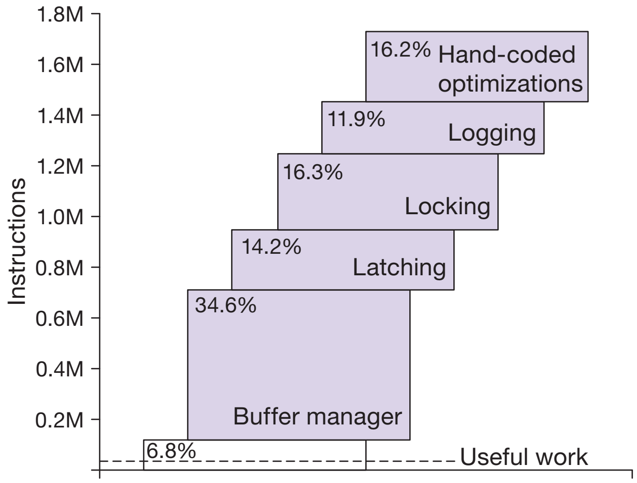


Figure 1 Breakdown of instruction count for various DBMS components for the New Order transaction from TPC-C. The top of the bar-graph is the original Shore performance with a main memory resident database and no thread contention. The bottom dashed line is the useful work, measured by executing the transaction on a no-overhead kernel.

instructions of the original system, or about 4 times the number of instructions in the useful work. The additional overheads in our implementation are due to call-stack depth in Shore and the fact that we could not completely strip out all references to transactions and the buffer manager.

1.4 Contributions and Paper Organization

The major contributions of this paper are to 1) dissect where time goes inside of a modern database system, 2) to carefully measure the performance of various stripped down variants of a modern database system, and 3) to use these measurements to speculate on the performance of different data management systems—for example, systems without transactions or logs—that one could build.

The remainder of this paper is organized as follows. In Section 2 we discuss OLTP features that may soon become (or are already becoming) obsolete. In Section 3 we review the Shore DBMS, as it was the starting point of our exploration, and describe the decomposition we performed. Section 4 contains our experimentation with Shore. Then, in Section 5, we use our measurements to discuss implications on future OLTP engines and speculate on the performance of some hypothetical data management systems. We present additional related work in Section 6 and conclude in Section 7.

2 Trends in OLTP

As mentioned in the introduction, most popular relational RDBMSs trace their roots to systems developed in the 1970's, and include features like disk-based indexing and heap files, log-based transactions, and locking-based concurrency control. However, 30 years have passed since these architectural decisions were made. At the present time, the computing world is quite different from when these traditional systems were designed; the purpose of this section is to explore the impact of these differences. We made a similar set of observations in [SMA+07].

2.1 Cluster Computing

Most current generation RDBMSs were originally written for shared memory multiprocessors in the 1970's. Many vendors added support for shared disk architectures in the 1980's. The last two decades have seen the advent of Gamma-style shared nothing databases [DGS+90] and the rise of clusters of commodity PCs for many large scale computing tasks. Any future database system must be designed from the ground up to run on such clusters.

2.2 Memory Resident Databases

Given the dramatic increase in RAM sizes over the past several decades, there is every reason to believe that many OLTP systems already fit or will soon fit into main memory, especially the aggregate main memory of a large cluster. This is largely because the sizes of most OTLP systems are not growing as dramatically as RAM capacity, as the number of customers, products, and other real world entities they record information about does not scale with Moore's law. Given this observation, it makes sense for database vendors to create systems that optimize for the common case of a memory resident system. In such systems, optimized indices [RR99, RR00] as well as eschewing disk-optimized tuple formats and page layouts (or lack thereof) [GS92] are important to consider.

2.3 Single Threading in OLTP Systems

All modern databases include extensive support for multi-threading, including a collection of transactional concurrency control protocols as well as extensive infiltration of their code with latching commands to support multiple threads accessing shared structures like buffer pools and index pages. The traditional motivations for multi-threading are to allow transaction processing to occur on behalf of one transaction while another waits for data to come from disk, and to prevent long-running transactions from keeping short transactions from making progress.

We claim that neither of these motivations is valid any more. First, if databases are memory resident, then there are never any disk waits. Furthermore, production transaction systems do not include any user waits—transactions are executed almost exclusively through stored procedures. Second, OLTP workloads are very simple. A typical transaction consists of a few index lookups and updates, which, in a memory resident system, can be completed in hundreds of microseconds. Moreover, with the bifurcation of the modern database industry into a transaction processing and a warehousing market, long running (analytical) queries are now serviced by warehouses.

One concern is that multi-threading is needed to support machines with multiple processors. We believe, however, that this can be addressed by treating one physical node with multiple processors as multiple nodes in a shared-nothing cluster, perhaps managed by a virtual machine monitor that dynamically allocates resources between these logical nodes [BDR97].

Another concern is that networks will become the new disks, introducing latency into distributed transactions and requiring the re-introduction of transactions. This is certainly true in the general case, but for many transaction applications, it is possible to partition the workload to be "single-sited" [Hel07, SMA+07], such that all transactions can be run entirely on a single node in a cluster.

Hence, certain classes of database applications will not need support for multithreading; in such systems, legacy locking and latching code becomes unnecessary overhead.

2.4 High Availability vs. Logging

Production transaction processing systems require 24x7 availability. For this reason, most systems use some form of high availability, essentially using two (or more) times the hardware to ensure that there is an available standby in the event of a failure.

Recent papers [LM06] have shown that, at least for warehouse systems, it is possible to exploit these available standbys to facilitate recovery. In particular, rather than using a REDO log, recovery can be accomplished by copying missing state from other database replicas. In our previous work we have claimed that this can be done for transaction systems as well [SMA+07]. If this is in fact the case, then the recovery code in legacy databases becomes also unnecessary overhead.

2.5 Transaction Variants

Although many OLTP systems clearly require transactional semantics, there have recently been proposals—particularly in the Internet domain—for data management systems with relaxed consistency. Typically, what is desired is some form of eventual consistency [Bre00, DHJ+07] in the belief that availability is more important than transactional semantics. Databases for such environments are likely to need little of the machinery developed for transactions (e.g., logs, locks, two-phase commit, etc.).

Even if one requires some form of strict consistency, many slightly relaxed models are possible. For example, the widespread adoption of snapshot isolation (which is non-transactional) suggests that many users are willing to trade transactional semantics for performance (in this case, due to the elimination of read locks).

And finally, recent research has shown that there are limited forms of transactions that require substantially less machinery than standard database transactions. For example, if all transactions are "two-phase"—that is, they perform all of their reads before any of their writes and are guaranteed not to abort after completing their reads—then UNDO logging is not necessary [AMS+07, SMA+07].

2.6 Summary

As our references suggest, several research groups, including Amazon [DHJ+07], HP [AMS+07], NYU [WSA97], and MIT [SMA+07] have demonstrated interest in building systems that differ substantially from the classic OTLP design. In particular, the MIT H-Store [SMA+07] system demonstrates that removing all of the above features can yield a two-order-of-magnitude speedup in transaction throughput, suggesting that some of these databases variants are likely to provide remarkable performance. Hence, it would seem to behoove the traditional database vendors to consider producing products with some of these features explicitly disabled. With the goal of helping these implementers understand the performance impact of different variants they may consider building, we proceed with our detailed performance study of Shore and the variants of it we created.

3 Shore

Shore (Scalable Heterogeneous Object Repository) was developed at the University of Wisconsin in the early 1990's and was designed to be a typed, persistent object system borrowing from both file system and object-oriented database technologies [CDF+94]. It had a layered architecture that allowed users to choose the appropriate level of support for their application from several components. These layers (type system, unix compatibility, language heterogeneity) were provided on top of the Shore Storage Manager (SSM). The storage manager provided features that are found in all modern DBMS: full concurrency control and recovery (ACID transaction properties) with two-phase locking and write-ahead logging, along with a robust implementation of B-trees. Its basic design comes from ideas described in Gray's and Reuter's seminal book on transaction processing [GR93], with many algorithms implemented straight from the ARIES papers [MHL+92, Moh89, ML89].

Support for the project ended in the late 1990's, but continued for the Shore Storage Manager; as of 2007, SSM version 5.0 is available for Linux on Intel x86 processors. Throughout the paper we use "Shore" to refer to the Shore Storage Manager. Information and source code of Shore is available online.[1] In the rest of this section we discuss the key components of Shore, its code structure, the characteristics of Shore that affect end-to-end performance, along with our set of modifications and the effect of these modifications to the code line.

3.1 Shore Architecture

There are several features of Shore that we do not describe as they are not relevant to this paper. These include disk volume management (we pre-load the entire database in main memory), recovery (we do not examine application crashes), distributed transactions, and access methods other than B-trees (such as R-trees). The remaining features can be organized roughly into the components shown in Figure 2.

Shore is provided as a library; the user code (in our case, the implementation of the TPC-C benchmark) is linked against the library and must use the threads library that Shore also uses. Each transaction runs inside a Shore thread, accessing both local user-space variables and Shore-provided data structures and methods. The methods relevant to OLTP are those needed to create and populate a database file, load it into the buffer pool, begin, commit, or abort a transaction, and perform record-level operations such as fetch, update, create, and delete, along with the associated operations on primary and secondary B-tree indexes.

Inside the transaction body (enclosed by begin and commit statements) the application programmer uses Shore's methods to access the storage structures: the file and indexes, along with a directory to find them. All the storage structures use *slotted pages* to store information. Shore's methods run under the transaction manager which closely interacts with all other components. Accessing the storage structures involves calls to the Log Manager, the Lock Manager, and the Buffer Pool Manager. These invocations always happen through a concurrency control layer, which oversees shared and mutually exclusive accesses to the various resources. This is not a separate module; rather, throughout the code, all accesses to shared structures happen by acquiring a *latch*. Latches are similar to database locks (in that

[1] http:// www.cs.wisc.edu/shore/

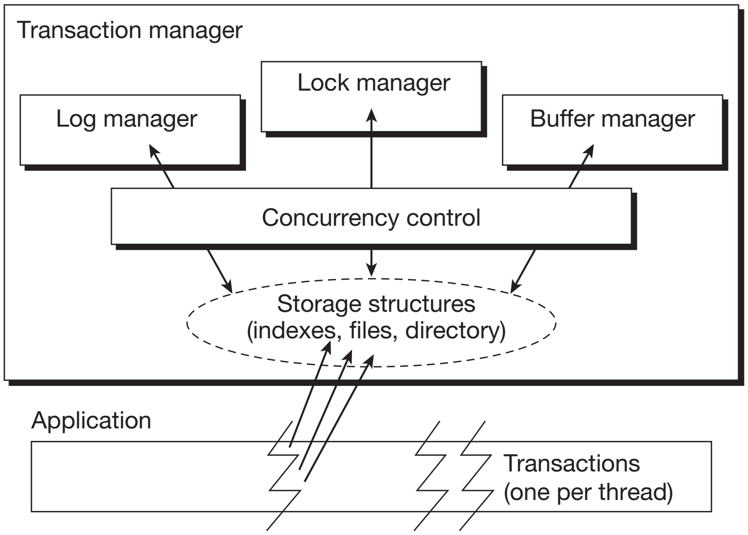


Figure 2 Basic components in Shore (see text for detailed description).

they can be shared or exclusive), but they are lightweight and come with no deadlock detection mechanisms. The application programmers need to ensure that latching will not lead to deadlock.

Next, we discuss the thread architecture and give more details on locking, logging, and the buffer pool management.

Thread support. Shore provides its own user-level, non-preemptive thread package that was derived from NewThreads (originally developed at the University of Washington), providing a portable OS interface API. The choice of the thread package had implications for the code design and behavior of Shore. Since threads are user-level, the application runs as a single process, multiplexing all Shore threads. Shore avoids blocking for I/O by spawning separate processes responsible for I/O devices (all processes communicate through shared memory). However, applications cannot take direct advantage of multicore (or SMP) systems, unless they are built as part of a distributed application; that, however, would add unnecessary overhead for multicore CPUs, when simple, non-user level threading would be sufficient.

Consequently, for the results reported throughout this paper, we use single-threaded operation. A system that uses multithreaded operation would consume a larger number of instructions and CPU cycles per transaction (since thread code would need to be executed in addition to transactional code). Since the primary goal of the paper is to look at the cost in CPU instructions of various database system components, the lack of a full multi-threading implementation in Shore only affects our results in that we begin at a lower starting point in total CPU instructions when we begin removing system components.

Locking and logging. Shore implements standard two-phase locking, with transactions having standard ACID properties. It supports hierarchical locking with the lock manager escalating up the hierarchy by default (record, page, store, volume). Each transaction keeps a list of the locks it holds, so that the locks can be logged when the transaction enters the prepared state and released at the end of the transaction. Shore also implements write ahead logging (WAL), which requires a close interaction between the log manager and the buffer manager. Before a page can be flushed from the buffer pool, the corresponding log record might have to be flushed. This also requires a close interaction between the transaction manager and the log manager. All three managers understand log sequence numbers (LSNs), which serve to identify and locate log records in the log, timestamp pages, identify the last update performed by a transaction, and find the last log record written by a transaction. Each page bears the LSN of the last update that affected that page. A page cannot be written to disk until the log record with that page's LSN has been written to stable storage.

Buffer Manager. The buffer manager is the means by which all other modules (except the log manager) read and write pages. A page is read by issuing a *fix* method call to the buffer manager. For a database that fits in main memory, the page is always found in the buffer pool (in the non-main memory case, if the requested page is not in the buffer pool, the thread gives up the CPU and waits for the process responsible for I/O to place the page in the buffer pool). The fix method updates the mapping between page IDs and buffer frames and usage statistics. To ensure consistency there is a latch to control access to the fix method. Reading a record (once a record ID has been found through an index lookup) involves

1. locking the record (and page, per hierarchical locking),
2. fixing the page in the buffer pool, and
3. computing the offset within the page of the record's tag.

Reading a record is performed by issuing a pin / unpin method call. Updates to records are accomplished by copying out part or all of the record from the buffer pool to the user's address space, performing the update there, and handing the new data to the storage manager.

Table 1 Possible set of optimizations for OLTP.

OLTP Properties and New Platforms	DBMS Modification
logless architectures	remove logging
partitioning, commutativity	remove locking (when applicable)
one transaction at a time	single thread, remove locking, remove latching
main memory resident	remove buffer manager, directory
transaction-less databases	avoid transaction bookkeeping

More details on the architecture of Shore can be found at the project's web site. Some additional mechanisms and features are also described in the following paragraphs, where we discuss our own modifications to Shore.

3.2 Removing Shore Components

Table 1 summarizes the properties and characteristics of modern OLTP systems (left column) that allow us to strip certain functionality from a DBMS (right column). We use these optimizations as a guideline for modifying Shore. Due to the tight integration of all managers in Shore, it was not possible to cleanly separate all components so that they could be removed in an arbitrary order. The next best thing was to remove features in an order dictated by the structure of the code, allowing for flexibility whenever possible. That order was the following:

1. Removing logging.
2. Removing locking OR latching.
3. Removing latching OR locking.
4. Removing code related to the buffer manager.

In addition, we found that the following optimizations could be performed at any point:

- Streamline and hardcode the B-tree key evaluation logic, as is presently done in most commercial systems.
- Accelerate directory lookups.
- Increase page size to avoid frequent allocations (subsumed by step 4 above).
- Remove transactional sessions (begin, commit, various checks).

Our approach to implementing the above-mentioned actions is described next. In general, to remove a certain component from the system, we either add a few if-statements to avoid executing code belonging to that component, or, if we find that if-statements add a measurable overhead, we rewrite entire methods to avoid invoking that component altogether.

Remove logging. Removing logging consists of three steps. The first is to avoid generating I/O requests along with the time associated to perform these requests (later, in Figure 7, we label this modification "*disk log*"). We achieve this by allowing group commit and then increasing the log buffer size so that it is not flushed to disk during our experiments. Then, we comment out all functions that are used to prepare and write log records (labeled "*main log*" in Figure 7). The last step was to add if-statements throughout the code to avoid processing Log Sequence Numbers (labeled "*LSN*" in Figure 7).

Remove locking (interchangeable with removing latching). In our experiments we found that we could safely interchange the order of removing locking and latching (once logging was already removed). Since latching is also performed inside locking, removing one also reduces the overhead of the other. To remove locking we first changed all Lock Manager methods to return immediately, as if the lock request was successful and all checks for locks were satisfied. Then, we modified methods related to pinning records, looking them up in a directory, and accessing them through a B-tree index. In each case, we eliminated code paths related to ungranted lock requests.

Remove latching (interchangeable with removing locking). Removing latching was similar to removing locking; we first changed all mutex requests to be immediately satisfied. We then added if-statements throughout the code to avoid requests for latches. We had to replace B-tree methods with ones that did not use latches, since adding if-statements would have increased overhead significantly because of the tight integration of latch code in the B-tree methods.

Remove buffer manager calls. The most widespread modification we performed was to remove the buffer manager methods, once we knew that logging, locking, and latching were already disabled. To create new records, we abandoned Shore's page allocation mechanism and instead used the standard malloc library. We call malloc for each new record (records no longer reside in pages) and use pointers for future accesses. Memory allocation can potentially be done more efficiently, especially when one knows in advance the sizes of the allocated objects. However, further optimization of main memory allocation is an incremental improvement relative to the overheads we are studying, and is left for future work. We were not able to completely remove the page interface to buffer frames, since its removal would invalidate most of the remaining Shore code. Instead, we accelerated the mappings between pages and buffer frames, reducing the overhead to a minimum. Similarly, pinning and updating a record will still go through a buffer manager layer, albeit a very thin one (we label this set of modifications "*page access*" in Figure 7).

Miscellaneous optimizations. There were four optimizations we made that can be invoked at any point during the process of removing the above-mentioned components. These were the following. (1) Accelerating the B-tree code by hand-coding node searches to optimize for the common case that keys are uncompressed integers (labeled "*Btree keys*" in Figures 5-8). (2) Accelerating directory lookups by using a single cache for all transactions (labeled "*dir lookup*" in Figure 7). (3) Increasing the page size from the default size of 8KB to 32KB, the maximum allowable in Shore (labeled "*small page*" in Figure 7). Larger pages, although not suitable for disk-based OLTP, can help in a main-memory resident database by reducing the number of levels in a B-tree (due to the larger node size), and result in less frequent page allocations for newly created records. An alternative would be to decrease the size of a B-tree node to the size of a cache line as proposed in [RR99], but this would have required removing the association between a B-tree node and a Shore page, or reducing a Shore page below 1KB (which Shore does not allow). (4) Removing the overhead of setting up and terminating a session for each transaction, along with the associated monitoring of running transactions, by consolidating transactions into a single session (labeled "*Xactions*" in Figure 7).

Our full set of changes/optimizations to Shore, along with the benchmark suite and documentation on how to run the experiments are available online.[2] Next, we move to the performance section of the paper.

4 Performance Study

The section is organized as follows. First we describe our variant of the TPC-C benchmark that we used (Section 4.1). In Section 4.2 we provide details of the hardware platform, the experimental setup, and the tools we used for collecting

[2] http://db.cs.yale.edu/hstore/

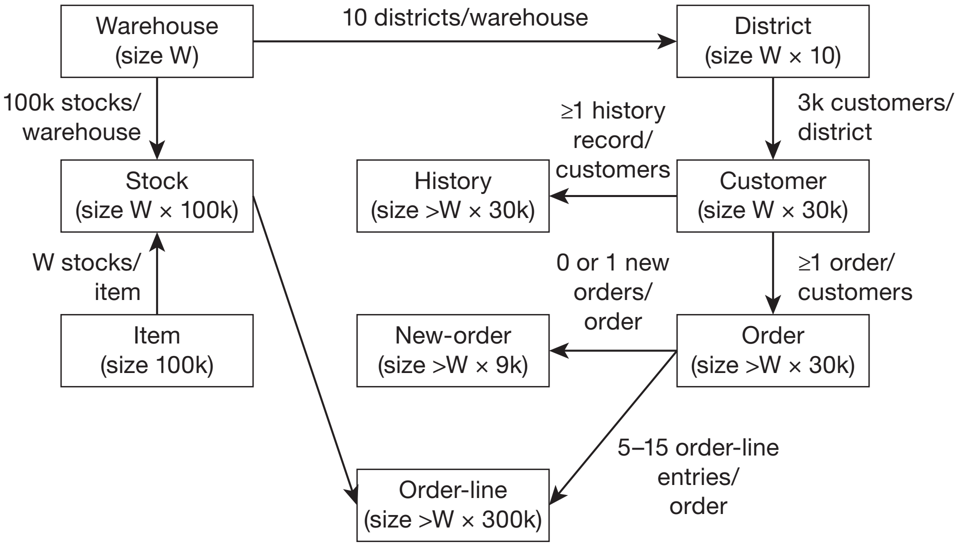


Figure 3 TPC-C Schema

the performance numbers. Section 4.3 presents a series of results, detailing Shore performance as we progressively apply optimizations and remove components.

4.1 OLTP Workload

Our benchmark is derived from TPC-C [TPCC], which models a wholesale parts supplier operating out of a number of warehouses and their associated sales districts. TPC-C is designed to represent any industry that must manage, sell, or distribute a product or service. It is designed to scale as the supplier expands and new warehouses are created. The scaling requirement is that each warehouse must supply 10 sales districts, and each district must serve 3000 customers. The database schema along with the scaling requirements (as a function of the number of warehouses W) is shown in Figure 3. The database size for one warehouse is approximately 100 MB (we experiment with five warehouses for a total size of 500MB).

TPC-C involves a mix of five concurrent transactions of different types and complexity. These transactions include entering orders (the New Order transaction), recording payments (Payment), delivering orders, checking the status of orders, and monitoring the level of stock at the warehouses. TPC-C also specifies that about 90% of the time the first two transactions are executed. For the purposes

New Order

```
begin
for loop(10)
.....Btree lookup(I), pin
Btree lookup(D), pin
Btree lookup (W), pin
Btree lookup (C), pin
update rec (D)
for loop (10)
.....Btree lookup(S), pin
.....update rec (S)
.....create rec (O-L)
.....insert Btree (O-L)
create rec (O)
insert Btree (O)
create rec (N-O)
insert Btree (N-O)
insert Btree 2ndary(N-O)
commit
```

Payment

```
begin
Btree lookup(D), pin
Btree lookup (W), pin
Btree lookup (C), pin
update rec (C)
update rec (D)
update rec (W)
create rec (H)
commit
```

Figure 4 Calls to Shore's methods for New Order and Payment transactions.

of the paper, and for better understanding the effect of our interventions, we experimented with a mix of only the first two transactions. Their code structure (calls to Shore) is shown in Figure 4. We made the following small changes to the original specifications, to achieve repeatability in the experiments:

New Order. Each New Order transaction places an order for 5-15 items, with 90% of all orders supplied in full by stocks from the customer's "home" warehouse (10% need to access stock belonging to a remote warehouse), and with 1% of the provided items being an invalid one (it is not found in the B-tree). To avoid variation in the results we set the number of items to 10, and always serve orders from a local warehouse. These two changes do not affect the throughput. The code in Figure 4 shows the two-phase optimization mentioned in Section 2.5, which allows us to avoid aborting a transaction; we read all items at the beginning, and if we find an invalid one we abort without redoing changes in the database.

Payment. This is a lightweight transaction; it updates the customer's balance and warehouse/district sales fields, and generates a history record. Again, there is a choice of home and remote warehouse which we always set to the home one.

Another randomly set input is whether a customer is looked up by name or ID, and we always use ID.

4.2 Setup and Measurement Methodology

All experiments are performed on a single-core Pentium 4 3.2GHz, with 1MB L2 cache, hyperthreading disabled, 1GB RAM, running Linux 2.6. We compiled with gcc version 3.4.4 and O2 optimizations. We use the standard linux utility iostat to monitor disk activity and verify in the main memory-resident experiments there is no generated disk traffic. In all experiments we pre-load the entire database into the main memory. Then we run a large number of transactions (40,000). Throughput is measured directly by dividing wall clock time by the number of completed transactions.

For detailed instruction and cycle counts we instrument the benchmark application code with calls to the PAPI library [MBD+99] http://icl.cs.utk.edu/papi/, which provides access to the CPU performance counters. Since we make a call to PAPI after every call to Shore, we have to compensate for the cost of PAPI calls when reporting the final numbers. These had an instruction count of 535-537 and were taking between 1350 and 1500 cycles in our machine. We measure each call to Shore for all 40,000 transactions and report the average numbers.

Most of the graphs reported in the paper are based on CPU instruction counts (as measured through the CPU performance counters) and not wall clock time. The reason is that instruction counts are representative of the total run-time code path length, and they are deterministic. Equal instruction counts among different components can of course result in different wall clock execution times (CPU cycles), because of different microarchitectural behavior (cache misses, TLB misses, etc.). In Section 4.3.4 we compare instruction counts to CPU cycles, illustrating the components where there is high micro-architectural efficiency that can be attributed to issues like few L2 cache misses and good instruction-level parallelism.

Cycle count, however, is susceptible to various parameters, ranging from CPU characteristics, such as cache size/associativity, branch predictors, TLB operation, to run-time variables such as concurrent processes. Therefore it should be treated as indicative of relative time breakdown. We do not expand on the issue of CPU cache performance in this paper, as our focus is to identify the set of DBMS components to remove that can produce up to two orders of magnitude better performance for certain classes of OLTP workloads. More information on the micro-architectural behavior of database workloads can be found elsewhere [Ail04].

Next, we begin the presentation of our results.

4.3 Experimental Results

In all experiments, our baseline Shore platform is a memory-resident database that is never flushed to disk (the only disk I/O that might be performed is from the Log Manager). There is only a single thread executing one transaction at a time. Masking I/O (in the case of disk-based logging) is not a concern as it only adds to overall response time and not to the instructions or cycles that the transaction has actually run.

We placed 11 different switches in Shore to allow us to remove functionality (or perform optimizations), which, during the presentation of the results, we organize into six components. For a list of the 11 switches (and the corresponding components) and the order we apply them, see Figure 7. These switches were described in more detail in Section 3.2 above. The last switch is to bypass Shore completely and run our own, minimal-overhead kernel, which we call "optimal" in our results. This kernel is basically a memory-resident, hand-built B-tree package with no additional transaction or query processing functionality.

4.3.1 Effect on Throughput

After all of these deletions and optimizations, Shore is left with a code residue, which is all CPU cycles since there is no I/O whatsoever; specifically, an average of about 80 microseconds per transaction (for a 50-50 mix of New Order and Payment transactions), or about 12,700 transactions per second.

In comparison, the useful work in our optimal system was about 22 microseconds per transaction, or about 46,500 transactions per second. The main causes of this difference are a deeper call stack depth in our kernel, and our inability to remove some of the transaction set up and buffer pool calls without breaking Shore. As a point of reference, "out of the box" Shore, with logging enabled but with the database cached in main memory, provides about 640 transactions per second (1.6 milliseconds per transaction), whereas Shore running in main memory, but without log flushing provides about 1,700 transactions per second, or about 588 microseconds per transaction. Hence, our modifications yield a factor of 20 improvement in overall throughput.

Given these basic throughput measurements, we now give detailed instruction breakdowns for the two transactions of our benchmark. Recall that the instruction and cycle breakdowns in the following sections do not include any impact of disk operations, whereas the throughput numbers for baseline Shore do include some log write operations.

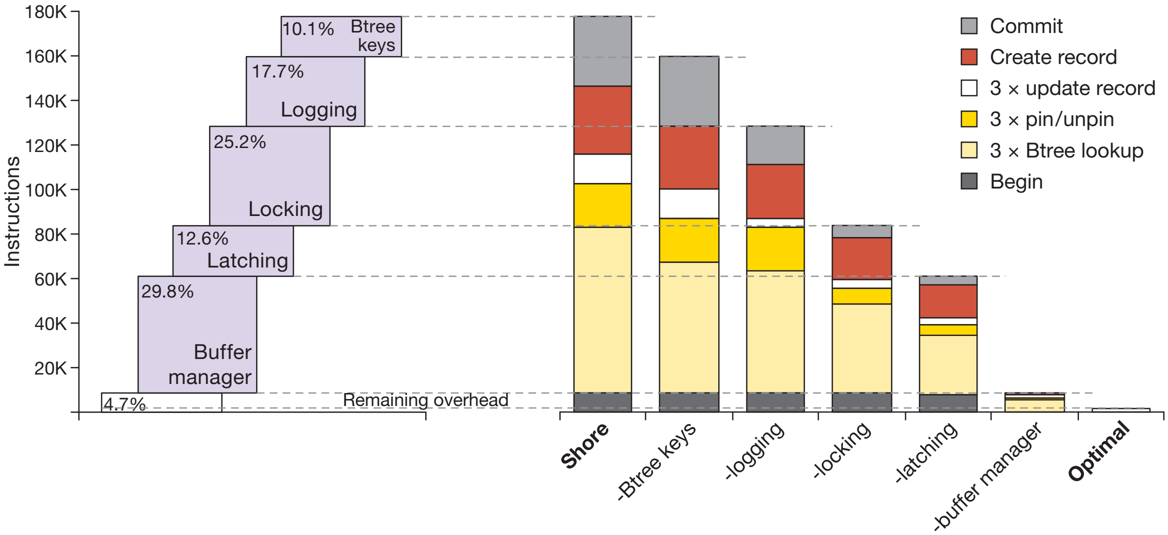


Figure 5 Detailed instruction count breakdown for Payment transaction.

4.3.2 Payment

Figure 5 (left side) shows the reductions in the instruction count of the Payment transaction as we optimized B-tree key evaluations and removed logging, locking, latching, and buffer manager functionality. The right part of the figure shows, for each feature removal we perform, its effect on the number of instructions spent in various portions of the transaction's execution. For the Payment transaction, these portions include a begin call, three B-tree lookups followed by three pin/unpin operations, followed by three updates (through the B-tree), one record creation and a commit call. The height of each bar is always the total number of instructions executed. The right-most bar is the performance of our minimal-overhead kernel.

Our B-tree key evaluation optimizations are reportedly standard practice in high-performance DBMS architectures, so we perform them first because any system should be able to do this. Removing logging affects mainly commits and updates, as those are the portions of the code that write log records, and to a lesser degree B-tree and directory lookups. These modifications remove about 18% of the total instruction count.

Locking takes the second most instructions, accounting for about 25% of the total count. Removing it affects all of the code, but is especially important in the pin/unpin operations, the lookups, and commits, which was expected as these are the operations that must acquire or release locks (the transaction already has locks on the updated records when the updates are performed).

Latching accounts for about 13% of the instructions, and is primarily important in the create record and B-tree lookup portions of the transaction. This is because the buffer pool (used in create) and B-trees are the primary shared data structures that must be protected with latches.

Finally, our buffer manager modifications account for about 30% of the total instruction count. Recall that with this set of modifications, new records are allocated directly with malloc, and page lookups no longer have to go through the buffer pool in most cases. This makes record allocation essentially free, and substantially improves the performance of other components that perform frequent lookups, like B-tree lookup and update.

At this point, the remaining kernel requires about 5% (for a 20x performance gain!) of the total initial instruction count, and is about 6 times the total instructions of our "optimal" system. This analysis leads to two observations: first, all six of the major components are significant, each accounting for 18 thousand or more instructions of the initial 180 thousand. Second, until all of our optimizations are applied, the reduction in instruction count is not dramatic: before our last step of removing the buffer manager, the remaining components used about a factor of three fewer instructions than the baseline system (versus a factor of 20 when the buffer manager is removed).

4.3.3 New Order

A similar breakdown of the instruction count in the New Order transaction is shown in Figure 6; Figure 7 shows a detailed accounting of all 11 modifications and optimizations we performed. This transaction uses about 10 times as many instructions as the Payment transaction, requiring 13 B-tree inserts, 12 record creation operations, 11 updates, 23 pin/unpin operations, and 23 B-tree lookups. The main differences in the allocation of instructions to major optimizations between New Order and Payment are in B-tree key code, logging, and locking. Since New Order adds B-tree insertions in the mix of operations, there is more relative benefit to be had by optimizing the key evaluation code (about 16%). Logging and locking now only account for about 12% and 16% of the total instructions; this is largely because the total fraction of time spent in operations where logging and locking perform a lot of work is much smaller in this case.

The buffer manager optimizations still represent the most significant win here, again because we are able to bypass the high overhead of record creation. Looking at the detailed breakdown in Figure 7 for the buffer manager optimization

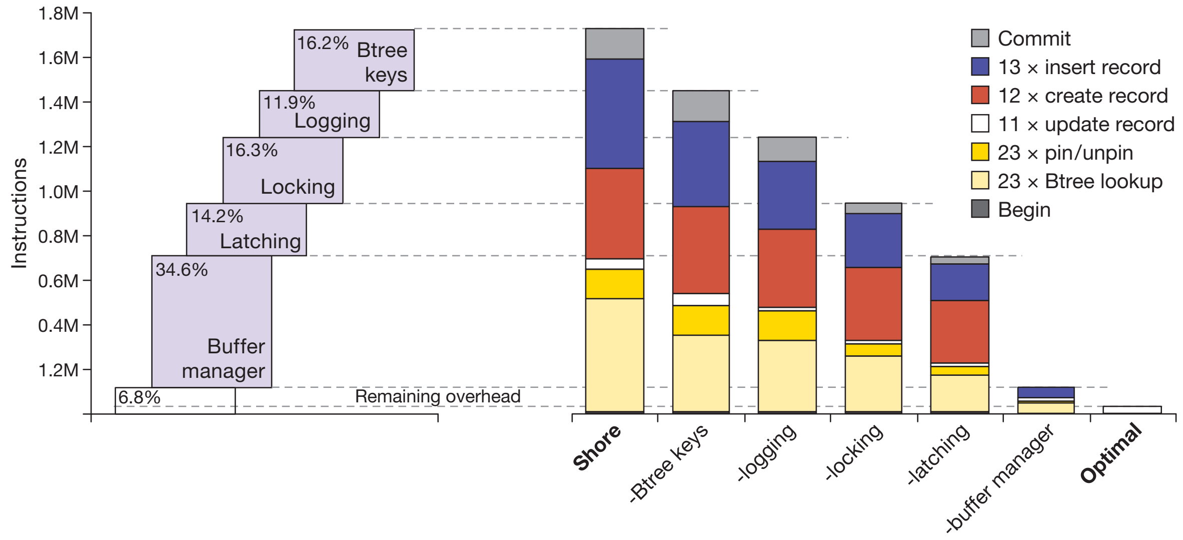


Figure 6 Detailed instruction count breakdown for New Order transaction.

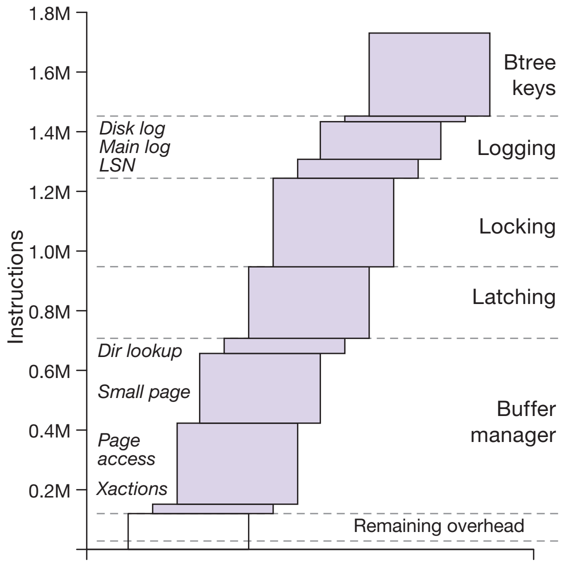


Figure 7 Expanding breakdown for New Order (see Section 3.2 for the labels on the left column).

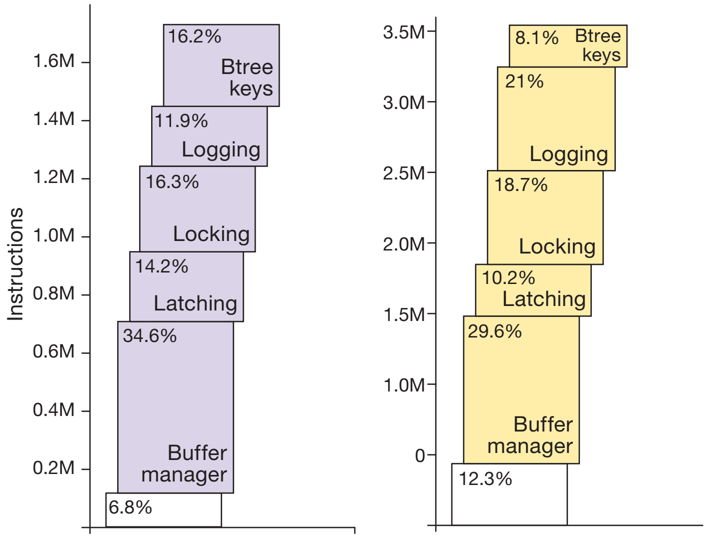


Figure 8 Instructions (left) vs. Cycles (right) for New Order.

reveals something surprising: changing from 8K to 32K pages (labelled "small page") provides almost a 14% reduction in the total instruction count. This simple optimization—which serves to reduce the frequency of page allocations and decrease B-tree depth—offers a sizeable gain.

4.3.4 Instructions vs. Cycles

Having looked at the detailed breakdown of instruction counts in the Payment and New Order transactions, we now compare the fraction of time (cycles) spent in each phase of the New Order transaction to the fraction of instructions used in each phase. The results are shown in Figure 8. As we noted earlier, we do not expect these two fractions to be identical for a given phase, because cache misses and pipeline stalls (typically due to branches) can cause some instructions to take more cycles than others. For example, B-tree optimizations reduce cycles less than they reduce instructions, because the Shore B-tree code overhead we remove is mainly offset calculations with few cache misses. Conversely, our residual "kernel" uses a larger fraction of cycles than it does instructions, because it is branch-heavy, consisting mostly of function calls. Similarly, logging uses significantly more cycles because it touches a lot of memory creating and writing log records (disk I/O time is not included in either graph). Finally, locking and the buffer manager consume about the same percentage of cycles as they do instructions.

5 Implications for Future OLTP Engines

Given the performance results in the previous section, we revisit our discussion of future OLTP designs from Section 2. Before going into the detailed implications of our results for the design of various database subsystems, we make two high level observations from our numbers:

- First, the benefit of stripping out any one of the components of the system has a relatively small benefit on overall performance. For example, our main memory optimizations improved the performance of Shore by about 30%, which is significant but unlikely to motivate the major database vendors to re-engineer their systems. Similar gains would be obtained by eliminating just latching or switching to a single-threaded, one-transaction-at-a-time approach.
- The most significant gains are to be had when multiple optimizations are applied. A fully stripped down system provides a factor of twenty or more performance gain over out-of-the-box Shore, which is truly significant. Note that such a system can still provide transactional semantics, if only one transaction is run at a time, all transactions are two phase, and recovery is implemented by copying state from other nodes in the network. Such a system is very, very different from what any of the vendors currently offers, however.

5.1 Concurrency Control

Our experiments showed a significant contribution (about 19% of cycles) of dynamic locking to total overhead. This suggests that there is a large gain to be had by identifying scenarios, such as application commutativity, or transaction-at-a-time processing, that allow concurrency control to be turned off. However, there are many DBMS applications which are not sufficiently well-behaved or where running only one transaction at a time per site will not work. In such cases, there is an interesting question as to what concurrency control protocol is best. Twenty years ago, various researchers [KR81, ACL87] performed exhaustive simulations that clearly showed the superiority of dynamic locking relative to other concurrency control techniques. However, this work assumed a disk-based load with disk stalls, which obviously impacts the results significantly. It would be highly desirable to redo these sorts of simulation studies with a main memory workload. We strongly suspect that some sort of optimistic concurrency control would be the prevailing option.

5.2 Multi-core Support

Given the increasing prevalence of many-core computers, an interesting question is how future OLTP engines should deal with multiple cores. One option is to run multiple transactions concurrently on separate cores within a single site (as it is done today); of course, such parallelism requires latching and implies a number of resource allocation issues. Our experiments show that although the performance overhead of latching is not particularly high (10% of cycles in the dominant transaction, New Order), it still remains an obstacle in achieving significant performance improvements in OLTP. As technologies (such as transactional memory [HM93]) for efficiently running highly concurrent programs on multicore machines mature and find their way into products, it will be very interesting to revisit new implementations for latching and reassess the overhead of multithreading in OLTP.

A second option is to use virtualization, either at the operating system or DBMS level, to make it appear that each core is a single-threaded machine. It is unclear what the performance implications of that approach would be, warranting a careful study of such virtualization. A third option, complementary to the other two, is to attempt to exploit intra-query parallelism, which may be feasible even if the system only runs one transaction at a time. However, the amount of intra-query parallelism available in a typical OLTP transaction is likely to be limited.

5.3 Replication Management

The traditional database wisdom is to support replication through a log-shipping based active-passive scheme; namely, every object has an “active” primary copy, to which all updates are first directed. The log of changes is then spooled over the network to one or more “passive” backup sites. Recovery logic rolls the remote database forward from the log. This scheme has several disadvantages. First, unless a form of two-phase commit is used, the remote copies are not transactionally consistent with the primary. Hence, reads cannot be directed to replicas if transaction-consistent reads are required. If reads are directed to replicas, nothing can be said about the accuracy of the answers. A second disadvantage is that failover is not instantaneous. Hence, the stall during failures is longer than it needs to be. Third, it requires the availability of a log; our experiments show that maintaining a log takes about 20% of total cycles. Hence, we believe it is interesting to consider alternatives to active-passive replication, such as an active-active approach.

The main reason that active-passive replication with log shipping has been used in the past is that the cost of rolling the log forward has been assumed to be far lower than the cost of performing the transaction logic on the replica.

However, in a main memory DBMS, the cost of a transaction is typically less than 1 msec, requiring so few cycles that it is likely not much slower than playing back a log. In this case, an alternate active-active architecture appears to make sense. In this case, all replicas are "active" and the transaction is performed synchronously on all replicas. The advantage of this approach is nearly instantaneous failover and there is no requirement that updates be directed to a primary copy first. Of course, in such a scenario, two-phase commit will introduce substantial additional latency, suggesting that techniques to avoid it are needed—perhaps by performing transactions in timestamp order.

5.4 Weak Consistency

Most large web-oriented OLTP shops insist on replicas, usually over a WAN, to achieve high availability and disaster recovery. However, seemingly nobody is willing to pay for transactional consistency over a WAN. As noted in Section 2, the common refrain in web applications is "eventual consistency" [Bre00, DHJ+07]. Typically, proponents of such approach advocate resolving inconsistencies through non-technical means; for example, it is cheaper to give a credit to a customer who complains than to ensure 100% consistency. In other words, the replicas eventually become consistent, presumably if the system is quiesced.

It should be clear that eventual consistency is impossible without transaction consistency under a general workload. For example, suppose transaction 1 commits at site 1 and aborts or is lost at site 2. Transaction 2 reads the result of transaction 1 and writes into the database, causing the inconsistency to propagate and pollute the system. That said, clearly, there must be workloads where eventual consistency is achievable, and it would be an interesting exercise to look for them, since, as noted above, our results suggest that removing transactional support—locking and logging—from a main memory system could yield a very high performance database.

5.5 Cache-conscious B-trees

In our study we did not convert Shore B-trees to a "cache-conscious" format [RR99, RR00]. Such an alteration, at least on a system without all of the other optimizations we present, would have only a modest impact. Cache-conscious research on B-trees targets cache misses that result from accessing key values stored in the B-tree nodes. Our optimizations removed between 80% to 88% of the time spent in B-tree operations, without changing the key access pattern. Switching from a stripped-down Shore to our minimal-overhead kernel—which still accesses the same data—removed three quarters of the remaining time. In other words, it appears to be more important to optimize other components, such as concurrency control and recovery, than to optimize data structures. However, once we strip a system down to a very basic kernel, cache misses in the B-tree code may well be the new bottleneck. In fact, it may be the case that other indexing structures, such as hash tables, perform better in this new environment. Again, these conjectures should be carefully tested.

6 Related Work

There have been several studies of performance bottlenecks in modern database systems. [BMK99] and [ADH+99] show the increasing contribution of main memory data stalls to database performance. [MSA+04] breaks down bottlenecks due to contention for various resources (such as locks, I/O synchronization, or CPU) from the client's point of view (which includes perceived latency due to I/O stalls and preemptive scheduling of other concurrent queries). Unlike the work presented here, these papers analyze complete databases and do not analyze performance per database component. Benchmarking studies such as TPC-B [Ano85] in the OLTP space and the Wisconsin Benchmark [BDT83] in general SQL processing, also characterize the performance of complete databases and not that of individual OLTP components.

Additionally, there has been a large amount of work on main memory databases. Work on main memory indexing structures has included AVL trees [AHU74] and T-trees [LC86]. Other techniques for main memory applicability appear in [BHT87]. Complete systems include TimesTen [Tim07], DataBlitz [BBK+98], and MARS [Eic87]. A survey of this area appears in [GS92]. However, none of this work attempts to isolate the components of overhead, which is the major contribution of this paper.

7 Conclusions

We performed a performance study of Shore motivated by our desire to understand where time is spent in modern database systems, and to help understand what the potential performance of several recently proposed alternative database architectures might be. By stripping out components of Shore, we were able to produce a system that could run our modified TPC-C benchmark about 20 times faster than the original system (albeit with substantially reduced functionality!). We found that buffer management and locking operations are the most significant contributors to system overhead, but that logging and latching operations are also significant.

Based on these results, we make several interesting observations. First, unless one strips out all of these components, the performance of a main memory-optimized database (or a database without transactions, or one without logging) is unlikely to be much better than a conventional database where most of the data fit into RAM. Second, when one does produce a fully stripped down system—e.g., that is single threaded, implements recovery via copying state from other nodes in the network, fits in memory, and uses reduced functionality transactions—the performance is orders of magnitude better than an unmodified system. This suggests that recent proposals for stripped down systems [WSA97, SMA+07] may be on to something.

8 Acknowledgments

We thank the SIGMOD reviewers for their helpful comments. This work was partially supported by the National Science Foundation under Grants 0704424 and 0325525.

9 Repeatability Assessment

All the results in this paper were verified by the SIGMOD repeatability committee. Code and/or data used in the paper are available at http://www.sigmod.org/codearchive/sigmod2008/

References

[ACL87] Agrawal, R., Carey, M. J., and Livny, M. "Concurrency control performance modeling: alternatives and implications." *ACM Trans. Database Syst. 12(4)*, Dec. 1987.

[AMS+07] Aguilera, M., Merchant, A., Shah, M., Veitch, A. C., and Karamanolis, C. T. "Sinfonia: a new paradigm for building scalable distributed systems." In *Proc. SOSP*, 2007.

[AHU74] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. "The Design and Analysis of Computer Algorithms." Addison-Wesley Publishing Company, 1974.

[ADH+99] Ailamaki, A., DeWitt, D. J., Hill, M. .D., and Wood, D. A. "DBMSs on a Modern Processor: Where Does Time Go?" In *Proc. VLDB*, 1999, 266-277.

[Ail04] Ailamaki, A. "Database Architecture for New Hardware." Tutorial. In *Proc. VLDB*, 2004.

[Ano85] Anon et al. "A Measure of Transaction Processing Power." In *Datamation*, February 1985.

[BBK+98] Baulier, J. D., Bohannon, P., Khivesara, A., et al. "The DataBlitz Main-Memory Storage Manager: Architecture, Performance, and Experience." In *The VLDB Journal*, 1998.

[BDT83] Bitton, D., DeWitt, D. J., and Turbyfill, C. “Benchmarking Database Systems, a Systematic Approach.” In *Proc. VLDB*, 1983.

[BHT87] Bitton, D., Hanrahan, M., and Turbyfill, C. “Performance of Complex Queries in Main Memory Database Systems.” In *Proc. ICDE*, 1987.

[BMK99] Boncz, P. A., Manegold, S., and Kersten, M. L. “Database Architecture Optimized for the New Bottleneck: Memory Access.” In *Proc. VLDB*, 1999.

[Bre00] Brewer, E. A. “Towards robust distributed systems (abstract).” In *Proc. PODC*, 2000.

[BDR97] Bugnion, E., Devine, S., and Rosenblum, M. “Disco: running commodity operating systems on scalable multiprocessors.” In *Proc. SOSP*, 1997.

[CDF+94] Carey, M. J., DeWitt, D. J., Franklin, M. J. et al. “Shoring up persistent applications.” In *Proc. SIGMOD*, 1994.

[CDG+06] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. “Bigtable: A Distributed Storage System for Structured Data.” In *Proc. OSDI*, 2006.

[DG04] Dean, J. and Ghemawat, S. “MapReduce: Simplified Data Processing on Large Clusters.” In *Proc. OSDI*, 2004.

[DHJ+07] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. “Dynamo: amazon’s highly available key-value store.” In *Proc. SOSP*, 2007.

[DGS+90] DeWitt, D. J., Ghandeharizadeh, S., Schneider, D. A., Bricker, A., Hsiao, H., and Rasmussen, R. “The Gamma Database Machine Project.” *IEEE Transactions on Knowledge and Data Engineering* 2(1):44-62, March 1990.

[Eic87] Eich, M. H. “MARS: The Design of A Main Memory Database Machine.” In *Proc. of the 1987 International workshop on Database Machines*, October, 1987.

[GS92] Garcia-Molina, H. and Salem, K. “Main Memory Database Systems: An Overview.” *IEEE Trans. Knowl. Data Eng.* 4(6): 509-516 (1992).

[GR93] Gray, J. and Reuter, A. “Transaction Processing: Concepts and Techniques.” Morgan Kaufmann Publishers, Inc., 1993.

[GBH+00] Gribble, S. D., Brewer, E. A., Hellerstein, J. M., and Culler, D .E. “Scalable, Distributed Data Structures for Internet Service Construction.” In *Proc. OSDI*, 2000.

[Hel07] Helland, P. “Life beyond Distributed Transactions: an Apostate’s Opinion.” In *Proc. CIDR*, 2007.

[HM93] Herlihy, M. P. and Moss, J. E. B. “Transactional Memory: architectural support for lock-free data structures.” In *Proc. ISCA*, 1993.

[KR81] Kung, H. T. and Robinson, J. T. “On optimistic methods for concurrency control.” *ACM Trans. Database Syst. 6(2):213–226*, June 1981.

[LM06] Lau, E. and Madden, S. “An Integrated Approach to Recovery and High Availability in an Updatable, Distributed Data Warehouse.” In *Proc. VLDB*, 2006.

[LC86] Lehman, T. J. and Carey, M. J. “A study of index structures for main memory database management systems.” In *Proc. VLDB*, 1986.

[LGG+91] Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., Shrira, L., and Williams, M. “Replication in the harp file system.” In *Proc. SOSP*, pages 226-238, 1991.

[MSA+04] McWherter, D. T., Schroeder, B., Ailamaki, A., and Harchol-Balter, M. “Priority Mechanisms for OLTP and Transactional Web Applications.” In *Proc.ICDE*, 2004.

[MHL+92] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. “ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging.” *ACM Trans. Database Syst. 17(1):94-162*, 1992.

[Moh89] Mohan, C. “ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes.” 1989, Research Report RJ 7008, Data Base Technology Institute, IBM Almaden Research Center.

[ML89] Mohan, C. and Levine, F. “ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging.” 1989, Research Report RJ 6846, Data Base Technology Institute, IBM Almaden Research Center.

[MBD+99] Mucci, P. J., Browne, S., Deane, C., and Ho, G. “PAPI: A Portable Interface to Hardware Performance Counters.” In *Proc. Department of Defense HPCMP Users Group Conference*, Monterey, CA, June 1999.

[RR99] Rao, J. and Ross, K. A. “Cache Conscious Indexing for Decision-Support in Main Memory.” In *Proc. VLDB*, 1999.

[RR00] Rao, J. and Ross, K. A. “Making B+-trees cache conscious in main memory.” In *SIGMOD Record, 29(2):475-486*, June 2000.

[SMK+01] Stoica, I., Morris, R., Karger, D. R., Kaashoek, M. F., and Balakrishnan, H. “Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications.” In *Proc. SIGCOMM*, 2001.

[SAB+05] Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O’Neil, E., O’Neil, P., Rasin, A., Tran, N., and Zdonik, S. “C-Store: A Column-oriented DBMS.” In *Proc. VLDB*, 2005.

[SMA+07] Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., and Helland, P. “The End of an Architectural Era (It’s Time for a Complete Rewrite).” In *Proc. VLDB*, 2007.

[Tim07] Oracle TimesTen. http://www.oracle.com/timesten/index.html. 2007.

[TPCC] The Transaction Processing Council. TPC-C Benchmark (Rev. 5.8.0), 2006. http://www.tpc.org/tpcc/spec/tpcc_current.pdf

[WSA97] Whitney, A., Shasha, D., and Apter, S. “High Volume Transaction Processing Without Concurrency Control, Two Phase Commit, SQL or C.” In *Proc. HPTPS*, 1997.

“One Size Fits All”: An Idea Whose Time Has Come and Gone

Michael Stonebraker (MIT CSAIL and StreamBase Systems, Inc)
Uğur Çetintemel (Brown University and StreamBase Systems, Inc.)

Abstract

The last 25 years of commercial DBMS development can be summed up in a single phrase: “One size fits all”. This phrase refers to the fact that the traditional DBMS architecture (originally designed and optimized for business data processing) has been used to support many data-centric applications with widely varying characteristics and requirements.

In this paper, we argue that this concept is no longer applicable to the database market, and that the commercial world will fracture into a collection of independent database engines, some of which may be unified by a common front-end parser. We use examples from the stream-processing market and the data-warehouse market to bolster our claims. We also briefly discuss other markets for which the traditional architecture is a poor fit and argue for a critical rethinking of the current factoring of systems services into products.

1 Introduction

Relational DBMSs arrived on the scene as research prototypes in the 1970’s, in the form of System R [10] and INGRES [27]. The main thrust of both prototypes

Originally published in *Proceedings of the 21st International Conference on Data Engineering*, pp. 2–11, 2005. Original DOI: 10.1109/ICDE.2005.1

was to surpass IMS in value to customers on the applications that IMS was used for, namely “business data processing”. Hence, both systems were architected for on-line transaction processing (OLTP) applications, and their commercial counterparts (i.e., DB2 and INGRES, respectively) found acceptance in this arena in the 1980’s. Other vendors (e.g., Sybase, Oracle, and Informix) followed the same basic DBMS model, which stores relational tables row-by-row, uses B-trees for indexing, uses a cost-based optimizer, and provides ACID transaction properties.

Since the early 1980’s, the major DBMS vendors have steadfastly stuck to a “one size fits all” strategy, whereby they maintain a single code line with all DBMS services. The reasons for this choice are straightforward—the use of multiple code lines causes various practical problems, including:

- *a cost problem*, because maintenance costs increase at least linearly with the number of code lines;
- *a compatibility problem*, because all applications have to run against every code line;
- *a sales problem*, because salespeople get confused about which product to try to sell to a customer; and
- *a marketing problem*, because multiple code lines need to be positioned correctly in the marketplace.

To avoid these problems, all the major DBMS vendors have followed the adage “put all wood behind one arrowhead”. In this paper we argue that this strategy has failed already, and will fail more dramatically off into the future.

The rest of the paper is structured as follows. In Section 2, we briefly indicate why the single code-line strategy has failed already by citing some of the key characteristics of the data warehouse market. In Section 3, we discuss stream processing applications and indicate a particular example where a specialized stream processing engine outperforms an RDBMS by two orders of magnitude. Section 4 then turns to the reasons for the performance difference, and indicates that DBMS technology is not likely to be able to adapt to be competitive in this market. Hence, we expect stream processing engines to thrive in the marketplace. In Section 5, we discuss a collection of other markets where one size is not likely to fit all, and other specialized database systems may be feasible. Hence, the fragmentation of the DBMS market may be fairly extensive. In Section 6, we offer some comments about the factoring of system software into products. Finally, we close the paper with some concluding remarks in Section 7.

2 Data Warehousing

In the early 1990's, a new trend appeared: Enterprises wanted to gather together data from multiple operational databases into a data warehouse for business intelligence purposes. A typical large enterprise has 50 or so operational systems, each with an on-line user community who expect fast response time. System administrators were (and still are) reluctant to allow business-intelligence users onto the same systems, fearing that the complex ad-hoc queries from these users will degrade response time for the on-line community. In addition, business-intelligence users often want to see historical trends, as well as correlate data from multiple operational databases. These features are very different from those required by on-line users.

For these reasons, essentially every enterprise created a large data warehouse, and periodically "scraped" the data from operational systems into it. Business-intelligence users could then run their complex ad-hoc queries against the data in the warehouse, without affecting the on-line users. Although most warehouse projects were dramatically over budget and ended up delivering only a subset of promised functionality, they still delivered a reasonable return on investment. In fact, it is widely acknowledged that historical warehouses of retail transactions pay for themselves within a year, primarily as a result of more informed stock rotation and buying decisions. For example, a business-intelligence user can discover that pet rocks are out and Barbie dolls are in, and then make appropriate merchandise placement and buying decisions.

Data warehouses are very different from OLTP systems. OLTP systems have been optimized for updates, as the main business activity is typically to sell a good or service. In contrast, the main activity in data warehouses is ad-hoc queries, which are often quite complex. Hence, periodic load of new data interspersed with ad-hoc query activity is what a typical warehouse experiences.

The standard wisdom in data warehouse schemas is to create a fact table, containing the "who, what, when, where" about each operational transaction. For example, Figure 1 shows the schema for a typical retailer. Note the central fact table, which holds an entry for each item that is scanned by a cashier in each store in its chain. In addition, the warehouse contains dimension tables, with information on each store, each customer, each product, and each time period. In effect, the fact table contains a foreign key for each of these dimensions, and a star schema is the natural result. Such star schemas are omnipresent in warehouse environments, but are virtually nonexistent in OLTP environments.

It is a well known homily that warehouse applications run much better using bit-map indexes while OLTP users prefer B-tree indexes. The reasons are straight-

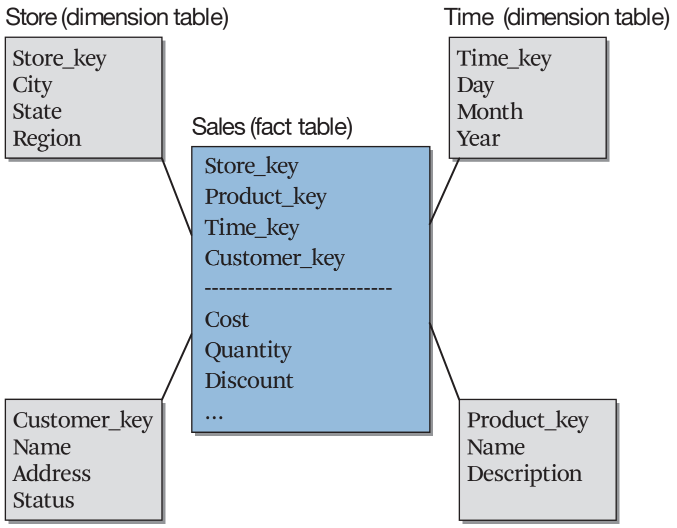


Figure 1 A typical star schema

forward: bit-map indexes are faster and more compact on warehouse workloads, while failing to work well in OLTP environments. As a result, many vendors support both B-tree indexes and bit-map indexes in their DBMS products.

In addition, materialized views are a useful optimization tactic in warehouse worlds, but never in OLTP worlds. In contrast, normal (“virtual”) views find acceptance in OLTP environments.

To a first approximation, most vendors have a warehouse DBMS (bit-map indexes, materialized views, star schemas and optimizer tactics for star schema queries) and an OLTP DBMS (B-tree indexes and a standard cost-based optimizer), which are united by a common parser, as illustrated in Figure 2.

Although this configuration allows such a vendor to market his DBMS product as a single system, because of the single user interface, in effect, she is selling multiple systems. Moreover, there will considerable pressure from both the OLTP and warehouse markets for features that are of no use in the other world. For example, it is common practice in OLTP databases to represent the state (in the United States) portion of an address as a two-byte character string. In contrast, it is obvious that 50 states can be coded into six bits. If there are enough queries and enough data to justify the cost of coding the state field, then the later representation is advantageous. This is usually true in warehouses and never true in OLTP. Hence, elaborate coding of fields will be a warehouse feature that has little or no utility in

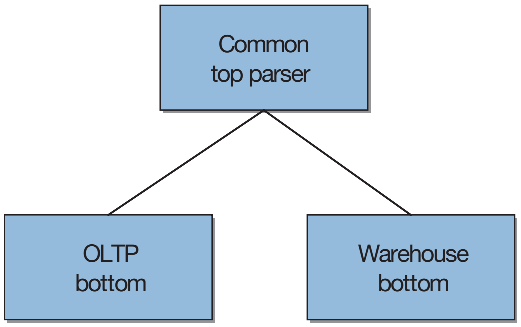


Figure 2 The architecture of current DBMSs

OLTP. The inclusion of additional market-specific features will make commercial products look increasingly like the architecture illustrated in Figure 2.

The illusion of "one size fits all" can be preserved as a marketing fiction for the two different systems of Figure 2, because of the common user interface. In the stream processing market, to which we now turn, such a common front end is impractical. Hence, not only will there be different engines but also different front ends. The marketing fiction of "one size fits all" will not fly in this world.

3 Stream Processing

Recently, there has been considerable interest in the research community in stream processing applications [7, 13, 14, 20]. This interest is motivated by the upcoming commercial viability of sensor networks over the next few years. Although RFID has gotten all the press recently and will find widespread acceptance in retail applications dealing with supply chain optimization, there are many other technologies as well (e.g., Lojack [3]). Many industry pundits see a "green field" of monitoring applications that will be enabled by this "sea change" caused by networks of low-cost sensor devices.

3.1 Emerging Sensor-based Applications

There are obvious applications of sensor network technology in the military domain. For example, the US Army is investigating putting vital-signs monitors on all soldiers, so that they can optimize medical triage in combat situations. In addition, there is already a GPS system in many military vehicles, but it is not connected yet into a closed-loop system. Instead, the army would like to monitor the position of all vehicles and determine, in real time, if they are off course. Additionally, they would like a sensor on the gun turret; together with location, this will allow the detection of crossfire situations. A sensor on the gas gauge will allow the optimization of refueling. In all, an army battalion of 30,000 humans and 12,000 vehicles will soon be a large-scale sensor network of several hundred thousand nodes delivering state and position information in real time.

Processing nodes in the network and downstream servers must be capable of dealing with this "firehose" of data. Required operations include sophisticated alerting, such as the platoon commander wishes to know when three of his four vehicles cross the front line. Also required are historical queries, such as "Where has vehicle 12 been for the last two hours?" Lastly, requirements encompass longitudinal queries, such as "What is the overall state of readiness of the force right now?"

Other sensor-based monitoring applications will also come over time in many non-military applications. Monitoring traffic congestion and suggesting alternate travel routes is one example. A related application is variable, congestion-based tolling on highway systems, which was the inspiration behind the Linear Road benchmark [9]. Amusement parks will soon turn passive wristbands on customers into active sensors, so that rides can be optimized and lost children located. Cell phones are already active devices, and one can easily imagine a service whereby the closest restaurant to a hungry customer can be located. Even library books will be sensor tagged, because if one is mis-shelved, it may be lost forever in a big library.

There is widespread speculation that conventional DBMSs will not perform well on this new class of monitoring applications. In fact, on Linear Road, traditional solutions are nearly an order of magnitude slower than a special purpose stream processing engine [9]. The inapplicability of the traditional DBMS technology to streaming applications is also bolstered by an examination of the current application areas with streaming data. We now discuss our experience with such an application, financial-feed processing.

3.2 An Existing Application: Financial-Feed Processing

Most large financial institutions subscribe to feeds that deliver real-time data on market activity, specifically news, consummated trades, bids and asks, etc. Reuters, Bloomberg and Infodyne are examples of vendors that deliver such feeds. Financial institutions have a variety of applications that process such feeds. These include systems that produce real-time business analytics, ones that perform electronic trading, ones that ensure legal compliance of all trades to the various company and SEC rules, and ones that compute real-time risk and market exposure to fluctuations in foreign exchange rates. The technology used to implement this class of applications is invariably “roll your own”, because application experts have not had good luck with off-the-shelf system software products.

In order to explore feed processing issues more deeply, we now describe in detail a specific prototype application, which was specified by a large mutual fund company. This company subscribes to several commercial feeds, and has a current production application that watches all feeds for the presence of late data. The idea is to alert the traders if one of the commercial feeds is delayed, so that the traders can know not to trust the information provided by that feed. This company is unhappy with the performance and flexibility of their “roll your own” solution and requested a pilot using a stream processing engine.

The company engineers specified a simplified version of their current application to explore the performance differences between their current system and a stream processing engine. According to their specification, they were looking for maximum message processing throughput on a single PC-class machine for a subset of their application, which consisted of two feeds reporting data from two exchanges.

Specifically, there are 4500 securities, 500 of which are “fast moving”. A stock tick on one of these securities is late if it occurs more than five seconds after the previous tick from the same security. The other 4000 symbols are slow moving, and a tick is late if 60 seconds have elapsed since the previous tick.

There are two feed providers and the company wished to receive an alert message each time there is a late tick from either provider. In addition, they wished to maintain a counter for each provider. When 100 late ticks have been received from either provider, they wished to receive a special “this is really bad” message and then to suppress the subsequent individual tick reports

The last wrinkle in the company’s specification was that they wished to accumulate late ticks from each of two exchanges, say NYSE and NASD, regardless of which feed vendor produced the late data. If 100 late messages were received from either exchange through either feed vendor, they wished to receive two additional special messages. In summary, they want four counters, each counting to 100, with a resulting special message. An abstract representation of the query diagram for this task is shown in Figure 3.

Although this prototype application is only a subset of the application logic used in the real production system, it represents a simple-to-specify task on which performance can be readily measured; as such, it is a representative example. We now turn to the speed of this example application on a stream processing engine as well as an RDBMS.

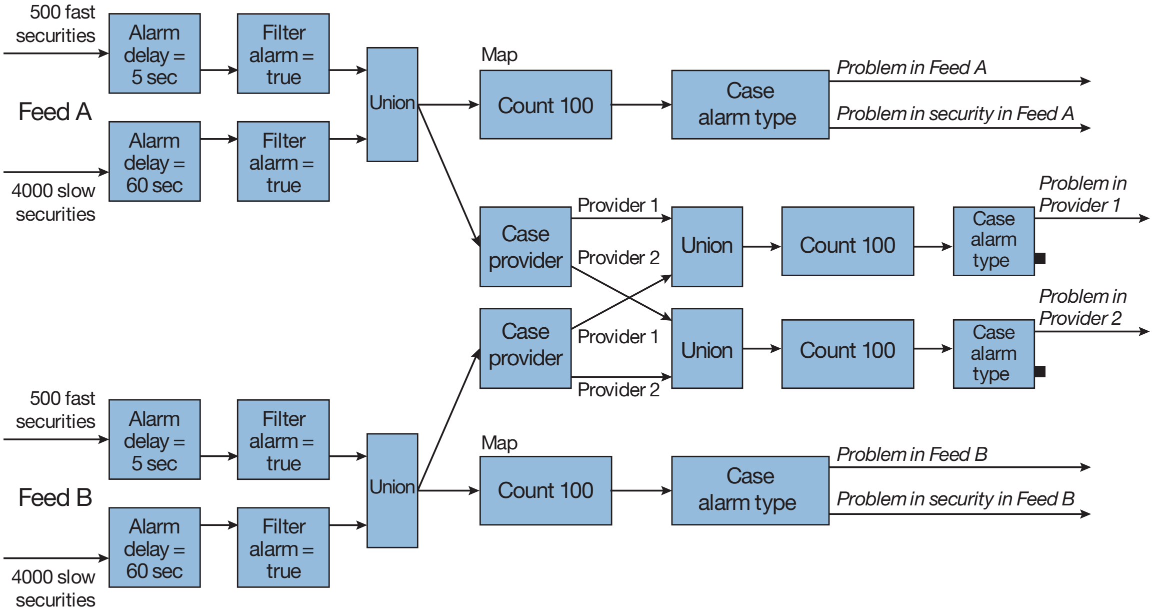


Figure 3 The Feed Alarm application in StreamBase

4 Performance Discussion

The example application discussed in the previous section was implemented in the StreamBase stream processing engine (SPE) [5], which is basically a commercial, industrial-strength version of Aurora [8, 13]. On a 2.8Ghz Pentium processor with 512 Mbytes of memory and a single SCSI disk, the workflow in Figure 3 can be executed at 160,000 messages per second, before CPU saturation is observed. In contrast, StreamBase engineers could only coax 900 messages per second from an implementation of the same application using a popular commercial relational DBMS.

In this section, we discuss the main reasons that result in the two orders of magnitude difference in observed performance. As we argue below, the reasons have to do with the inbound processing model, correct primitives for stream processing, and seamless integration of DBMS processing with application processing. In addition, we also consider transactional behavior, which is often another major consideration.

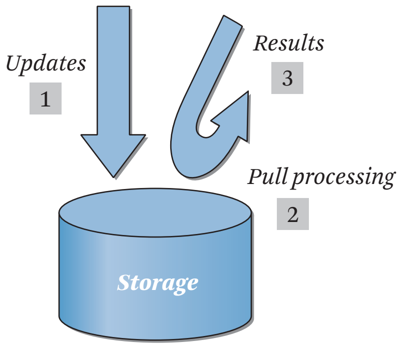


Figure 4 “Outbound” processing

4.1 “Inbound” versus “Outbound” Processing

Built fundamentally into the DBMS model of the world is what we term “outbound” processing, illustrated in Figure 4. Specifically, one inserts data into a database as a first step (step 1). After indexing the data and committing the transaction, that data is available for subsequent query processing (step 2) after which results are presented to the user (step 3). This model of “process-after-store” is at the heart of all conventional DBMSs, which is hardly surprising because, after all, the main function of a DBMS is to accept and then never lose data.

In real-time applications, the storage operation, which must occur before processing, adds significantly both to the delay (i.e., latency) in the application, as well as to the processing cost per message of the application. An alternative processing model that avoids this storage bottleneck is shown graphically in Figure 5. Here, input streams are pushed to the system (step 1) and get processed (step 2) as they “fly by” in memory by the query network. The results are then pushed to the client application(s) for consumption (step 3). Reads or writes to storage are optional and can be executed asynchronously in many cases, when they are present. The fact that storage is absent or optional saves both on cost and latency, resulting in significantly higher performance. This model, called “inbound” processing, is what is employed by a stream processing engine such as StreamBase.

One is, of course, led to ask “Can a DBMS do inbound processing?” DBMSs were originally designed as outbound processing engines, but grafted triggers onto their engines as an afterthought many years later. There are many restrictions on triggers (e.g., the number allowed per table) and no way to ensure trigger safety (i.e., ensuring that triggers do not go into an infinite loop). Overall, there is very

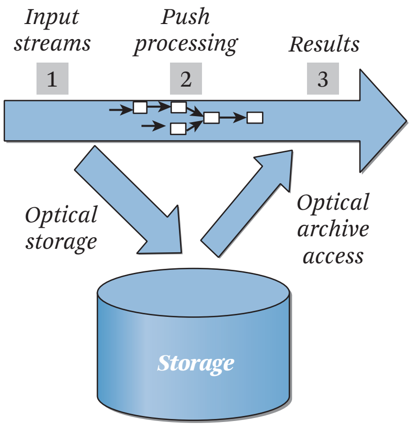


Figure 5 "Inbound" processing

little or no programming support for triggers. For example, there is no way to see what triggers are in place in an application, and no way to add a trigger to a table through a graphical user interface. Moreover, virtual views and materialized views are provided for regular tables, but not for triggers. Lastly, triggers often have performance problems in existing engines. When StreamBase engineers tried to use them for the feed alarm application, they still could not obtain more than 900 messages per second. In summary, triggers are incorporated to the existing designs as an afterthought and are thus second-class citizens in current systems.

As such, relational DBMSs are outbound engines onto which limited inbound processing has been grafted. In contrast, stream processing engines, such as Aurora and StreamBase are fundamentally inbound processing engines. From the ground up, an inbound engine looks radically different from an outbound engine. For example, an outbound engine uses a "pull" model of processing, i.e., a query is submitted and it is the job of the engine to efficiently pull records out of storage to satisfy the query. In contrast, an inbound engine uses a "push" model of processing, and it is the job of the engine to efficiently push incoming messages through the processing steps entailed in the application.

Another way to view the distinction is that an outbound engine stores the data and then executes the queries against the data. In contrast, an inbound engine stores the queries and then passes the incoming data (messages) through the queries.

Although it seems conceivable to construct an engine that is either an inbound or an outbound engine, such a design is clearly a research project. In the meantime, DBMSs are optimized for outbound processing, and stream processing engines for inbound processing. In the feed alarm application, this difference in philosophy accounts for a substantial portion of the performance difference observed.

4.2 The Correct Primitives

SQL systems contain a sophisticated aggregation system, whereby a user can run a statistical computation over groupings of the records from a table in a database. The standard example is:

```
Select avg (salary)
From employee
Group by department
```

When the execution engine processes the last record in the table, it can emit the aggregate calculation for each group of records. However, this construct is of little benefit in streaming applications, where streams continue forever and there is no notion of “end of table”.

Consequently, stream processing engines extend SQL (or some other aggregation language) with the notion of time windows. In StreamBase, windows can be defined based on clock time, number of messages, or breakpoints in some other attribute. In the feed alarm application, the leftmost box in each stream is such an aggregate box. The aggregate groups stocks by symbol and then defines windows to be ticks 1 and 2, 2 and 3, 3 and 4, etc. for each stock. Such “sliding windows” are often very useful in real-time applications.

In addition, StreamBase aggregates have been constructed to deal intelligently with messages which are late, out-of-order, or missing. In the feed alarm application, the customer is fundamentally interested in looking for late data. StreamBase allows aggregates on windows to have two additional parameters. The first is a *timeout* parameter, which instructs the StreamBase execution engine to close a window and emit a value even if the condition for closing the window has not been satisfied. This parameter effectively deals with late or missing tuples. The second parameter is *slack*, which is a directive to the execution engine to keep a window open, after its closing condition has been satisfied. This parameter addresses disorder in tuple arrivals. These two parameters allow the user to specify how to deal with stream abnormalities and can be effectively utilized to improve system resilience.

In the feed alarm application each window is two ticks, but has a timeout of either 5 or 60 seconds. This will cause windows to be closed if the inter-arrival time between successive ticks exceeds the maximum defined by the user. This is a very efficient way to discover late data; i.e., as a side effect of the highly-tuned aggregate logic. In the example application, the box after each aggregate discards the valid data and keeps only the timeout messages. The remainder of the application performs the necessary bookkeeping on these timeouts.

Having the right primitives at the lower layers of the system enables very high performance. In contrast, a relational engine contains no such built-in constructs. Simulating their effect with conventional SQL is quite tedious, and results in a second significant difference in performance.

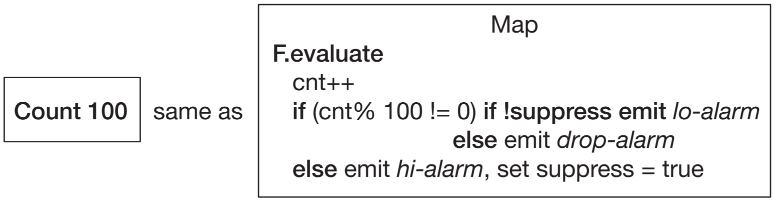
It is possible to add time windows to SQL, but these make no sense on stored data. Hence, window constructs would have to be integrated into some sort of an inbound processing model.

4.3 Seamless Integration of DBMS Processing and Application Logic

Relational DBMSs were all designed to have client-server architectures. In this model, there are many client applications, which can be written by arbitrary people, and which are therefore typically untrusted. Hence, for security and reliability reasons, these client applications are run in a separate address space from the DBMS. The cost of this choice is that the application runs in one address space while DBMS processing occurs in another, and a process switch is required to move from one address space to the other.

In contrast, the feed alarm application is an example of an embedded system. It is written by one person or group, who is trusted to “do the right thing”. The entire application consists of (1) DBMS processing—for example the aggregation and filter boxes, (2) control logic to direct messages to the correct next processing step, and (3) application logic. In StreamBase, these three kinds of functionality can be freely interspersed. Application logic is supported with user-defined boxes, the Count100 box in our example financial-feed processing application. The actual code, shown in Figure 6, consists of four lines of C++ that counts to 100 and sets a flag that ensures that the correct messages are emitted. Control logic is supported by allowing multiple predicates in a filter box, and thereby multiple exit arcs. As such, a filter box performs “if-then-else” logic in addition to filtering streams.

In effect, the feed alarm application is a mix of DBMS-style processing, conditional expressions, and user-defined functions in a conventional programming language. This combination is performed by StreamBase within a single address space without any process switches. Such a seamless integration of DBMS logic with conventional programming facilities was proposed many years ago in Rigel [23] and Pascal-R [25], but was never implemented in commercial relational systems. Instead, the major vendors implemented stored procedures, which are much more



```
F.evaluate
  cnt++
  if (cnt% 100 != 0) if !suppress emit lo-alarm
                     else emit drop-alarm
  else emit hi-alarm, set suppress = true
```

Figure 6 "Count100" Logic

limited programming systems. More recently, the emergence of object-relational engines provided blades or extenders, which are more powerful than stored procedures, but still do not facilitate flexible control logic.

Embedded systems do not need the protection provided by client-server DBMSs, and a two-tier architecture merely generates overhead. This is a third source of the performance difference observed in our example application.

Another integration issue, not exemplified by the feed alarm example, is the storage of state information in streaming applications. Most stream processing applications require saving some state, anywhere from modest numbers of megabytes to small numbers of gigabytes. Such state information may include (1) reference data (i.e., what stocks are of interest), (2) translation tables (in case feeds use different symbols for the same stock), and (3) historical data (e.g., "how many late ticks were observed every day during the last year?"). As such, tabular storage of data is a requirement for most stream processing applications.

StreamBase embeds BerkeleyDB [4] for state storage. However, there is approximately one order of magnitude performance difference between calling BerkeleyDB in the StreamBase address space and calling it in client-server mode in a different address space. This is yet another reason to avoid process switches by mixing DBMS and application processing in one address space.

Although one might suggest that DBMSs enhance their programming models to address this performance problem, there are very good reasons why client-server DBMSs were designed the way they are. Most business data processing applications need the protection that is afforded by this model. Stored procedures and object-relational blades were an attempt to move some of the client logic into the server to gain performance. To move further, a DBMS would have to implement both an embedded and a non-embedded model, with different run time systems. Again, this would amount to giving up on "one size fits all".

In contrast, feed processing systems are invariably embedded applications. Hence, the application and the DBMS are written by the same people, and driven from external feeds, not from human-entered transactions. As such, there is no reason to protect the DBMS from the application, and it is perfectly acceptable to run both in the same address space. In an embedded processing model, it is reasonable to freely mix application logic, control logic and DBMS logic, which is exactly what StreamBase does.

4.4 High Availability

It is a requirement of many stream-based applications to have high availability (HA) and stay up 7x24. Standard DBMS logging and crash recovery mechanisms (e.g., [22]) are ill-suited for the streaming world as they introduce several key problems.

First, log-based recovery may take large number of seconds to small numbers of minutes. During this period, the application would be "down". Such behavior is clearly undesirable in many real-time streaming domains (e.g., financial services). Second, in case of a crash, some effort must be made to buffer the incoming data streams, as otherwise this data will be irretrievably lost during the recovery process. Third, DBMS recovery will only deal with tabular state and will thus ignore operator states. For example, in the feed alarm application, the counters are not in stored in tables; therefore their state would be lost in a crash. One straightforward fix would be to force all operator state into tables to use DBMS-style recovery; however, this solution would significantly slow down the application.

The obvious alternative to achieve high availability is to use techniques that rely on Tandem-style process pairs [11]. The basic idea is that, in the case of a crash, the application performs failover to a backup machine, which typically operates as a "hot standby", and keeps going with small delay. This approach eliminates the overhead of logging. As a case in point, StreamBase turns off logging in BerkeleyDB.

Unlike traditional data-processing applications that require precise recovery for correctness, many stream-processing applications can tolerate and benefit from weaker notions of recovery. In other words, failover does not always need to be "perfect". Consider monitoring applications that operate on data streams whose values are periodically refreshed. Such applications can often tolerate tuple losses when a failure occurs, as long as such interruptions are short. Similarly, if one loses a couple of ticks in the feed alarm application during failover, the correctness would probably still be preserved. In contrast, applications that trigger alerts when certain combinations of events happen, require that no tuples be lost, but may tolerate temporary duplication. For example, a patient monitoring application may be able to tolerate duplicate tuples ("heart rate is 79") but not lost tuples ("heart rate has changed to zero"). Of course, there will always be a class of applications that require strong, precise recovery guarantees. A financial application that performs portfolio management based on individual stock transactions falls into this category.

As a result, there is an opportunity to devise simplified and low overhead failover schemes, when weaker correctness notions are sufficient. A collection of detailed options on how to achieve high availability in a streaming world has recently been explored [17].

4.5 Synchronization

Many stream-based applications rely on shared data and computation. Shared data is typically contained in a table that one query updates and another one reads. For example, the Linear Road application requires that vehicle-position data be used to update statistics on highway usage, which in turn are read to determine tolls for each segment on the highway. Thus, there is a basic need to provide isolation between messages.

Traditional DBMSs use ACID transactions to provide isolation (among others things) between concurrent transactions submitted by multiple users. In streaming systems, which are not multi-user, such isolation can be effectively achieved through simple critical sections, which can be implemented through light-weight semaphores. Since full-fledged transactions are not required, there is no need to use heavy-weight locking-based mechanisms anymore.

In summary, ACID properties are not required in most stream processing applications, and simpler, specialized performance constructs can be used to advantage.

5 One Size Fits All?

The previous section has indicated a collection of architectural issues that result in significant differences in performance between specialized stream processing engines and traditional DBMSs. These design choices result in a big difference between the internals of the two engines. In fact, the run-time code in StreamBase looks nothing like a traditional DBMS run-time. The net result is vastly better performance on a class of real-time applications. These considerations will lead to a separate code line for stream processing, of course assuming that the market is large enough to facilitate this scenario.

In the rest of the section, we outline several other markets for which specialized database engines may be viable.

5.1 Data Warehouses

The architectural differences between OLTP and warehouse database systems discussed in Section 2 are just the tip of the iceberg, and additional differences will occur over time. We now focus on probably the biggest architectural difference, which is to store the data by column, rather than by row.

All major DBMS vendors implement record-oriented storage systems, where the attributes of a record are placed contiguously in storage. Using this “row-store” architecture, a single disk write is all that is required to push all of the attributes of a single record out to disk. Hence, such a system is “write-optimized” because high performance on record writes is easily achievable. It is easy to see that write-optimized systems are especially effective on OLTP-style applications, the primary reason why most commercial DBMSs employ this architecture.

In contrast, warehouse systems need to be “read-optimized” as most workload consists of ad-hoc queries that touch large amounts of historical data. In such systems, a “column-store” model where the values for all of the rows of a single attribute are stored contiguously is drastically more efficient (as demonstrated by Sybase IQ [6], Addamark [1], and KDB [2]).

With a column-store architecture, a DBMS need only read the attributes required for processing a given query, and can avoid bringing into memory any other irrelevant attributes. Given that records with hundreds of attributes (with many null values) are becoming increasingly common, this approach results in a sizeable performance advantage for warehouse workloads where typical queries involve aggregates that are computed on a small number of attributes over large data sets. The first author of this paper is engaged in a research project to explore the performance benefits of a column-store system.

5.2 Sensor Networks

It is not practical to run a traditional DBMS in the processing nodes that manage sensors in a sensor network [21, 24]. These emerging platforms of device networks are currently being explored for applications such as environmental and medical monitoring, industrial automation, autonomous robotic teams, and smart homes [16, 19, 26, 28, 29].

In order to realize the full potential of these systems, the components are designed to be wireless, with respect to both communication and energy. In this environment, bandwidth and power become the key resources to be conserved. Furthermore, communication, as opposed to processing or storage access, is the main consumer of energy. Thus, standard DBMS optimization tactics do not apply and need to be critically rethought. Furthermore, transactional capabilities seem to be irrelevant in this domain.

In general, there is a need to design flexible, light-weight database abstractions (such as TinyDB [18]) that are optimized for data movement as opposed to data storage.

5.3 Text Search

None of the current text search engines use DBMS technology for storage, even though they deal with massive, ever-increasing data sets. For instance, Google built its own storage system (called GFS [15]) that outperforms conventional DBMS technology (as well as file system technology) for some of the reasons discussed in Section 4.

A typical search engine workload [12, 15] consists of a combination of inbound streaming data (coming from web crawlers), which needs to be cleaned and incorporated into the existing search index, and ad hoc look-up operations on the existing index. In particular, the write operations are mostly append-only and read operations sequential. Concurrent writes (i.e., appends) to the same file are necessary for good performance. Finally, the large number of storage machines, made up of commodity parts, ensure that failure is the norm rather than the exception. Hence, high availability is a key design consideration and can only be achieved through fast recovery and replication.

Clearly, these application characteristics are much different from those of conventional business-processing applications. As a result, even though some DBMSs has built-in text search capabilities, they fall short of meeting the performance and availability requirements of this domain: they are simply too heavy-weight and inflexible.

5.4 Scientific Databases

Massive amounts of data are continuously being gathered from the real-world by sensors of various types, attached to devices such as satellites and microscopes, or are generated artificially by high-resolution scientific and engineering simulations.

The analysis of such data sets is the key to better understanding physical phenomena and is becoming increasingly commonplace in many scientific research domains. Efficient analysis and querying of these vast databases require highly-efficient multi-dimensional indexing structures and application-specific aggregation techniques. In addition, the need for efficient data archiving, staging, lineage, and error propagation techniques may create a need for yet another specialized engine in this important domain.

5.5 XML Databases

Semi-structured data is everywhere. Unfortunately, such data does not immediately fit into the relational model. There is a heated ongoing debate regarding how to best store and manipulate XML data. Even though some believe that relational DBMSs (with proper extensions) are the way to go, others would argue that a specialized engine is needed to store and process this data format.

6 A Comment on Factoring

Most stream-based applications require three basic services:

- *Message transport:* In many stream applications, there is a need to transport data efficiently and reliably among multiple distributed machines. The reasons for these are threefold. First, data sources and destinations are typically geographically dispersed. Second, high performance and availability requirements dictate the use of multiple cooperating server machines. Third, virtually all big enterprise systems consist of a complicated network of business applications running on a large number of machines, in which an SPE is embedded. Thus, the input and outputs messages to the SPE need to be properly routed from and to the appropriate external applications.
- *Storage of state:* As discussed in Section 4.3, in all but the most simplistic applications, there is a need to store state, typically in the form of read-only reference and historical tables, and read-write translation (e.g., hash) tables.
- *Execution of application logic:* Many streaming applications demand domain-specific message processing to be interspersed with query activity. In general, it is neither possible nor practical to represent such application logic using only the built-in query primitives (e.g., think legacy code).

A traditional design for a stream-processing application spreads the entire application logic across three diverse systems: (1) *a messaging system* (such as MQSeries, WebMethods, or Tibco) to reliably connect the component systems, typically using a publish/subscribe paradigm; (2) *a DBMS* (such as DB2 or Oracle) to provide persistence for state information; and (3) *an application server* (such as WebSphere or WebLogic) to provide application services to a set of custom-coded programs. Such a three-tier configuration is illustrated in Figure 7.

Unfortunately, such a design that spreads required functionality over three heavyweight pieces of system software will not perform well. For example, every message that requires state lookup and application services will entail multiple process switches between these different services.

In order to illustrate this per message overhead, we trace the steps taken when processing a message. An incoming message is first picked up by the bus and then forwarded to the custom application code (step 1), which cleans up and then processes the message. If the message needs to be correlated with historical data or

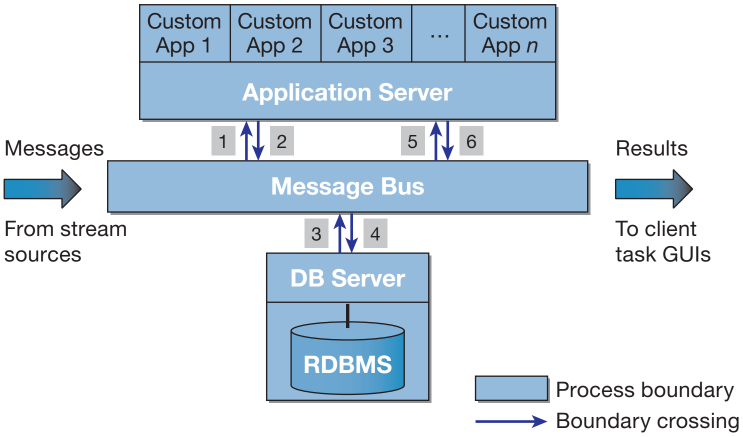


Figure 7 A multi-tier stream processing architecture

requires access to persistent data, then a request is sent to the DB server (steps 2-3), which accesses the DBMS. The response follows the reverse path to the application code (steps 4-5). Finally, the outcome of the processed message is forwarded to the client task GUI (step 6). Overall, there are six "boundary crossings" for processing a single message. In addition to the obvious context switches incurred, messages also need to transformed on-the-fly, by the appropriate adapters, to and from the native formats of the systems, each time they are picked up from and passed on to the message bus. The result is a very low useful work to overhead ratio. Even if there is some batching of messages, the overhead will be high and limit achievable performance.

To avoid such a performance hit, a stream processing engine must provide all three services in a single piece of system software that executes as one multi-threaded process on each machine that it runs. Hence, an SPE must have elements of a DBMS, an application server, and a messaging system. In effect, an SPE should provide specialized capabilities from all three kinds of software "under one roof".

This observation raises the question of whether the current factoring of system software into components (e.g., application server, DBMS, Extract-Transform-Load system, message bus, file system, web server, etc.) is actually an optimal one. After all, this particular decomposition arose partly as a historical artifact and partly from marketing happenstance. It seems like other factoring of systems services seems equally plausible, and it should not be surprising to see considerable evolution of component definition and factoring off into the future.

7 Concluding Remarks

In summary, there may be a substantial number of domain-specific database engines with differing capabilities off into the future. We are reminded of the curse “may you live in interesting times”. We believe that the DBMS market is entering a period of very interesting times. There are a variety of existing and newly-emerging applications that can benefit from data management and processing principles and techniques. At the same time, these applications are very much different from business data processing and from each other—there seems to be no obvious way to support them with a single code line. The “one size fits all” theme is unlikely to successfully continue under these circumstances.

References

[1] Addamark Scalable Log Server. http://www.addamark.com/products/sls.htm.

[2] Kx systems. http://www.kx.com/.

[3] Lojack.com, 2004. http://www.lojack.com/.

[4] Sleepycat software. http://www.sleepycat.com/.

[5] StreamBase Inc. http://www.streambase.com/.

[6] Sybase IQ. http://www.sybase.com/products/databaseservers/sybaseiq.

[7] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, J. Hwang, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Zing, R. Yan, and S. Zdonik. Aurora: A Data Stream Management System (demo description). In *Proceedings of the 2003 ACM SIGMOD Conference on Management of Data*, San Diego, CA, 2003.

[8] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 2003.

[9] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear Road: A Benchmark for Stream Data Management Systems. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, Toronto, CA, 2004.

[10] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. Wade, and V. Watson. System R: A Relational Approach to Database Management. *ACM Transactions on Database Systems*, 1976.

[11] J. Barlett, J. Gray, and B. Horst. Fault tolerance in Tandem computer systems. Tandem Computers Technical Report 86.2., 1986.

[12] E. Brewer, "Combining systems and databases: a search engine retrospective," in *Readings in Database Systems*, M. Stonebraker and J. Hellerstein, Eds., 4 ed, 2004.

[13] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring Streams: A New Class of Data Management Applications. In *proceedings of the 28th International Conference on Very Large Data Bases* (VLDB'02), Hong Kong, China, 2002.

[14] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proc. of the 1st CIDR Conference*, Asilomar, CA, 2003.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (SOSP), Bolton Landing, NY, USA, 2003.

[16] T. He, S. Krishnamurthy, J. A. Stankovic, T. Abdelzaher, L. Luo, R. Stoleru, T. Yan, L. Gu, J. Hui, and B. Krogh. An Energy-Efficient Surveillance System Using Wireless Sensor Networks. In *MobiSys'04*, 2004.

[17] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *Proceedings of the International Conference on Data Engineering*, Tokyo, Japan, 2004.

[18] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. The Design of an Acquisitional Query Processor for Sensor Networks. In *Proceedings of SIGMOD*, San Diego, CA, 2003.

[19] D. Malan, T. Fulford-Jones, M. Welsh, and S. Moulton. CodeBlue: An Ad Hoc Sensor Network Infrastructure for Emergency Medical Care. In *WAMES'04*, 2004.

[20] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query Processing, Resource Management, and Approximation and in a Data Stream Management System. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, Asilomar, CA, 2003.

[21] G. Pottie and W. Kaiser. Wireless Integrated Network Sensors. *Communications of the ACM*.

[22] K. Rothermel and C. Mohan. ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions. In *Proc. 15th International Conference on Very Large Data Bases (VLDB)*, Amsterdam, Holland, 1989.

[23] L. A. Rowe and K. A. Shoens. Data abstraction, views and updates in RIGEL. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data* (SIGMOD), Boston, Massachusetts, 1979.

[24] P. Saffo. Sensors: The Next Wave of Information. *Communications of the ACM*.

[25] J. W. Schmidth. Some High-Level Language Constructs for Data of Type Relation. *Transactions on Database Systems*, 2(247-261, 1977.

[26] L. Schwiebert, S. Gupta, and J. Weinmann. Research Challenges in Wireless Networks of Biomedical Sensors. In *Mobicom'01*, 2001.

[27] M. Stonebraker, E. Wong, P. Kreps, and G. Held. The Design and Implementation of INGRES. *ACM Trans. Database Systems*, 1(3):189-222, 1976.

[28] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. Lessons from a Sensor Network Expedition. In *EWSN'04*, 2004.

[29] C. S. Ting Liu, Pei Zhang and Margaret Martonosi. Implementing Software on Resource-Constrained Mobile Sensors: Experiences with Impala and ZebraNet. In *MobiSys'04*, 2004.

The End of an Architectural Era (It's Time for a Complete Rewrite)

Michael Stonebraker (MIT CSAIL), **Samuel Madden** (MIT CSAIL),
Daniel J. Abadi (MIT CSAIL), **Stavros Harizopoulos** (MIT CSAIL),
Nabil Hachem (AvantGarde Consulting, LLC),
Pat Helland (Microsoft Corporation)

Abstract

In previous papers [SC05, SBC+07], some of us predicted the end of "one size fits all" as a commercial relational DBMS paradigm. These papers presented reasons and experimental evidence that showed that the major RDBMS vendors can be outperformed by 1–2 orders of magnitude by specialized engines in the data warehouse, stream processing, text, and scientific database markets.

Assuming that specialized engines dominate these markets over time, the current relational DBMS code lines will be left with the business data processing

VLDB '07, September 23–28, 2007, Vienna, Austria.

Originally published in *Proceedings of the 33rd International Conference on Very Large Data Bases,* pp. 1150–1160, 2007.

(OLTP) market and hybrid markets where more than one kind of capability is required. In this paper we show that current RDBMSs can be beaten by nearly two orders of magnitude in the OLTP market as well. The experimental evidence comes from comparing a new OLTP prototype, H-Store, which we have built at M.I.T. to a popular RDBMS on the standard transactional benchmark, TPC-C.

We conclude that the current RDBMS code lines, while attempting to be a "one size fits all" solution, in fact, excel at nothing. Hence, they are 25 year old legacy code lines that should be retired in favor of a collection of "from scratch" specialized engines. The DBMS vendors (and the research community) should start with a clean sheet of paper and design systems for yesterday's needs.

1 Introduction

The popular relational DBMSs all trace their roots to System R from the 1970s. For example, DB2 is a direct descendent of System R, having used the RDS portion of System R intact in their first release. Similarly, SQL Server is a direct descendent of Sybase System 5, which, borrowed heavily from System R. Lastly, the first release of Oracle implemented the user interface from System R.

All three systems were architected more than 25 years ago, when hardware characteristics were much different than today. Processors are thousands of times faster and memories are thousands of times larger. Disk volumes have increased enormously, making it possible to keep essentially everything, if one chooses to. However, the bandwidth between disk and main memory has increased much more slowly. One would expect this relentless pace of technology to have changed the architecture of database systems dramatically over the last quarter of a century, but surprisingly the architecture of most DBMSs is essentially identical to that of System R.

Moreover, at the time relational DBMSs were conceived, there was only a single DBMS market, business data processing. In the last 25 years, a number of other markets have evolved, including data warehouses, text management, and stream processing. These markets have very different requirements than business data processing.

Lastly, the main user interface device at the time RDBMSs were architected was the dumb terminal, and vendors imagined operators inputting queries through an interactive terminal prompt. Now it is a powerful personal computer connected to the World Wide Web. Web sites that use OLTP DBMSs rarely run interactive transactions or present users with direct SQL interfaces.

In summary, the current RDBMSs were architected for the business data processing market in a time of different user interfaces and different hardware characteristics. Hence, they all include the following System R architectural features:

- Disk oriented storage and indexing structures
- Multithreading to hide latency
- Locking-based concurrency control mechanisms
- Log-based recovery

Of course, there have been some extensions over the years, including support for compression, shared-disk architectures, bitmap indexes, support for user-defined data types and operators, etc. However, no system has had a complete redesign since its inception. This paper argues that the time has come for a complete rewrite.

A previous paper [SBC+07] presented benchmarking evidence that the major RDBMSs could be beaten by specialized architectures by an order of magnitude or more in several application areas, including:

- Text (specialized engines from Google, Yahoo, etc.)
- Data Warehouses (column stores such as Vertica, Monet [Bon02], etc.)
- Stream Processing (stream processing engines such as StreamBase and Coral8)
- Scientific and intelligence databases (array storage engines such as MATLAB and ASAP [SBC+07])

Based on this evidence, one is led to the following conclusions:

1. RDBMSs were designed for the business data processing market, which is their sweet spot
2. They can be beaten handily in most any other market of significant enough size to warrant the investment in a specialized engine

This paper builds on [SBC+07] by presenting evidence that the current architecture of RDBMSs is not even appropriate for business data processing. Our methodology is similar to the one employed in [SBC+07]. Specifically, we have designed a new DBMS engine for OLTP applications. Enough of this engine, H-Store, is running to enable us to conduct a performance bakeoff between it and a popular commercial RDBMSs. Our experimental data shows H-Store to be a factor of 82 faster on TPC-C (almost two orders of magnitude).

Because RDBMSs can be beaten by more than an order of magnitude on the standard OLTP benchmark, then there is no market where they are competitive. As such, they should be considered as legacy technology more than a quarter of a century in age, for which a complete redesign and re-architecting is the appropriate next step.

Section 2 of this paper explains the design considerations that can be exploited to achieve this factor of 82 on TPC-C. Then, in Section 3, we present specific application characteristics which can be leveraged by a specialized engine. Following that, we sketch some of the H-store design in Section 4. We then proceed in Section 5 to present experimental data on H-Store and a popular RDBMS on TPC-C. We conclude the paper in Section 6 with some radical suggestions for the research agenda for the DBMS community.

2 OLTP Design Considerations

This section presents five major issues, which a new engine such as H-Store can leverage to achieve dramatically better performance than current RDBMSs.

2.1 Main Memory

In the late 1970's a large machine had somewhere around a megabyte of main memory. Today, several Gbytes are common and large machines are approaching 100 Gbytes. In a few years a terabyte of main memory will not be unusual. Imagine a shared nothing grid system of 20 nodes, each with 32 Gbytes of main memory now, (soon to be 100 Gbytes), and costing less than $50,000. As such, any database less than a terabyte in size, is capable of main memory deployment now or in the near future.

The overwhelming majority of OLTP databases are less than 1 Tbyte in size and growing in size quite slowly. For example, it is a telling statement that TPC-C requires about 100 Mbytes per physical distribution center (warehouse). A very large retail enterprise might have 1000 warehouses, requiring around 100 Gbytes of storage, which fits our envelope for main memory deployment.

As such, we believe that OLTP should be considered a main memory market, if not now then within a very small number of years. Consequently, the current RDBMS vendors have disk-oriented solutions for a main memory problem. In summary, 30 years of Moore's law has antiquated the disk-oriented relational architecture for OLTP applications.

Although there are some main memory database products on the market, such as TimesTen and SolidDB, these systems inherit the baggage of System R as well. This includes such features as a disk-based recovery log and dynamic locking, which, as we discuss in the following sections, impose substantial performance overheads.

2.2 Multi-threading and Resource Control

OLTP transactions are very lightweight. For example, the heaviest transaction in TPC-C reads about 200 records. In a main memory environment, the useful work of such a transaction consumes less than one millisecond on a low-end machine. In addition, most OLTP environments we are familiar with do not have "user stalls". For example, when an Amazon user clicks "buy it", he activates an OLTP transaction which will only report back to the user when it finishes. Because of an absence of disk operations and user stalls, the elapsed time of an OLTP transaction is minimal. In such a world it makes sense to run each SQL command in a transaction to completion with a single-threaded execution model, rather than paying for the overheads of isolation between concurrently executing statements.

Current RDBMSs have elaborate multi-threading systems to try to fully utilize CPU and disk resources. This allows several-to-many queries to be running in parallel. Moreover, they also have resource governors to limit the multiprogramming load, so that other resources (IP connections, file handles, main memory for sorting, etc.) do not become exhausted. These features are irrelevant in a single threaded execution model. No resource governor is required in a single threaded system.

In a single-threaded execution model, there is also no reason to have multi-threaded data structures. Hence the elaborate code required to support, for example, concurrent B-trees can be completely removed. This results in a more reliable system, and one with higher performance.

At this point, one might ask "What about long running commands?" In real-world OLTP systems, there aren't any for two reasons: First, operations that appear to involve long-running transactions, such as a user inputting data for a purchase on a web store, are usually split into several transactions to keep transaction time short. In other words, good application design will keep OLTP queries small. Second, longer-running ad-hoc queries are not processed by the OLTP system; instead such queries are directed to a data warehouse system, optimized for this activity. There is no reason for an OLTP system to solve a non-OLTP problem. Such thinking only applies in a "one size fits all" world.

2.3 Grid Computing and Fork-lift Upgrades

Current RDBMSs were originally written for the prevalent architecture of the 1970s, namely shared-memory multiprocessors. In the 1980's shared disk architectures were spearheaded by Sun and HP, and most DBMSs were expanded to include capabilities for this architecture. It is obvious that the next decade will bring domination by shared-nothing computer systems, often called grid computing or blade computing. Hence, any DBMS must be optimized for this configuration. An obvious strategy is to horizontally partition data over the nodes of a grid, a tactic first investigated in Gamma [DGS+90].

In addition, no user wants to perform a "fork-lift" upgrade. Hence, any new system should be architected for incremental expansion. If N grid nodes do not provide enough horsepower, then one should be able to add another K nodes, producing a system with N+K nodes. Moreover, one should perform this upgrade, without a hiccup, i.e. without taking the DBMS down. This will eliminate every system administrator's worst nightmare; a fork-lift upgrade with a requirement for a complete data reload and cutover.

To achieve incremental upgrade without going down requires significant capabilities, not found in existing systems. For example, one must be able to copy portions of a database from one site to another without stopping transactions. It is not clear how to bolt such a capability onto most existing systems. However, this can be made a requirement of a new design and implemented efficiently, as has been demonstrated by the existence of exactly this feature in the Vertica[1] codeline.

2.4 High Availability

Relational DBMSs were designed in an era (1970s) when an organization had a single machine. If it went down, then the company lost money due to system unavailability. To deal with disasters, organizations typically sent log tapes off site. If a disaster occurred, then the hardware vendor (typically IBM) would perform heroics to get new hardware delivered and operational in small numbers of days. Running the log tapes then brought the system back to something approaching where it was when the disaster happened.

A decade later in the 1980's, organizations executed contracts with disaster recovery services, such as Comdisco, for backup machine resources, so the log tapes could be installed quickly on remote backup hardware. This strategy minimized the time that an enterprise was down as a result of a disaster.

[1] http://www.vertica.com

Today, there are numerous organizations that run a **hot standby** within the enterprise, so that real-time failover can be accomplished. Alternately, some companies run multiple primary sites, so failover is even quicker. The point to be made is that businesses are much more willing to pay for multiple systems in order to avoid the crushing financial consequences of down time, often estimated at thousands of dollars per minute.

In the future, we see high availability and built-in disaster recovery as essential features in the OLTP (and other) markets. There are a few obvious conclusions to be drawn from this statement. First, every OLTP DBMS will need to keep multiple replicas consistent, requiring the ability to run seamlessly on a grid of geographically dispersed systems.

Second, most existing RDBMS vendors have glued multi-machine support onto the top of their original SMP architectures. In contrast, it is clearly more efficient to start with shared-nothing support at the bottom of the system.

Third, the best way to support shared nothing is to use multiple machines in a peer-to-peer configuration. In this way, the OLTP load can be dispersed across multiple machines, and inter-machine replication can be utilized for fault tolerance. That way, all machine resources are available during normal operation. Failures only cause degraded operation with fewer resources. In contrast, many commercial systems implement a “hot standby”, whereby a second machine sits effectively idle waiting to take over if the first one fails. In this case, normal operation has only half of the resources available, an obviously worse solution. These points argue for a complete redesign of RDBMS engines so they can implement peer-to-peer HA in the guts of a new architecture.

In an HA system, regardless of whether it is hot-standby or peer-to-peer, logging can be dramatically simplified. One must continue to have an undo log, in case a transaction fails and needs to roll back. However, the undo log does not have to persist beyond the completion of the transaction. As such, it can be a main memory data structure that is discarded on transaction commit. There is never a need for redo, because that will be accomplished via network recovery from a remote site. When the dead site resumes activity, it can be refreshed from the data on an operational site.

A recent paper [LM06] argues that failover/rebuild is as efficient as redo log processing. Hence, there is essentially no downside to operating in this manner. In an HA world, one is led to having no persistent redo log, just a transient undo one. This dramatically simplifies recovery logic. It moves from an Aries-style [MHL+92] logging system to new functionality to bring failed sites up to date from operational sites when they resume operation.

Again, a large amount of complex code has been made obsolete, and a different capability is required.

2.5 No Knobs

Current systems were built in an era where resources were incredibly expensive, and every computing system was watched over by a collection of wizards in white lab coats, responsible for the care, feeding, tuning and optimization of the system. In that era, computers were expensive and people were cheap. Today we have the reverse. Personnel costs are the dominant expense in an IT shop.

As such "self-everything" (self-healing, self-maintaining, self-tuning, etc.) systems are the only answer. However, all RDBMSs have a vast array of complex tuning knobs, which are legacy features from a bygone era. True; all vendors are trying to provide automatic facilities which will set these knobs without human intervention. However, legacy code cannot ever remove features. Hence, "no knobs" operation will be in addition to "human knobs" operation, and result in even more system documentation. Moreover, at the current time, the automatic tuning aids in the RDBMSs that we are familiar with do not produce systems with anywhere near the performance that a skilled DBA can produce. Until the tuning aids get vastly better in current systems, DBAs will turn the knobs.

A much better answer is to completely rethink the tuning process and produce a new system with no visible knobs.

3 Transaction, Processing and Environment Assumptions

If one assumes a grid of systems with main memory storage, built-in high availability, no user stalls, and useful transaction work under 1 millisecond, then the following conclusions become evident:

1. A persistent redo log is almost guaranteed to be a significant performance bottleneck. Even with group commit, forced writes of commit records can add milliseconds to the runtime of each transaction. The HA/failover system discussed earlier dispenses with this expensive architectural feature.
2. With redo gone, getting transactions into and out of the system is likely to be the next significant bottleneck. The overhead of JDBC/ODBC style interfaces will be onerous, and something more efficient should be used. In particular, we advocate running application logic—in the form of stored procedures—"in process" inside the database system, rather than the interprocess overheads implied by the traditional database client / server model.

3. An undo log should be eliminated wherever practical, since it will also be a significant bottleneck.
4. Every effort should be made to eliminate the cost of traditional dynamic locking for concurrency control, which will also be a bottleneck.
5. The latching associated with multi-threaded data structures is likely to be onerous. Given the short runtime of transactions, moving to a single threaded execution model will eliminate this overhead at little loss in performance.
6. One should avoid a two-phase commit protocol for distributed transactions, wherever possible, as network latencies imposed by round trip communications in 2PC often take on the order of milliseconds.

Our ability to remove concurrency control, commit processing and undo logging depends on several characteristics of OLTP schemas and transaction workloads, a topic to which we now turn.

3.1 Transaction and Schema Characteristics

H-Store requires the complete workload to be specified in advance, consisting of a collection of transaction *classes*. Each class contains transactions with the same SQL statements and program logic, differing in the run-time constants used by individual transactions. Since there are assumed to be no ad-hoc transactions in an OLTP system, this does not appear to be an unreasonable requirement. Such transaction classes must be registered with H-Store in advance, and will be disallowed if they contain user stalls (transactions may contain stalls for other reasons—for example, in a distributed setting where one machine must wait for another to process a request.) Similarly, H-Store also assumes that the collection of tables (logical schema) over which the transactions operate is known in advance.

We have observed that in many OLTP workloads every table except a single one called the *root*, has exactly one join term which is a 1-n relationship to its ancestor. Hence, the schema is a *tree* of 1-n relationships. We denote this class of schemas as *tree schemas*. Such schemas are popular; for example, customers produce orders, which have line items and fulfillment schedules. Tree schemas have an obvious horizontal partitioning over the nodes in a grid. Specifically, the root table can be range or hash partitioned on the primary key(s). Every descendent table can be partitioned such that all equi-joins in the tree span only a single site. In the discussion to follow, we will consider both tree and non-tree schemas.

In a tree schema, suppose every command in every transaction class has equality predicates on the primary key(s) of the root node (for example, in an e-commerce application, many commands will be rooted with a specific customer, so will include predicates like `customer_id` = 27). Using the horizontal partitioning discussed above, it is clear that in this case every SQL command in every transaction is local to one site. If, in addition, every command in each transaction class is limited to the same single site, then we call the application a *constrained tree application (CTA)*. A CTA application has the valuable feature that every transaction can be run to completion at a single site. The value of such *single-sited* transactions, as will be discussed in Section 4.3, is that transactions can execute without any stalls for communication with another grid site (however, in some cases, replicas will have to synchronize so that transactions are executed in the same order).

If every command in every transaction of a CTA specifies an equality match on the primary key(s) of one or more direct descendent nodes in addition to the equality predicate on the root, then the partitioning of a tree schema can be extended hierarchically to include these direct descendent nodes. In this case, a finer granularity partitioning can be used, if desired.

CTAs are an important class of single-sited applications which can be executed very efficiently. Our experience with many years of designing database applications in major corporations suggests that OLTP applications are often designed explicitly to be CTAs, or that decompositions to CTAs are often possible [Hel07]. Besides simply arguing that CTAs are prevalent, we are also interested in techniques that can be used to make non-CTA applications single-sited; it is an interesting research problem to precisely characterize the situations in which this is possible. We mention two possible schema transformations that can be systematically applied here.

First, consider all of the read-only tables in the schema, i.e. ones which are not updated by any transaction class. These tables can be replicated at all sites. If the application becomes CTA with these tables removed from consideration, then the application becomes single-sited after replication of the read-only tables.

Another important class of applications are *one-shot*. These applications have the property that all of their transactions can be executed in parallel without requiring intermediate results to be communicated among sites. Moreover, the result of previous SQL queries are never required in subsequent commands. In this case, each transaction can be decomposed into a collection of single-site plans which can be dispatched to the appropriate sites for execution.

Applications can often be made one-shot with vertical partitioning of tables amongst sites (columns that are not updated are replicated); this is true of TPC-C, for example (as we discuss in Section 5.)

Some transaction classes are *two-phase* (or can be made to be two phase.) In phase one there are a collection of read-only operations. Based on the result of these queries, the transaction may be aborted. Phase two then consists of a collection of queries and updates where there can be no possibility of an integrity violation. H-Store will exploit the two-phase property to eliminate the undo log. We have observed that many transactions, including those in TPC-C, are two-phase.

A transaction class is *strongly two-phase* if it is two-phase and additionally has the property that phase 1 operations on all replicas result in all replica sites aborting or all continuing.

Additionally, for every transaction class, we find all other classes whose members *commute* with members of the indicated class. Our specific definition of commutativity is:

> Two concurrent transactions from the same or different classes commute when any interleaving of their single-site sub-plans produces the same final database state as any other interleaving (assuming both transactions commit).

A transaction class which commutes with all transaction classes (including itself) will be termed *sterile*.

We use single-sited, sterile, two-phase, and strong two-phase properties in the H-Store algorithms, which follow. We have identified these properties as being particularly relevant based on our experience with major commercial online retail applications, and are confident that they will be found in many real world environments.

4 H-Store Sketch

In this section, we describe how H-Store exploits the previously described properties to implement a very efficient OLTP database.

4.1 System Architecture

H-Store runs on a grid of computers. All objects are partitioned over the nodes of the grid. Like C-Store [SAB+05], the user can specify the level of K-safety that he wishes to have.

At each site in the grid, rows of tables are placed contiguously in main memory, with conventional B-tree indexing. B-tree block size is tuned to the width of an L2 cache line on the machine being used. Although conventional B-trees can be beaten by cache conscious variations [RR99, RR00], we feel that this is an optimization to be performed only if indexing code ends up being a significant performance bottleneck.

Every H-Store site is single threaded, and performs incoming SQL commands to completion, without interruption. Each site is decomposed into a number of logical sites, one for each available core. Each logical site is considered an independent physical site, with its own indexes and tuple storage. Main memory on the physical site is partitioned among the logical sites. In this way, every logical site has a dedicated CPU and is single threaded.

In an OLTP environment most applications use stored procedures to cut down on the number of round trips between an application and the DBMS. Hence, H-Store has only one DBMS capability, namely to execute a predefined transaction (transactions may be issued from any site):

```
Execute transaction (parameter_list)
```

In the current prototype, stored procedures are written in C++, though we have suggestions on better languages in Section 6. Our implementation mixes application logic with direct manipulation of the database in the same process; this provides comparable performance to running the whole application inside a single stored procedure, where SQL calls are made as local procedure calls (not JDBC) and data is returned in a shared data array (again not JDBC). Like C-Store there is no redo log, and an undo log is written only if required, as discussed in Section 4.4. If written, the undo log is main memory resident, and discarded on transaction commit.

4.2 Query Execution

We expect to build a conventional cost-based query optimizer which produces query plans for the SQL commands in transaction classes at transaction definition time. We believe that this optimizer can be rather simple, as 6 way joins are never done in OLTP environments. If multi-way joins occur, they invariably identify a unique tuple of interest (say a purchase order number) and then the tuples that join to this record (such as the line items). Hence, invariably one proceeds from an *anchor tuple* through a small number of 1-to-*n* joins to the tuples of ultimate interest. GROUP BY and aggregation rarely occur in OLTP environments. The net result is, of course, a simple query execution plan.

The query execution plans for all commands in a transaction may be:

Single-sited: In this case the collection of plans can be dispatched to the appropriate site for execution.

One shot: In this case, all transactions can be decomposed into a set of plans that are executed only at a single site.

General: In the general case, there will be commands which require intermediate results to be communicated among sites in the grid. In addition, there may be commands whose run-time parameters are obtained from previous commands. In this case, we need the standard Gamma-style run time model of an *execution supervisor* at the site where the transaction enters the system, communicating with *workers* at the sites where data resides.

For general transactions, we compute the *depth* of the transaction class to be the number of times in the collection of plans, where a message must be sent between sites.

4.3 Database Designer

To achieve no-knobs operation, H-Store will build an automatic physical database designer which will specify horizontal partitioning, replication locations, and indexed fields.

In contrast to C-Store which assumed a world of overlapping materialized views appropriate in a read-mostly environment, H-Store implements the tables specified by the user and uses standard replication of user-specified tables to achieve HA. Most tables will be horizontally partitioned across all of the nodes in a grid. To achieve HA, such *table fragments* must have one or more *buddies*, which contain exactly the same information, possibly stored using a different physical representation (e.g., sort order).

The goal of the database designer is to make as many transaction classes as possible single-sited. The strategy to be employed is similar to the one used by C-Store [SAB+05]. That system constructed automatic designs for the omnipresent star or snowflake schemas in warehouse environments, and is now in the process of generalizing these algorithms for schemas that are "near snowflakes". Similarly, H-Store will construct automatic designs for the common case in OLTP environments (constrained tree applications), and will use the previously mentioned strategy of partitioning the database across sites based on the primary key of the root table and assigning tuples of other tables to sites based on root tuples they descend from. We will also explore extensions, such as optimizations for read-only tables and vertical partitioning mentioned in Section 3. It is a research task to see how far this approach can be pushed and how successful it will be.

In the meantime, horizontal partitioning and indexing options can be specified manually by a knowledgeable user.

4.4 Transaction Management, Replication and Recovery

Since H-Store implements two (or more) copies of each table, replicas must be transactionally updated. This is accomplished by directing each SQL read command to any replica and each SQL update to all replicas.

Moreover, every transaction receives a **timestamp** on entry to H-Store, which consists of a (site_id, local_unique_timestamp) pair. Given an ordering of sites, timestamps are unique and form a total order. We assume that the local clocks which generate local timestamps are kept nearly in sync with each other, using an algorithm like NTP [Mil89].

There are multiple situations which H-Store leverages to streamline concurrency control and commit protocols.

Single-sited/one shot. If all transaction classes are single-sited or one-shot, then individual transaction can be dispatched to the correct replica sites and executed to completion there. Unless all transaction classes are sterile, each execution site must wait a small period of time (meant to account for network delays) for transactions arriving from other initiators, so that the execution is in timestamp order. By increasing latency by a small amount, all replicas will by updated in the same order; in a local area network, maximum delays will be sub-millisecond. This will guarantee the identical outcome at each replica. Hence, data inconsistency between the replicas cannot occur. Also, all replicas will commit or all replicas will abort. Hence, each transaction can commit or abort locally, confident that the same outcome will occur at the other replicas. There is no redo log, no concurrency control, and no distributed commit processing.

Two-phase. No undo-log is required. Thus, if combined with the above properties, no transaction facilities are required at all.

Sterile. If all transaction classes are sterile, then execution can proceed normally with no concurrency control. Further, the need to issue timestamps and execute transactions in the same order on all replicas is obviated. However, if multiple sites are involved in query processing, then there is no guarantee that all sites will abort or all sites will continue. In this case, workers must respond "abort" or "continue" at the end of the first phase, and the execution supervisor must communicate this information to worker sites. Hence, standard commit distributed processing must be done at the end of phase one. This extra overhead can be avoided if the transaction is strongly two-phase.

Other cases. For other cases (non-sterile, non-single-sited, non one-shot), we need to endure the overhead of some sort of concurrency control scheme. All RDBMSs we are familiar with use dynamic locking to achieve transaction consistency. This decision followed pioneering simulation work in the 1980's [ACL87] that showed that locking worked better than other alternatives. However, we believe that dynamic locking is a poor choice for H-Store for the following reasons:

1. Transactions are very short-lived. There are no user-stalls and no disk activity. Hence, transactions are alive for very short time periods. This favors optimistic methods over pessimistic methods, like dynamic locking. Others, for example architects and programming language designers using transactions in memory models [HM93], have reached the same conclusion.
2. Every transaction is decomposed into collections of sub-commands, which are local to a given site. As noted earlier, the collection of sub commands are run in a single threaded fashion at each site. Again, this results in no latch waits, smaller total execution times, and again favors more optimistic methods.
3. We assume that we receive the entire collection of transaction classes in advance. This information can be used to advantage, as has been done previously by systems such as the SDD-1 scheme from the 1970's [BSR80] to reduce the concurrency control overhead.
4. In a well designed system there are **very few** transaction collisions and **very very few** deadlocks. These situations degrade performance and the workload is invariably modified by application designers to remove them. Hence, one should design for the "no collision" case, rather than using pessimistic methods.

The H-Store scheme takes advantage of these factors.

Every (non-sterile, non single-sited, non one-shot) transaction class has a collection of transaction classes with which it might conflict and arrives at some site in the grid and interacts with a transaction coordinator at that site. The transaction coordinator acts as the execution supervisor at the arrival site and sends out the subplan pieces to the various sites. A *worker* site receives a subplan and waits for the same small period of time mentioned above for other possibly conflicting transactions with lower timestamps to arrive. Then, the worker:

- Executes the subplan, if there is no uncommitted, potentially conflicting transaction at his site with a lower timestamp, and then sends his output data to the site requiring it, which may be an intermediate site or the transaction coordinator.
- Issues an abort to the coordinator otherwise

If the coordinator receives an "ok" from all sites, it continues with the transaction by issuing the next collection of subplans, perhaps with C++ logic interspersed. If there are no more subplans, then it commits the transaction. Otherwise, it aborts.

The above algorithm is the *basic* H-Store strategy. During execution, a transaction monitor watches the percentage of successful transactions. If there are too many aborts, H-Store dynamically moves to the following more sophisticated strategy.

Before executing or aborting the subplan, noted above, each worker site stalls by a length of time approximated by `MaxD * average_round_trip_message_delay` to see if a subplan with an earlier timestamp appears. If so, the worker site correctly sequences the subplans, thereby lowering the probability of abort. MaxD is the maximum depth of a conflicting transaction class.

This *intermediate* strategy lowers the abort probability, but at a cost of some number of msecs of increased latency. We are currently running simulations to demonstrate the circumstances under which this results in improved performance.

Our last *advanced* strategy keeps track of the read set and write set of each transaction at each site. In this case, a worker site runs each subplan, and then aborts the subplan if necessary according to standard optimistic concurrency control rules. At some extra overhead in bookkeeping and additional work discarded on aborts, the probability of conflict can be further reduced. Again, simulations are in progress to determine when this is a winning strategy.

In summary, our H-Store concurrency control algorithm is:

- Run sterile, single-sited and one-shot transactions with no controls
- Other transactions are run with the basic strategy
- If there are too many aborts, escalate to the intermediate strategy
- If there are still too many aborts, further escalate to the advanced strategy.

It should be noted that this strategy is a sophisticated optimistic concurrency control scheme. Optimistic methods have been extensively investigated previously [KR81, ACL87]. Moreover, the Ants DBMS [Ants07] leverages commutativity to lower locking costs. Hence, this section should be considered as a very low overhead consolidation of known techniques.

Notice that we have not yet employed any sophisticated scheduling techniques to lower conflict. For example, it is possible to run examples from all pairs of transaction classes and record the conflict frequency. Then, a scheduler could take this information into account, and try to avoid running transactions together with a high probability of conflict.

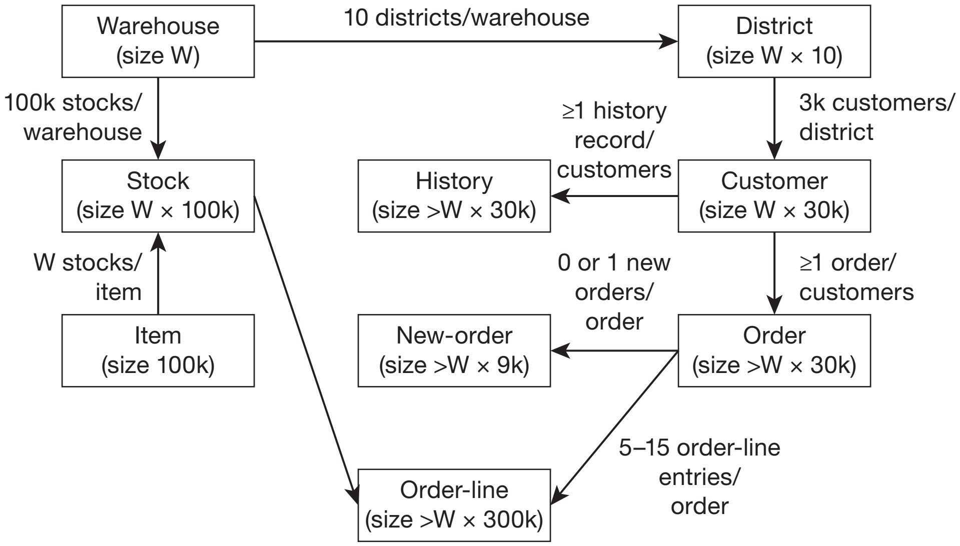


Figure 1 TPC-C Schema (reproduced from the TPC-C specification version 5.8.0, page 10)

The next section shows how these techniques and the rest of the H-Store design works on TPC-C.

5 A Performance Comparison

TPC-C runs on the schema diagramed in Figure 1, and contains 5 transaction classes (`new_order`, `payment`, `order status`, `delivery`, and `stock_level`).

Because of space limitations, we will not include the code for these transactions; the interested reader is referred to the TPC-C specification [TPCC]. Table 1 summarizes their behavior.

There are three possible strategies for an efficient H-Store implementation of TPC-C. First, we could run on a single core, single CPU machine. This automatically makes every transaction class single-sited, and each transaction can be run to completion in a single-threaded environment. The paired-HA site will achieve the same execution order, since, as will be seen momentarily, all transaction classes can be made strongly two-phase, meaning that all transactions will either succeed at both sites or abort at both sites. Hence, on a single site with a paired HA site, ACID properties are achieved with no overhead whatsoever. The other two strategies are for parallel operation on multi-core and/or multi-CPUs systems. They involve making the workload either sterile or one-shot, which, as we

Table 1 TPC-C Transaction Classes

new_order	Place an order for a customer. 90% of all orders can be supplied in full by stocks from the customer's "home" warehouse; 10% need to access stock belonging to a remote warehouse. Read/write transaction. No minimum percentage of mix required, but about 50% of transactions are new_order transactions.
payment	Updates the customer's balance and warehouse/district sales fields. 85% of updates go to customer's home warehouse; 15% to a remote warehouse. Read/write transaction. Must be at least 43% of transaction mix.
order_ status	Queries the status of a customer's last order. Read only. Must be at least 4% of transaction mix.
delivery	Select a warehouse, and for each of 10 districts "deliver" an order, which means removing a record from the new-order table and updating the customer's account balance. Each delivery can be a separate transaction; Must be at least 4% of transaction mix.
stock_level	Finds items with a stock level below a threshold; read only, must read committed data but does not need serializability. Must be at least 4% of transaction mix.

discussed in the previous section, are sufficient to allow us to run queries without conventional concurrency control. To do this, we will need to perform some trickery with the TPC-C workload; before describing this, we first address data partitioning.

TPC-C is not a tree-structured schema. The presence of the Item table as well as the relationship of Order-line with Stock make it a non-tree schema. The Item table, however, is read-only and can be replicated at each site. The Order-line table can be partitioned according to Warehouse to each site. With such replication and partitioning, the schema is decomposed such that each site has a subset of the records rooted at a distinct partition of the warehouses. This will be termed the *basic* H-Store strategy for partitioning and replication.

5.1 Query Classes

All transaction classes except new_order are already two-phase since they never need to abort. New_order may need to abort, since it is possible that its input contains invalid item numbers. However, it is permissible in the TPC-C specification to run a query for each item number at the beginning of the transaction to check for valid item numbers. By rearranging the transaction logic, all transaction classes become two-phase. It is also true that all transaction classes are strongly two-phase. This is because the Item table is never updated, and therefore all `new_order` transactions sent to all replicas always reach the same decision of whether to abort or not.

All 5 transaction classes appear to be sterile when considered with the basic partitioning and replication strategy. We make three observations in this regard.

First, the `new_order` transaction inserts a tuple in both the Orders table and New_Orders table as well as line items in the Line_order table. At each site, these operations will be part of a single sub-plan, and there will be no interleaved operations. This will ensure that the `order_status` transaction does not see partially completed new orders. Second, because `new_order` and `payment` transactions in TPC-C are strongly two-phase, no additional coordination is needed between sites in the event that one of these transactions updates a "remote" warehouse relative to the customer making the order or payment.

Third, the `stock_level` transaction is allowed to run as multiple transactions which can see stock levels for different items at different points in time, as long as the stock level results from committed transactions. Because `new_orders` are aborted, if necessary, before they perform any updates, any stock information read comes from committed transactions (or transactions that will be committed soon).

Hence, all transaction classes can be made sterile and strongly two-phase. As such, they achieve a valid execution of TPC-C with no concurrency control. Although we could have tested this configuration, we decided to employ additional manipulation of the workload to also make all transaction classes one-shot, doing so improves performance.

With the basic strategy, all transaction classes, except `new_order` and `payment` are single-sited, and therefore one-shot. `Payment` is already one shot, since there is no need to exchange data when updating a remote warehouse. `New_order`, however, needs to insert in Order-line information about the district of a stock entry which may reside in a remote site. Since that field is never updated, and there are no deletes/inserts into the Stock table, we can vertically partition Stock and replicate the read-only parts of it across all sites. With this replication trick added to the basic strategy, `new_order` becomes one shot.

As a result, with the basic strategy augmented with the tricks described above, all transaction classes become one-shot and strongly two-phase. As long as we add a short delay as mentioned in Section 4.4, ACID properties are achieved with no concurrency control overhead whatsoever. This is the configuration on which benchmark results are reported in Section 5.3

It is difficult to imagine that an automatic program could figure out what is required to make TPC-C either one-shot or sterile. Hence, a knowledgeable human would have to carefully code the transactions classes. It is likely, however, that most transaction classes will be simpler to analyze. As such, it is an open question how successful automatic transaction class analysis will be.

5.2 Implementation

We implemented a variant of TPC-C on H-Store and on a very popular commercial RDBMS. The same driver was used for both systems and generated transactions at the maximum rate without modeling think time. These transactions were delivered to both systems using TCP/IP. All transaction classes were implemented as stored procedures. In H-Store the transaction logic was coded in C++, with local procedure calls to H-Store query execution. In contrast, the transaction logic for the commercial system was written using their proprietary stored procedure language. High availability and communication with user terminals was not included for either system.

Both DBMSs were run on a dual-core 2.8GHz CPU computer system, with 4 Gbytes of main memory and four 250 GB SATA disk drives. Both DBMSs used horizontal partitioning to advantage.

5.3 Results

On this configuration, H-Store ran 70,416 TPC-C transactions per second. In contrast, we could only coax 850 transactions per second from the commercial system, in spite of several days of tuning by a professional DBA, who specializes in this vendor's product. Hence, H-Store ran a factor of 82 faster (almost two orders of magnitude).

Per our earlier discussion, the bottleneck for the commercial system was logging overhead. That system spent about 2/3 of its total elapsed time inside the logging system. One of us spent many hours trying to tune the logging system (log to a dedicated disk, change the size of the group commit; all to no avail). If logging was turned off completely, and assuming no other bottleneck creeps up, then throughput would increase to about 2,500 transactions per second.

The next bottleneck appears to be the concurrency control system. In future experiments, we plan to tease apart the overhead contributions which result from:

- Redo logging
- Undo logging

- Latching
- Locking

Finally, though we did not implement all of the TPC-C specification (we did not, for example, model wait times), it is also instructive to compare our partial TPC-C implementation with TPC-C performance records on the TPC website.[2] The highest performing TPC-C implementation executes about 4 million new-order transactions per minute, or a total of about 133,000 total transactions per second. This is on a 128 core shared memory machine, so this implementation is getting about 1000 transactions per core. Contrast this with 400 transactions per core in our benchmark on a commercial system on a (rather pokey) desktop machine, or 35,000 transactions per core in H-Store! Also, note that H-Store is within a factor of two of the best TPC-C results on a machine costing around $1000.00

In summary, the conclusion to be reached is that nearly two orders of magnitude in performance improvement are available to a system designed along the lines of H-Store.

6 Some Comments about a "One Size Does Not Fit All" World

If the results of this paper are to be believed, then we are heading toward a world with at least 5 (and probably more) specialized engines and the death of the "one size fits all" legacy systems. This section considers some of the consequences of such an architectural shift.

6.1 The Relational Model Is not Necessarily the Answer

Having survived the great debate of 1974 [Rus74] and the surrounding arguments between the advocates of the Codasyl and relational models, we are reluctant to bring up this particular "sacred cow". However, it seems appropriate to consider the data model (or data models) that we build systems around. In the 1970's the DBMS world contained only business data processing applications, and Ted Codd's idea of normalizing data into flat tables has served our community well over the subsequent 30 years. However, there are now other markets, whose needs must be considered. These include data warehouses, web-oriented search, real-time analytics, and semi-structured data markets.

We offer the following observations.

[2] http://www.tcp.org/tpcc/results/tpcc_perf_results.asp

1. In the data warehouse market, nearly 100% of all schemas are stars or snowflakes, containing a central fact table with 1-n joins to surrounding dimension tables, which may in turn participate in further 1-n joins to second level dimension tables, and so forth. Although stars and snowflakes are easily modeled using relational schemas, in fact, an entity-relationship model would be simpler in this environment and more natural. Moreover, warehouse queries would be simpler in an E-R model. Lastly, warehouse operations that are incredibly expensive with a relational implementation, for example changing the key of a row in a dimension table, might be made faster with some sort of E-R implementation.

2. In the stream processing market, there is a need to:

    (a) Process streams of messages at high speed

    (b) Correlate such streams with stored data

    To accomplish both tasks, there is widespread enthusiasm for StreamSQL, a generalization of SQL that allows a programmer to mix stored tables and streams in the FROM clause of a SQL statement. This work has evolved from the pioneering work of the Stanford Stream group [ABW06] and is being actively discussed for standardization. Of course, StreamSQL supports relational schemas for both tables and streams.

    However, commercial feeds, such as Reuters, Infodyne, etc., have all chosen some data model for their messages to obey. Some are flat and fit nicely into a relational schema. However, several are hierarchical, such as the FX feed for foreign exchange. Stream processing systems, such as StreamBase and Coral8, currently support only flat (relational) messages. In such systems, a front-end adaptor must normalize hierarchical objects into several flat message types for processing. Unfortunately, it is rather painful to join the constituent pieces of a source message back together when processing on multiple parts of a hierarchy is necessary.

    To solve this problem, we expect the stream processing vendors to move aggressively to hierarchical data models. Hence, they will assuredly deviate from Ted Codd's principles.

3. Text processing obviously has never used a relational model.

4. Any scientific-oriented DBMS, such as ASAP [SBC+07], will probably implement arrays, not tables as their basic data type.

5. There has recently been considerable debate over good data models for semi-structured data. There is certainly fierce debate over the excessive complexity

of XMLSchema [SC05]. There are fans of using RDF for such data [MM04], and some who argue that RDF can be efficiently implemented by a relational column store [AMM+07]. Suffice it to say that there are many ideas on which way to go in this area.

In summary, the relational model was developed for a "one size fits all" world. The various specialized systems which we envision can each rethink what data model would work best for their particular needs.

6.2 SQL is Not the Answer

SQL is a "one size fits all" language. In an OLTP world one never asks for the employees who earn more than their managers. In fact, there are no ad-hoc queries, as noted earlier. Hence, one can implement a smaller language than SQL. For performance reasons, stored procedures are omni-present. In a data warehouse world, one needs a different subset of SQL, since there are complex ad-hoc queries, but no stored procedures. Hence, the various storage engines can implement vertical-market specific languages, which will be simpler than the daunting complexity of SQL.

Rethinking how many query languages should exist as well as their complexity will have a huge side benefit. At this point SQL is a legacy language with many known serious flaws, as noted by Chris Date two decades ago [Dat84]. Next time around, we can do a better job.

When rethinking data access languages, we are reminded of a raging discussion from the 1970's. On the one-hand, there were advocates of a data sublanguage, which could be interfaced to any programming language. This has led to high overhead interfaces, such as JDBC and ODBC. In addition, these interfaces are very difficult to use from a conventional programming language.

In contrast, some members of the DBMS community proposed much nicer embedding of database capabilities in programming languages, typified in the 1970s by Pascal R [Sch80] and Rigel [RS79]. Both had clean integration with programming language facilities, such as control flow, local variables, etc. Chris Date also proposed an extension to PL/1 with the same purpose [Dat76].

Obviously none of these languages ever caught on, and the data sublanguage camp prevailed. The couplings between a programming language and a data sublanguage that our community has designed are ugly beyond belief and are low productivity systems that date from a different era. Hence, we advocate scrapping sublanguages completely, in favor of much cleaner language embeddings.

In the programming language community, there has been an explosion of "little languages" such as Python, Perl, Ruby and PHP. The idea is that one should use the best language available for any particular task at hand. Also little languages are attractive because they are easier to learn than general purpose languages. From afar, this phenomenon appears to be the death of "one size fits all" in the programming language world.

Little languages have two very desirable properties. First, they are mostly open source, and can be altered by the community. Second they are less daunting to modify than the current general purpose languages. As such, we are advocates of modifying little languages to include clean embeddings of DBMS access.

Our current favorite example of this approach is Ruby-on-Rails.[3] This system is the little language, Ruby, extended with integrated support for database access and manipulation through the "model-view-controller" programming pattern. Ruby-on-Rails compiles into standard JDBC, but hides all the complexity of that interface.

Hence, H-Store plans to move from C++ to Ruby-on-Rails as our stored procedure language. Of course, the language run-time must be linked into the DBMS address space, and must be altered to make calls to DBMS services using high performance local procedure calls, not JDBC.

7 Summary and Future Work

In the last quarter of a century, there has been a dramatic shift in:

1. DBMS markets: from business data processing to a collection of markets, with varying requirements
2. Necessary features: new requirements include shared nothing support and high availability
3. Technology: large main memories, the possibility of hot standbys, and the web change most everything

The result is:

1. The predicted demise of "one size fits all"
2. The inappropriateness of current relational implementations for any segment of the market

[3] http://www.rubyonrails.org

3. The necessity of rethinking both data models and query languages for the specialized engines, which we expect to be dominant in the various vertical markets

Our H-Store prototype demonstrates the performance gains that can be had when this conventional thinking is questioned. Of course, beyond these encouraging initial performance results, there are a number of areas where future work is needed. In particular:

- More work is needed to identify when it is possible to automatically identify single-sited, two-phase, and one-shot applications. "Auto-everything" tools that can suggest partitions that lead to these properties are also essential.
- The rise of multi-core machines suggests that there may be interesting optimizations related to sharing of work between logical sites physically co-located on the same machine.
- A careful study of the performance of the various transaction management strategies outlined in Section 3 is needed.
- A study of the overheads of the various components of a OLTP system—logging, transaction processing and two-phase commit, locking, JDBC/ODBC, etc—would help identify which aspects of traditional DBMS design contribute most to the overheads we have observed.
- After stripping out all of these overheads, our H-Store implementation is now limited by the performance of in-memory data structures, suggesting that optimizing these structures will be important. For example, we found that the simple optimization of representing read-only tables as arrays offered significant gains in transaction throughput in our H-Store implementation.
- Integration with data warehousing tools—for example, by using no-overwrite storage and occasionally dumping records into a warehouse—will be essential if H-Store-like systems are to seamlessly co-exist with data warehouses.

In short, the current situation in the DBMS community reminds us of the period 1970-1985 where there was a "group grope" for the best way to build DBMS engines and dramatic changes in commercial products and DBMS vendors ensued. The 1970-1985 period was a time of intense debate, a myriad of ideas, and considerable upheaval.

We predict the next fifteen years will have the same feel.

References

[ABW06] A. Arasu, S. Babu, and J. Widom. "The CQL Continuous Query Language: Semantic Foundations and Query Execution." *The VLDB Journal*, 15(2), June 2006.

[ACL87] R. Agrawal, M. J. Carey, and M. Livny. "Concurrency control performance modeling: alternatives and implications." *ACM Trans. Database Syst. 12(4)*, Nov. 1987.

[AMM+07] D. Abadi, A. Marcus, S. Madden, and K. Hollenbach. "Scalable Semantic Web Data Management Using Vertical Partitioning." In *Proc. VLDB*, 2007.

[Ants07] ANTs Software. ANTs Data Server-Technical White Paper, http://www.ants.com, 2007.

[BSR80] P. A. Bernstein, D. Shipman, and J. B. Rothnie. "Concurrency Control in a System for Distributed Databases (SDD-1)." *ACM Trans. Database Syst. 5(1)*, March 1980.

[Bon02] P. A. Boncz. "Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications." Ph.D. Thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, May 2002.

[Dat76] C. J. Date. "An Architecture for High-Level Language Database Extensions." In *Proc. SIGMOD*, 1976.

[Dat84] C. J. Date. "A critique of the SQL database language." In *SIGMOD Record* 14(3):8-54, Nov. 1984.

[DGS+90] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. "The Gamma Database Machine Project." *IEEE Transactions on Knowledge and Data Engineering 2(1):44-62*, March 1990.

[Hel07] P. Helland. "Life beyond Distributed Transactions: an Apostate's Opinion." In *Proc. CIDR*, 2007.

[HM93] M. Herlihy and J. E. Moss. "Transactional memory: architectural support for lock-free data structures." In *Proc. ISCA*, 1993.

[KL81] H. T. Kung and J. T. Robinson. "On optimistic methods for concurrency control." *ACM Trans. Database Syst. 6(2):213–226*, June 1981.

[LM06] E. Lau and S. Madden. "An Integrated Approach to Recovery and High Availability in an Updatable, Distributed Data Warehouse." In *Proc. VLDB*, 2006.

[MHL+92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging." *ACM Trans. Database Syst. 17(1):94-162*, March 1992.

[Mil89] D. L. Mills. "On the Accuracy and Stability of Clocks Synchronized by the Network Time Protocol in the Internet System." *SIGCOMM Comput. Commun. Rev. 20(1):65-75*, Dec. 1989.

[MM04] F. Manola and E. Miller, (eds). RDF Primer. W3C Specification, February 10, 2004. http://www.w3.org/TR/REC-rdf-primer-20040210/

[RR99] J. Rao and K. A. Ross. "Cache Conscious Indexing for Decision-Support in Main Memory." In *Proc. VLDB*, 1999.

[RR00] J. Rao and K. A. Ross. “Making B+-trees cache conscious in main memory.” In *SIGMOD Record, 29(2):475-486*, June 2000.

[RS79] L. A. Rowe and K. A. Shoens. “Data Abstractions, Views and Updates in RIGEL.” In *Proc. SIGMOD*, 1979.

[Rus74] Randall Rustin (Ed.): Proceedings of 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Michigan, May 1-3, 1974, 2 Volumes.

[SAB+05] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O’Neil, P. O’Neil, A. Rasin, N. Tran, and S. Zdonik. “C-Store: A Column-oriented DBMS.” In *Proc. VLDB*, 2005.

[SBC+07] M. Stonebraker, C. Bear, U. Cetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. Zdonik. “One Size Fits All?-Part 2: Benchmarking Results.” In *Proc. CIDR*, 2007.

[SC05] M. Stonebraker and U. Cetintemel. “One Size Fits All: An Idea whose Time has Come and Gone.” In *Proc. ICDE*, 2005.

[Sch80] J. W. Schmidt, et al. “Pascal/R Report.” U Hamburg, Fachbereich Informatik, Report 66, Jan 1980.

[TPCC] The Transaction Processing Council. TPC-C Benchmark (Revision 5.8.0), 2006. http://www.tpc.org/tpcc/spec/tpcc_current.pdf

C-Store: A Column-Oriented DBMS

Mike Stonebraker (MIT CSAIL), **Daniel J. Abadi** (MIT CSAIL), **Adam Batkin** (Brandeis University), **Xuedong Chen** (UMass Boston), **Mitch Cherniack** (Brandeis University), **Miguel Ferreira** (MIT CSAIL), **Edmond Lau** (MIT CSAIL), **Amerson Lin** (MIT CSAIL), **Sam Madden** (MIT CSAIL), **Elizabeth O'Neil** (UMass Boston), **Pat O'Neil** (UMass Boston), **Alex Rasin** (Brown University), **Nga Tran** (Brandeis University), **Stan Zdonik** (Brown University)

Abstract

This paper presents the design of a read-optimized relational DBMS that contrasts sharply with most current systems, which are write-optimized. Among the many differences in its design are: storage of data by column rather than by row, careful coding and packing of objects into storage including main memory during query processing, storing an overlapping collection of column-oriented projections, rather than the current fare of tables and indexes, a non-traditional implementation of transactions which includes high availability and snapshot isolation for read-only transactions, and the extensive use of bitmap indexes to complement B-tree structures.

We present preliminary performance data on a subset of TPC-H and show that the system we are building, C-Store, is substantially faster than popular commercial products. Hence, the architecture looks very encouraging.

1 Introduction

Most major DBMS vendors implement record-oriented storage systems, where the attributes of a record (or tuple) are placed contiguously in storage. With this *row store* architecture, a single disk write suffices to push all of the fields of a single record out to disk. Hence, high performance writes are achieved, and we call a DBMS with a row store architecture a *write-optimized* system. These are especially effective on OLTP-style applications.

In contrast, systems oriented toward ad-hoc querying of large amounts of data should be *read-optimized*. Data warehouses represent one class of read-optimized system, in which periodically a bulk load of new data is performed, followed by a relatively long period of ad-hoc queries. Other read-mostly applications include customer relationship management (CRM) systems, electronic library card catalogs, and other ad-hoc inquiry systems. In such environments, a *column store* architecture, in which the values for each single column (or attribute) are stored contiguously, should be more efficient. This efficiency has been demonstrated in the warehouse marketplace by products like Sybase IQ [FREN95, SYBA04], Addamark [ADDA04], and KDB [KDB04]. In this paper, we discuss the design of a column store called C-Store that includes a number of novel features relative to existing systems.

With a column store architecture, a DBMS need only read the values of columns required for processing a given query, and can avoid bringing into memory irrelevant attributes. In warehouse environments where typical queries involve aggregates performed over large numbers of data items, a column store has a sizeable performance advantage. However, there are several other major distinctions that can be drawn between an architecture that is read-optimized and one that is write-optimized.

Current relational DBMSs were designed to pad attributes to byte or word boundaries and to store values in their native data format. It was thought that it was too expensive to shift data values onto byte or word boundaries in main memory for processing. However, CPUs are getting faster at a much greater rate than disk bandwidth is increasing. Hence, it makes sense to trade CPU cycles, which are abundant, for disk bandwidth, which is not. This tradeoff appears especially profitable in a read-mostly environment.

There are two ways a column store can use CPU cycles to save disk bandwidth. First, it can code data elements into a more compact form. For example, if one is storing an attribute that is a customer's state of residence, then US states can be coded into six bits, whereas the two-character abbreviation requires 16 bits and a variable length character string for the name of the state requires many more. Second, one should *densepack* values in storage. For example, in a column store it is straightforward to pack N values, each K bits long, into N * K bits. The coding and compressibility advantages of a column store over a row store have been previously pointed out in [FREN95]. Of course, it is also desirable to have the DBMS query executor operate on the compressed representation whenever possible to avoid the cost of decompression, at least until values need to be presented to an application.

Commercial relational DBMSs store complete tuples of tabular data along with auxiliary B-tree indexes on attributes in the table. Such indexes can be *primary*, whereby the rows of the table are stored in as close to sorted order on the specified attribute as possible, or *secondary*, in which case no attempt is made to keep the underlying records in order on the indexed attribute. Such indexes are effective in an OLTP write-optimized environment but do not perform well in a read-optimized world. In the latter case, other data structures are advantageous, including bit map indexes [ONEI97], cross table indexes [ORAC04], and materialized views [CERI91]. In a read-optimized DBMS one can explore storing data using only these read-optimized structures, and not support write-optimized ones at all.

Hence, C-Store physically stores a collection of columns, each sorted on some attribute(s). Groups of columns sorted on the same attribute are referred to as "projections"; the same column may exist in multiple projections, possibly sorted on a different attribute in each. We expect that our aggressive compression techniques will allow us to support many column sort-orders without an explosion in space. The existence of multiple sort-orders opens opportunities for optimization.

Clearly, collections of off-the-shelf "blade" or "grid" computers will be the cheapest hardware architecture for computing and storage intensive applications such as DBMSs [DEWI92]. Hence, any new DBMS architecture should assume a grid environment in which there are G nodes (computers), each with private disk and private memory. We propose to horizontally partition data across the disks of the various nodes in a "shared nothing" architecture [STON86]. Grid computers in the near future may have tens to hundreds of nodes, and any new system should be architected for grids of this size. Of course, the nodes of a grid computer may be physically co-located or divided into clusters of co-located nodes. Since database administrators are hard pressed to optimize a grid environment, it is essential to allocate data structures to grid nodes automatically. In addition, intra-query parallelism is facilitated by horizontal partitioning of stored data structures, and we follow the lead of Gamma [DEWI90] in implementing this construct.

Many warehouse systems (e.g. Walmart [WEST00]) maintain two copies of their data because the cost of recovery via DBMS log processing on a very large (terabyte) data set is prohibitive. This option is rendered increasingly attractive by the declining cost per byte of disks. A grid environment allows one to store such replicas on different processing nodes, thereby supporting a Tandem-style highly-available system [TAND89]. However, there is no requirement that one store multiple copies in the exact same way. C-Store allows redundant objects to be stored in different sort orders providing higher retrieval performance in addition to high availability. In general, storing overlapping projections further improves performance, as long as redundancy is crafted so that all data can be accessed even if one of the G sites fails. We call a system that tolerates K failures *K-safe*. C-Store will be configurable to support a range of values of K.

It is clearly essential to perform transactional updates, even in a read-mostly environment. Warehouses have a need to perform on-line updates to correct errors. As well, there is an increasing push toward real-time warehouses, where the delay to data visibility shrinks toward zero. The ultimate desire is on-line update to data warehouses. Obviously, in read-mostly worlds like CRM, one needs to perform general on-line updates.

There is a tension between providing updates and optimizing data structures for reading. For example, in KDB and Addamark, columns of data are maintained in entry sequence order. This allows efficient insertion of new data items, either in batch or transactionally, at the end of the column. However, the cost is a less-than-optimal retrieval structure, because most query workloads will run faster with the data in some other order. However, storing columns in non-entry sequence will make insertions very difficult and expensive.

C-Store approaches this dilemma from a fresh perspective. Specifically, we combine in a single piece of system software, both a read-optimized column store and an update/insert-oriented writeable store, connected by a *tuple mover*, as noted in Figure 1. At the top level, there is a small Writeable Store (WS) component, which is architected to support high performance inserts and updates. There is also a much larger component called the Read-optimized Store (RS), which is capable of supporting very large amounts of information. RS, as the name implies, is optimized for read and supports only a very restricted form of insert, namely the batch movement of records from WS to RS, a task that is performed by the tuple mover of Figure 1.

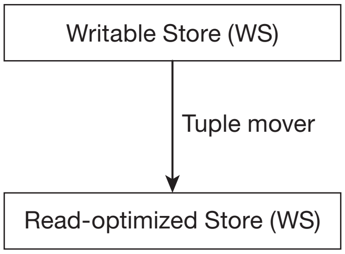


Figure 1 Architecture of C-Store

Of course, queries must access data in both storage systems. Inserts are sent to WS, while deletes must be marked in RS for later purging by the tuple mover. Updates are implemented as an insert and a delete. In order to support a high-speed tuple mover, we use a variant of the LSM-tree concept [ONEI96], which supports a *merge out* process that moves tuples from WS to RS in bulk by an efficient method of merging ordered WS data objects with large RS blocks, resulting in a new copy of RS that is installed when the operation completes.

The architecture of Figure 1 must support transactions in an environment of many large ad-hoc queries, smaller update transactions, and perhaps continuous inserts. Obviously, blindly supporting dynamic locking will result in substantial read-write conflict and performance degradation due to blocking and deadlocks.

Instead, we expect read-only queries to be run in *historical* mode. In this mode, the query selects a timestamp, T, less than the one of the most recently committed transactions, and the query is semantically guaranteed to produce the correct answer as of that point in history. Providing such *snapshot isolation* [BERE95] requires C-Store to timestamp data elements as they are inserted and to have careful programming of the runtime system to ignore elements with timestamps later than T.

Lastly, most commercial optimizers and executors are row-oriented, obviously built for the prevalent row stores in the marketplace. Since both RS and WS are column-oriented, it makes sense to build a column-oriented optimizer and executor. As will be seen, this software looks nothing like the traditional designs prevalent today.

In this paper, we sketch the design of our updatable column store, C-Store, that can simultaneously achieve very high performance on warehouse-style queries and achieve reasonable speed on OLTP-style transactions. C-Store is a column-oriented DBMS that is architected to reduce the number of disk accesses per query. The innovative features of C-Store include:

1. A hybrid architecture with a WS component optimized for frequent insert and update and an RS component optimized for query performance.
2. Redundant storage of elements of a table in several overlapping projections in different orders, so that a query can be solved using the most advantageous projection.
3. Heavily compressed columns using one of several coding schemes.
4. A column-oriented optimizer and executor, with different primitives than in a row-oriented system.
5. High availability and improved performance through K-safety using a sufficient number of overlapping projections.
6. The use of snapshot isolation to avoid 2PC and locking for queries.

It should be emphasized that while many of these topics have parallels with things that have been studied in isolation in the past, it is their combination in a real system that make C-Store interesting and unique.

The rest of this paper is organized as follows. In Section 2 we present the data model implemented by C-Store. We explore in Section 3 the design of the RS portion of C-Store, followed in Section 4 by the WS component. In Section 5 we consider the allocation of C-Store data structures to nodes in a grid, followed by a presentation of C-Store updates and transactions in Section 6. Section 7 treats the tuple mover component of C-Store, and Section 8 presents the query optimizer and executor. In Section 9 we present a comparison of C-Store performance to that achieved by both a popular commercial row store and a popular commercial column store. On TPC-H style queries, C-Store is significantly faster than either alternate system. However, it must be noted that the performance comparison is not fully completed; we have not fully integrated the WS and tuple mover, whose overhead may be significant. Finally, Sections 10 and 11 discuss related previous work and our conclusions.

2 Data Model

C-Store supports the standard relational *logical data model*, where a database consists of a collection of named tables, each with a named collection of attributes (columns). As in most relational systems, attributes (or collections of attributes) in C-Store tables can form a unique *primary key* or be a *foreign key* that references a primary key in another table. The C-Store query language is assumed to be SQL, with standard SQL semantics. Data in C-Store is not physically stored using this logical data model. Whereas most row stores implement physical tables directly and then add various indexes to speed access, C-Store implements only *projections*.

Table 1 Sample EMP data

Name	Age	Dept	Salary
Bob	25	Math	10K
Bill	27	EECS	50K
Jill	24	Biology	80K

Specifically, a C-Store projection is *anchored* on a given logical table, T, and contains one or more attributes from this table. In addition, a projection can contain any number of other attributes from other tables, as long as there is a sequence of n:1 (*i.e.*, foreign key) relationships from the anchor table to the table containing an attribute.

To form a projection, we project the attributes of interest from T, retaining any duplicate rows, and perform the appropriate sequence of value-based foreign-key joins to obtain the attributes from the non-anchor table(s). Hence, a projection has the same number of rows as its anchor table. Of course, much more elaborate projections could be allowed, but we believe this simple scheme will meet our needs while ensuring high performance. We note that we use the term projection slightly differently than is common practice, as we do not store the base table(s) from which the projection is derived.

We denote the *ith* projection over table *t* as *ti*, followed by the names of the fields in the projection. Attributes from other tables are prepended with the name of the logical table they come from. In this section, we consider an example for the standard EMP(name, age, salary, dept) and DEPT(dname, floor) relations. Sample EMP data is shown in Table 1. One possible set of projections for these tables could be as shown in Example 1.

```
EMP1  (name, age)
EMP2  (dept, age, DEPT.floor)
EMP3  (name, salary)
DEPT1 (dname, floor)
```

Example 1: Possible projections for EMP and DEPT.

Tuples in a projection are stored column-wise. Hence, if there are K attributes in a projection, there will be K data structures, each storing a single column, each of which is sorted on the same *sort key*. The sort key can be any column or columns in the projection. Tuples in a projection are sorted on the key(s) in left to right order.

We indicate the sort order of a projection by appending the sort key to the projection separated by a vertical bar. A possible ordering for the above projections would be:

```
EMP1  (name, age| age)
EMP2  (dept, age, DEPT.floor| DEPT.floor)
EMP3  (name, salary| salary)
DEPT1 (dname, floor| floor)
```

Example 2: Projections in Example 1 with sort orders.

Lastly, every projection is *horizontally partitioned* into 1 or more *segments*, which are given a *segment identifier*, Sid, where Sid > 0. C-Store supports only value-based partitioning on the sort key of a projection. Hence, each segment of a given projection is associated with a *key range* of the sort key for the projection. Moreover, the set of all key ranges *partitions* the key space.

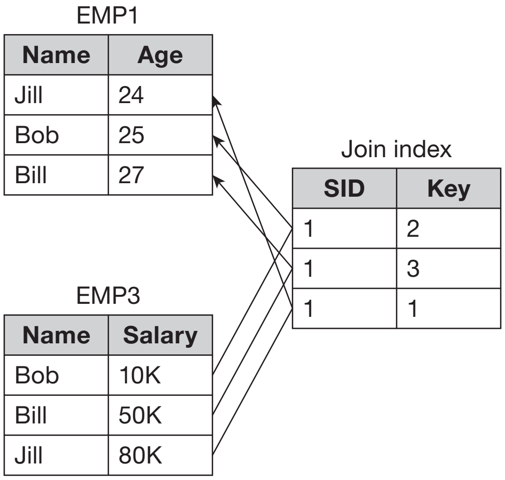
Clearly, to answer any SQL query in C-Store, there must be a *covering set* of projections for every table in the database such that every column in every table is stored in at least one projection. However, C-Store must also be able to reconstruct complete rows of tables from the collection of stored segments. To do this, it will need to join segments from different projections, which we accomplish using *storage keys* and *join indexes*.

Storage Keys. Each segment associates every data value of every column with a storage key, SK. Values from different columns in the same segment with matching storage keys belong to the same logical row. We refer to a row of a segment using the term *record* or *tuple*. Storage keys are numbered 1, 2, 3, . . . in RS and are not physically stored, but are inferred from a tuple's physical position in the column (see Section 3 below.) Storage keys are physically present in WS and are represented as integers, larger than the largest integer storage key for any segment in RS.

Join Indices. To reconstruct all of the records in a table T from its various projections, C-Store uses *join indexes*. If *T1* and *T2* are two projections that cover a table *T*, a join index from the M segments in *T1* to the N segments in *T2* is logically a collection of M tables, one per segment, S, of *T1* consisting of rows of the form:

```
(s: SID in T2, k: Storage Key in Segment s)
```

Here, an entry in the join index for a given tuple in a segment of *T1* contains the segment ID and storage key of the corresponding (joining) tuple in *T2*. Since all join indexes are between projections anchored at the same table, this is always a one-to-one mapping. An alternative view of a join index is that it takes *T1*, sorted in some order O, and logically resorts it into the order, O' of *T2*.



EMP1

Name	Age
Jill	24
Bob	25
Bill	27

EMP3

Name	Salary
Bob	10K
Bill	50K
Jill	80K

Join index

SID	Key
1	2
1	3
1	1

Figure 2 A join index from EMP3 to EMP1.

In order to reconstruct *T* from the segments of *T1*, . . . , *Tk* it must be possible to find a *path* through a set of join indices that maps each attribute of *T* into some sort order O*. A path is a collection of join indexes originating with a sort order specified by some projection, *Ti*, that passes through zero or more intermediate join indices and ends with a projection sorted in order O*. For example, to be able to reconstruct the EMP table from projections in Example 2, we need at least two join indices. If we choose age as a common sort order, we could build two indices that map EMP2 and EMP3 to the ordering of EMP1. Alternatively, we could create a join index that maps EMP2 to EMP3 and one that maps EMP3 to EMP1. Figure 2 shows a simple example of a join index that maps EMP3 to EMP1, assuming a single segment (SID = 1) for each projection. For example, the first entry of EMP3, (Bob, 10K), corresponds to the second entry of EMP1, and thus the first entry of the join index has storage key 2. In practice, we expect to store each column in several projections, thereby allowing us to maintain relatively few join indices. This is because join indexes are very expensive to store and maintain in the presence of updates, since each modification to a projection requires every join index that points into or out of it to be updated as well.

The segments of the projections in a database and their connecting join indexes must be allocated to the various nodes in a C-Store system. The C-Store administrator can optionally specify that the tables in a database must be *K-safe*. In this case, the loss of K nodes in the grid will still allow all tables in a database to be reconstructed (i.e., despite the K failed sites, there must exist a covering set of projections and a set of join indices that map to some common sort order.) When a failure occurs, C-Store simply continues with K-1 safety until the failure is repaired and the node is brought back up to speed. We are currently working on fast algorithms to accomplish this.

Thus, the C-Store physical DBMS design problem is to determine the collection of projections, segments, sort keys, and join indices to create for the collection of logical tables in a database. This physical schema must give K-safety as well as the best overall performance for a given *training workload*, provided by the C-Store administrator, subject to requiring no more than a given *space budget*, B. Additionally, C-Store can be instructed to keep a log of all queries to be used periodically as the training workload. Because there are not enough skilled DBAs to go around, we are writing an automatic schema design tool. Similar issues are addressed in [PAPA04]

We now turn to the representation of projections, segments, storage keys, and join indexes in C-Store.

3 RS

RS is a read-optimized column store. Hence any segment of any projection is broken into its constituent columns, and each column is stored in order of the sort key for the projection. The storage key for each tuple in RS is the ordinal number of the record in the segment. This storage key is not stored but calculated as needed.

3.1 Encoding Schemes

Columns in the RS are compressed using one of 4 encodings. The encoding chosen for a column depends on its *ordering* (i.e., is the column ordered by values in that column (self-order) or by corresponding values of some other column in the same projection (foreign-order), and the proportion of *distinct values* it contains. We describe these encodings below.

Type 1: Self-order, few distinct values. A column encoded using Type 1 encoding is represented by a sequence of triples, (v, f, n) such that v is a value stored in the column, f is the position in the column where v first appears, and n is the number of times v appears in the column. For example, if a group of 4's appears in positions 12-18, this is captured by the entry, (4, 12, 7). For columns that are self-ordered, this requires one triple for each distinct value in the column. To support search queries over values in such columns, Type 1-encoded columns have clustered B-tree indexes over their value fields. Since there are no online updates to RS, we can *densepack* the index leaving no empty space. Further, with large disk blocks (e.g., 64-128K), the height of this index can be kept small (e.g., 2 or less).

Type 2: Foreign-order, few distinct values. A column encoded using Type 2 encoding is represented by a sequence of tuples, (v, b) such that v is a value stored in the column and b is a bitmap indicating the positions in which the value is stored. For example, given a column of integers 0,0,1,1,2,1,0,2,1, we can Type 2-encode this as three pairs: (0, 110000100), (1, 001101001), and (2,000010010). Since each bitmap is sparse, it is run length encoded to save space. To efficiently find the i-th value of a type 2-encoded column, we include "offset indexes": B-trees that map positions in a column to the values contained in that column.

Type 3: Self-order, many distinct values. The idea for this scheme is to represent every value in the column as a delta from the previous value in the column. Thus, for example, a column consisting of values 1,4,7,7,8,12 would be represented by the sequence: 1,3,3,0,1,4, such that the first entry in the sequence is the first value in the column, and every subsequent entry is a delta from the previous value. Type-3 encoding is a block-oriented form of this compression scheme, such that the first entry of every block is a value in the column and its associated storage key, and every subsequent value is a delta from the previous value. This scheme is reminiscent of the way VSAM codes B-tree index keys [VSAM04]. Again, a densepack B-tree tree at the block-level can be used to index these coded objects.

Type 4: Foreign-order, many distinct values. If there are a large number of values, then it probably makes sense to leave the values unencoded. However, we are still investigating possible compression techniques for this situation. A densepack B-tree can still be used for the indexing.

3.2 Join Indexes

Join indexes must be used to connect the various projections anchored at the same table. As noted earlier, a join index is a collection of (sid, storage_key) pairs. Each of these two fields can be stored as normal columns.

There are physical database design implications concerning where to store join indexes, and we address these in the next section. In addition, join indexes must integrate RS and WS; hence, we revisit their design in the next section as well.

4 WS

In order to avoid writing two optimizers, WS is also a column store and implements the identical physical DBMS design as RS. Hence, the same projections and join indexes are present in WS. However, the storage representation is drastically different because WS must be efficiently updatable transactionally.

The storage key, SK, for each record is explicitly stored in each WS segment. A unique SK is given to each insert of a logical tuple in a table T. The execution engine must ensure that this SK is recorded in each projection that stores data for the logical tuple. This SK is an integer, larger than the number of records in the largest segment in the database.

For simplicity and scalability, WS is horizontally partitioned in the same way as RS. Hence, there is a 1:1 mapping between RS segments and WS segments. A (sid, storage_key) pair identifies a record in either of these containers.

Since we assume that WS is trivial in size relative to RS, we make no effort to compress data values; instead we represent all data directly. Therefore, each projection uses B-tree indexing to maintain a logical sort-key order.

Every column in a WS projection is represented as a collection of pairs, (v, sk), such that v is a value in the column and sk is its corresponding storage key. Each pair is represented in a conventional B-tree on the second field. The sort key(s) of each projection is additionally represented by pairs (s, sk) such that s is a sort key value and sk is the storage key describing where s first appears. Again, this structure is represented as a conventional B-tree on the sort key field(s). To perform searches using the sort key, one uses the latter B-tree to find the storage keys of interest, and then uses the former collection of B-trees to find the other fields in the record.

Join indexes can now be fully described. Every projection is represented as a collection of pairs of segments, one in WS and one in RS. For each record in the "sender," we must store the sid and storage key of a corresponding record in the "receiver." It will be useful to horizontally partition the join index in the same way as the "sending" projection and then to co-locate join index partitions with the sending segment they are associated with. In effect, each (sid, storage key) pair is a pointer to a record which can be in either the RS or WS.

5 Storage Management

The storage management issue is the allocation of segments to nodes in a grid system; C-Store will perform this operation automatically using a *storage allocator*. It seems clear that all columns in a single segment of a projection should be co-located. As noted above, join indexes should be co-located with their "sender" segments. Also, each WS segment will be co-located with the RS segments that contain the same key range.

Using these constraints, we are working on an allocator. This system will perform initial allocation, as well as reallocation when load becomes unbalanced. The details of this software are beyond the scope of this paper.

Since everything is a column, storage is simply the persistence of a collection of columns. Our analysis shows that a raw device offers little benefit relative to today's file systems. Hence, big columns (megabytes) are stored in individual files in the underlying operating system.

6 Updates and Transactions

An insert is represented as a collection of new objects in WS, one per column per projection, plus the sort key data structure. All inserts corresponding to a single logical record have the same storage key. The storage key is allocated at the site where the update is received. To prevent C-Store nodes from needing to synchronize with each other to assign storage keys, each node maintains a locally unique counter to which it appends its local site id to generate a globally unique storage key. Keys in the WS will be consistent with RS storage keys because we set the initial value of this counter to be one larger than the largest key in RS.

We are building WS on top of BerkeleyDB [SLEE04]; we use the B-tree structures in that package to support our data structures. Hence, every insert to a projection results in a collection of physical inserts on different disk pages, one per column per projection. To avoid poor performance, we plan to utilize a very large main memory buffer pool, made affordable by the plummeting cost per byte of primary storage. As such, we expect "hot" WS data structures to be largely main memory resident.

C-Store's processing of deletes is influenced by our locking strategy. Specifically, C-Store expects large numbers of ad-hoc queries with large read sets interspersed with a smaller number of OLTP transactions covering few records. If C-Store used conventional locking, then substantial lock contention would likely be observed, leading to very poor performance.

Instead, in C-Store, we isolate read-only transactions using *snapshot isolation*. Snapshot isolation works by allowing read-only transactions to access the database as of some time in the recent past, before which we can guarantee that there are no uncommitted transactions. For this reason, when using snapshot isolation, we do not need to set any locks. We call the most recent time in the past at which snapshot isolation can run the *high water mark* (HWM) and introduce a low-overhead mechanism for keeping track of its value in our multi-site environment. If we let read-only transactions set their effective time arbitrarily, then we would have to support general time travel, an onerously expensive task. Hence, there is also a *low water mark* (LWM) which is the earliest effective time at which a read-only transaction can run. Update transactions continue to set read and write locks and obey strict two-phase locking, as described in Section 6.2.

6.1 Providing Snapshot Isolation

The key problem in snapshot isolation is determining which of the records in WS and RS should be visible to a read-only transaction running at effective time ET. To provide snapshot isolation, we cannot perform updates in place. Instead, an update is turned into an insert and a delete. Hence, a record is visible if it was inserted before ET and deleted after ET. To make this determination without requiring a large space budget, we use coarse granularity "epochs," to be described in Section 6.1.1, as the unit for timestamps. Hence, we maintain an *insertion vector (IV)* for each projection segment in WS, which contains for each record the epoch in which the record was inserted. We program the tuple mover (described in Section 7) to ensure that no records in RS were inserted after the LWM. Hence, RS need not maintain an insertion vector. In addition, we maintain a *deleted record vector (DRV)* for each projection, which has one entry per projection record, containing a 0 if the tuple has not been deleted; otherwise, the entry contains the epoch in which the tuple was deleted. Since the DRV is very sparse (mostly zeros), it can be compactly coded using the type 2 algorithm described earlier. We store the DRV in the WS, since it must be updatable. The runtime system can now consult *IV* and *DRV* to make the visibility calculation for each query on a record-by-record basis.

6.1.1 Maintaining the High Water Mark

To maintain the HWM, we designate one site the *timestamp authority* (TA) with the responsibility of allocating timestamps to other sites. The idea is to divide time into a number of *epochs*; we define the *epoch number* to be the number of epochs that have elapsed since the beginning of time. We anticipate epochs being relatively long – e.g., many seconds each, but the exact duration may vary from deployment to deployment. We define the initial HWM to be epoch 0 and start *current epoch* at 1. Periodically, the TA decides to move the system to the next epoch; it sends a *end of epoch* message to each site, each of which increments *current epoch* from e to $e + 1$, thus causing new transactions that arrive to be run with a timestamp $e + 1$. Each site waits for all the transactions that began in epoch e (or an earlier epoch)

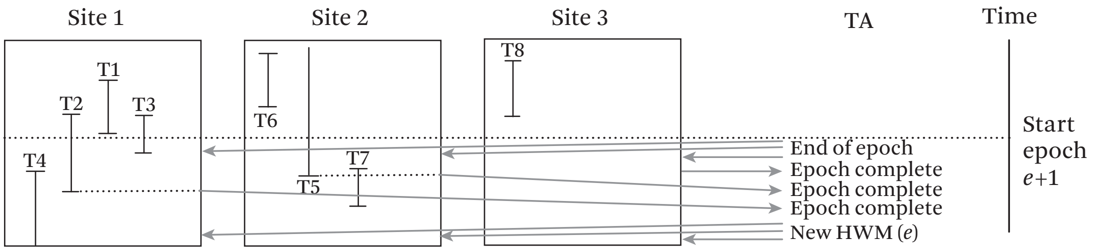


Figure 3 Illustration showing how the HWM selection algorithm works. Gray arrows indicate messages from the TA to the sites or vice versa. We can begin reading tuples with timestamp e when all transactions from epoch e have committed. Note that although T4 is still executing when the HWM is incremented, read-only transactions will not see its updates because it is running in epoch $e + 1$.

to complete and then sends an *epoch complete* message to the TA. Once the TA has received *epoch complete* messages from all sites for epoch e, it sets the HWM to be e, and sends this value to each site. Figure 3 illustrates this process.

After the TA has broadcast the new HWM with value e, read-only transactions can begin reading data from epoch e or earlier and be assured that this data has been committed. To allow users to refer to a particular real-world time when their query should start, we maintain a table mapping epoch numbers to times, and start the query as of the epoch nearest to the user-specified time.

To avoid epoch numbers from growing without bound and consuming extra space, we plan to "reclaim" epochs that are no longer needed. We will do this by "wrapping" timestamps, allowing us to reuse old epoch numbers as in other protocols, e.g., TCP. In most warehouse applications, records are kept for a specific amount of time, say 2 years. Hence, we merely keep track of the oldest epoch in any DRV, and ensure that wrapping epochs through zero does not overrun.

To deal with environments for which epochs cannot effectively wrap, we have little choice but to enlarge the "wrap length" of epochs or the size of an epoch.

6.2 Locking-based Concurrency Control

Read-write transactions use strict two-phase locking for concurrency control [GRAY92]. Each site sets locks on data objects that the runtime system reads or writes, thereby implementing a distributed lock table as in most distributed databases. Standard write-ahead logging is employed for recovery purposes; we use a NO-FORCE, STEAL policy [GRAY92] but differ from the traditional implementation of logging and locking in that we only log UNDO records, performing REDO as described in Section 6.3, and we do not use strict two-phase commit, avoiding the PREPARE phase as described in Section 6.2.1 below.

Locking can, of course, result in deadlock. We resolve deadlock via timeouts through the standard technique of aborting one of the deadlocked transactions.

6.2.1 Distributed COMMIT Processing

In C-Store, each transaction has a *master* that is responsible for assigning units of work corresponding to a transaction to the appropriate sites and determining the ultimate commit state of each transaction. The protocol differs from two-phase commit (2PC) in that no PREPARE messages are sent. When the master receives a COMMIT statement for the transaction, it waits until all workers have completed all outstanding actions and then issues a *commit* (or *abort*) message to each site. Once a site has received a commit message, it can release all locks related to the transaction and delete the UNDO log for the transaction. This protocol differs from 2PC because the master does not PREPARE the worker sites. This means it is possible for a site the master has told to commit to crash before writing any updates or log records related to a transaction to stable storage. In such cases, the failed site will recover its state, which will reflect updates from the committed transaction, from other projections on other sites in the system during recovery.

6.2.2 Transaction Rollback

When a transaction is aborted by the user or the C-Store system, it is undone by scanning backwards in the UNDO log, which contains one entry for each logical update to a segment. We use logical logging (as in ARIES [MOHA92]), since physical logging would result in many log records, due to the nature of the data structures in WS.

6.3 Recovery

As mentioned above, a crashed site recovers by running a query (copying state) from other projections. Recall that C-Store maintains K-safety; i.e. sufficient projections and join indexes are maintained, so that K sites can fail within t, the time to recover, and the system will be able to maintain transactional consistency. There are three cases to consider. If the failed site suffered no data loss, then we can bring it up to date by executing updates that will be queued for it elsewhere in the network. Since we anticipate read-mostly environments, this roll forward operation should not be onerous. Hence, recovery from the most common type of crash is straightforward. The second case to consider is a catastrophic failure which destroys both the RS and WS. In this case, we have no choice but to reconstruct both segments from other projections and join indexes in the system. The only needed functionality is the ability to retrieve auxiliary data structures (IV, DRV) from remote sites. After restoration, the queued updates must be run as above. The third case occurs if WS is damaged but RS is intact. Since RS is written only by the tuple mover, we expect it will typically escape damage. Hence, we discuss this common case in detail below.

6.3.1 Efficiently Recovering the WS

Consider a WS segment, Sr, of a projection with a sort key K and a key range R on a recovering site r along with a collection C of other projections, $M1, \ldots, Mb$ which contain the sort key of Sr. The tuple mover guarantees that each WS segment, S, contains all tuples with an insertion timestamp later than some time $t_{lastmove}(S)$, which represents the most recent insertion time of any record in S's corresponding RS segment.

To recover, the recovering site first inspects every projection in C for a collection of columns that covers the key range K with each segment having $t_{lastmove}(S) \leq t_{lastmove}(Sr)$. If it succeeds, it can run a collection of queries of the form:

```
SELECT desired_fields,
       insertion_epoch,
       deletion_epoch
FROM recovery_segment

WHERE insertion_epoch > $t_{lastmove}(Sr)$
       AND insertion_epoch <= HWM
       AND deletion_epoch = 0
           OR deletion_epoch >= LWM
       AND sort_key in K
```

As long as the above queries return a storage key, other fields in the segment can be found by following appropriate join indexes. As long as there is a collection of segments that cover the key range of Sr, this technique will restore Sr to the current HWM. Executing queued updates will then complete the task.

On the other hand, if there is no cover with the desired property, then some of the tuples in Sr have already been moved to RS on the remote site. Although we can still query the remote site, it is challenging to identify the desired tuples without retrieving everything in RS and differencing against the local RS segment, which is obviously an expensive operation.

To efficiently handle this case, if it becomes common, we can force the tuple mover to log, for each tuple it moves, the storage key in RS that corresponds to the storage key and epoch number of the tuple before it was moved from WS. This log can be truncated to the timestamp of the oldest tuple still in the WS on any site, since no tuples before that will ever need to be recovered. In this case, the recovering site can use a remote WS segment, S, plus the tuple mover log to solve the query above, even though $t_{lastmove}(S)$ comes after $t_{lastmove}(Sr)$.

At r, we must also reconstruct the WS portion of any join indexes that are stored locally, i.e. for which Sr is a "sender." This merely entails querying remote "receivers," which can then compute the join index as they generate tuples, transferring the WS partition of the join index along with the recovered columns.

7 Tuple Mover

The job of the tuple mover is to move blocks of tuples in a WS segment to the corresponding RS segment, updating any join indexes in the process. It operates as a background task looking for *worthy* segment pairs. When it finds one, it performs a *merge-out process*, *MOP* on this (RS, WS) segment pair.

MOP will find all records in the chosen WS segment with an insertion time at or before the LWM, and then divides them into two groups:

- Ones deleted at or before LWM. These are discarded, because the user cannot run queries as of a time when they existed.
- Ones that were not deleted, or deleted after LWM. These are moved to RS.

MOP will create a new RS segment that we name RS'. Then, it reads in blocks from columns of the RS segment, deletes any RS items with a value in the DRV less than or equal to the LWM, and merges in column values from WS. The merged data is then written out to the new RS' segment, which grows as the merge progresses. The most recent insertion time of a record in RS' becomes the segment's new $t_{lastmove}$ and is always less than or equal to the LWM. This old-master/new-master approach will be more efficient than an update-in-place strategy, since essentially all data objects will move. Also, notice that records receive new storage keys in RS', thereby requiring join index maintenance. Since RS items may also be deleted, maintenance of the DRV is also mandatory. Once RS' contains all the WS data and join indexes are modified on RS', the system cuts over from RS to RS'. The disk space used by the old RS can now be freed.

Periodically the timestamp authority sends out to each site a new LWM epoch number. Hence, LWM "chases" HWM, and the delta between them is chosen to mediate between the needs of users who want historical access and the WS space constraints.

8 C-Store Query Execution

The query optimizer will accept a SQL query and construct a query plan of execution nodes. In this section, we describe the nodes that can appear in a plan and then the architecture of the optimizer itself.

8.1 Query Operators and Plan Format

There are 10 node types and each accepts operands or produces results of type projection (`Proj`), column (`Col`), or bitstring (`Bits`). A projection is simply a set of columns with the same cardinality and ordering. A bitstring is a list of zeros and ones indicating whether the associated values are present in the record subset being described. In addition, C-Store query operators accept predicates (`Pred`), join indexes (`JI`), attribute names (`Att`), and expressions (`Exp`) as arguments.

Join indexes and bitstrings are simply special types of columns. Thus, they also can be included in projections and used as inputs to operators where appropriate.

We briefly summarize each operator below.

1. **Decompress** converts a compressed column to an uncompressed (Type 4) representation.
2. **Select** is equivalent to the selection operator of the relational algebra (σ), but rather than producing a restriction of its input, instead produces a bitstring representation of the result.
3. **Mask** accepts a bitstring `B` and projection `Cs`, and restricts `Cs` by emitting only those values whose corresponding bits in `B` are 1.
4. **Project** equivalent to the projection operator of the relational algebra (π).
5. **Sort** sorts all columns in a projection by some subset of those columns (the *sort* columns).
6. **Aggregation Operators** compute SQL-like aggregates over a named column, and for each group identified by the values in a projection.
7. **Concat** combines one or more projections sorted in the same order into a single projection
8. **Permute** permutes a projection according to the ordering defined by a join index.
9. **Join** joins two projections according to a predicate that correlates them.
10. **Bitstring Operators** `BAnd` produces the bitwise AND of two bitstrings. `BOr` produces a bitwise OR. `BNot` produces the complement of a bitstring.

A C-Store query plan consists of a tree of the operators listed above, with access methods at the leaves and iterators serving as the interface between connected nodes. Each non-leaf plan node consumes the data produced by its children via a modified version of the standard iterator interface [GRAE93] via calls of "get_next." To reduce communication overhead (i.e., number of calls of "get_next") between plan nodes, C-Store iterators return 64K blocks from a single column. This approach preserves the benefit of using iterators (coupling data flow with control flow), while changing the granularity of data flow to better match the column-based model.

8.2 Query Optimization

We plan to use a Selinger-style [SELI79] optimizer that uses cost-based estimation for plan construction. We anticipate using a two-phase optimizer [HONG92] to limit the complexity of the plan search space. Note that query optimization in this setting differs from traditional query optimization in at least two respects: the need to consider compressed representations of data and the decisions about when to mask a projection using a bitstring.

C-Store operators have the capability to operate on both compressed and uncompressed input. As will be shown in Section 9, the ability to process compressed data is the key to the performance benefits of C-Store. An operator's execution cost (both in terms of I/O and memory buffer requirements) is dependent on the compression type of the input. For example, a `Select` over Type 2 data (foreign order/few values, stored as a delta-encoded bitmaps, with one bitmap per value) can be performed by reading only those bitmaps from disk whose values match the predicate (despite the column itself not being sorted). However, operators that take Type 2 data as input require much larger memory buffer space (one page of memory for each possible value in the column) than any of the other three types of compression. Thus, the cost model must be sensitive to the representations of input and output columns.

The major optimizer decision is which set of projections to use for a given query. Obviously, it will be time consuming to construct a plan for each possibility, and then select the best one. Our focus will be on pruning this search space. In addition, the optimizer must decide where in the plan to mask a projection according to a bitstring. For example, in some cases it is desirable to push the `Mask` early in the plan (e.g, to avoid producing a bitstring while performing selection over Type 2 compressed data) while in other cases it is best to delay masking until a point where it is possible to feed a bitstring to the next operator in the plan (e.g., `COUNT`) that can produce results solely by processing the bitstring.

9 Performance Comparison

At the present time, we have a storage engine and the executor for RS running. We have an early implementation of the WS and tuple mover; however they are not at the point where we can run experiments on them. Hence, our performance analysis is limited to read-only queries, and we are not yet in a position to report on updates. Moreover, RS does not yet support segments or multiple grid nodes. As such, we report single-site numbers. A more comprehensive performance study will be done once the other pieces of the system have been built.

Our benchmarking system is a 3.0 Ghz Pentium, running RedHat Linux, with 2 Gbytes of memory and 750 Gbytes of disk.

In the decision support (warehouse) market TPC-H is the gold standard, and we use a simplified version of this benchmark, which our current engine is capable of running. Specifically, we implement the **lineitem**, **order**, and **customer** tables as follows:

```
CREATE TABLE LINEITEM (
L_ORDERKEY INTEGER NOT NULL,
L_PARTKEY  INTEGER NOT NULL,
L_SUPPKEY  INTEGER NOT NULL,
L_LINENUMBER       INTEGER NOT NULL,
L_QUANTITY INTEGER NOT NULL,
L_EXTENDEDPRICE    INTEGER NOT NULL,
L_RETURNFLAG       CHAR(1) NOT NULL,
L_SHIPDATE INTEGER NOT NULL);

CREATE TABLE ORDERS (
O_ORDERKEY  INTEGER NOT NULL,
O_CUSTKEY   INTEGER NOT NULL,
O_ORDERDATE INTEGER NOT NULL);

CREATE TABLE CUSTOMER (
C_CUSTKEY   INTEGER NOT NULL,
C_NATIONKEY INTEGER NOT NULL);
```

We chose columns of type INTEGER and CHAR(1) to simplify the implementation. The standard data for the above table schema for TPC-H scale_10 totals 60,000,000 line items (1.8GB), and was generated by the data generator available from the TPC website.

We tested three systems and gave each of them a storage budget of 2.7 GB (roughly 1.5 times the raw data size) for all data plus indices. The three systems were C-Store as described above and two popular commercial relational DBMS systems, one that implements a row store and another that implements a column

C-Store	Row Store	Column Store
1.987 GB	4.480 GB	2.650 GB

store. In both of these systems, we turned off locking and logging. We designed the schemas for the three systems in a way to achieve the best possible performance given the above storage budget. The row-store was unable to operate within the space constraint so we gave it 4.5 GB which is what it needed to store its tables plus indices. The actual disk usage numbers are shown below. Obviously, C-Store uses 40% of the space of the row store, even though it uses redundancy and the row store does not. The main reasons are C-Store compression and absence of padding to word or block boundaries. The column store requires 30% more space than C-Store. Again, C-Store can store a redundant schema in less space because of superior compression and absence of padding.

We ran the following seven queries on each system:

Q1. *Determine the total number of lineitems shipped for each day after day D.*

```
SELECT l_shipdate, COUNT (*)
FROM lineitem
WHERE l_shipdate > D
GROUP BY l_shipdate
```

Q2. *Determine the total number of lineitems shipped for each supplier on day D.*

```
SELECT l_suppkey, COUNT (*)
FROM lineitem
WHERE l_shipdate = D
GROUP BY l_suppkey
```

Q3. *Determine the total number of lineitems shipped for each supplier after day D.*

```
SELECT l_suppkey, COUNT (*)
FROM lineitem
WHERE l_shipdate > D
GROUP BY l_suppkey
```

Q4. *For every day after D, determine the latest shipdate of all items ordered on that day.*

```
SELECT o_orderdate, MAX (l_shipdate)
FROM lineitem, orders
WHERE l_orderkey = o_orderkey AND
      o_orderdate > D
GROUP BY o_orderdate
```

Q5. *For each supplier, determine the latest shipdate of an item from an order that was made on some date, D.*

```
SELECT l_suppkey, MAX (l_shipdate)
FROM lineitem, orders
WHERE l_orderkey = o_orderkey AND
      o_orderdate = D
GROUP BY l_suppkey
```

Q6. *For each supplier, determine the latest shipdate of an item from an order made after some date, D.*

```
SELECT l_suppkey, MAX (l_shipdate)
FROM lineitem, orders
WHERE l_orderkey = o_orderkey AND
      o_orderdate > D
GROUP BY l_suppkey
```

Q7. *Return a list of identifiers for all nations represented by customers along with their total lost revenue for the parts they have returned.* This is a simplified version of query 10 (Q10) of TPC-H.

```
SELECT c_nationkey, sum(l_extendedprice)
FROM lineitem, orders, customers
WHERE l_orderkey=o_orderkey AND
      o_custkey=c_custkey AND
      l_returnflag='R'
GROUP BY c_nationkey
```

We constructed schemas for each of the three systems that best matched our seven-query workload. These schema were tuned individually for the capabilities of each system. For C-Store, we used the following schema:

```
D1: (l_orderkey, l_partkey, l_suppkey,
     l_linenumber, l_quantity,
     l_extendedprice, l_returnflag, l_shipdate
     | l_shipdate, l_suppkey)

D2: (o_orderdate, l_shipdate, l_suppkey |
     o_orderdate, l_suppkey)

D3: (o_orderdate, o_custkey, o_orderkey |
     o_orderdate)

D4: (l_returnflag, l_extendedprice,
     c_nationkey | l_returnflag)

D5: (c_custkey, c_nationkey | c_custkey)
```

D2 and D4 are materialized (join) views. D3 and D5 are added for completeness since we don't use them in any of the seven queries. They are included so that we can answer arbitrary queries on this schema as is true for the product schemas.

On the commercial row-store DBMS, we used the common relational schema given above with a collection of system-specific tuning parameters. We also used system-specific tuning parameters for the commercial column-store DBMS. Although we believe we chose good values for the commercial systems, obviously, we cannot guarantee they are optimal.

The following table indicates the performance that we observed. All measurements are in seconds and are taken on a dedicated machine.

Query	C-Store	Row Store	Column Store
Q1	0.03	6.80	2.24
Q2	0.36	1.09	0.83
Q3	4.90	93.26	29.54
Q4	2.09	722.90	22.23
Q5	0.31	116.56	0.93
Q6	8.50	652.90	32.83
Q7	2.54	265.80	33.24

As can be seen, C-Store is much faster than either commercial product. The main reasons are:

- *Column representation* – avoids reads of unused attributes (same as competing column store).
- *Storing overlapping projections, rather than the whole table* – allows storage of multiple orderings of a column as appropriate.
- *Better compression of data* – allows more orderings in the same space.
- *Query operators operate on compressed representation* – mitigates the storage barrier problem of current processors.

In order to give the other systems every possible advantage, we tried running them with the materialized views that correspond to the projections we used with C-Store. This time, the systems used space as follows (C-Store numbers, which did not change, are included as a reference):

C-Store	Row Store	Column Store
1.987 GB	11.900 GB	4.090 GB

The relative performance numbers in seconds are as follows:

Query	C-Store	Row Store	Column Store
Q1	0.03	0.22	2.34
Q2	0.36	0.81	0.83
Q3	4.90	49.38	29.10
Q4	2.09	21.76	22.23
Q5	0.31	0.70	0.63
Q6	8.50	47.38	25.46
Q7	2.54	18.47	6.28

As can be seen, the performance gap closes, but at the same time, the amount of storage needed by the two commercial systems grows quite large.

In summary, for this seven query benchmark, C-Store is on average 164 times faster than the commercial row-store and 21 times faster than the commercial column-store in the space-constrained case. For the case of unconstrained space, C-Store is 6.4 times faster than the commercial row-store, but the row-store takes 6 times the space. C-Store is on average 16.5 times faster than the commercial column-store, but the column-store requires 1.83 times the space.

Of course, this performance data is very preliminary. Once we get WS running and write a tuple mover, we will be in a better position to do an exhaustive study.

10 Related Work

One of the thrusts in the warehouse market is in maintaining so-called "data cubes." This work dates from Essbase by Arbor software in the early 1990's, which was effective at "slicing and dicing" large data sets [GRAY97]. Efficiently building and maintaining specific aggregates on stored data sets has been widely studied [KOTI99, ZHAO97]. Precomputation of such aggregates as well as more general materialized views [STAU96] is especially effective when a prespecified set of queries is run at regular intervals. On the other hand, when the workload cannot be anticipated in advance, it is difficult to decide what to precompute. C-Store is aimed entirely at this latter problem.

Including two differently architected DBMSs in a single system has been studied before in data mirrors [RAMA02]. However, the goal of data mirrors was to achieve better query performance than could be achieved by either of the two underlying systems alone in a warehouse environment. In contrast, our goal is to simultaneously achieve good performance on update workloads and ad-hoc queries. Consequently, C-Store differs dramatically from a data mirror in its design.

Storing data via columns has been implemented in several systems, including Sybase IQ, Addamark, Bubba [COPE88], Monet [BONC04], and KDB. Of these, Monet is probably closest to C-Store in design philosophy. However, these systems typically store data in entry sequence and do not have our hybrid architecture nor do they have our model of overlapping materialized projections.

Similarly, storing tables using an inverted organization is well known. Here, every attribute is stored using some sort of indexing, and record identifiers are used to find corresponding attributes in other columns. C-Store uses this sort of organization in WS but extends the architecture with RS and a tuple mover.

There has been substantial work on using compressed data in databases; Roth and Van Horn [ROTH93] provide an excellent summary of many of the techniques that have been developed. Our coding schemes are similar to some of these techniques, all of which are derived from a long history of work on the topic in the broader field of computer science [WITT87]. Our observation that it is possible to operate directly on compressed data has been made before [GRAE91, WESM00].

Lastly, materialized views, snapshot isolation, transaction management, and high availability have also been extensively studied. The contribution of C-Store is an innovative combination of these techniques that simultaneously provides improved performance, K-safety, efficient retrieval, and high performance transactions.

11 Conclusions

This paper has presented the design of C-Store, a radical departure from the architecture of current DBMSs. Unlike current commercial systems, it is aimed at the “read-mostly” DBMS market. The innovative contributions embodied in C-Store include:

- A column store representation, with an associated query execution engine.
- A hybrid architecture that allows transactions on a column store.
- A focus on economizing the storage representation on disk, by coding data values and dense-packing the data.

- A data model consisting of overlapping projections of tables, unlike the standard fare of tables, secondary indexes, and projections.
- A design optimized for a shared nothing machine environment.
- Distributed transactions without a redo log or two phase commit.
- Efficient snapshot isolation.

Acknowledgements and References

We would like to thank David DeWitt for his helpful feedback and ideas.

This work was supported by the National Science Foundation under NSF Grant numbers IIS-0086057 and IIS-0325525.

[ADDA04] http://www.addamark.com/products/sls.htm

[BERE95] Hal Berenson et al. A Critique of ANSI SQL Isolation Levels. In *Proceedings of SIGMOD*, 1995.

[BONC04] Peter Boncz et. al. MonetDB/X100: Hyper-pipelining Query Execution. In *Proceedings CIDR 2004*.

[CERI91] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In *VLDB*, 1991.

[COPE88] George Copeland et. al. Data Placement in Bubba. *In Proceedings SIGMOD 1988*.

[DEWI90] David Dewitt et. al. The GAMMA Database machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March, 1990.

[DEWI92] David Dewitt and Jim Gray. Parallel Database Systems: The Future of High Performance Database Processing. *Communications of the ACM*, 1992.

[FREN95] Clark D. French. One Size Fits All Database Architectures Do Not Work for DSS. In *Proceedings of SIGMOD*, 1995.

[GRAE91] Goetz Graefe, Leonard D. Shapiro. Data Compression and Database Performance. In *Proceedings of the Symposium on Applied Computing*, 1991.

[GRAE93] G. Graefe. Query Evaluation Techniques for Large Databases. *Computing Surveys*, 25(2), 1993.

[GRAY92] Jim Gray and Andreas Reuter. *Transaction Processing Concepts and Techniques*, Morgan Kaufman, 1992.

[GRAY97] Gray et al. DataCube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery*, 1(1), 1997.

[HONG92] Wei Hong and Michael Stonebraker. Exploiting Interoperator Parallelism in XPRS. In *SIGMOD*, 1992.

[KDB04] http://www.kx.com/products/database.php

[KOTI99] Yannis Kotidis, Nick Roussopoulos. DynaMat: A Dynamic View Management System for Data Warehouses. In *Proceedings of SIGMOD*, 1999.

[MOHA92] C. Mohan et. al: ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *TODS*, March 1992.

[ONEI96] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil, The Log-Structured Merge-Tree. *Acta Informatica* 33, June 1996.

[ONEI97] P. O'Neil and D. Quass. Improved Query Performance with Variant Indexes, In *Proceedings of SIGMOD*, 1997.

[ORAC04] Oracle Corporation. *Oracle 9i Database for Data Warehousing and Business Intelligence*. White Paper. http://www.oracle.com/solutions/business_intelligence/Oracle9idw_bwp.

[PAPA04] Stratos Papadomanolakis and Anastassia Ailamaki. AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning. In *SSDBM 2004*.

[RAMA02] Ravishankar Ramamurthy, David Dewitt. Qi Su: A Case for Fractured Mirrors. In *Proceedings of VLDB*, 2002.

[ROTH93] Mark A. Roth, Scott J. Van Horn: Database Compression. *SIGMOD Record* 22(3). 1993.

[SELI79] Patricia Selinger, Morton Astrahan, Donald Chamberlain, Raymond Lorie, Thomas Price. Access Path Selection in a Relational Database. In *Proceedings of SIGMOD*, 1979.

[SLEE04] http://www.sleepycat.com/docs/

[STAU96] Martin Staudt, Matthias Jarke. Incremental Maintenance of Externally Materialized Views. In *VLDB*, 1996.

[STON86] Michael Stonebraker. The Case for Shared Nothing. In *Database Engineering*, 9(1), 1986.

[SYBA04] http://www.sybase.com/products/databaseservers/sybaseiq

[TAND89] Tandem Database Group: NonStop SQL, A Distributed High Performance, High Availability Implementation of SQL. In Proceedings *of HPTPS*, 1989.

[VSAM04] http://www.redbooks.ibm.com/redbooks.nsf/0/8280b48d5e3997bf85256cbd007e4a96?OpenDocument

[WESM00] Till Westmann, Donald Kossmann, Sven Helmer, Guido Moerkotte. The Implementation and Performance of Compressed Databases. *SIGMOD Record* 29(3), 2000.

[WEST00] Paul Westerman. *Data Warehousing: Using the Wal-Mart Model*. Morgan-Kaufmann Publishers , 2000.

[WITT87] I. Witten, R. Neal, and J. Cleary. Arithmetic coding for data compression. *Comm. of the ACM*, 30(6), June 1987.

[ZHAO97] Y. Zhao, P. Deshpande, and J. Naughton. An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. In *Proceedings of SIGMOD*, 1997.

The Implementation of POSTGRES

Michael Stonebraker, Lawrence A. Rowe, Michael Hirohama

Abstract—Currently, POSTGRES is about 90 000 lines of code in C and is being used by assorted "bold and brave" early users. The system has been constructed by a team of five part-time students led by a full-time chief programmer over the last three years. During this period, we have made a large number of design and implementation choices. Moreover, in some areas we would do things quite differently if we were to start from scratch again. The purpose of this paper is to reflect on the design and implementation decisions we made and to offer advice to implementors who might follow some of our paths. In this paper, we restrict our attention to the DBMS "backend" functions. In another paper, some of us treat Picasso, the application development environment that is being built on top of POSTGRES.

Index Terms—Extensible databases, next-generation DBMS's, no-overwrite storage managers, object-oriented databases, rule systems.

Manuscript received August 15, 1989; revised December 1, 1989. This work was supported by the Defense Advanced Research Projects Agency under NASA Grant NAG 2-530 and by the Army Research Office under Grant DAAL03-87-K-0083.

The authors are with the Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720.

IEEE Log Number 8933788.

Paper originally published in *IEEE Transactions on Knowledge and Data Engineering,* 2(1): 125–142, 1990. Original DOI: 10.1109/69.50912

I Introduction

Current relational DBMS's are oriented toward efficient support for business data processing applications where large numbers of instances of fixed format records must be stored and accessed. Traditional transaction management and query facilities for this application area will be termed *data management.*

To satisfy the broader application community outside of business applications, DBMS's will have to expand to offer services in two other dimensions, namely *object management* and *knowledge management*. Object management entails efficiently storing and manipulating nontraditional data types such as bitmaps, icons, text, and polygons. Object management problems abound in CAD and many other engineering applications. Object-oriented programming languages and databases provide services in this area.

Knowledge management entails the ability to store and enforce a collection of *rules* that are part of the semantics of an application. Such rules describe integrity constraints about the application, as well as allowing the derivation of data that are not directly stored in the database.

We now indicate a simple example which requires services in all three dimensions. Consider an application that stores and manipulates text and graphics to facilitate the layout of newspaper copy. Such a system will be naturally integrated with subscription and classified advertisement data. Billing customers for these services will require traditional data management services. In addition, this application must store nontraditional objects including text, bitmaps (pictures), and icons (the banner across the top of the paper). Hence, object management services are required. Lastly, there are many rules that control newspaper layout. For example, the ad copy for two major department stores can never be on facing pages. Support for such rules is desirable in this application.

We believe that *most* real world data management problems are *three dimensional*. Like the newspaper application, they will require a three-dimensional solution. The fundamental goal of POSTGRES [26], [35] is to provide support for such three-dimensional applications. To the best of our knowledge it is the first three-dimensional data manager. However, we expect that most DBMS's will follow the lead of POSTGRES into these new dimensions.

To accomplish this objective, object and rule management capabilities were added to the services found in a traditional data manager. In the next two sections, we describe the capabilities provided and comment on our implementation decisions. Then, in Section IV we discuss the novel *no-overwrite* storage manager that we implemented in POSTGRES. Other papers have explained the major POSTGRES design decisions in these areas, and we assume that the reader is familiar with [21] on the data model, [30] on rule management, and [28] on storage management. Hence, in these three sections we stress considerations that led to our design, what we liked about the design, and the mistakes that we felt we made. Where appropriate we make suggestions for future implementors based on our experience.

Section V of the paper comments on specific issues in the implementation of POSTGRES and critiques the choices that we made. In this section, we discuss how we interfaced to the operating system, our choice of programming languages, and some of our implementation philosophy.

The final section concludes with some performance measurements of POSTGRES. Specifically, we report the results of some of the queries in the Wisconsin benchmark [7].

II The POSTGRES Data Model and Query Language

II.A Introduction

Traditional relational DBMS's support a data model consisting of a collection of named relations, each attribute of which has a specific type. In current commercial systems, possible types are floating point numbers, integers, character strings, and dates. It is commonly recognized that this data model is insufficient for nonbusiness data processing applications. In designing a new data model and query language, we were guided by the following three design criteria.

1) *Orientation toward database access from a query language:* We expect POSTGRES users to interact with their databases primarily by using the set-oriented query language, POSTQUEL. Hence, inclusion of a query language, an optimizer, and the corresponding run-time system was a primary design goal.

It is also possible to interact with a POSTGRES database by utilizing a navigational interface. Such interfaces were popularized by the CODASYL proposals of the 1970's and are enjoying a renaissance in recent object-oriented proposals such as ORION [6] or O2 [34]. Because POSTGRES gives each record a unique identifier (OID), it is possible to use the identifier for one record as a data item in a second record. Using optionally definable indexes on OID's, it is then possible to navigate from one record to the next by running one query per navigation step. In addition, POSTGRES allows a user to define functions (methods) to the DBMS. Such functions can intersperse statements in a programming language, a query language, and direct calls to internal POSTGRES interfaces. The ability to directly execute functions which we call *fast path* is provided in POSTGRES and allows a user to navigate the database by executing a sequence of functions.

However, we do not expect this sort of mechanism to become popular. All navigational interfaces have the same disadvantages of CODASYL systems, namely the application programmer must construct a query plan for each task he wants to accomplish and substantial application maintenance is required whenever the schema changes.

2) *Orientation toward multilingual access:* We could have picked our favorite programming language and then tightly coupled POSTGRES to the compiler and run-time environment of that language. Such an approach would offer *persistence* for variables in this programming language, as well as a query language integrated with the control statements of the language. This approach has been followed in ODE [l] and many of the recent commercial startups doing object-oriented databases.

Our point of view is that most databases are accessed by programs written in several different languages, and we do not see any programming language Esperanto on the horizon. Therefore, most application development organizations are *multilingual* and require access to a database from different languages. In addition, database application packages that a user might acquire, for example to perform statistical or spreadsheet services, are often not coded in the language being used for developing applications. Again, this results in a multilingual environment.

Hence, POSTGRES is programming language *neutral*, that is, it can be called from many different languages. Tight integration of POSTGRES to a particular language requires compiler extensions and a run-time system specific to that programming language. One of us has built an implementation of persistent CLOS (Common LISP Object System) on top of POSTGRES. Persistent CLOS (or persistent *X* for any programming language *X*) is inevitably language specific. The run-time system must map the disk representation for language objects, including pointers, into the main memory representation expected by the language. Moreover, an object cache must be maintained in the program address space, or performance will suffer badly. Both tasks are inherently language specific.

We expect many language specific interfaces to be built for POSTGRES and believe that the query language plus the *fast path* interface available in POSTGRES offer a powerful, convenient abstraction against which to build these programming language interfaces.

3) *Small number of concepts:* We tried to build a data model with as few concepts as possible. The relational model succeeded in replacing previous data models in part because of its simplicity. We wanted to have as few concepts as possible so that users would have minimum complexity to contend with. Hence, POSTGRES leverages the following three constructs:

types

functions

inheritance.

In the next subsection, we briefly review the POSTGRES data model. Then, we turn to a short description of POSTQUEL and fast path. We conclude the section with a discussion of whether POSTGRES is object-oriented followed by a critique of our data model and query language.

II.B The POSTGRES Data Model

As mentioned in the previous section, POSTGRES leverages *types* and *functions* as fundamental constructs. There are three kinds of types in POSTGRES and three kinds of functions and we discuss the six possibilities in this section.

Some researchers, e.g., [27], [19], have argued that one should be able to construct new *base types* such as bits, bit-strings, encoded character strings, bitmaps, compressed integers, packed decimal numbers, radix 50 decimal numbers, money, etc. Unlike most next-generation DBMS's which have a hardwired collection of base types (typically integers, floats, and character strings), POSTGRES contains an *abstract data type* facility whereby any user can construct an arbitrary number of new base types. Such types can be added to the system while it is executing and require the defining user to specify functions to convert instances of the type to and from the character string data type. Details of the syntax appear in [35].

The second kind of type available in POSTGRES is a *constructed type*.[1] A user can create a new type by constructing a *record* of base types and instances of other constructed types. For example,

```
create DEPT (dname = c10, floor = integer, floor-
    space = polygon)
create EMP (name = c12, dept = DEPT, salary =
    float)
```

Here, DEPT is a type constructed from an instance of each of three base types: a character string, an integer, and a polygon. EMP, on the other hand, is fabricated from base types and other constructed types.

[1] In this section, the reader can use the words *constructed type*, *relation*, and *class* interchangeably. Moreover. the words *record*, *instance*, and *tuple* are similarly interchangeable. This section has been purposely written with the chosen notation to illustrate a point about object-oriented databases which is discussed in Section II-E.

A constructed type can optionally *inherit* data elements from other constructed types. For example, a SALESMAN type can be created as follows:

```
create SALESMAN (quota = float) inherits (EMP)
```

In this case, an instance of SALESMAN has a quota and inherits all data elements from EMP, namely name, dept, and salary. We had the standard discussion about whether to include single or multiple inheritance and concluded that a single inheritance scheme would simply be too restrictive. As a result, POSTGRES allows a constructed type to inherit from an arbitrary collection of other constructed types.

When ambiguities arise because an object has multiple parents with the same field name, we elected to refuse to create the new type. However, we isolated the resolution semantics in a single routine, which can be easily changed to track multiple inheritance semantics as they unfold over time in programming languages.

We now turn to the POSTGRES notion of functions. There are three different classes of POSTGRES functions:

normal functions

operators

POSTQUEL functions

and we discuss each in turn.

A user can define an arbitrary collection of *normal functions* whose operands are base types or constructed types. For example, he can define a function, area, which maps an instance of a polygon into an instance of a floating point number. Such functions are automatically available in the query language as illustrated in the following example:

```
retrieve (DEPT.dname)
    where area (DEPT.floorspace) > 500
```

Normal functions can be defined to POSTGRES while the system is running and are dynamically loaded when required during query execution.

Functions are allowed on constructed types, e.g.,

```
retrieve (EMP.name) where overpaid (EMP)
```

In this case, overpaid has an operand of type EMP and returns a Boolean. Functions whose operands are constructed types are inherited down the type hierarchy in the standard way.

Normal functions are arbitrary procedures written in a general purpose programming language (in our case C or LISP). Hence, they have arbitrary semantics and can run other POSTQUEL commands during execution. Therefore, queries with normal functions in the qualification cannot be optimized by the POSTGRES query optimizer. For example, the above query on overpaid employees will result in a sequential scan of all employees.

To utilize indexes in processing queries, POSTGRES supports a second class of functions, called *operators*. Operators are functions with one or two operands which use the standard operator notation in the query language. For example, the following query looks for departments whose floor space has a greater area than that of a specific polygon:

```
retrieve (DEPT.dname) where DEPT. floorspace AGT
    polygon["(0,0), (1, 1), (0,2)"].
```

The "area greater than" operator AGT is defined by indicating the token to use in the query language as well as the function to call to evaluate the operator. Moreover, several *hints* can also be included in the definition which assist the query optimizer. One of these hints is that ALE is the negator of this operator. Therefore, the query optimizer can transform the query:

```
retrieve (DEPT.dname) where not (DEPT.floorspace
    ALE polygon[ "(0,0), (1,1), (0,2)"])
```

which cannot be optimized into the one above which can be optimized. In addition, the design of the POSTGRES access methods allows a B+-tree index to be constructed for the instances of floorspace appearing in DEPT records. This index can support efficient access for the class of operators {ALT, ALE, AE, AGT , AGE}. Information on the access paths available to the various operators is recorded in the POSTGRES system catalogs.

As pointed out in [29], it is imperative that a user be able to construct new access method s lO provide efficient access to instances of nontraditional base types. For example, suppose a user introduces a new operator "!!" defined on polygons that returns true if two polygons overlap. Then, he might ask a query such as

```
retrieve (DEPT.dname) where DEPT.floorspace !!
    polygon["(0,0), (1, 1), (0,2)"]
```

There is no B+-tree or hash access method that will allow this query to be rapidly executed. Rather, the query must be supported by some multidimensional access method such as R-trees, grid files, K-D-B trees, etc. Hence, POSTGRES was designed to allow new access methods to be written by POSTGRES users and then dynamically added to the system. Basically, an access method to POSTGRES is a collection of 13 normal functions which perform record level operations such as fetching the next record in a scan, inserting a new record, deleting a specific record, etc. All a user need do is define implementations for each of these functions and make a collection of entries in the system catalogs.

Operators are only available for operands which are base types because access methods traditionally support fast access to specific fields in records. It is unclear what an access method for a constructed type should do, and therefore POSTGRES does not include this capability.

The third kind of function available in POSTGRES is *POSTQUEL functions*. Any collection of commands in the POSTQUEL query language can be packaged together and defined as a function. For example, the following function defines the overpaid employees:

```
define function high-pay as retrieve (EMP.all) where
    EMP.salary > 50000
```

POSTQUEL functions can also have parameters, for example,

```
define function ret-sal as retrieve (EMP.salary) where
    EMP.name = $1
```

Notice that ret-sal has one parameter in the body of the function, the name of the person involved. Such parameters must be provided at the time the function is called. A third example POSTQUEL function is

```
define function set-of-DEPT as retrieve (DEPT.all)
    where DEPT.floor = $.floor
```

This function has a single parameter "$.floor." It is expected to appear in a record and receives the value of its parameter from the floor field defined elsewhere in the same record.

Each POSTQUEL function is automatically a constructed type. For example, one can define a FLOORS type as follows:

```
create FLOORS (floor = i2, depts = set-of-DEPT)
```

This constructed type uses the set-of-DEPT function as a constructed type. In this case, each instance of FLOORS has a value for depts which is the value of the function set-of-DEPT for that record.

In addition, POSTGRES allows a user to form a constructed type, one or more of whose fields has the special type POSTQUEL. For example, a user can construct the following type:

```
create PERSON (name = c12, hobbies = POST-
    QUEL)
```

In this case, each instance of hobbies contains a different POSTQUEL function, and therefore each person has a name and a POSTQUEL function that defines his particular hobbies. This support for POSTQUEL as a type allows the system to simulate nonnormalized relations as found in NF**2 [11].

POSTQUEL functions can appear in the query language in the same manner as normal functions. The following example ensures that Joe has the same salary as Sam:

```
replace EMP (salary = ret-sal("Joe")) where
EMP.name = "Sam"
```

In addition, since POSTQUEL functions are a constructed type, queries can be executed against POSTQUEL functions just like other constructed types. For example, the following query can be run on the constructed type, high-pay:

```
retrieve (high-pay.salary) where high-pay.name =
    "george"
```

If a POSTQUEL function contains a single retrieve command, then it is very similar to a relational view definition, and this capability allows retrieval operations to be performed on objects which are essentially relational views.

Lastly, every time a user defines a constructed type, a POSTQUEL function is automatically defined with the same name. For example, when DEPT is constructed, the following function is automatically defined:

```
define function DEPT as retrieve (DEPT.all) where DEPT.OID = $1
```

When EMP was defined earlier in this section, it contained a field dept which was of type DEPT. In fact, DEPT was the above automatically defined POSTQUEL function. As a result, an instance of a constructed type is available as a type because POSTGRES automatically defines a POSTQUEL function for each such type.

POSTQUEL functions are a very powerful notion because they allow arbitrary collections of instances of types to be returned as the value of the function. Since POSTQUEL functions can reference other POSTQUEL functions, arbitrary structures of complex objects can be assembled. Lastly, POSTQUEL functions allow collections of commands such as the five SQL commands that make up TP1 [3] to be assembled into a single function and stored inside the DBMS. Then, one can execute TP1 by executing the single function. This approach is preferred to having to submit the five SQL commands in TP1 one by one from an application program. Using a POSTQUEL function, one replaces five roundtrips between the application and the DBMS with 1, which results in a 25% performance improvement in a typical OLTP application.

II.C The POSTGRES Query Language

The previous section presented several examples of the POSTQUEL language. It is a set-oriented query language that resembles a superset of a relational query language. Besides user-defined functions and operators which were illustrated earlier, the features which have been added to a traditional relational language include

path expressions

support for nested queries

transitive closure

support for inheritance

support for time travel.

Path expressions are included because POSTQUEL allows constructed types which contain other constructed types to be hierarchically referenced. For example, the EMP type defined above contains a field which is an instance of the constructed type, DEPT. Hence, one can ask for the names of employees who work on the first floor as follows:

```
retrieve (EMP.name) where EMP.dept.floor = 1
```

rather than being forced to do a join, e.g.,

```
retrieve (EMP.name) where EMP.dept = DEPT.OID
    and DEPT.floor = l
```

POSTQUEL also allows queries to be nested and has operators that have sets of instances as operands. For example to find the departments which occupy an entire floor, one would query

```
retrieve (DEPT.dname)
where DEPT.floor NOTIN {D.floor using D in DEPT
    where D.dname ! = DEPT.dname}
```

In this case, the expression inside the curly braces represents a set of instances and NOTIN is an operator which takes a set of instances as its right operand.

The transitive closure operation allows one to explode a parts or ancestor hierarchy. Consider for example the constructed type

```
parent (older, younger)
```

One can ask for all the ancestors of John as follows.

```
retrieve* into answer (parent.older)
using a in answer
where parent.younger = "John"
or parent.younger = a.older
```

In this case, the * after retrieve indicates that the associated query should be run until the answer fails to grow.

If one wishes to find the names of all employees over 40, one would write

```
retrieve (E.name) using E in EMP
where E.age > 40
```

On the other hand, if one wanted the names of all salesmen or employees over 40, the notation is

```
retrieve (E.name) using E in EMP*
where E.age > 40
```

Here the * after the constructed type EMP indicates that the query should be run over EMP and all constructed types under EMP in the inheritance hierarchy. This use of * allows a user to easily run queries over a constructed type and all its descendents.

Lastly, POSTGRES supports the notion of *time travel*. This feature allows a user to run historical queries. For example to find the salary of Sam at time T one would query

```
retrieve (EMP.salary)
using EMP [T]
where EMP.name = "Sam"
```

POSTGRES will automatically find the version of Sam's record valid at the correct time and get the appropriate salary.

Like relational systems, the result of a POSTQUEL command can be added to the database as a new constructed type. In this case, POSTQUEL follows the lead of relational systems by removing duplicate records from the result. The user who is interested in retaining duplicates can do so by ensuring that the OID field of some instance is included in the target list being selected. For a full description of POSTQUEL the interested reader should consult [35].

II.D Fast Path

There are three reasons why we chose to implement a *fast path* feature. First, a user who wishes to interact with a database by executing a sequence of functions to navigate to desired data can use fast path to accomplish his objective. Second, there are a variety of decision support applications in which the end user is given a specialized query language. In such environments, it is often easier for the application developer to construct a parse tree representation for a query rather than an ASCII one. Hence, it would be desirable for the application designer to be able to directly call the POSTGRES optimizer or executor. Most DBMS's do not allow direct access to internal system modules.

The third reason is a bit more complex. In the persistent CLOS layer of **Picasso**, it is necessary for the run-time system to assign a unique identifier (OID) to every constructed object that is persistent. It is undesirable for the system to synchronously insert each object directly into a POSTGRES database and thereby assign a POSTGRES identifier to the object. This would result in poor performance in executing a persistent CLOS program. Rather, persistent CLOS maintains a cache of objects in the address space of the program and only inserts a persistent object into this cache synchronously. There are several options which control how the cache is written out to the database at a later time. Unfortunately, it is essential that a persistent object be assigned a unique identifier at the time it enters the cache, because other objects may have to point to the newly created object and use its OID to do so.

If persistent CLOS assigns unique identifiers, then there will be a complex mapping that must be performed when objects are written out to the database and real POSTGRES unique identifiers are assigned. Alternately, persistent CLOS must maintain its own system for unique identifiers, independent of the POSTGRES one, an obvious duplication of effort. The solution chosen was to allow persistent CLOS to access the POSTGRES routine that assigns unique identifiers and allow it to preassign N POSTGRES object identifiers which it can subsequently assign to cached objects. At a later time, these objects can be written to a POSTGRES database using the preassigned unique identifiers. When the supply of identifiers is exhausted, persistent CLOS can request another collection.

In all of these examples, an application program requires direct access to a user-defined or internal POSTGRES function, and therefore the POSTGRES query language has been extended with function-name (param-list)

In this case, besides running queries in POSTQUEL, a user can ask that any function known to POSTGRES be executed. This function can be one that a user has previously defined as a normal, operator, or POSTQUEL function or it can be one that is included in the POSTGRES implementation.

Hence, the user can directly call the parser, the optimizer, the executor, the access methods, the buffer manager, or the utility routines. In addition, he can define functions which in turn make calls on POSTGRES internals. In this way, he can have considerable control over the low-level flow of control, much as is available through a DBMS toolkit such as Exodus [20], but without all the effort involved in configuring a tailored DBMS from the toolkit. Moreover, should the user wish to interact with his database by making a collection of function calls (method invocations), this facility allows the possibility. As noted in the Introduction, we do not expect this interface to be especially popular.

The above capability is called *fast path* because it provided direct access to specific functions without checking the validity of parameters. As such, it is effectively a remote procedure call facility and allows a user program to call a function in another address space rather than in its own address space.

II.E Is POSTGRES Object-Oriented?

There have been many next-generation data models proposed in the last few years. Some are characterized by the term “extended relational,” others are considered “object-oriented,” while yet others are termed “nested relational.” POSTGRES could be accurately described as an object-oriented system because it includes unique identity for objects, abstract data types, classes (constructed types), methods (functions), and inheritance for both data and functions. Others (e.g., [2]) are suggesting definitions for the word “object-oriented,” and POSTGRES satisfies virtually all of the proposed litmus tests.

On the other hand, POSTGRES could also be considered an extended relational system. As noted in a previous footnote, Section II could have been equally well written with the word “constructed type” and “instance” replaced by the words “relation” and “tuple.” In fact, in previous descriptions of POSTGRES [26], this notation was employed. Hence, others, e.g., [18], have characterized POSTGRES as an extended relational system.

Lastly, POSTGRES supports the POSTQUEL type, which is exactly a nested relational structure. Consequently, POSTGRES could be classified as a nested relational system as well.

As a result, POSTGRES could be described using any of the three adjectives above. In our opinion, we can interchangeably use the words *relations*, *classes*, and *constructed types* in describing POSTGRES. Moreover, we can also interchangeably use the words *function* and *method*. Lastly, we can interchangeably use the words *instance*, *record*, and *tuple*. Hence, POSTGRES seems to be either object-oriented or not object-oriented, depending on the choice of a few tokens in the parser. As a result, we feel that most of the efforts to classify the extended data models in next-generation database systems are silly exercises in surface syntax.

In the remainder of this section, we comment briefly on the POSTGRES implementation of OID's and inheritance. POSTGRES gives each record a unique identifier (OID), and then allows the application designer to decide for each constructed type whether he wishes to have an index on the OID field. This decision should be contrasted with most object-oriented systems which construct an OID index for all constructed types in the system automatically. The POSTGRES scheme allows the cost of the index to be paid only for those types of objects for which it is profitable. In our opinion, this flexibility has been an excellent decision.

Second, there are several possible ways to implement an inheritance hierarchy. Considering the SALESMEN and EMP example noted earlier, one can store instances of SALESMAN by storing them as EMP records and then only storing the extra quota information in a separate SALESMAN record. Alternately, one can store no information on each salesman in EMP and then store complete SALESMAN records elsewhere. Clearly, there are a variety of additional schemes.

POSTGRES chose one implementation, namely storing all SALESMAN fields in a single record. However, it is likely that applications designers will demand several other representations to give them the flexibility to optimize their particular data. Future implementations of inheritance will likely require several storage options.

II.F A Critique of the POSTGRES Data Model

There are five areas where we feel we made mistakes in the POSTGRES data model:

union types

access method interface

functions

large objects

arrays.

We discuss each in turn.

A desirable feature in any next-generation DBMS would be to support union types, i.e., an instance of a type can be an instance of one of several given types. A persuasive example (similar to one from [10]) is that employees can be on loan to another plant or on loan to a customer. If two base types, customer and plant, exist, one would like to change the EMP type to

create EMP (name = *c*12, dept = DEPT, salary =
  float, on-loan-to = plant or customer)

Unfornately including union types makes a query optimizer more complex. For example, to find all the employees on loan to the same organization one would state the query

retrieve (EMP.name, *E*.name)
using *E* in EMP
where EMP.on-loan-to = *E*.on-loan-to

However, the optimizer must construct two different plans, one for employees on loan to a customer and one for employees on loan to a different plant. The reason for two plans is that the equality operator may be different for the two types. In addition, one must construct indexes on union fields, which entails substantial complexity in the access methods.

Union types are *highly* desirable in certain applications, and we considered three possible stances with respect to union types:

1) support only through abstract data types

2) support through POSTQUEL functions

3) full support.

Union types can be easily constructed using the POSTGRES abstract data type facility. If a user wants a specific union type, he can construct it and then write appropriate operators and functions for the type. The implementation complexity of union types is thus forced into the routines for the operators and functions and onto the implementor of the type. Moreover, it is clear that there are a vast number of union types and an extensive type library must be constructed by the application designer. The **Picasso** team stated that this approach placed an unacceptably difficult burden on them, and therefore position 1) was rejected.

Position 2) offers some support for union types but has problems. Consider the example of employees and their hobbies from [26].

create EMP (name = *c*12, hobbies = POSTQUEL)

Here the hobbies field is a POSTQUEL function, one per employee, which retrieves all hobby information about that particular employee. Now consider the following POSTQUEL query:

```
retrieve (EMP.hobbies.average)
    where EMP.name = “Fred”
```

In this case, the field average for each hobby record will be returned whenever it is defined. Suppose, however, that average is a float for the softball hobby and an integer for the cricket hobby. In this case, the application program must be prepared to accept values of different types.

The more difficult problem is the following legal POSTQUEL query:

```
retrieve into TEMP (result = EMP.hobbies.average)
    where EMP.name = “Fred”
```

In this case, a problem arises concerning the type of the result field, because it is a union type. Hence, adopting position 2) leaves one in an awkward position of not having a reasonable type for the result of the above query.

Of course, position 3) requires extending the indexing and query optimization routines to deal with union types. Our solution was to adopt position 2) and to add an abstract data type, ANY, which can hold an instance of any type. This solution which turns the type of the result of the above query from

one-of {integer, float}

into ANY is not very satisfying. Not only is information lost, but we are also forced to include with POSTGRES this universal type.

In our opinion, the only realistic alternative is to adopt position 3), swallow the complexity increase, and that is what we would do in any next system.

Another failure concerned the access method design and was the decision to support indexing only on the value of a field and not on a function of a value. The utility of indexes on functions of values is discussed in [17], and the capability was retrofitted, rather inelegantly, into one version of POSTGRES [4].

Another comment on the access method design concerns extendibility. Because a user can add new base types dynamically, it is essential that he also be able to add new access methods to POSTGRES if the system does not come with an access method that supports efficient access to his types. The standard example of this capability is the use of *R*-trees [15] to speed access to geometric objects. We have now designed and/or coded three access methods for POSTGRES in addition to *B*+-trees. Our experience has consistently been that adding an access method is *very hard*. There are four problems that complicate the situation. First, the access method must include explicit calls to the POSTGRES locking subsystem to set and release locks on access method objects. Hence, the designer of a new access method must understand locking and how to use the particular POSTGRES facilities. Second, the designer must understand how to interface to the buffer manager and be able to get, put, pin, and unpin pages. Next, the POSTGRES execution engine contains the "state" of the execution of any query and the access methods must understand portions of this state and the data structures involved. Last, but not least, the designer must write 13 nontrivial routines. Our experience so far is that novice programmers can add new types to POSTGRES; however, it requires a highly skilled programmer to add a new access method. Put differently, the manual on how to add new data types to POSTGRES is two pages long, the one for access methods is 50 pages.

We failed to realize the difficulty of access method construction. Hence, we designed a system that allows end users to add access methods dynamically to a running system. However, access methods will be built by sophisticated system programmers who could have used a simpler interface.

A third area where our design is flawed concerns POSTGRES support for POSTQUEL functions. Currently, such functions in POSTGRES are collections of commands in the query language POSTQUEL. If one defined budget in DEPT as a POSTQUEL function, then the value for the shoe department budget might be the following command:

```
retrieve (DEPT.budget) where DEPT.dname =
    "candy"
```

In this case, the shoe department will automatically be assigned the same budget as the candy department. However, it is impossible for the budget of the shoe department to be specified as

```
if floor = 1 then
    retrieve (DEPT.budget) where DEPT.dname =
        "candy"
else
    retrieve (DEPT.budget) where DEPT.dname =
        "toy"
```

This specification defines the budget of the shoe department to the candy department budget if it is on the first floor. Otherwise, it is the same as the toy department. This query is not possible because POSTQUEL has no conditional expressions. We had *extensive* discussions about this and other extensions to POSTQUEL. Each such extension was rejected because it seemed to turn POSTQUEL into a programming language and not a query language.

A better solution would be to allow a POSTQUEL function to be expressible in a general purpose programming language enhanced with POSTQUEL queries. Hence, there would be no distinction between normal functions and POSTQUEL functions. Put differently, normal functions would be able to be constructed types and would support path expressions.

There are three problems with this approach. First, path expressions for normal functions cannot be optimized by the POSTGRES query optimizer because they have arbitrary semantics. Hence, most of the optimizations planned for POSTQUEL functions would have to be discarded. Second, POSTQUEL functions are much easier to define than normal functions because a user need not know a general purpose programming language. Also, he need not specify the types of the function arguments or the return type because POSTGRES can figure these out from the query specification. Hence, we would have to give up ease of definition in moving from POSTQUEL functions to normal functions. Lastly, normal functions have a protection problem because they can do arbitrary things, such as zeroing the database. POSTGRES deals with this problem by calling normal functions in two ways:

trusted—loaded into the POSTGRES address space
untrusted—loaded into a separate address space.

Hence, normal functions are either called quickly with no security or slowly in a protected fashion. No such security problem arises with POSTQUEL functions.

A better approach might have been to support POSTQUEL functions written in the fourth generation language (4GL) being designed for **Picasso** [22]. This programming system leaves type information in the system catalogs. Consequently, there would be no need for a separate registration step to indicate type information to POSTGRES. Moreover, a processor for the language is available for integration in POSTGRES. It is also easy to make a 4GL "safe," i.e., unable to perform wild branches or malicious actions. Hence, there would be no security problem. Also, it seems possible that path expressions could be optimized for 4GL functions.

Current commercial relational products seem to be moving in this direction by allowing database procedures to be coded in their proprietary fourth generation languages (4GL's). In retrospect, we probably should have looked seriously at designing POSTGRES to support functions written in a 4GL.

Next, POSTGRES allows types to be constructed that are of arbitrary size. Hence, large bitmaps are a perfectly acceptable POSTGRES data type. However, the current POSTGRES user interface (portals) allows a user to fetch one or more instances of a constructed type. It is currently impossible to fetch only a portion of an instance. This presents an application program with a severe buffering problem; it must be capable of accepting an entire instance, no matter how large it is. We should extend the portal syntax in a straightforward way to allow an application to position a portal on a specific field of an instance of a constructed type and then specify a byte count that he would like to retrieve. These changes would make it much easier to insert and retrieve big fields.

Lastly, we included arrays in the POSTGRES data model. Hence, one could have specified the SALESMAN type as

```
create SALESMAN (name = c12, dept = DEPT, sal-
    ary = float, quota = float[12])
```

Here, the SALESMAN has all the fields of EMP plus a quota which is an array of 12 floats, one for each month of the year. In fact, character strings are really an array of characters, and the correct notation for the above type is

```
create SALESMAN (name = c[12], dept = DEPT,
    salary = float, quota = float[12])
```

In POSTGRES, we support fixed and variable length arrays of base types, along with an array notation in POSTQUEL. For example, to request all salesmen who have an April quota over 1000, one would write

```
retrieve (SALESMAN.name) where SALES-
    MAN.quota[4] > 1000
```

However, we do not support arrays of constructed types; hence, it is not possible to have an array of instances of a constructed type. We omitted this capability only because it would have made the query optimizer and executor somewhat harder. In addition, there is no built-in search mechanism for the elements of an array. For example, it is not possible to find the names of all salesmen who have a quota over 1000 during any month of the year. In retrospect, we should have included general support for arrays or no support at all.

III The Rules System

III.A Introduction

It is clear to us that all DBMS's need a rules system. Current commercial systems are required to support referential integrity [12], which is merely a simple-minded collection of rules. In addition, most current systems have special purpose rules systems to support relational views, protection, and integrity constraints. Lastly, a rules system allows users to do event-driven programming as well as enforce integrity constraints that cannot be performed in other ways. There are three high-level decisions that the POSTGRES team had to make concerning the philosophy of rule systems.

First, a decision was required concerning how many rule syntaxes there would be. Some approaches, e.g., [13], [36], propose rule systems oriented toward application designers that would augment other rule systems present for DBMS internal purposes. Hence, such systems would contain several independently functioning rules systems. On the other hand, [25] proposed a rule system that tried to support user functionality as well as needed DBMS internal functions in a single syntax.

From the beginning, a goal of the POSTGRES rules system was to have only one syntax. It was felt that this would simplify the user interface, since application designers need learn only one construct. Also, they would not have to deal with deciding which system to use in the cases where a function could be performed by more than one rules system. It was also felt that a single rules system would ease the implementation difficulties that would be faced.

Second, there are two implementation philosophies by which one could support a rule system. The first is a *query rewrite* implementation. Here, a rule would be applied by converting a user query to an alternate form prior to execution. This transformation is performed between the query language parser and the optimizer. Support for views [24] is done this way along with many of the proposals for recursive query support [5], [33]. Such an implementation will be very efficient when there are a small number of rules on any given constructed type and most rules cover the whole constructed type. For example, a rule such as

```
EMP [dept] contained-in DEPT[dname]
```

expresses the referential integrity condition that employees cannot be in a nonexistent department and applies to all EMP instances. However, a query rewrite implementation will not work well if there are a large number of rules on each constructed type, each of them covering only a few instances. Consider, for example, the following three rules:

employees in the shoe department have a steel desk

employees over 40 have a wood desk

employees in the candy department do not have a desk.

To retrieve the kind of a desk that Sam has, one must run the following three queries:

```
retrieve (desk = “steel”) where EMP.name = “Sam”
    and EMP.dept = “shoe”
retrieve (desk = “wood”) where EMP.name= “Sam”
    and EMP.age > 40
retrieve (desk = null) where EMP.name = “Sam” and
    EMP.dept = “candy”
```

Hence, a user query must be rewritten for each rule, resulting in a serious degradation of performance unless all queries are processed as a group using multiple query optimization techniques [23].

Moreover, a query rewrite system has great difficulty with *exceptions* [8]. For example, consider the rule “all employees have a steel desk” together with the exception “Jones is an employee who has a wood desk.” If one asks for the kind of desk and age for all employees over 35, then the query must be rewritten as the following two queries:

```
retrive (desk = “steel,” EMP.age) where EMP.age
    > 35 and EMP.name ! = “Jones”
retrieve (desk = “wood,” EMP.age) where EMP.age
    > 35 and EMP.name = “Jones”
```

In general, the number of queries as well as the complexity of their qualifications increases linearly with the number of rules. Again, this will result in bad performance unless multiple query optimization techniques are applied.

Lastly, a query rewrite system does not offer any help in resolving situations when the rules are violated. For example, the above referential integrity rule is silent on what to do if a user tries to insert an employee into a nonexistent department.

On the other hand, one could adopt a *trigger* implementation based on individual record accesses and updates to the database. Whenever a record is accessed, inserted, deleted, or modified, the low-level execution code has both the old record and the new record readily available. Hence, assorted actions can easily be taken by the low-level code. Such an implementation requires the rule firing code to be placed deep in the query execution routines. It will work well if there are many rules each affecting only a few instances, and it is easy to deal successfully with conflict resolution at this level. However, rule firing is deep in the executor, and it is thereby impossible for the query optimizer to construct an efficient execution plan for a chain of rules that are awakened.

Hence, this implementation complements a query rewrite scheme in that it excels where a rewrite scheme is weak and vice-versa. Since we wanted to have a single rule system, it was clear that we needed to provide both styles of implementation.

A third issue that we faced was the paradigm for the rules system. A conventional production system consisting of collections of if-then rules has been explored in the past [13], [25] and is a readily available alternative. However, such a scheme lacks expressive power. For example, suppose one wants to enforce a rule that Joe makes the same salary as Fred. In this case, one must specify two different if-then rules. The first one indicates the action to take if Fred receives a raise, namely to propagate the change on to Joe. The second rule specifies that any update to Joe's salary must be refused. Hence, many user rules require two or more if-then specifications to achieve the desired effect.

The intent in POSTGRES was to explore a more powerful paradigm. Basically, any POSTGRES command can be turned into a rule by changing the semantics of the command so that it is logically either *always* running or *never* running. For example, Joe may be specified to have the same salary as Fred by the rule

```
always replace EMP (salary = E.salary)
using E in EMP
where EMP.name = "Fred" and E.name = "Joe"
```

This single specification will propagate Joe's salary on to Fred as well as refuse direct updates to Fred's salary. In this way, a single "always" rule replaces the two statements needed in a production rule syntax.

Moreover, to efficiently support the triggering implementation where there are a large number of rules present for a single constructed type, each of which applies to only a few instances, the POSTGRES team designed a sophisticated marking scheme whereby rule wakeup information is placed on individual instances. Consequently, regardless of the number of rules present for a single constructed type, only those which actually must fire will be awakened. This should be contrasted to proposals without such data structures, which will be hopelessly inefficient whenever a large number of rules are present for a single constructed type.

Lastly, the decision was made to support the query rewrite scheme by escalating markers to the constructed type level. For example, consider the rule

```
always replace EMP (age = 40) where name ! =
    "Bill"
```

This rule applies to all employees except Bill and it would be a waste of space to mark each individual employee. Rather, one would prefer to set a single marker in the system catalogs to cover the whole constructed type implicitly. In this case, any query, e.g.,

```
retrieve (EMP.age) where EMP.name = "Sam"
```

will be altered prior to execution by the query rewrite implementation to

```
retrieve (age = 40) where EMP.name = "Sam" and
    EMP.name ! = "Bill"
```

At the current time, much of the POSTGRES rules system (PRS) as described in [30] is operational, and there are three aspects of the design which we wish to discuss in the next three subsections, namely,

complexity

absence of needed function

efficiency.

Then, we close with the second version of the POSTGRES rules system (PRS II) which we are currently designing. This rules system is described in more detail in [31], [32].

III.B Complexity

The first problem with PRS is that the implementation is exceedingly complex. It is difficult to explain the marking mechanisms that cause rule wakeup even to a sophisticated person. Moreover, some of us have an uneasy feeling that the implementation may not be quite correct. The fundamental problem can be illustrated using the Joe–Fred example above. First, the rule must be awakened and run whenever Fred's salary changes. This requires that one kind of marker be placed on the salary of Fred. However, if Fred is given a new name, say Bill, then the rule must be deleted and reinstalled. This requires a second kind of marker on the name of Fred. Additionally, it is inappropriate to allow any update to Joe's salary; hence, a third kind of marker is required on that field. Furthermore, if Fred has not yet been hired, then the rule must take effect on the insertion of his record. This requires a marker to be placed in the index for employee names. To support rules that deal with ranges of values, for example,

```
always replace EMP (age = 40)
where EMP.salary > 50000 and EMP.salary < 60000
```

we require that two “stub” markers be placed in the index to denote the ends of the scan. In addition, each intervening index record must also be marked. Ensuring that all markers are correctly installed and appropriate actions taken when record accesses and updates occur has been a challenge.

Another source of substantial complexity is the necessity to deal with priorities. For example, consider a second rule:

```
always replace EMP (age = 50) where EMP.dept =
    “shoe”
```

In this case, a highly paid shoe department employee would be given two different ages. To alleviate this situation, the second rule could be given a higher priority, e.g.,

```
always replace EMP (age = 50) where EMP.dept =
    ”shoe”
priority = 1
```

The default priority for rules is 0; hence, the first rule would set the age of highly paid employees to 40 unless they were in the shoe department, in which case their age would be set to 50 by the second rule. Priorities, of course, add complications to the rules system. For example, if the second rule above is deleted, then the first rule must be awakened to correct the ages of employees in the shoe department.

Another aspect of complexity is our decision to support both early and late evaluation of rules. Consider the example rule that Joe makes the same salary as Fred. This rule can be awakened when Fred gets a salary adjustment, or activation can be delayed until a user requests the salary of Joe. Activation can be delayed as long as possible in the second case, and we term this *late* evaluation while the former case is termed *early* evaluation. This flexibility also results in substantial extra complexity. For example, certain rules cannot be activated late. If salaries of employees are indexed, then the rule that sets Joe’s salary to that of Fred must be activated early because the index must be kept correct. Moreover, it is impossible for an early rule to read data that are written by a late rule. Hence, additional restrictions must be imposed.

Getting PRS correct has entailed uncounted hours of discussion and considerable implementation complexity. The bottom line is that the implementation of a rule system that is clean and simple to the user is, in fact, extremely complex and tricky. Our personal feeling is that we should have embarked on a more modest rules system.

III.C Absence of Needed Function

The definition of a *useful* rules system is one that can handle at least all of the following problems in one integrated system:

support for views

protection

referential integrity

other integrity constraints.

We focus in this section on support for views. The query rewrite implementation of a rules system should be able to translate queries on views into queries on real objects. In addition, updates to views should be similarly mapped to updates on real objects.

There are various special cases of view support that can be performed by PRS, for example materialized views. Consider the following view definitions:

```
define view SHOE-EMP (name = EMP.name, age =
    EMP.age, salary = EMP.salary)
where EMP.dept = "shoe"
```

The following two PRS rules specify a materialization of this view:

```
always append to SHOE-EMP (name = EMP.name,
    salary = EMP.salary) where EMP.dept = "shoe"
always delete SHOE-EMP where SHOE-EMP.name
    NOTIN {EMP.name where EMP.dept = "shoe")
```

In this case, SHOE-EMP will always contain a correct materialization of the shoe department employees, and queries can be directed to this materialization.

However, there seemed to be no way to support updates on views that are not materialized. One of us has spent countless hours attempting to support this function through PRS and failed. Hence, inability to support operations provided by conventional views is a major weakness of PRS.

III.D Implementation Efficiency

The current POSTGRES implementation uses markers on individual fields to support rule activation. The only escalation supported is to convert a collection of field level markers to a single marker on the entire constructed type. Consequently, if a rule covers a single instance, e.g.,

```
always replace EMP (salary = 1000) where EMP.name
    = “Sam”
```

then a total of three markers will be set, one in the index, one on the salary field, and one on the name field. Each marker is composed of

rule-id	6 bytes
priority	1 byte
marker-type	1 byte.

Consequently, the marker overhead for the rule is 24 bytes. Now consider a more complex rule:

```
always replace EMP (salary = 1000) where EMP.dept
    = “shoe”
```

If 1000 employees work in the shoe department, then 24K bytes of overhead will be consumed in markers. The only other option is to escalate to a marker on the entire constructed type, in which case the rule will be activated if any salary is read or written and not just for employees in the shoe department. This will be an overhead intensive option. Hence, for rules which cover many instances but not a significant fraction of all instances, the POSTGRES implementation will not be very space efficient.

We are considering several solutions to this problem. First, we have generalized B+-trees to efficiently store interval data as well as point data. Such “segmented B+-trees” are the subject of a separate paper [16]. This will remove the space overhead in the index for the dominant form of access method. Second, to lower the overhead on data records, we will probably implement markers at the physical block level as well as at the instance and constructed type levels. The appropriate extra granularities are currently under investigation.

III.E The Second POSTGRES Rules System

Because of the inability of the current rules paradigm to support views and to a lesser extent the fundamental complexity of the implementation, we are converting to a second POSTGRES rules system (PRS II). This rules system has much in common with the first implementation, but returns to the traditional production rule paradigm to obtain sufficient control to perform view updates correctly. This section outlines our thinking, and a complete proposal appears in [32].

The production rule syntax we are using in PRS II has the form

```
ON event TO object
    WHERE POSTQUEL-qualification
THEN DO POSTQUEL-command(s)
```

Here, event is RETRIEVE, REPLACE, DELETE, APPEND, UPDATE, NEW (i.e., replace or append) or old (i.e., delete or replace). Moreover, object is either the name of a constructed type or constructed-type.column. POSTQUEL-qualification is a normal qualification, with no additions or changes. Lastly, POSTQUEL-commands is a set of POSTQUEL commands with the following two changes:

NEW, OLD, or CURRENT can appear instead of the
    name of a constructed type in front of any attribute
refuse (target-list) is added as a new POSTQUEL command

In this notation, we would specify the "Fred-Joe" rule as

```
on NEW EMP.salary where EMP.name = "Fred"
then do
    replace E (salary = CURRENT.salary)
    using E in EMP
    where E.name = "Joe"

on NEW EMP.salary where EMP.name = "Joe"
then do
    refuse
```

Notice, that PRS II is less powerful than the "always" system because the Fred–Joe rule requires two specifications instead of one.

PRS II has both a query rewrite implementation and a trigger implementation, and it is an optimization decision which one to use as noted in [32]. For example, consider the rule

```
on RETRIEVE to SHOE-EMP
then do
retrieve (EMP.name, EMP.age, EMP.salary)
    where EMP.dept = “shoe”
```

Any query utilizing such a rule, e.g.,

```
retrieve (SHOE-EMP.name) where SHOE-EMP.age
    < 40
```

would be processed by the rewrite implementation to

```
retrieve (EMP.name) where EMP.age < 40 and
    EMP.dept = “shoe”
```

As can be seen, this is identical to the query modification performed in relational view processing techniques [24]. This rule could also be processed by the triggering system, in which case the rule would materialize the records in SHOE-EMP iteratively.

Moreover, it is straightforward to support additional functionality, such as allowing multiple queries in the definition of a view. Supporting materialized views can be efficiently done by *caching* the action part of the above rule, i.e., executing the command before a user requests evaluation. This corresponds to moving the rule to early evaluation. Lastly, supporting views that are partly materialized and partly specified as procedures as well as views that involve recursion appears fairly simple. In [32], we present details on these extensions.

Consider the following collection of rules that support updates to SHOE-EMP:

```
on NEW SHOE-EMP
then do
    append to EMP (name = NEW.name, salary =
        NEW.salary)

on OLD SHOE-EMP
then do
    delete EMP where EMP.name = OLD.name and
    EMP.salary = OLD.salary

on update to SHOE-EMP
then do
    replace EMP (name = NEW.name, salary =
        NEW.salary)
    where EMP.name = NEW.name
```

If these rules are processed by the trigger implementation, then an update to SHOE-EMP, e.g.,

replace SHOE-EMP (salary = 1000) where SHOE-
EMP.name = "Mike"

will be processed normally until it generates a collection of

[new-record, old-record]

pairs. At this point the triggering system can be activated to make appropriate updates to underlying constructed types. Moreover, if a user wishes nonstandard view update semantics, he can perform any particular actions he desires by changing the action part of the above rules.

PRS II thereby allows a user to use the rules system to define semantics for retrievals and updates to views. In fact, we expect to build a compiler that will convert a higher level view notation into the needed collection of PRS II rules. In addition, PRS II retains all the functionality of the first rules system, so protection, alerters, integrity constraints, and arbitrary triggers are readily expressed. The only disadvantage is that PRS II requires two rules to perform many tasks expressible as a single PRS rule. To overcome this disadvantage, we will likely continue to support the PRS syntax in addition to the PRS II syntax and compile PRS into PRS II support.

PRS II can be supported by the same implementation that we proposed for the query rewrite implementation of PRS, namely marking instances in the system catalogs. Moreover, the query rewrite algorithm is nearly the same as in the first implementation. The triggering system can be supported by the same instance markers as in PRS. In fact, the implementation is bit simpler because a couple of the types of markers are not required. Because the implementation of PRS II is so similar to our initial rules system, we expect to have the conversion completed in the near future.

IV Storage System

IV.A Introduction

When considering the POSTGRES storage system, we were guided by a missionary zeal to do something different. All current commercial systems use a storage manager with a write-ahead log (WAL), and we felt that this technology was well understood. Moreover, the original INGRES prototype from the 1970's used a similar storage manager, and we had no desire to do another implementation.

Hence, we seized on the idea of implementing a “no-overwrite” storage manager. Using this technique, the old record remains in the database whenever an update occurs, and serves the purpose normally performed by a write-ahead log. Consequently, POSTGRES has no log in the conventional sense of the term. Instead the POSTGRES log is simply 2 bits per transaction indicating whether each transaction committed, aborted, or is in progress.

Two very nice features can be exploited in a no-overwrite system. First, aborting a transaction can be instantaneous because one does not need to process the log undoing the effects of updates; the previous records are readily available in the database. More generally, to recover from a crash, one must abort all the transactions in progress at the time of the crash. This process can be effectively instantaneous in POSTGRES.

The second benefit of a no-overwrite storage manager is the possibility of *time travel*. As noted earlier, a user can ask a historical query and POSTGRES will automatically return information from the record valid at the correct time.

This storage manager should be contrasted with a conventional one where the previous record is overwritten with a new one. In this case, a write-ahead log is required to maintain the previous version of each record. There is no possibility of time travel because the log cannot be queried since it is in a different format. Moreover, the database must be restored to a consistent state when a crash occurs by processing the log to undo any partially completed transactions. Hence, there is no possibility of instantaneous crash recovery.

Clearly a no-overwrite storage manager is superior to a conventional one if it can be implemented at comparable performance. There is a brief hand-wave argument in [28] that alleges this might be the case. In our opinion, the argument hinges around the existence of *stable* main memory. In the absence of stable memory, a no-overwrite storage manager must force to disk at commit time all pages written by a transaction. This is required because the effects of a committed transaction must be durable in case a crash occurs and main memory is lost. A conventional data manager, on the other hand, need only force to disk at commit time the log pages for the transaction’s updates. Even if there are as many log pages as data pages (a highly unlikely occurrence), the conventional storage manager is doing sequential I/O to the log while a no-overwrite storage manager is doing random I/O. Since sequential I/O is substantially faster than random I/O, the no-overwrite solution is guaranteed to offer worse performance.

However, if stable main memory is present, then neither solution must force pages to disk. In this environment, performance should be comparable. Hence, with stable main memory it appears that a no-overwrite solution is competitive. As computer manufacturers offer some form of stable main memory, a no-overwrite solution may become a viable storage option.

In designing the POSTGRES storage system, we were guided by two philosophical premises. First, we decided to make a clear distinction between current data and historical data. We expected access patterns to be highly skewed toward current records. In addition, queries to the archive might look very different from those accessing current data. For both reasons, POSTGRES maintains two different physical collections of records, one for the current data and one for historical data, each with its own indexes.

Second, our design assumes the existence of a randomly addressable archive device on which historical records are placed. Our intuitive model for this archive is an optical disk. Our design was purposely made consistent with an archive that has a write-once-read-many (WORM) orientation. This characterizes many of the optical disks on the market today.

In the next subsection, we indicate two problems with the POSTGRES design. Then, in Section 1.5.3 we make additional comments on the storage manager.

IV.B Problems in the POSTGRES Design

There are at least two problems with our design. First, it is unstable under heavy load. An asynchronous demon, known as vacuum cleaner, is responsible for moving historical records from the magnetic disk structure holding the current records to the archive where historical records remain. Under normal circumstances, the magnetic disk portion of each constructed type is (say) only 1.1 times the minimum possible size of the constructed type. Of course, the vacuum cleaner consumes CPU and I/O resources running in background achieving this goal. However, if the load on a POSTGRES database increases, then the vacuum cleaner may not get to run. In this case, the magnetic disk portion of a constructed type will increase, and performance will suffer because the execution engine must read historical records on the magnetic disk during the (presumably frequent) processing of queries to the current database. As a result, performance will degrade proportionally to the excess size of the magnetic disk portion of the database. As load increases, the vacuum cleaner gets less resources, and performance degrades as the size of the magnetic disk database increases. This will ultimately result in a POSTGRES database going into meltdown.

Obviously, the vacuum cleaner should be run in background if possible so that it can consume resources at 2:00 A.M. when there is little other activity. However, if there is consistent heavy load on a system, then the vacuum cleaner must be scheduled at the same priority as other tasks, so the above instability does not occur.

The bottom line is that scheduling the vacuum cleaner is a tricky optimization problem.

The second comment which we wish to make is that future archive systems are likely to be read/write, and rewritable optical disks have already appeared on the market. Consequently, there is no reason for us to have restricted ourselves to WORM technology. Certain POSTGRES assumptions were therefore unnecessary, such as requiring the current portion of any constructed type to be on magnetic disk.

IV.C Other Comments

Historical indexes will usually be on a combined key consisting of a time range together with one or more keys from the record itself. Such two-dimensional indexes can be stored using the technology of *R*-trees [15], *R*+-trees [14], or perhaps in some new way. We are not particularly comfortable that good ways to index time ranges have been found, and we encourage additional work in this area. A possible approach is segmented R-trees which we are studying [16].

Another comment concerns POSTGRES support for time travel. There are many tasks that are very difficult to express with our mechanisms. For example, the query to find the time at which Sam's salary increased from $5000 to $6000 is very tricky in POSTQUEL.

A last comment is that time travel can be implemented with a conventional transaction system using a write ahead log. For example, one need only have an "archive" constructed type for each physical constructed type for which time travel is desired. When a record is updated, its previous value is written in the archive with the appropriate timestamps. If the transaction fails to commit, this archive insert and the corresponding record update is unwound using a conventional log. Such an implementation may well have substantial benefits, and we should have probably considered such a possibility. In making storage system decisions, we were guided by a missionary zeal to do something different than a conventional write ahead log scheme. Hence, we may have overlooked other intriguing options.

V The POSTGRES Implementation

V.A Introduction

POSTGRES contains a fairly conventional parser, query optimizer, and execution engine. Two aspects of the implementation deserve special mention,

dynamic loading and the process structure
choice of implementation language

and we discuss each in turn.

V.B Dynamic Loading and Process Structure

POSTGRES assumes that data types, operators, and functions can be added and subtracted dynamically, i.e., while the system is executing. Moreover, we have designed the system so that it can accommodate a potentially very large number of types and operators. Consequently, the user functions that support the implementation of a type must be dynamically loaded and unloaded. Hence, POSTGRES maintains a cache of currently loaded functions and dynamically moves functions into the cache and then ages them out of the cache. Moreover, the parser and optimizer run off of a main memory cache of information about types and operators. Again this cache must be maintained by POSTGRES software. It would have been much easier to assume that all types and operators were linked into the system at POSTGRES initialization time and have required a user to reinstall POSTGRES when he wished to add or drop types. Moreover, users of prototype software are not running systems which cannot go down for rebooting. Hence, the function is not essential.

Second, the rules system forces significant complexity on the design. A user can add a rule such as

```
always retrieve (EMP.salary)
where EMP.name = “Joe”
```

In this case, his application process wishes to be notified of any salary adjustment to Joe. Consider a second user who gives Joe a raise. The POSTGRES process that actually does the adjustment will notice that a marker has been placed on the salary field. However, in order to alert the first user, one of four things must happen.

1. POSTGRES could be designed as a single server process. In this case, within the current process the first user’s query could simply be activated. However, such a design is incompatible with running on a shared memory multiprocessor, where a so-called multiserver is required. Hence, this design was discarded.
2. The POSTGRES process for the second user could run the first user’s query and then connect to his application process to deliver results. This requires that an application process be coded to expect communication from random other processes. We felt this was too difficult to be a reasonable solution.

3. The POSTGRES process for the second user could connect to the input socket for the first user's POSTGRES and deliver the query to be run. The first POSTGRES would run the query and then send results to the user. This would require careful synchronization of the input socket among multiple independent command streams. Moreover, it would require the second POSTGRES to know the portal name on which the first user's rule was running.
4. The POSTGRES process for the second user could alert a special process called the *POSTMASTER*. This process would in turn alert the process for the first user where the query would be run and the results delivered to the application process.

We have adopted the fourth design as the only one we thought was practical. However, we have thereby constructed a process through which everybody must channel communications. If the POSTMASTER crashes, then the whole POSTGRES environment must be restarted. This is a handicap, but we could think of no better solution. Moreover, there are a collection of system demons, including the vacuum cleaner mentioned above, which need a place to run. In POSTGRES, they are run as subprocesses managed by the POSTMASTER.

A last aspect of our design concerns the operating system process structure. Currently, POSTGRES runs as one process for each active user. This was done as an expedient to get a system operational as quickly as possible. We plan on converting POSTGRES to use lightweight processes available in the operating systems we are using. These include PRESTO for the Sequent Symmetry and threads in Version 4 of Sun/OS.

V.C Programming Language Used

At the beginning of the project, we were forced to make a commitment to a programming language and machine environment. The machine was an easy one, since SUN workstations were nearly omnipresent at Berkeley, and any other choice would have been nonstandard. However, we were free to choose any language in which to program. We considered the following:

C

C++

MODULA 2+

LISP

ADA

SMALLTALK.

We dismissed SMALLTALK quickly because we felt it was too slow and compilers were not readily available for a wide variety of platforms. We felt it desirable to keep open the option of distributing our software widely. We felt ADA and MODULA 2+ offered limited advantages over C++ and were not widely used in the Berkeley environment. Hence, obtaining pretrained programmers would have been a problem. Lastly, we were not thrilled to use C, since INGRES had been coded in C and we were anxious to choose a different language, if only for the sake of doing something different. At the time we started (10/85), there was not a stable C++ compiler, so we did not seriously consider this option.

By a process of elimination, we decided to try writing POSTGRES in LISP. We expected that it would be especially easy to write the optimizer and inference engine in LISP, since both are mostly tree processing modules. Moreover, we were seduced by AI claims of high programmer productivity for applications written in LISP.

We soon realized that parts of the system were more easily coded in C, for example the buffer manager which moves 8K pages back and forth to the disk and uses a modified LRU algorithm to control what pages are resident. Hence, we adopted the policy that we would use both C and LISP and code modules of POSTGRES in whichever language was most appropriate. By the time Version 1 was operational, it contained about 17K lines in LISP and about 63K lines of C.

Our feeling is that the use of LISP has been a terrible mistake for several reasons. First, current LISP environments are very large. To run a "nothing" program in LISP requires about 3 mbytes of address space. Hence, POSTGRES exceeds 4 mbytes in size, all but 1 mbyte is the LISP compiler, editor and assorted other nonrequired (or even desired) functions. Hence, we suffer from a gigantic footprint. Second, a DBMS never wants to stop when garbage collection happens. Any response time sensitive program must therefore allocate and deallocate space manually, so that garbage collection never happens during normal processing. Consequently, we spent extra effort ensuring that LISP garbage collection is not used by POSTGRES. Hence, this aspect of LISP, which improves programmer productivity, was not available to us. Third, LISP execution is slow. As noted in the performance figures in the next section, our LISP code is more than twice as slow as the comparable C code. Of course, it is possible that we are not skilled LISP programmers or do not know how to optimize the language; hence, our experience should be suitably discounted.

However, none of these irritants was the real disaster. We have found that debugging a two-language system is extremely difficult. The C debugger, of course, knows nothing about LISP while the LISP debugger knows nothing about C. As a result, we have found debugging POSTGRES to be a painful and frustrating task. Memory allocation bugs were among the most painful since LISP and C have very different models of dynamic memory. Of course, it is true that the optimizer and inference engine were easier to code in LISP. Hence, we saved some time there. However, this was more than compensated by the requirement of writing a lot of utility code that would convert LISP data structures into C and vice versa. In fact, our assessment is that the primary productivity increases in LISP come from the nice programming environment (e.g., interactive debugger, nice workstation tools, etc.) and not from the language itself. Hence, we would encourage the implementors of other programming languages to study the LISP environment carefully and implement the better ideas.

As a result we have just finished moving our 17K lines of LISP to C to avoid the debugging hassle and secondarily to avoid the performance and footprint problems in LISP. Our experience with LISP and two-language systems has not been positive, and we would caution others not to follow in our footsteps.

VI Status and Performance

At the current time (October 1989) the LISP-less Version 1 of POSTGRES has been in the hands of users for a short time, and we are shaking the last bugs out of the C port. In addition, we have designed all of the additional functionality to appear in Version 2. The characteristics of Version 1 are the following.

1) The query language POSTQUEL runs except for aggregates, functions, and set operators.

2) All object management capabilities are operational except POSTQUEL types.

3) Some support for rules exists. Specifically, replace always commands are operational; however, the implementation currently only supports early evaluation and only with markers on whole columns.

4) The storage system is complete. However, we are taking delivery shortly on an optical disk jukebox, and so the archive is currently not implemented on a real optical disk. Moreover, *R*-trees to support time travel are not yet implemented.

5) Transaction management runs.

The focus has been on getting the function in POSTGRES to run. So far, only minimal attention has been paid in performance. Figure 1 shows assorted queries in the Wisconsin benchmark and gives results for three systems running on a Sun 3/280. All numbers are run on a nonquiescent system so there may be significant fluctuations. The first two are the C and LISP versions of POSTGRES. These are functionally identical systems with the same algorithms embodied in the code. The footprint of the LISP system is about 4.5 megabytes while the C system is about I megabyte. For comparison purposes we also include the performance numbers for the commercial version of INGRES in the third column. As can be seen, the LISP

	POSTGRES C-based	POSTGRES LISP-based	INGRES RTI 5.0
nullqry	0.4	0.3	0.2
scan 10Ktups	36.	180.	5.2
retrieve into query 1% selectivity	38.	n/a	9.9
append to 10Ktup	4.7	180.	0.4
delete from 10Ktup	37.	n/a	5.7
replace in 10Ktup	42.	280.	5.7

Figure 1 Comparison of INGRES and POSTGRES (times are listed in seconds per query).

system is several times slower than the C system. In various other benchmarks, we have never seen the C system less than twice as fast as the LISP system. Moreover, the C system is several times slower than a commercial system. The public domain version of INGRES that we worked on the mid 1970's is about a factor of two slower than commercial INGRES. Hence, it appears that POSTGRES is about one-half the speed of the original INGRES. There are substantial inefficiencies in POSTGRES, especially in the code which checks that a retrieved record is valid. We expect that subsequent tuning will get us somewhere in between the performance of public domain INGRES and RTI INGRES.

VII Conclusions

In this section, we summarize our opinions about certain aspects of the design of POSTGRES. First, we are uneasy about the complexity of the POSTGRES data model. The comments in Section II all contain suggestions to make it more complex. Moreover, other research teams have tended to construct even more complex data models, e.g., EXTRA [9]. Consequently, a simple concept such as referential integrity, which can be done in only one way in existing commercial systems, can be done in several different ways in POSTGRES. For example, the user can implement an abstract data type and then do the required checking in the input conversion routine. Alternately, he can use a rule in the POSTGRES rules system. Lastly, he can use a POSTQUEL function for the field that corresponds to the foreign key in a current relational system. There are complex performance tradeoffs between these three solutions, and a decision must be made by a sophisticated application designer. We fear that real users, who have a hard time with database design for existing relational systems, will find the next-generation data models, such as the one in POSTGRES, impossibly complex. The problem is that applications exist where each representation is the only acceptable one. The demand for wider application of database technology ensures that vendors will produce systems with these more complex data models.

Another source of uneasiness is the fact that rules and POSTQUEL functions have substantial overlap in function. For example, a POSTQUEL function can be simulated by one rule per record, albeit at some performance penalty. On the other hand, all rules, except retrieve always commands, can be alternately implemented using POSTQUEL functions. We expect to merge the two concepts in Version 2, and our proposal appears in [32].

In the areas of rules and storage management, we are basically satisfied with POSTGRES capabilities. The syntax of the rule system should be changed as noted in Section III; however, this is not a significant issue and it should be available easily in Version 2. The storage manager has been quite simple to implement. Crash recovery code has been easy to write because the only routine which must be carefully written is the vacuum cleaner. Moreover, access to past history seems to be a highly desirable capability.

Furthermore, the POSTGRES implementation certainly erred in the direction of excessive sophistication. For example, new types and functions can be added on-the-fly without recompiling POSTGRES. It would have been much simpler to construct a system that required recompilation to add a new type. Second, we have implemented a complete transaction system in Version 1. Other prototypes tend to assume a single user environment. In these and many other ways, we strove for substantial generality; however, the net effect has been to slow down the implementation effort and make the POSTGRES internals much more complex. As a result, POSTGRES has taken us considerably longer to build than the original version of INGRES. One could call this the "second system" effect. It was essential that POSTGRES be more usable than the original INGRES prototype in order for us to feel like we were making a contribution.

A last comment concerns technology transfer to commercial systems. It appears that the process is substantially accelerating. For example, the relational model was constructed in 1970, first prototypes of implementations appeared around 1976–1977, commercial versions first surfaced around 1981 and popularity of relational systems in the marketplace occurred around 1985. Hence, there was a 15 year period during which the ideas were transferred to commercial systems. Most of the ideas in POSTGRES and in other next-generation systems date from 1984 or later.

Commercial systems embodying some of these ideas have already appeared and major vendors are expected to have advanced systems within the next year or two. Hence, the 15 year period appears to have shrunk to less than half that amount. This acceleration is impressive, but it will lead to rather short lifetimes for the current collection of prototypes.

References

[1] R. Agrawal and N. Gehani, "ODE: The language and the data model," in *Proc. 1989 ACM SIGMOD. Conference Management Data*, Portland, OR, May 1989.

[2] M. Atkinson *et al.*, "The object-oriented database system manifesto," Altair Tech. Rep. 30-89, Rocquencourt, France, Aug. 1989.

[3] Anon *et al.*, "A measure of transaction processing power," Tandem Computers, Tech. Rep. 85.1, Cupertino, CA, 1985.

[4] P. Aoki, "Implementation of extended indexes in POSTGRES," Electron. Res. Lab. Univ. of California, Tech. Rep. 89-62, July 1989.

[5] F. Bancilhon and R. Ramakrishnan, "An amateur's introduction to recursive query processing," in *Proc. 1986 ACM SIGMOD Conf. Management Data*, Washington, DC. May 1986.

[6] J. Banerjee *et al.*, "Semantics and implementation of schema evolution in object-oriented databases," in *Proc. 1987 ACM SIGMOD Conf. Management Data*, San Francisco, CA, May 1987.

[7] D. Bitton *et al.*, "Benchmarking database systems: A systematic approach," in *Proc. 1983 VLDB Conf*., Cannes, France, Sept. 1983.

[8] A. Borgida, "Language features for flexible handling of exceptions in information systems," *ACM TODS*, Dec. 1985.

[9] M. Carey *et al.*, "A data model and query language for EXODUS," in *Proc. 1988 ACM SJGMOD Conf. Management Data*, Chicago, IL, June 1988.

[10] G. Copeland and D. Maier, "Making Smalltalk a database system," in *Proc. 1984 ACM SIGMOD Conf. Management Data*, Boston, MA, June 1984.

[11] P. Dadam *et al.*, "A DBMS prototype to support NF2 relations," in *Proc. 1986 ACM SIGMOD Conf. Management Data*, Washington, DC, May 1986.

[12] C. Date, "Referential integrity," in *Proc. Seventh Int. VLDB Conf*., Cannes, France, Sept. 1981.

[13] K. Eswaren, "Specification, implementation and interactions of a rule subsystem in an integrated database system," IBM Res., San Jose, CA, Res. Rep. RJ1820, Aug. 1976.

[14] C. Faloutsos *et al.*, "Analysis of object oriented spatial access methods," in *Proc. 1987 ACM SJGMOD Conf. Management Data*, San Francisco, CA, May 1987.

[15] A. Gutman, "R-trees: A dynamic index structure for spatial searching," in *Proc. 1984 ACM SJGMOD Conf. Management Data*, Boston, MA, June 1984.

[16] C. Kolovson and M. Stonebraker, “Segmented search trees and their application to data bases,” in preparation.

[17] C. Lynch and M. Stonebraker, “Extended user-defined indexing with application to textual databases,” in *Proc. 1988 VLDB Conf*., Los Angeles, CA, Sept. 1988.

[18] D. Maier, “Why isn’t there an object-oriented data model?” in *Proc. 11th IFIP World Congress*, San Francisco, CA, Aug. 1989.

[19] S. Osborne and T. Heaven, “The design of a relational system with abstract data types as domains,” *ACM TODS*, Sept. 1986.

[20] J. Richardson and M. Carey, “Programming constructs for database system implementation in EXODUS,” in *Proc. 1987 ACM SIGMOD Conf. Management Data*, San Francisco, CA, May 1987.

[21] L. Rowe and M. Stonebraker, “The POSTGRES data model,” in *Proc. 1987 VLDB Conf*., Brighton, England, Sept. 1987.

[22] L. Rowe *et al*., “The design and implementation of Picasso,” in preparation.

[23] T. Sellis, “Global query optimization,” in *Proc. 1986 ACM SIGMOD Conf. Management Data*, Washington, DC, June 1986.

[24] M. Stonebraker, “Implementation of integrity constraints and views by query modification,” in *Proc. 1975 ACM SIGMOD Conf*., San Jose, CA, May 1975.

[25] M. Stonebraker *et al*., “A rules system for a relational data base management system,” in *Proc. 2nd Int. Conf. Databases*, Jerusalem, Israel, June 1982. New York: Academic.

[26] M. Stonebraker and L. Rowe, “The design of POSTGRES,” in *Proc. 1986 ACM-SJGMOD Conf*., Washington, DC, June 1986.

[27] M. Stonebraker, “Inclusion of new types in relational data base systems,” in *Proc. Second Int. Conf. Data Eng*., Los Angeles, CA, Feb. 1986.

[28] ———, “The POSTGRES storage system,” in *Proc. 1987 VLDB Conf*., Brighton, England, Sept. I987.

[29] M. Stonebraker *et al*., “Extensibility in POSTGRES,” *IEEE Database Eng*., Sept. 1987.

[30] M. Stonebraker *et al*., ”The POSTGRES rules system,” *IEEE Trans. Software Eng*., July 1988.

[31] M. Stonebraker *et al*., “Commentary on the POSTGRES rules system.” SIGMOD Rec., Sept. 1989.

[32] M. Stonebraker *et al*., “Rules, procedures and views,” in preparation.

[33] J. Ullman, “Implementation of logical query languages for databases,” *ACM TODS*, Sept. 1985.

[34] F. Velez *et al*., “The O2 object manager: An overview,” GIP ALTAIR, Le Chesnay, France, Tech. Rep. 27-89, Feb. 1989.

[35] S. Wensel, Ed., “The POSTGRES reference manual,” Electron. Res. Lab., Univ. of California, Berkeley. CA, Rep. M88/20, Mar. 1988.

[36] J. Widom and S. Finkelstein, “A syntax and semantics for set-oriented production rules in relational data bases,” IBM Res., San Jose, CA, June 1989.

Michael Stonebraker, for a photograph and biography, see this issue, p. 3

Lawrence A. Rowe received the B.A. degree in mathematics and the Ph.D. degree in information and computer science from the University of California, Irvine, in 1970 and 1976, respectively.

Since 1976, he has been on the faculty at the University of California, Berkeley. His primary research interests are database application development tools and integrated circuit (IC) computer-integrated manufacturing. He designed and implemented Rigel, a Pascal-like database programming language, and the Forms Application Development System, a forms-based 4GL with integrated application generators. He is currently implementing an object-oriented, graphical user-interface development system called Picasso and a programming language to specify IC fabrication process plans called the Berkeley Process-Flow Language. He is a co-founder and director of Ingres., Inc., which markets the INGRES relational database management system.

Michael Hirohama received the B.A. degree in computer science from the University of California, Berkeley, in 1987.

Since that time, he has been the lead programmer for the POSTGRES project.

The Design and Implementation of INGRES

Michael Stonebraker (University of California, Berkeley),
Eugene Wong (University of California, Berkeley),
Peter Kreps (University of California, Berkeley),
Gerald Held (Tandem Computers, Inc.)

The currently operational (March 1976) version of the INGRES database management system is described. This multiuser system gives a relational view of data, supports two high level nonprocedural data sublanguages, and runs as a collection of user processes on top of the UNIX operating system for Digital Equipment Corporation PDP 11/40, 11/45, and 11/70 computers. Emphasis is on the design decisions and tradeoffs related to (1) structuring the system into processes, (2) embedding

This research was sponsored by Army Research Office Grant DAHCO4-74-G0087, the Naval Electronic Systems Command Contract N00039-76-C-0022, the Joint Services Electronics Program Contract F44620-71-C-0087, National Science Foundation Grants DCR75-03839 and ENG74-06651-A01, and a grant from the Sloan Foundation.
Authors' addresses: M. Stonebraker and E. Wong, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley, CA 94720; P. Kreps, Department of Computer Science and Applied Mathematics, Building 50B, Lawrence Berkeley Laboratories, University of California, Berkeley, Berkeley, CA 94720; G. Held, Tandem Computers, Inc., Cupertino, CA 95014.
Originally published in *ACM Transactions on Database Systems,* 1(3): 189–222, 1976. Original DOI: 10.1145/320473.320476

one command language in a general purpose programming language, (3) the algorithms implemented to process interactions, (4) the access methods implemented, (5) the concurrency and recovery control currently provided, and (6) the data structures used for system catalogs and the role of the database administrator.

Also discussed are (1) support for integrity constraints (which is only partly operational), (2) the not yet supported features concerning views and protection, and (3) future plans concerning the system.

Key Words and Phrases: relational database, nonprocedural language, query language, data sublanguage, data organization, query decomposition, database optimization, data integrity, protection, concurrency
CR Categories: 3.50, 3.70, 4.22, 4.33, 4.34

1 Introduction

INGRES (Interactive Graphics and Retrieval System) is a relational database system which is implemented on top of the UNIX operating system developed at Bell Telephone Laboratories [22] for Digital Equipment Corporation PDP 11/40, 11/45, and 11/70 computer systems. The implementation of INGRES is primarily programmed in C, a high level language in which UNIX itself is written. Parsing is done with the assistance of YACC, a compiler-compiler available on UNIX [19].

The advantages of a relational model for database management systems have been extensively discussed in the literature [7, 10, 11] and hardly require further elaboration. In choosing the relational model, we were particularly motivated by (a) the high degree of data independence that such a model affords, and (b) the possibility of providing a high level and entirely procedure free facility for data definition, retrieval, update, access control, support of views, and integrity verification.

1.1 Aspects Described in This Paper

In this paper we describe the design decisions made in INGRES. In particular we stress the design and implementation of: (a) the system process structure (see Section 2 for a discussion of this UNIX notion); (b) the embedding of all INGRES commands in the general purpose programming language C; (c) the access methods implemented; (d) the catalog structure and the role of the database administrator; (e) support for views, protection, and integrity constraints; (f) the decomposition procedure implemented; (g) implementation of updates and consistency of secondary indices; (h) recovery and concurrency control.

In Section 1.2 we briefly describe the primary query language supported, Quel, and the utility commands accepted by the current system. The second user interface, Cupid, is a graphics oriented, casual user language which is also operational [20, 21] but not discussed in this paper. In Section 1.3 we describe the Equel (Embedded Quel) precompiler, which allows the substitution of a user supplied C program for the "front end" process. This precompiler has the effect of embedding all of INGRES in the general purpose programming language C. In Section 1.4 a few comments on Quel and Equel are given.

In Section 2 we describe the relevant factors in the UNIX environment which have affected our design decisions. Moreover, we indicate the structure of the four processes into which INGRES is divided and the reasoning behind the choices implemented.

In Section 3 we indicate the catalog (system) relations which exist and the role of the database administrator with respect to all relations in a database. The implemented access methods, their calling conventions, and, where appropriate, the actual layout of data pages in secondary storage are also presented.

Sections 4, 5, and 6 discuss respectively the various functions of each of the three "core" processes in the system. Also discussed are the design and implementation strategy of each process. Finally, Section 7 draws conclusions, suggests future extensions, and indicates the nature of the current applications run on INGRES.

Except where noted to the contrary, this paper describes the INGRES system operational in March 1976.

1.2 QUEL and the Other INGRES Utility Commands

Quel (QUEry Language) has points in common with Data Language/Alpha [8], Square [3], and Sequel [4] in that it is a complete query language which frees the programmer from concern for how data structures are implemented and what algorithms are operating on stored data [9]. As such it facilitates a considerable degree of data independence [24].

The Quel examples in this section all concern the following relations.

EMPLOYEE	(NAME, DEPT, SALARY, MANAGER, AGE)
DEPT	(DEPT, FLOOR#)

A Quel interaction includes at least one RANGE statement of the form

RANGE OF variable-list IS relation-name

The purpose of this statement is to specify the relation over which each variable ranges. The variable-list portion of a RANGE statement declares variables which will be used as arguments for tuples. These are called *tuple variables.*

An interaction also includes one or more statements of the form

```
Command    [result-name] (target-list)
           [WHERE Qualification]
```

Here Command is either RETRIEVE, APPEND, REPLACE, or DELETE. For RETRIEVE and APPEND, result-name is the name of the relation which qualifying tuples will be retrieved into or appended to. For REPLACE and DELETE, result-name is the name of a tuple variable which, through the qualification, identifies tuples to be modified or deleted. The target-list is a list of the form

```
result-domain = QUEL Function . . . .
```

Here the result-domains are domain names in the result relation which are to be assigned the values of the corresponding functions.

The following suggest valid Quel interactions. A complete description of the language is presented in [15].

Example 1 Compute salary divided by age-18 for employee Jones.

```
RANGE OF E IS EMPLOYEE
RETRIEVE INTO W
(COMP = E.SALARY/(E.AGE-18))
WHERE E.NAME = "Jones"
```

Here E is a tuple variable which ranges over the EMPLOYEE relation, and all tuples in that relation are found which satisfy the qualification E.NAME = "Jones." The result of the query is a new relation W, which has a single domain COMP that has been calculated for each qualifying tuple.

If the result relation is omitted, qualifying tuples are written in display format on the user's terminal or returned to a calling program.

Example 2 Insert the tuple (Jackson,candy,13000,Baker,30) into EMPLOYEE.

```
APPEND TO EMPLOYEE(NAME = "Jackson", DEPT = "candy",
    SALARY = 13000, MGR = "Baker", AGE = 30)
```

Here the result relation EMPLOYEE is modified by adding the indicated tuple to the relation. Domains which are not specified default to zero for numeric domains and null for character strings. A shortcoming of the current implemenation is that 0 is not distinguished from "no value" for numeric domains.

Example 3 Fire everybody on the first floor.

```
RANGE OF E IS EMPLOYEE
RANGE OF D IS DEPT
DELETE E WHERE  E.DEPT = D.DEPT
                AND    D.FLOOR# = 1
```

Here E specifies that the EMPLOYEE relation is to be modified. All tuples are to be removed which have a value for DEPT which is the same as some department on the first floor.

Example 4 Give a 10-percent raise to Jones if he works on the first floor.

```
RANGE OF E IS EMPLOYEE
RANGE OF D IS DEPT
REPLACE E(SALARY = 1.1*E.SALARY)
WHERE   E.NAME = "Jones" AND
        E.DEPT = D.DEPT AND D.FLOOR# = I
```

Here E.SALARY is to be replaced by 1.1*E.SALARY for those tuples in EMPLOYEE where the qualification is true.

In addition to the above Quel commands, INGRES supports a variety of utility commands. These utility commands can be classified into seven major categories.

(a) Invocation of INGRES:

```
INGRES data-base-name
```

This command executed from UNIX "logs in" a user to a given database. (A database is simply a named collection of relations with a given database administrator who has powers not available to ordinary users.) Thereafter the user may issue all other commands (except those executed directly from UNIX) within the environment of the invoked database.

(b) Creation and destruction of databases:

```
CREATEDB data-base-name
DESTROYDB data-base-name
```

These two commands are called from UNIX. The invoker of CREATEDB must be authorized to create databases (in a manner to be described presently), and he automatically becomes the database administrator. DESTROYDB successfully destroys a database only if invoked by the database administrator.

(c) Creation and destruction of relations:

```
CREATE relname(domain-name IS format, domain-name IS format, . . . )
DESTROY relname
```

These commands create and destroy relations within the current database. The invoker of the CREATE command becomes the "owner" of the relation created. A user may only destroy a relation that he owns. The current formats accepted by INGRES are 1-, 2-, and 4-byte integers, 4- and 8-byte floating point numbers, and 1- to 255-byte fixed length ASCII character strings.

(d) Bulk copy of data:

```
COPY relname(domain-name IS format, domain-name IS format, . . . )
    direction "file-name"
PRINT relname
```

The command COPY transfers an entire relation to or from a UNIX file whose name is "filename." Direction is either TO or FROM. The format for each domain is a description of how it appears (or is to appear) in the UNIX file. The relation relname must exist and have domain names identical to the ones appearing in the COPY command. However, the formats need not agree and COPY will automatically convert data types. Support is also provided for dummy and variable length fields in a UNIX file.

PRINT copies a relation onto the user's terminal, formatting it as a report. In this sense it is stylized version of COPY.

(e) Storage structure modification:

```
MODIFY relname TO storage-structure ON (key1, key2, . . . )
INDEX ON relname IS indexname(key1, key2, . . . )
```

The MODIFY command changes the storage structure of a relation from one access method to another. The five access methods currently supported are discussed in Section 3. The indicated keys are domains in relname which are concatenated left to right to form a combined key which is used in the organization of tuples in all but one of the access methods. Only the owner of a relation may modify its storage structure.

INDEX creates a secondary index for a relation. It has domains of key1, key2, . . . , pointer. The domain "pointer" is the unique identifier of a tuple in the indexed relation having the given values for key1, key2, ... An index named AGEINDEX for EMPLOYEE might be the following binary relation (assuming that there are six tuples in EMPLOYEE with appropriate names and ages).

	Age	Pointer
	25	identifier for Smith's tuple
	32	identifier for Jones's tuple
AGEINDEX	36	identifier for Adams's tuple
	29	identifier for Johnson's tuple
	47	identifier for Baker's tuple
	58	identifier for Harding's tuple

The relation indexname is in turn treated and accessed just like any other relation, except it is automatically updated when the relation it indexes is updated. Naturally, only the owner of a relation may create and destroy secondary indexes for it.

(f) Consistency and integrity control:

INTEGRITY CONSTRAINT is qualification
INTEGRITY CONSTRAINT LIST relname
INTEGRITY CONSTRAINT OFF relname
INTEGRITY CONSTRAINT OFF (integer, . . . , integer)
RESTORE data-base-name

The first four commands support the insertion, listing, deletion, and selective deletion of integrity constraints which are to be enforced for all interactions with a relation. The mechanism for handling this enforcement is discussed in Section 4. The last command restores a database to a consistent state after a system crash. It must be executed from UNIX, and its operation is discussed in Section 6. The RESTORE command is only available to the database administrator.

(g) Miscellaneous:

HELP [relname or manual-section]
SAVE relname UNTIL expiration-date
PURGE data-base-name

HELP provides information about the system or the database invoked. When called with an optional argument which is a command name, HELP returns the appropriate page from the INGRES reference manual [31]. When called with a relation name as an argument, it returns all information about that relation. With no argument at all, it returns information about all relations in the current database.

SAVE is the mechanism by which a user can declare his intention to keep a relation until a specified time. PURGE is a UNIX command which can be invoked by a database administrator to delete all relations whose “expiration-dates” have passed. This should be done when space in a database is exhausted. (The database administrator can also remove any relations from his database using the DESTROY command, regardless of who their owners are.)

Two comments should be noted at this time.

(a) The system currently accepts the language specified as QUEL_1 in [15]; extension is in progress to accept QUEL_n. (b) The system currently does not accept views or protection statements. Although the algorithms have been specified [25, 27], they are not yet operational. For this reason no syntax for these statements is given in this section; however the subject is discussed further in Section 4.

1.3 EQUEL

Although QUEL alone provides the flexibility for many data management requirements, there are applications which require a customized user interface in place of the QUEL language. For this as well as other reasons, it is often useful to have the flexibility of a general purpose programming language in addition to the database facilities of QUEL. To this end, a new language, EQUEL (Embedded QUEL), which consists of QUEL embedded in the general purpose programming language C, has been implemented.

In the design of EQUEL the following goals were set: (a) The new language must have the full capabilities of both C and QUEL. (b) The C program should have the capability for processing each tuple individually, thereby satisfying the qualification in a RETRIEVE statement. (This is the “piped” return facility described in Data Language/ALPHA [8].)

With these goals in mind, EQUEL was defined as follows:

(a) Any C language statement is a valid EQUEL statement.

(b) Any QUEL statement (or INGRES utility command) is a valid EQUEL statement as long as it is prefixed by two number signs (##).

(c) C program variables may be used anywhere in QUEL statements except as command names. The declaration statements of C variables used in this manner must also be prefixed by double number signs.

(d) RETRIEVE statements without a result relation have the form

```
RETRIEVE    (target-list)
            [WHERE qualification]
```

```
##{
C-block
##}
```

which results in the C-block being executed once for each qualifying tuple.

Two short examples illustrate EQUEL syntax.

Example 5 The following program implements a small front end to INGRES which performs only one query. It reads in the name of an employee and prints out the employee's salary in a suitable format. It continues to do this as long as there are names to be read in. The functions READ and PRINT have the obvious meaning.

```
main()
{
## char EMPNAME[20];
## int SAL;
while (READ(EMPNAME))
          {
##        RANGE OF X IS EMP
##        RETRIEVE (SAL = X.SALARY)
##        WHERE X.NAME = EMPNAME
                    ##{
                    PRINT("The salary of", EMPNAME, "is", SAL);
                    ##}
          }
}
```

In this example the C variable *EMPNAME* is used in the qualification of the QUEL statement, and for each qualifying tuple the C variable *SAL* is set to the appropriate value and then the PRINT statement is executed.

Example 6 Read in a relation name and two domain names. Then for each of a collection of values which the second domain is to assume, do some processing on all values which the first domain assumes. (We assume the function PROCESS exists and has the obvious meaning.) A more elaborate version of this program could serve as a simple report generator.

```
main()
{
## int VALUE;
## char RELNAME[13], DOMNAME[13], DOMVAL[80];
## char DOMNAME 2[13];
READ(RELNAME);
READ(DOMNAME);
READ(DOMNAME 2);
## RANGE OF X IS RELNAME
while (READ(DOMVAL))
            {
##          RETRIEVE (VALUE= X.DOMNAME)
##              WHERE X.DOMNAME 2 = DOMVAL
                    ##{
                    PROCESS(VALUE);
                    ##}
            }
}
```

Any RANGE declaration (in this case the one for X) is assumed by INGRES to hold until redefined. Hence only one RANGE statement is required, regardless of the number of times the RETRIEVE statement is executed. Note clearly that anything except the name of an INGRES command can be a C variable. In the above example *RELNAME* is a C variable used as a relation name, while *DOMNAME* and *DOMNAME* 2 are used as domain names.

1.4 Comments on QUEL and EQUEL

In this section a few remarks are made indicating differences between QUEL and EQUEL and selected other proposed data sublanguages and embedded data sublanguages.

QUEL borrows much from Data Language/ALPHA. The primary differences are: (a) Arithmetic is provided in QUEL; Data Language/ALPHA suggests reliance on a host language for this feature. (b) No quantifiers are present in QUEL. This results in a consistent semantic interpretation of the language in terms of functions on the crossproduct of the relations declared in the RANGE statements. Hence, QUEL is considered by its designers to be a language based on functions and not on a first order predicate calculus. (c) More powerful aggregation capabilities are provided in QUEL.

The latest version of SEQUEL [2] has grown rather close to QUEL. The reader is directed to Example 1(b) of [2], which suggests a variant of the QUEL syntax. The main differences between QUEL and SEQUEL appear to be: (a) SEQUEL allows statements with no tuple variables when possible using a block oriented notation.

(b) The aggregation facilities of SEQUEL appear to be different from those defined in QUEL.

System R [2] contains a proposed interface between SEQUEL and PL/1 or other host language. This interface differs substantially from EQUEL and contains explicit cursors and variable binding. Both notions are implicit in EQUEL. The interested reader should contrast the two different approaches to providing an embedded data sublanguage.

2 The INGRES Process Structure

INGRES can be invoked in two ways: First, it can be directly invoked from UNIX by executing INGRES database-name; second, it can be invoked by executing a program written using the EQUEL precompiler. We discuss each in turn and then comment briefly on why two mechanisms exist. Before proceeding, however, a few details concerning UNIX must be introduced.

2.1 The UNIX Environment

Two points concerning UNIX are worthy of mention in this section.

(a) The UNIX file system. UNIX supports a tree structured file system similar to that of MULTICS. Each file is either a directory (containing references to descendant files in the file system) or a data file. Each file is divided physically into 512-byte blocks (pages). In response to a read request, UNIX moves one or more pages from secondary memory to UNIX core buffers and then returns to the user the actual byte string desired. If the same page is referenced again (by the same or another user) while it is still in a core buffer, no disk I/O takes place.

It is important to note that UNIX pages data from the file system into and out of system buffers using a “least recently used” replacement algorithm. In this way the entire file system is managed as a large virtual store.

The INGRES designers believe that a database system should appear as a user job to UNIX. (Otherwise, the system would operate on a nonstandard UNIX and become less portable.) Moreover the designers believe that UNIX should manage the system buffers for the mix of jobs being run. Consequently, INGRES contains no facilities to do its own memory management.

(b) The UNIN process structure. A process in UNIX is an address space (64K bytes or less on an 11/40, 128K bytes or less on an 11/45 or 11/70) which is associated with a user-id and is the unit of work scheduled by the UNIX scheduler. Processes may “fork” subprocesses; consequently a parent process can be the root of a process subtree. Furthermore, a process can request that UNIX execute a file

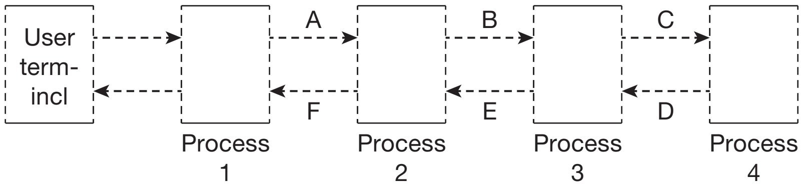


Figure 1 INGRES process structure

in a descendant process. Such processes may communicate with each other via an interprocess communication facility called "pipes." A pipe may be declared as a one direction communication link which is written into by one process and read by a second one. UNIX maintains synchronization of pipes so no messages are lost. Each process has a "standard input device" and a "standard output device." These are usually the user's terminal, but may be redirected by the user to be files, pipes to other processes, or other devices.

Last, UNIX provides a facility for processes executing reentrant code to share procedure segments if possible. INGRES takes advantage of this facility so the core space overhead of multiple concurrent users is only that required by data segments.

2.2 Invocation from UNIX

Issuing INGRES as a UNIX command causes the process structure shown in Figure 1 to be created. In this section the functions in the four processes will be indicated. The justification of this particular structure is given in Section 2.4.

Process 1 is an interactive terminal monitor which allows the user to formulate, print, edit, and execute collections of INGRES commands. It maintains a workspace with which the user interacts until he is satisfied with his interaction. The contents of this workspace are passed down pipe A as a string of ASCII characters when execution is desired. The set of commands accepted by the current terminal monitor is indicated in [31].

As noted above, UNIX allows a user to alter the standard input and output devices for his processes when executing a command. As a result the invoker of INGRES may direct the terminal monitor to take input from a user file (in which case he runs a "canned" collection of interactions) and direct output to another device (such as the line printer) or file.

Process 2 contains a lexical analyzer, a parser, query modification routines for integrity control (and, in the future, support of views and protection), and concurrency control. Because of size constraints, however, the integrity control routines are not in the currently released system. When process 2 finishes, it passes a string of tokens to process 3 through pipe B. Process 2 is discussed in Section 4.

Process 3 accepts this token string and contains execution routines for the commands RETRIEVE, REPLACE, DELETE, and APPEND. Any update is turned into a RETRIEVE command to isolate tuples to be changed. Revised copies of modified tuples are spooled into a special file. This file is then processed by a "deferred update processor" in process 4, which is discussed in Section 6.

Basically, process 3 performs two functions for RETRIEVE commands. (a) A multivariable query is *decomposed* into a sequence of interactions involving only a single variable. (b) A one-variable query is executed by a one-variable query processor (OVQP). The OVQP in turn performs its function by making calls on the access methods. These two functions are discussed in Section 5; the access methods are indicated in Section 3.

All code to support utility commands (CREATE, DESTROY, INDEX, etc.) resides in process 4. Process 3 simply passes to process 4 any commands which process 4 will execute. Process 4 is organized as a collection of overlays which accomplish the various functions. Some of these functions are discussed in Section 6.

Error messages are passed back through pipes D, E, and F to process 1, which returns them to the user. If the command is a RETRIEVE with no result relation specified, process 3 returns qualifying tuples in a stylized format directly to the "standard output device" of process 1. Unless redirected, this is the user's terminal.

2.3 Invocation from EQUEL

We now turn to the operation of INGRES when invoked by code from the precompiler.

In order to implement EQUEL, a translator (precompiler) was written to convert an EQUEL program into a valid C program with QUEL statements converted to appropriate C code and calls to INGRES. The resulting C program is then compiled by the normal C compiler, producing an executable module. Moreover, when an EQUEL program is run, the executable module produced by the C compiler is used as the front end process in place of the interactive terminal monitor, as noted in Figure 2.

During execution of the front end program, database requests (QUEL statements in the EQUEL program) are passed through pipe A and processed by INGRES. Note that unparsed ASCII strings are passed to process 2; the rationale behind this decision is given in [1]. If tuples must be returned for tuple at a time processing, then they are returned through a special data pipe set up between process 3 and the

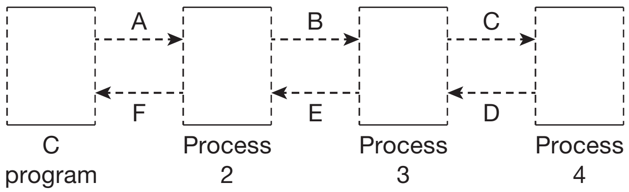


Figure 2 The forked process structure

C program. A condition code is also returned through pipe F to indicate success or the type of error encountered.

The functions performed by the EQUEL translator are discussed in detail in [1].

2.4 Comments on the Process Structure

The process structure shown in Figures 1 and 2 is the fourth different process structure implemented. The following considerations suggested this final choice:

(a) Address space limitations. To run on an 11/40, the 64K address space limitation must be adhered to. Processes 2 and 3 are essentially their maximum size; hence they cannot be combined. The code in process 4 is in several overlays because of size constraints.

Were a large address space available, it is likely that processes 2, 3, and 4 would be combined into a single large process. However, the necessity of 3 "core" processes should not degrade performance substantially for the following reasons.

If one large process were resident in main memory, there would be no necessity of swapping code. However, were enough real memory available (~300K bytes) on a UNIX system to hold processes 2 and 3 and all overlays of process 4, no swapping of code would necessarily take place either. Of course, this option is possible only on an 11/70.

On the other hand, suppose one large process was paged into and out of main memory by an operating system and hardware which supported a virtual memory. It is felt that under such conditions page faults would generate I/O activity at approximately the same rate as the swapping/overlaying of processes in INGRES (assuming the same amount of real memory was available in both cases).

Consequently the only sources of overhead that appear to result from multiple processes are the following: (1) Reading or writing pipes require system calls which are considerably more expensive than subroutine calls (which could be used in a single-process system). There are at least eight such system calls needed to execute an INGRES command. (2) Extra code must be executed to format information for transmission on pipes. For example, one cannot pass a pointer to a data structure through a pipe; one must linearize and pass the whole structure.

(b) Simple control flow. The grouping of functions into processes was motivated by the desire for simple control flow. Commands are passed only to the right; data and errors only to the left. Process 3 must issue commands to various overlays in process 4; therefore, it was placed to the left of process 4. Naturally, the parser must precede process 3.

Previous process structures had a more complex interconnection of processes. This made synchronization and debugging much harder.

The structure of process 4 stemmed from a desire to overlay little-used code in a single process. The alternative would have been to create additional processes 5, 6, and 7 (and their associated pipes), which would be quiescent most of the time. This would have required added space in UNIX core tables for no real advantage.

The processes are all synchronized (i.e. each waits for an error return from the next process to the right before continuing to accept input from the process to the left), simplifying the flow of control. Moreover, in many instances the various processes *must* be synchronized. Future versions of INGRES may attempt to exploit parallelism where possible. The performance payoff of such parallelism is unknown at the present time.

(c) Isolation of the front end process. For reasons of protection the C program which replaces the terminal monitor as a front end must run with a user-id different from that of INGRES. Otherwise it could tamper directly with data managed by INGRES. Hence, it must be either overlayed into a process or run in its own process. The latter was chosen for efficiency and convenience.

(d) Rationale for two process structures. The interactive terminal monitor could have been written in Equel. Such a strategy would have avoided the existence of two process structures which differ only in the treatment of the data pipe. Since the terminal monitor was written prior to the existence of Equel, this option could not be followed. Rewriting the terminal monitor in Equel is not considered a high priority task given current resources. Moreover, an Equel monitor would be slightly slower because qualifying tuples would be returned to the calling program and then displayed rather than being displayed directly by process 3.

3 Data Structures and Access Methods

We begin this section with a discussion of the files that INGRES manipulates and their contents. Then we indicate the five possible storage structures (file formats) for relations. Finally we sketch the access methods language used to interface uniformly to the available formats.

3.1 The INGRES File Structure

Figure 3 indicates the subtree of the UNIX file system that INGRES manipulates. The root of this subtree is a directory made for the UNIX user "INGRES." (When the INGRES system is initially installed such a user must be created. This user is known as the "superuser" because of the powers available to him. This subject is discussed further in [28].) This root has six descendant directories. The AUX directory has descendant files containing tables which control the spawning of processes (shown in Figures 1 and 2) and an authorization list of users who are allowed to create databases. Only the INGRES superuser may modify these files (by using the UNIX editor). BIN and SOURCE are directories indicating descendant files of respectively object and source code. TMP has descendants which are temporary files for the workspaces used by the interactive terminal monitor. DOC is the root of a subtree with system documentation and the reference manual. Last, there is a directory entry in DATADIR for each database that exists in INGRES. These directories contain the database files in a given database as descendants.

These database files are of four types:

(a) Administration file. This contains the user-id of the database administrator (DBA) and initialization information.

(b) Catalog (system) relations. These relations have predefined names and are created for every database. They are owned by the DBA and constitute the system catalogs. They may be queried by a knowledgeable user issuing RETRIEVE statements; however, they may be updated only by the INGRES utility commands (or directly by the INGRES superuser in an emergency). (When protection statements are implemented the DBA will be able to selectively restrict RETRIEVE access to these relations if he wishes.) The form and content of some of these relations will be discussed presently.

(c) DBA relations. These are relations owned by the DBA and are shared in that any user may access them. When protection is implemented the DBA can "authorize" shared use of these relations by inserting protection predicates (which will be in one of the system relations and may be unique for each user) and deauthorize use by removing such predicates. This mechanism is discussed in [28].

(d) Other relations. These are relations created by other users (by RETRIEVE INTO W or CREATE) and are *not shared.*

Three comments should be made at this time.

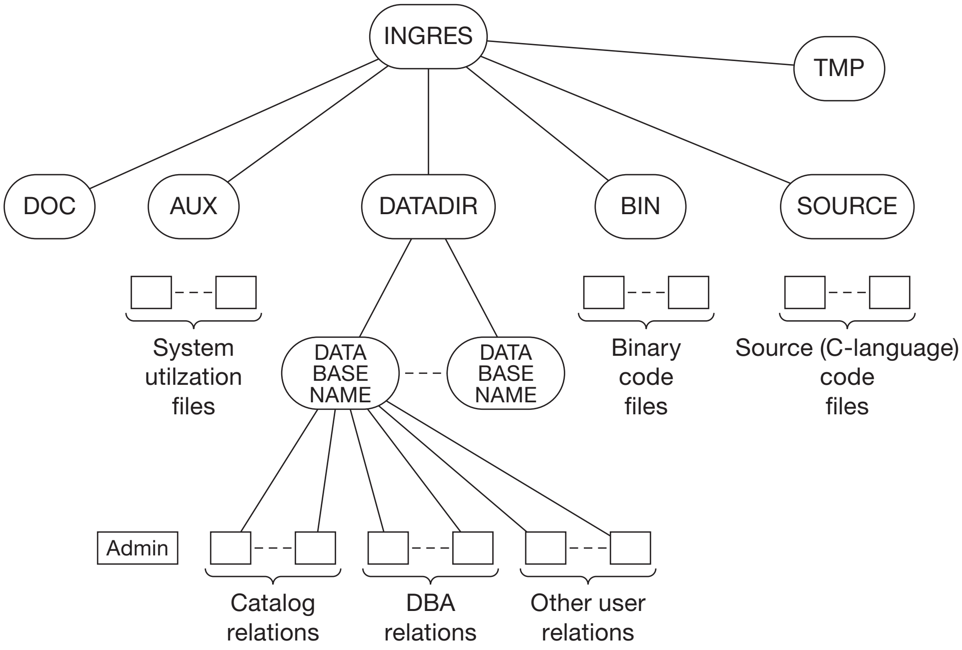


Figure 3 The INGRES subtree

(a) The DBA has the following powers not available to ordinary users: the ability to create shared relations and to specify access control for them; the ability to run PURGE; the ability to destroy any relations in his database (except the system catalogs).

This system allows "one-level sharing" in that only the DBA has these powers, and he cannot delegate any of them to others (as in the file systems of most time sharing systems). This strategy was implemented for three reasons: (1) The need for added generality was not perceived. Moreover, added generality would have created tedious problems (such as making revocation of access privileges nontrivial). (2) It seems appropriate to entrust to the DBA the duty (and power) to resolve the policy decision which must be made when space is exhausted and some relations must be destroyed or archived. This policy decision becomes much harder (or impossible) if a database is not in the control of one user. (3) Someone must be entrusted with the policy decision concerning which relations are physically stored and which are defined as "views." This "database design" problem is best centralized in a single DBA.

(b) Except for the single administration file in each database, every file is treated as a relation. Storing system catalogs as relations has the following advantages: (1) Code is economized by sharing routines for accessing both catalog and data relations. (2) Since several storage structures are supported for accessing data relations quickly and flexibly under various interaction mixes, these same storage choices may be utilized to enhance access to catalog information. (3) The ability to execute QUEL statements to examine (and patch) system relations where necessary has greatly aided system debugging.

(c) Each relation is stored in a separate file, i.e. no attempt is made to "cluster" tuples from *different* relations which may be accessed together on the same or on a nearby page.

Note clearly that this clustering is analogous to DBTG systems in declaring a record type to be accessed via a set type which associates records of that record type with a record of a different record type. Current DBTG implementations usually attempt to physically cluster these associated records.

Note also that clustering tuples from one relation in a given file has obvious performance implications. The clustering techniques of this nature that INGRES supports are indicated in Section 3.3.

The decision not to cluster tuples from different relations is based on the following reasoning. (1) UNIX has a small (512-byte) page size. Hence it is expected that the number of tuples which can be grouped on the same page is small. Moreover, logically adjacent pages in a UNIX file are *not necessarily* physically adjacent. Hence clustering tuples on "nearby" pages has no meaning in UNIX; the next logical page in a file may be further away (in terms of disk arm motion) than a page in a different file. In keeping with the design decision of *not* modifying UNIX, these considerations were incorporated in the design decision not to support clustering. (2) The access methods would be more complicated if clustering were supported. (3) Clustering of tuples only makes sense if associated tuples can be linked together using "sets" [6], "links" [29], or some other scheme for identifying clusters. Incorporating these access paths into the decomposition scheme would have greatly increased its complexity.

It should be noted that the designers of System R have reached a different conclusion concerning clustering [2].

3.2 System Catalogs

We now turn to a discussion of the system catalogs. We discuss two relations in detail and indicate briefly the contents of the others.

The RELATION relation contains one tuple for every relation in the database (including all the system relations). The domains of this relation are:

relid the name of the relation.

owner the UNIX user-id of the relation owner; when appended to relid it produces a unique file name for storing the relation.

spec indicates one of five possible storage schemes or else a special code indicating a virtual relation (or "view").

indexd flag set if secondary index exists for this relation. (This flag and the following two are present to improve performance by avoiding catalog lookups when possible during query modification and one variable query processing.)

protect flag set if this relation has protection predicates.

integ flag set if there are integrity constraints.

save scheduled lifetime of relation.

tuples number of tuples in relation (kept up to date by the routine "closer" discussed in the next section).

atts number of domains in relation.

width width (in bytes) of a tuple.

prim number of primary file pages for this relation.

The ATTRIBUTE catalog contains information relating to individual domains of relations. Tuples of the ATTRIBUTE catalog contain the following items for each domain of every relation in the database:

relid name of relation in which attribute appears.

owner relation owner.

domain_name domain name.

domain_no domain number (position) in relation. In processing interactions INGRES uses this number to reference this domain.

offset offset in bytes from beginning of tuple to beginning of domain.

type data type of domain (integer, floating point, or character string).

length length (in bytes) of domain.

keyno if this domain is part of a key, then "keyno" indicates the ordering of this domain within the key.

These two catalogs together provide information about the structure and content of each relation in the database. No doubt items will continue to be added or deleted as the system undergoes further development. The first planned extensions are the minimum and maximum values assumed by domains. These will be used by a more sophisticated decomposition scheme being developed, which is discussed briefly in Section 5 and in detail in [30]. The representation of the catalogs as relations has allowed this restructuring to occur very easily.

Several other system relations exist which provide auxiliary information about relations. The INDEX catalog contains a tuple for every secondary index in the database. Since secondary indices are themselves relations, they are independently cataloged in the RELATION and ATTRIBUTE relations. However, the INDEX catalog provides the association between a primary relation and its secondary indices and records which domains of the primary relation are in the index.

The PROTECTION and INTEGRITY catalogs contain respectively the protection and integrity predicates for each relation in the database. These predicates are stored in a partially processed form as character strings. (This mechanism exists for INTEGRITY and will be implemented in the same way for PROTECTION.) The VIEW catalog will contain, for each virtual relation, a partially processed QUEL-like description of the view in terms of existing relations. The use of these last three catalogs is described in Section 4. The existence of any of this auxiliary information for a given relation is signaled by the appropriate flag(s) in the RELATION catalog.

Another set of system relations consists of those used by the graphics subsystem to catalog and process maps, which (like everything else) are stored as relations in the database. This topic has been discussed separately in [13].

3.3 Storage Structures Available

We will now describe the five storage structures currently available in INGRES. Four of the schemes are keyed, i.e. the storage location of a tuple within the file is a function of the value of the tuple's key domains. They are termed "hashed," "ISAM," "compressed hash," and "compressed ISAM." For all four structures the key may be any ordered collection of domains. These schemes allow rapid access to specific portions of a relation when key values are supplied. The remaining nonkeyed scheme (a "heap") stores tuples in the file independently of their values and provides a low overhead storage structure, especially attractive in situations requiring a complete scan of the relation.

The nonkeyed storage structure in INGRES is a randomly ordered sequential file. Fixed length tuples are simply placed sequentially in the file in the order supplied. New tuples added to the relation are merely appended to the end of the file. The unique tuple identifier for each tuple is its byte-offset within the file. This mode is intended mainly for (a) very small relations, for which the overhead of other schemes is unwarranted; (b) transitional storage of data being moved into or out of the system by COPY; (c) certain temporary relations created as intermediate results during query processing.

In the remaining four schemes the key-value of a tuple determines the page of the file on which the tuple will be placed. The schemes share a common "page-structure" for managing tuples on file pages, as shown in Figure 4.

A tuple must fit entirely on a single page. Its unique tuple identifier (TID) consists of a page number (the ordering of its page in the UNIX file) plus a line number. The line number is an index into a line table, which grows upward from the bottom of the page, and whose entries contain pointers to the tuples on the page. In this way the physical arrangement of tuples on a page can be reorganized without affecting TIDs.

Initially the file contains all its tuples on a number of primary pages. If the relation grows and these pages fill, overflow pages are allocated and chained by pointers to the primary pages with which they are associated. Within a chained group of pages no special ordering of tuples is maintained. Thus in a keyed access which locates a particular primary page, tuples matching the key may actually appear on any page in the chain.

As discussed in [16], two modes of key-to-address transformation are used—randomizing (or "hashing") and order preserving. In a "hash" file tuples are distributed randomly throughout the primary pages of the file according to a hashing function on a key. This mode is well suited for situations in which access is to be conditioned on a specific key value.

As an order preserving mode, a scheme similar to IBM's ISAM [18] is used. The relation is sorted to produce the ordering on a particular key. A multilevel directory is created which records the high key on each primary page. The directory, which is static, resides on several pages following the primary pages within the file itself. A primary page and its overflow pages are *not* maintained in sort order. This decision is discussed in Section 4.2. The "ISAM-like" mode is useful in cases where the key value is likely to be specified as falling within a range of values, since a near ordering of the keys is preserved. The index compression scheme discussed in [16] is currently under implementation.

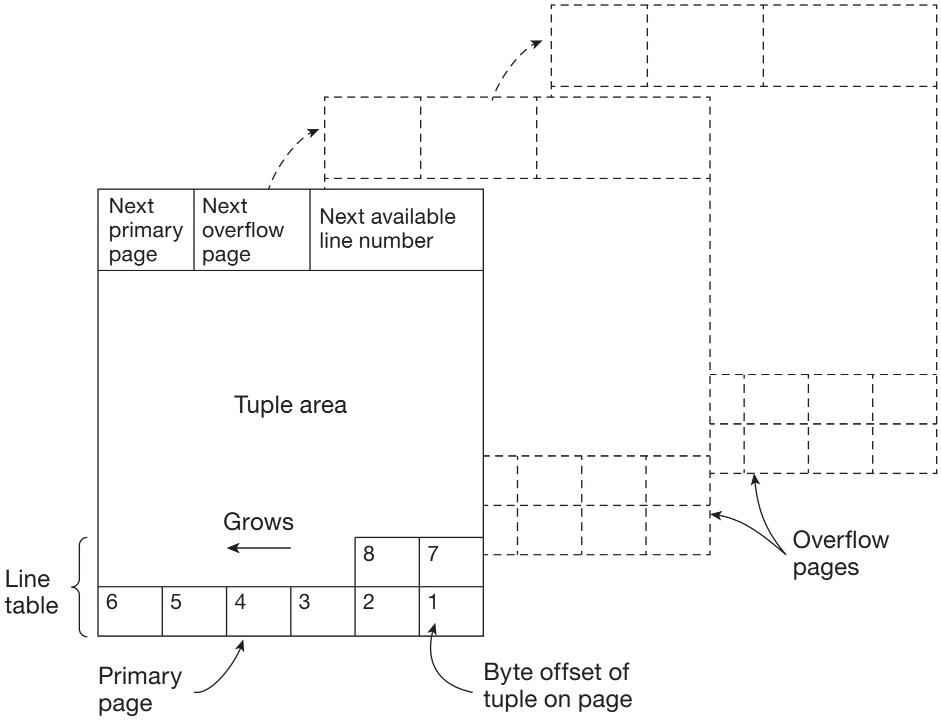


Figure 4 Page layout for keyed storage structures

In the above-mentioned keyed modes, fixed length tuples are stored. In addition, both schemes can be used in conjunction with data compression techniques [14] in cases where increased storage utilization outweighs the added cost of encoding and decoding data during access. These modes are known as "compressed hash" and "compressed ISAM."

The current compression scheme suppresses blanks and portions of a tuple which match the preceding tuple. This compression is applied to each page independently. Other schemes are being experimented with. Compression appears to be useful in storing variable length domains (which must be declared their maximum length). Padding is then removed during compression by the access method. Compression may also be useful when storing secondary indices.

3.4 Access Methods Interface

The Access Methods Interface (AMI) handles all actual accessing of data from relations. The AMI language is implemented as a set of functions whose calling conventions are indicated below. A separate copy of these functions is loaded with each of processes 2, 3, and 4.

Each access method must do two things to support the following calls. First, it must provide some linear ordering of the tuples in a relation so that the concept of “next tuple” is well defined. Second, it must assign to each tuple a unique tuple-id (TID).

The nine implemented calls are as follows:

(a) OPENR(descriptor, mode, relation_name)
Before a relation may be accessed it must be “opened.” This function opens the UNIX file for the relation and fills in a “descriptor” with information about the relation from the RELATION and ATTRIBUTE catalogs. The descriptor (storage for which must be declared in the calling routine) is used in subsequent calls on AMI routines as an input parameter to indicate which relation is involved. Consequently, the AMI data accessing routines need not themselves check the system catalogs for the description of a relation. “Mode” specifies whether the relation is being opened for update or for retrieval only.

(b) GET(descriptor, tid, limit_tid, tuple, next_flag)
This function retrieves into “tuple,” a single tuple from the relation indicated by “descriptor.” “Tid” and “limit_tid” are tuple identifiers. There are two modes of retrieval, “scan” and “direct.” In “scan” mode GET is intended to be called successively to retrieve all tuples within a range of tuple-ids. An initial value of “tid” sets the low end of the range desired and “limit_tid” sets the high end. Each time GET is called with “next-flag” = TRUE, the tuple following “tid” is retrieved and its tuple-id is placed into “tid” in readiness for the next call. Reaching “limit_ tid” is indicated by a special return code, The initial settings of “tid” and “limit_tid“ are done by calling the FIND function. In ”direct“ mode (”next_flag“ = FALSE), GET retrieves the tuple with tuple-id = “tid.”

(c) FIND(descriptor, key, tid, key_type)
When called with a negative “key-type,” FIND returns in “tid” the lowest tuple-id on the lowest page which could possibly contain tuples matching the key supplied. Analogously, the highest tuple-id is returned when “key-type” is positive. The objective is to restrict the scan of a relation by eliminating tuples from consideration which are known from their placement not to satisfy a given qualification.

“Key-type” also indicates (through its absolute value) whether the key, if supplied, is an EXACTKEY or a RANGEKEY. Different criteria for matching are applied in each case. An EXACTKEY matches only those tuples containing exactly the value of the key supplied. A RANGEKEY represents the low (or high) end of a range of possible key values and thus matches any tuple with a key value greater than or equal to (or less than or equal to) the key supplied. Note that only with an order preserving storage structure can a RANGEKEY be used to successfully restrict a scan.

In cases where the storage structure of the relation is incompatible with the “key-type,” the “tid” returned will be as if no key were supplied (that is, the lowest or highest tuple in the relation). Calls to FIND invariably occur in pairs, to obtain the two tuple-ids which establish the low and high ends of the scan done in subsequent calls to GET.

Two functions are available for determining the access characteristics of the storage structure of a primary data relation or secondary index, respectively.

(d) PARAMD (descriptor, access_characteristics_structure)

(e) PARAMI (index-descriptor, access_characteristics_structure)
The “access-characteristics-structure” is filled in with information regarding the type of key which may be utilized to restrict the scan of a given relation: It indicates whether exact key values or ranges of key values can be used, and whether a partially specified key may be used. This determines the “key-type” used in a subsequent call to FIND. The ordering of domains in the key is also indicated. These two functions allow the access optimization routines to be coded independently of the specific storage structures currently implemented.

Other AMI functions provide a facility for updating relations.

(f) INSERT(descriptor, tuple)
The tuple is added to the relation in its “proper” place according to its key value and the storage mode of the relation.

(g) REPLACE(descriptor, tid, new_tuple)

(h) DELETE(descriptor, tid)
The tuple indicated by “tid” is either replaced by new values or deleted from the relation altogether. The tuple-id of the affected tuple will have been obtained by a previous GET.

Finally, when all access to a relation is complete it must be closed:

(i) CLOSER(descriptor)
This closes the relation's UNIX file and rewrites the information in the descriptor back into the system catalogs if there has been any change.

3.5 Addition of New Access Methods

One of the goals of the AMI design was to insulate higher level software from the actual functioning of the access methods, thereby making it easier to add different ones. It is anticipated that users with special requirements will take advantage of this feature.

In order to add a new access method, one need only extend the AMI routines to handle the new case. If the new method uses the same page layout and TID scheme, only FIND, PARAMI, and PARAMD need to be extended. Otherwise new procedures to perform the mapping of TIDs to physical file locations must be supplied for use by GET, INSERT, REPLACE, and DELETE.

4 The Structure of Process 2

Process 2 contains four main components:

(a) a lexical analyzer;

(b) a parser (written in YACC [19]);

(c) concurrency control routines;

(d) query modification routines to support protection, views, and integrity control (at present only partially implemented).

Since (a) and (b) are designed and implemented along fairly standard lines, only (c) and (d) will be discussed in detail. The output of the parsing process is a tree structured representation of the input query used as the internal form in subsequent processing. Furthermore, the qualification portion of the query has been converted to an equivalent Boolean expression in conjunctive normal form. In this form the query tree is then ready to undergo what has been termed "query modification."

4.1 Query Modification

Query modification includes adding integrity and protection predicates to the original query and changing references to virtual relations into references to the appropriate physical relations. At the present time only a simple integrity scheme has been implemented.

In [27] algorithms of several levels of complexity are presented for performing integrity control on updates. In the present system only the simplest case, involving single-variable, aggregate free integrity assertions, has been implemented, as described in detail in [23].

Briefly, integrity assertions are entered in the form of QUEL qualification clauses to be applied to interactions updating the relation over which the variable in the assertion rangrs. A parse tree is created for the qualification and a representation of this tree is stored in the INTEGRITY catalog together with an indication of the relation and the specific domains involved. At query modification time, updates are checked for any possible integrity assertions on the affected domains. Relevant assertions are retrieved, rebuilt into tree form, and grafted onto the update tree so as to AND the assertions with the existing qualification of the interaction.

Algorithms for the support of views are also given in [27]. Basically a view is a virtual relation defined in terms of relations which physically exist. Only the view definition will be stored, and it will be indicated to INGRES by a DEFINE command. This command will have a syntax identical to that of a RETRIEVE statement. Thus legal views will be those relations which it is possible to materialize by a RETRIEVE statement. They will be allowed in INGRES to support EQUEL programs written for obsolete versions of the database and for user convenience.

Protection will be handled according to the algorithm described in [25]. Like integrity control, this algorithm involves adding qualifications to the user's interaction. The details of the implementation (which is in progress) are given in [28], which also includes a discussion of the mechanisms being implemented to physically protect INGRES files from tampering in any way other than by executing the INGRES object code. Last, [28] distinguishes the INGRES protection scheme from the one based on views in [5] and indicates the rationale behind its use.

In the remainder of this section we give an example of query modification at work.

Suppose at a previous point in time all employees in the EMPLOYEE relation were under 30 and had no manager recorded. If an EQUEL program had been written for this previous version of EMPLOYEE which retrieved ages of employees coded into 5 bits, it would now fail for employees over 31.

If one wishes to use the above program without modification, then the following view must be used:

```
RANGE OF E IS EMPLOYEE
DEFINE OLDEMP (E.NAME, E.DEPT, E.SALARY, E.AGE)
WHERE   E.AGE < 30
```

Suppose that all employees in the EMPLOYEE relation must make more than $8000. This can be expressed by the integrity constraint:

```
RANGE OF E IS EMPLOYEE
INTEGRITY CONSTRAINT IS E.SALARY > 8000
```

Last, suppose each person is only authorized to alter salaries of employees whom he manages. This is expressed as follows:

```
RANGE OF E IS EMPLOYEE
PROTECT EMPLOYEE FOR ALL (E.SALARY; E.NAME)
WHERE   E.MANAGER = *
```

The * is a surrogate for the logon name of the current UNIX user of INGRES. The semicolon separates updatable from nonupdatable (but visible) domains.

Suppose Smith through an EQUEL program or from the terminal monitor issues the following interaction:

```
RANGE OF L IS OLDEMP
REPLACE L(SALARY = .9*L.SALARY)
WHERE   L.NAME = "Brown"
```

This is an update on a view. Hence the view algorithm in [27] will first be applied to yield:

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY = .9*E.SALARY)
WHERE   E.NAME = "Brown"
AND   E.AGE < 30
```

Note Brown is only in OLDEMP if he is under 30. Now the integrity algorithm in [27] must be applied to ensure that Brown's salary is not being cut to as little as $8000. This involves modifying the interaction to:

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY = .9*E.SALARY)
WHERE   E.NAME = "Brown"
    AND   E.AGE < 30
    AND   .9*E.SALARY > $8000
```

Since .9*E.SALARY will be Brown's salary after the update, the added qualification ensures this will be more than $8000.

Last, the protection algorithm of [28] is applied to yield:

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY = .9*E.SALARY)
WHERE   E.NAME = “Brown”
    AND   E.AGE < 30
    AND   .9*E.SALARY > $8000
    AND   E.MANAGER = “Smith”
```

Notice that in all three cases more qualification is ANDed onto the user's interaction. The view algorithm must in addition change tuple variables.

In all cases the qualification is obtained from (or is an easy modification of) predicates stored in the VIEW, INTEGRITY, and PROTECTION relations. The tree representation of the interaction is simply modified to AND these qualifications (which are all stored in parsed form).

It should be clearly noted that only one-variable, aggregate free integrity assertions are currently supported. Moreover, even this feature is not in the released version of INGRES. The code for both concurrency control and integrity control will not fit into process 2 without exceeding 64K words. The decision was made to release a system with concurrency control.

The INGRES designers are currently adding a fifth process (process 2.5) to hold concurrency and query modification routines. On PDP 11/45s and 11/70s that have a 128K address space this extra process will not be required.

4.2 Concurrency Control

In any multiuser system provisions must be included to ensure that multiple concurrent updates are executed in a manner such that some level of data integrity can be guaranteed. The following two updates illustrate the problem.

```
            RANGE OF E IS EMPLOYEE
U1          REPLACE E(DEPT = “toy”)
                WHERE E.DEPT = “candy”

            RANGE OF F IS EMPLOYEE
U2          REPLACE F(DEPT = “candy”)
                WHERE F.DEPT = “toy”
```

If U1 and U2 are executed concurrently with no controls, some employees may end up in each department and the particular result may not be repeatable if the database is backed up and the interactions reexecuted.

The control which must be provided is to guarantee that some database operation is “atomic” (occurs in such a fashion that it *appears* instantaneous and before or after any other database operation). This atomic unit will be called a "transaction."

In INGRES there are five basic choices available for defining a transaction:

(a) something smaller than one INGRES command;

(b) one INGRES command;

(c) a collection of INGRES commands with no intervening C code;

(d) a collection of INGRES commands with C code but no system calls;

(e) an arbitrary Equel program.

If option (a) is chosen, INGRES could not guarantee that two concurrently executing update commands would give the same result as if they were executed sequentially (in either order) in one collection of INGRES processes. In fact, the outcome could fail to be repeatable, as noted in the example above. This situation is clearly undesirable.

Option (e) is, in the opinion of the INGRES designers, impossible to support. The following transaction could be declared in an Equel program.

```
BEGIN TRANSACTION
    FIRST QUEL UPDATE
    SYSTEM CALLS TO CREATE AND DESTROY FILES
    SYSTEM CALLS TO FORK A SECOND COLLECTION OF INGRES PROCESSES
        TO WHICH COMMANDS ARE PASSED
    SYSTEM CALLS TO READ FROM A TERMINAL
    SYSTEM CALLS TO READ FROM A TAPE
    SECOND QUEL UPDATE (whose form depends on previous two system calls)
END TRANSACTION
```

Suppose T1 is the above transaction and runs concurrently with a transaction T2 involving commands of the same form. The second update of each transaction may well conflict with the first update of the other. Note that there is no way to tell a priori that T1 and T2 conflict, since the form of the second update is not known in advance. Hence a deadlock situation can arise which can only be resolved by aborting one transaction (an undesirable policy in the eyes of the INGRES designers) or attempting to back out one transaction. The overhead of backing out through the intermediate system calls appears prohibitive (if it is possible at all).

Restricting a transaction to have no system calls (and hence no I/O) cripples the power of a transaction in order to make deadlock resolution possible. This was judged undesirable.

For example, the following transaction requires such system calls:

```
BEGIN TRANSACTION
    QUEL RETRIEVE to find all flights on a particular day from San Francisco to Los
            Angeles with space available.
    Display flights and times to user.
    Wait for user to indicate desired flight.
    QUEL REPLACE to reserve a seat on the flight of the user's choice.
END TRANSACTION
```

If the above set of commands is not a transaction, then space on a flight may not be available when the REPLACE is executed even though it was when the RETRIEVE occurred.

Since it appears impossible to support multi-QUEL statement transactions (except in a crippled form), the INGRES designers have chosen Option (b), one QUEL statement, as a transaction.

Option (c) can be handled by a straightforward extension of the algorithms to follow and will be implemented if there is sufficient user demand for it. This option can support "triggers" [2] and may prove useful.

Supporting Option (d) would considerably increase system complexity for what is perceived to be a small generalization. Moreover, it would be difficult to enforce in the EQUEL translator unless the translator parsed the entire C language.

The implementation of (b) or (c) can be achieved by physical locks on data items, pages, tuples, domains, relations, etc. [12] or by predicate locks [26]. The current implementation is by relatively crude physical locks (on domains of a relation) and avoids deadlock by not allowing an interaction to proceed to process 3 until it can lock all required resources. Because of a problem with the current design of the REPLACE access method call, all domains of a relation must currently be locked (i.e. a whole relation is locked) to perform an update. This situation will soon be rectified.

The choice of avoiding deadlock rather than detecting and resolving it is made primarily for implementation simplicity.

The choice of a crude locking unit reflects our environment where core storage for a large lock table is not available. Our current implementation uses a LOCK relation into which a tuple for each lock requested is inserted. This entire relation is physically locked and then interrogated for conflicting locks. If none exist, all needed locks are inserted. If a conflict exists, the concurrency processor "sleeps" for a fixed interval and then tries again. The necessity to lock the entire relation and to sleep for a fixed interval results from the absence of semaphores (or an equivalent mechanism) in UNIX. Because concurrency control can have high overhead as currently implemented, it can be turned off.

The INGRES designers are considering writing a device driver (a clean extension to UNIX routinely written for new devices) to alleviate the lack of semaphores. This driver would simply maintain core tables to implement desired synchronization and physical locking in UNIX.

The locks are held by the concurrency processor until a termination message is received on pipe E. Only then does it delete its locks.

In the future we plan to experimentally implement a crude (and thereby low CPU overhead) version of the predicate locking scheme described in [26]. Such an approach may provide considerable concurrency at an acceptable overhead in lock table space and CPU time, although such a statement is highly speculative.

To conclude this section, we briefly indicate the reasoning behind not sorting a page and its overflow pages in the "ISAM-like" access method. This topic is also discussed in [17].

The proposed device driver for locking in UNIX must at least ensure that read-modify-write of a single UNIX page is an atomic operation. Otherwise, INGRES would still be required to lock the whole LOCK relation to insert locks. Moreover, any proposed predicate locking scheme could not function without such an atomic operation. If the lock unit is a UNIX page, then INGRES can insert and delete a tuple from a relation by holding only one lock at a time if a primary page and its overflow page are unordered. However, maintenance of the sort order of these pages may require the access method to lock more than one page when it inserts a tuple. Clearly deadlock may be possible given concurrent updates, and the size of the lock table in the device driver is not predictable. To avoid both problems these pages remain unsorted.

5 Process 3

As noted in Section 2, this process performs the following two functions, which will be discussed in turn:

(a) Decomposition of queries involving more than one variable into sequences of one-variable queries. Partial results are accumulated until the entire query is evaluated. This program is called DECOMP. It also turns any updates into the appropriate queries to isolate qualifying tuples and spools modifications into a special file for deferred update.

(b) Processing of single-variable queries. The program is called the one-variable query processor (OVQP).

5.1 DECOMP

Because INGRES allows interactions which are defined on the crossproduct of perhaps several relations, efficient execution of this step is of crucial importance in searching as small a portion of the appropriate crossproduct space as possible. DECOMP uses three techniques in processing interactions. We describe each technique, and then give the actual algorithm implemented followed by an example which illustrates all features. Finally we indicate the role of a more sophisticated decomposition scheme under design.

(a) Tuple substitution. The basic technique used by DECOMP to reduce a query to fewer variables is tuple substitution. One variable (out of possibly many) in the query is selected for substitution. The AMI language is used to scan the relation associated with the variable one tuple at a time. For each tuple the values of domains in that relation are substituted into the query. In the resulting modified query, all previous references to the substituted variable have now been replaced by values (constants) and the query has thus been reduced to one less variable. Decomposition is repeated (recursively) on the modified query until only one variable remains, at which point the OVQP is called to continue processing.

(b) One-variable detachment. If the qualification Q of the query is of the form

$$\mathrm{Q}_1(V_1) \quad \text{AND} \quad \mathrm{Q}_2(V_1, \ldots, V_n)$$

for some tuple variable V_1, the following two steps can be executed:

1. Issue the query

   RETRIEVE INTO W (TL[V_1])
   WHERE Q$_1$[V_1]

   Here TL[V_1] are those domains required in the remainder of the query. Note that this is a one-variable query and may be passed directly to OVQP.

2. Replace R_1, the relation over which V_1 ranges, by W in the range declaration and delete Q$_1$[V_1] from Q.

The query formed in step 1 is called a "one-variable, detachable subquery," and the technique for forming and executing it is called "one-variable detachment" (OVD). This step has the effect of reducing the size of the relation over which V_1

ranges by restriction and projection. Hence it may reduce the complexity of the processing to follow.

Moreover, the opportunity exists in the process of creating new relations through OVD, to choose storage structures, and particularly keys, which will prove helpful in further processing.

(c) Reformatting. When a tuple variable is selected for substitution, a large number of queries, each with one less variable, will be executed. If (b) is a possible operation after the substitution for some remaining variable V_1, then the relation over which V_1 ranges, R_1, can be reformatted to have domains used in Q$_1(V_1)$ as a key. This will expedite (b) each time it is executed during tuple substitution.

We can now state the complete decomposition algorithm. After doing so, we illustrate all steps with an example.

Step 1. If the number of variables in the query is 0 or 1, call OVQP and then return; else go on to step 2.

Step 2. Find all variables, $\{V_1, \ldots, V_n\}$, for which the query contains a one-variable clause.

Perform OVD to create new ranges for each of these variables. The new relation for each variable V_i is stored as a hash file with key K_i chosen as follows:

2.1. For each j select from the remaining multivariable clauses in the query the collection, C_{ij}, which have the form $V_i \cdot d_i = V_j \cdot d_j$, where d_i, d_j are domains of V_i and V_j.

2.2. From the key K_i to be the concatenation of domains $d_{i1}, d_{i2}, \ldots$ of V_i appearing in clauses in C_{ij}.

2.3. If more than one j exists, for which C_{ij} is nonempty, one C_{ij} is chosen arbitrarily for forming the key. If C_{ij} is empty for all j, the relation is stored as an unsorted table.

Step 3. Choose the variable V_s with the smallest number of tuples as the next one for which to perform tuple substitution.

Step 4. For each tuple variable V_j for which C_{js} is nonnull, reformat if necessary the storage structure of the relation R_j over which it ranges so that the key of R_j is the concatenation of domains $d_{j1}, \ldots$ appearing in C_{js}. This ensures that when the clauses in C_{js} become one-variable after substituting for V_s, subsequent calls to OVQP to restrict further the range of V_j will be done as efficiently as possible.

Step 5. Iterate the following steps over all tuples in the range of the variable selected in step 3 and then return:
5.1. Substitute values from tuple into query.
5.2. Invoke decomposition algorithm recursively on a copy of resulting query which now has been reduced by one variable.
5.3. Merge the results from 5.2 with those of previous iterations.

We use the following query to illustrate the algorithm:

```
RANGE OF E, M IS EMPLOYEE
RANGE OF D IS DEPT
RETRIEVE (E.NAME)
WHERE  E.SALARY   > M.SALARY AND
       E.MANAGER = M.NAME    AND
       E.DEPT    = D.DEPT    AND
       D.FLOOR#  = 1         AND
       E.AGE     > 40
```

This request is for employees over 40 on the first floor who earn more than their manager.

LEVEL 1.

Step 1. Query is not one variable.

Step 2. Issue the two queries:

```
RANGE OF D IS DEPT
RETRIEVE INTO T1(D.DEPT)                                        (1)
WHERE D.FLOOR# = 1

RANGE OF E IS EMPLOYEE
RETRIEVE INTO T2(E.NAME, E.SALARY, E.MANAGER, E.DEPT)           (2)
WHERE E.AGE > 40
```

T1 is stored hashed on DEPT; however, the algorithm must choose arbitrarily between hashing T2 on MANAGER or DEPT. Suppose it chooses MANAGER. The original query now becomes:

```
RANGE OF D IS TI
RANGE OF E IS T2
RANGE OF M IS EMPLOYEE
RETRIEVE (E.NAME)
```

```
WHERE   E.SALARY     > M.SALARY  AND
        E.MANAGER = M.NAME     AND
        E.DEPT        = D.DEPT
```

Step 3. Suppose T1 has smallest cardinality. Hence D is chosen for substitution.

Step 4. Reformat T2 to be hashed on DEPT; the guess chosen in step 2 above was a poor one.

Step 5. Iterate for each tuple in T1 and then quit:
5.1 Substitute value for D. DEPT yielding

```
RANGE OF E IS T1
RANGE OF M IS EMPLOYEE
RETRIEVE (E.NAME)
WHERE   E.SALARY     > M.SALARY  AND
        E.MANAGER = M.NAME     AND
        E.DEPT        = value
```

5.2. Start at step 1 with the above query as input (Level 2 below).
5.3. Cumulatively merge results as they are obtained.

LEVEL 2.

Step 1. Query is not one variable.

Step 2. Issue the query

```
RANGE OF E IS T2
RETRIEVE INTO T3 (E.NAME, E.SALARY, E.NAME)                    (3)
WHERE   E.DEPT = value
```

T3 is constructed hashed on MANAGER. T2 in step 4 in Level 1 above is reformatted so that this query (which will be issued once for each tuple in T1) will be done efficiently by OVQP. Hopefully the cost of reformatting is small compared to the savings at this step. What remains is

```
RANGE OF E IS T3
RANGE IF M IS EMPLOYEE
RETRIEVE (E.NAME)
WHERE   E.SALARY     > M.SALARY  AND
        E.MANAGER = M.NAME
```

Step 3. T3 has less tuples than EMPLOYEE; therefore choose T3.

Step 4. [unnecessary]

Step 5. Iterate for each tuple in T3 and then return to previous level:

5.1. Substitute values for E.NAME, E.SALARY, and E.MANAGER, yielding

```
RANGE OF M IS EMPLOYEE
RETRIEVE (VALUE 1)                                              (4)
WHERE   Value2 > M.SALARY     AND
        Value3 = M.NAME
```

5.2. Start at step 1 with this query as input (Level 3 below).

5.3. Cumulatively merge results as obtained.

LEVEL 3.

Step 1. Query has one variable; invoke OVQP and then return to previous level.

The algorithm thus decomposes the original query into the four prototype, one-variable queries labeled (1)–(4), some of which are executed repetitively with different constant values and with results merged appropriately. Queries (1) and (2) are executed once, query (3) once for each tuple in T1, and query (4) the number of times equal to the number of tuples in T1 times the number of tuples in T3.

The following comments on the algorithm are appropriate.

(a) OVD is almost always assured of speeding processing. Not only is it possible to choose the storage structure of a temporary relation wisely, but also the cardinality of this relation may be much less than the one it replaces as the range for a tuple variable. It only fails if little or no reduction takes place and reformatting is unproductive.

It should be noted that a temporary relation is created rather than a list of qualifying tuple-id's. The basic tradeoff is that OVD must copy qualifying tuples but can remove duplicates created during the projection. Storing tuple-id's avoids the copy operation at the expense of reaccessing qualifying tuples and retaining duplicates. It is clear that cases exist where each strategy is superior. The INGRES designers have chosen OVD because it does not appear to offer worse performance than the alternative, allows a more accurate choice of the variable with the smallest range in step 3 of the algorithm above, and results in cleaner code.

(b) Tuple substitution is done when necessary on the variable associated with the smallest number of tuples. This has the effect of reducing the number of eventual calls on OVQP.

(c) Reformatting is done (if necessary) with the knowledge that it will usually replace a collection of complete sequential scans of a relation by a collection of limited scans. This almost always reduces processing time.

(d) It is believed that this algorithm efficiently handles a large class of interactions. Moreover, the algorithm does not require excessive CPU overhead to perform. There are, however, cases where a more elaborate algorithm is indicated. The following comment applies to such cases.

(e) Suppose that we have two or more strategies $ST_0, ST_1, \ldots, ST_n$, each one being better than the previous one but also requiring a greater overhead. Suppose further that we begin an interaction on ST_0 and run it for an amount of time equal to a fraction of the estimated overhead of ST_1. At the end of that time, by simply counting the number of tuples of the first substitution variable which have already been processed, we can get an estimate for the total processing time using ST_0. If this is significantly greater than the overhead of ST_1, then we switch to ST_1. Otherwise we stay and complete processing the interaction using ST_0. Obviously, the procedure can be repeated on ST_1 to call ST_2 if necessary, and so forth.

The algorithm detailed in this section may be thought of as ST_0. A more sophisticated algorithm is currently under development [30].

5.2 One-Variable Query Processor (OVQP)

This module is concerned solely with the efficient accessing of tuples from a single relation given a particular one-variable query. The initial portion of this program, known as STRATEGY, determines what key (if any) may be used profitably to access the relation, what value(s) of that key will be used in calls to the AMI routine FIND, and whether access may be accomplished directly through the AMI to the storage structure of the primary relation itself or if a secondary index on the relation should be used. If access is to be through a secondary index, then STRATEGY must choose which *one* of possibly many indices to use.

Tuples are then retrieved according to the access strategy selected and are processed by the SCAN portion of OVQP. These routines evaluate each tuple against the qualification part of the query, create target list values for qualifying tuples, and dispose of the target list appropriately.

Since SCAN is relatively straightforward, we discuss only the policy decisions made in STRATEGY.

First STRATEGY examines the qualification for clauses which specify the value of a domain, i.e. clauses of the form

V.domain op constant

or

constant op *V*.domain

where “op” is one of the set $\{=, <, >, \leq, \geq\}$. Such clauses are termed “simple” clauses and are organized into a list. The constants in simple clauses will determine the key values input to FIND to limit the ensuing scan.

Obviously a nonsimple clause may be equivalent to a simple one. For example, E.SALARY/2 = 10000 is equivalent to E.SALARY = 20000. However, recognizing and converting such clauses requires a general algebraic symbol manipulator. This issue has been avoided by ignoring all nonsimple clauses.

STRATEGY must select one of two accessing strategies: (a) issuing two AMI FIND commands on the primary relation followed by a sequential scan of the relation (using GET in “scan” mode) between the limits set, or (b) issuing two AMI FIND commands on some index relation followed by a sequential scan of the index between the limits set. For each tuple retrieved the “pointer” domain is obtained; this is simply the tuple-id of a tuple in the primary relation. This tuple is fetched (using GET in “direct” mode) and processed.

To make the choice, the access possibilities available must be determined. Keying information about the primary relation is obtained using the AMI function PARAMD. Names of indices are obtained from the INDEX catalog and keying information about indices is obtained with the function PARAMI.

Further, a compatability between the available access possibilities and the specification of key values by simple clauses must be established. A hashed relation requires that a simple clause specify equality as the operator in order to be useful; for combined (multidomain) keys, all domains must be specified. ISAM structures, on the other hand, allow range specifications; additionally, a combined ISAM key requires only that the most significant domains be specified.

STRATEGY checks for such a compatability according to the following priority order of access possibilities: (1) hashed primary relation, (2) hashed index, (3) ISAM primary relation, (4) ISAM index. The rationale for this ordering is related to the expected number of page accesses required to retrieve a tuple from the source relation in each case. In the following analysis the effect of overflow pages is ignored (on the assumption that the four access possibilities would be equally affected).

In case (1) the key value provided locates a desired source tuple in one access via calculation involving a hashing function. In case (2) the key value similarly locates an appropriate index relation tuple in one access, but an additional access is required to retrieve the proper primary relation tuple. For an ISAM-structured scheme a directory must be examined. This lookup itself incurs at least one access but possibly more if the directory is multilevel. Then the tuple itself must be accessed. Thus case (3) requires at least two (but possibly more) total accesses. In case (4) the use of an index necessitates yet another access in the primary relation, making the total at least three.

To illustrate STRATEGY, we indicate what happens to queries (1)–(4) from Section 5.1.

Suppose EMPLOYEE is an ISAM relation with a key of NAME, while DEPT is hashed on FLOOR#. Moreover a secondary index for AGE exists which is hashed on AGE, and one for SALARY exists which uses ISAM with a key of SALARY.

Query (1): One simple clause exists (D.FLOOR# = 2). Hence Strategy (a) is applied against the hashed primary relation.

Query (2): One simple clause exists (E.AGE > 40). However, it is not usable to limit the scan on a hashed index. Hence a complete (unkeyed) scan of EMPLOYEE is required. Were the index for AGE an ISAM relation, then Strategy (b) would be used on this index.

Query (3): One simple clause exists and T1 has been reformatted to allow Strategy (a) against the hashed primary relation.

Query (4): Two simple clauses exist (value2 > M.SALARY; value3 = M.NAME). Strategy (a) is available on the hashed primary relation, as is Strategy (b) for the ISAM index. The algorithm chooses Strategy (a).

6 Utilities in Process 4

6.1 Implementation of Utility Commands

We have indicated in Section 1 several database utilities available to users. These commands are organized into several overlay programs as noted previously. Bringing the required overlay into core as needed is done in a straightforward way.

Most of the utilities update or read the system relations using AMI calls. MODIFY contains a sort routine which puts tuples in collating sequence according to the concatenation of the desired keys (which need not be of the same data type). Pages are initially loaded to approximately 80 percent of capacity. The sort routine is a recursive *N*-way merge-sort where *N* is the maximum number of files process 4 can have open at once (currently eight). The index building occurs in an obvious way. To convert to hash structures, MODIFY must specify the number of primary pages to be allocated. This parameter is used by the AMI in its hash scheme (which is a standard modulo division method).

It should be noted that a user who creates an empty hash relation using the CREATE command and then copies a large UNIX file into it using COPY creates a very inefficient structure. This is because a relatively small default number of primary pages will have been specified by CREATE, and overflow chains will be long. A better strategy is to COPY into an unsorted table so that MODIFY can subsequently make a good guess at the number of primary pages to allocate.

6.2 Deferred Update and Recovery

Any updates (APPEND, DELETE, REPLACE) are processed by writing the tuples to be added, changed, or modified into a temporary file. When process 3 finishes, it calls process 4 to actually perform the modifications requested and any updates to secondary indices which may be required as a final step in processing. Deferred update is done for four reasons.

(a) Secondary index considerations. Suppose the following QUEL statement is executed:

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY = 1.1*E.SALARY)
WHERE   E.SALARY > 20000
```

Suppose further that there is a secondary index on the salary domain and the primary relation is keyed on another domain.

OVQP, in finding the employees who qualify for the raise, will use the secondary index. If one employee qualifies and his tuple is modified and the secondary index updated, then the scan of the secondary index will find his tuple a second time since it has been moved forward. (In fact, his tuple will be found an arbitrary number of times.) Either secondary indexes cannot be used to identify qualifying tuples when range qualifications are present (a rather unnatural restriction), or secondary indices must be updated in deferred mode.

(b) Primary relation considerations. Suppose the QUEL statement

```
RANGE OF E, M IS EMPLOYEE
REPLACE E(SALARY = .9*E.SALARY)
Where    E.MGR       = M.NAME AND
         E.SALARY    > M.SALARY
```

is executed for the following EMPLOYEE relation:

NAME	SALARY	MANAGER
Smith	10K	Jones
Jones	8K	
Brown	9.5K	Smith

Logically Smith should get the pay cut and Brown should not. However, if Smith's tuple is updated before Brown is checked for the pay cut, Brown will qualify. This undesirable situation must be avoided by deferred update.

(c) Functionality of updates. Suppose the following QUEL statement is executed:

```
RANGE OF E, M IS EMPLOYEE
REPLACE E(SALARY = M.SALARY)
```

This update attempts to assign to each employee the salary of every other employee, i.e. a single data item is to be replaced by multiple values. Stated differently, the REPLACE statement does not specify a function. In certain cases (such as a REPLACE involving only one tuple variable) functionality is guaranteed. However, in general the functionality of an update is data dependent. This nonfunctionality can only be checked if deferred update is performed.

To do so, the deferred update processor must check for duplicate TIDs in REPLACE calls (which requires sorting or hashing the update file). This potentially expensive operation does not exist in the current implementation, but will be optionally available in the future.

(d) Recovery considerations. The deferred update file provides a log of updates to be made. Recovery is provided upon system crash by the RESTORE command. In this case the deferred update routine is requested to destroy the temporary file if it has not yet started processing it. If it has begun processing, it reprocesses the entire update file in such a way that the effect is the same as if it were processed exactly once from start to finish.

Hence the update is "backed out" if deferred updating has not yet begun; otherwise it is processed to conclusion. The software is designed so the update file can be optionally spooled onto tape and recovered from tape. This added feature should soon be operational.

If a user from the terminal monitor (or a C program) wishes to stop a command he can issue a "break" character. In this case all processes reset except the deferred update program, which recovers in the same manner as above.

All update commands do deferred update; however the INGRES utilities have not yet been modified to do likewise. When this has been done, INGRES will recover from all crashes which leave the disk intact. In the meantime there can be disk-intact crashes which cannot be recovered in this manner (if they happen in such a way that the system catalogs are left inconsistent).

The INGRES "superuser" can checkpoint a database onto tape using the UNIX backup scheme. Since INGRES logs all interactions, a consistent system can always be obtained, albeit slowly, by restoring the last checkpoint and running the log of interactions (or the tape of deferred updates if it exists).

It should be noted that deferred update is a very expensive operation. One INGRES user has elected to have updates performed directly in process 3, cognizant that he must avoid executing interactions which will run incorrectly. Like checks for functionality, direct update may be optionally available in the future. Of course, a different recovery scheme must be implemented.

7 Conclusion and Future Extensions

The system described herein is in use at about fifteen installations. It forms the basis of an accounting system, a system for managing student records, a geodata system, a system for managing cable trouble reports and maintenance calls for a large telephone company, and assorted other smaller applications. These applications have been running for periods of up to nine months.

7.1 Performance

At this time no detailed performance measurements have been made, as the current version (labeled Version 5) has been operational for less than two months. We have instrumented the code and are in the process of collecting such measurements.

The sizes (in bytes) of the processes in INGRES are indicated below. Since the access methods are loaded with processes 2 and 3 and with many of the utilities, their contribution to the respective process sizes has been noted separately.

access methods (AM)	11K
terminal monitor	10K
EQUEL	30K + AM
process 2	45K + AM
process 3 (query processor)	45K + AM
utilities (8 overlays)	160K + AM

7.2 User Feedback

The feedback from internal and external users has been overwhelmingly positive.

In this section we indicate features that have been suggested for future systems.

(a) Improved performance. Earlier versions of INGRES were very slow; the current version should alleviate this problem.

(b) Recursion. QUEL does not support recursion, which must be tediously programmed in C using the precompiler; recursion capability has been suggested as a desired extension.

(c) Other language extensions. These include user defined functions (especially counters), multiple target lists for a single qualification statement, and if-then-else control structures in QUEL; these features may presently be programmed, but only very inefficiently, using the precompiler.

(d) Report generator. PRINT is a very primitive report generator and the need for augmented facilities in this area is clear; it should be written in EQUEL.

(e) Bulk copy. The COPY routine fails to handle easily all situations that arise.

7.3 Future Extensions

Noted throughout the paper are areas where system improvement is in progress, planned, or desired by users. Other areas of extension include: (a) a multicomputer system version of INGRES to operate on distributed databases; (b) further performance enhancements; (c) a higher level user language including recursion and user defined functions; (d) better data definition and integrity features; and (e) a database administrator advisor.

The database administrator advisor program would run at idle priority and issue queries against a statistics relation to be kept by INGRES. It could then offer advice to a DBA concerning the choice of access methods and the selection of indices. This topic is discussed further in [16].

Acknowledgment

The following persons have played active roles in the design and implementation of INGRES: Eric Allman, Rick Berman, Jim Ford, Angela Go, Nancy McDonald, Peter Rubinstein, Iris Schoenberg, Nick Whyte, Carol Williams, Karel Youssefi, and Bill Zook.

References

[1] Allman, E., Stonebraker, M., and Held, G. Embedding a relational data sublanguage in a general purpose programming language. Proc. Conf. on Data, SIGPLAN Notices (ACM) 8, 2 (1976), 25–35.

[2] Astrahan, M. M., et al. System R: Relational approach to database management. *ACM Trans. on Database Systems 1*, 2 (June 1976), 97–137.

[3] Boyce, R., et al. Specifying queries as relational expessions: SQUARE. Rep. RJ 1291, IBM Res. Lab., San Jose, Calif., Oct. 1973.

[4] Chamberlin, D., and Boyce, R. SEQUEL: A structured English query language. Proc. 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974, pp. 249–264.

[5] Chamberlin, D., Gray, J.N., and Traiger, I.L. Views, authorization and locking in a relational data base system. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., May 1975, pp. 425–430.

[6] Comm. on Data Systems Languages. CODASYL Data Base Task Group Rep., ACM, New York, 1971.

[7] Codd, E.F. A relational model of data for large shared data banks. *Comm. ACM 13*, 6 (June 1970), 377–387.

[8] Codd, E.F. A data base sublanguage founded on the relational calculus. Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, Calif., Nov. 1971, pp. 35–68.

[9] Codd, E.F. Relational completeness of data base sublanguages. Courant Computer Science Symp. 6, May 1971, Prentice-Hall, Englewood Cliffs, N.J., pp. 65–90.

[10] Codd, E.F., and Date, C.J. Interactive support for non-programmers, the relational and network approaches. Proc. 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

[11] Date, C.J., and Codd, E.F. The relational and network approaches: Comparison of the application programming interfaces. Proc. 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Vol. II, Ann Arbor, Mich., May 1974, pp. 85–113.

[12] Gray, J.N., Lorie, R.A., and Putzolu, G.R. Granularity of Locks in a Shared Data Base. Proc. Int. Conf. of Very Large Data Bases, Framingham, Mass., Sept. 1975, pp. 428–451. (Available from ACM, New York.)

[13] Go, A., Stonebraker, M., and Williams, C. An approach to implementing a geo-data system. Proc. ACM SIGGRAPH/SIGMOD Conf. for Data Bases in Interactive Design, Waterloo, Ont., Canada, Sept. 1975, pp. 67–77.

[14] Gottlieb, D., et al. A classification of compression methods and their usefulness in a large data processing center. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., May 1975, pp. 453–458.

[15] Held, G.D., Stonebraker, M., and Wong, E. INGRES—A relational data base management system. Proc. AFIPS 1975 NCC, Vol. 44, AFIPS Press, Montvale, N.J., 1975, pp. 409–416.

[16] Held, G.D. Storage Structures for Relational Data Base Management Systems. Ph.D. Th., Dep. of Electrical Eng. and Computer Science, U. of California, Berkeley, Calif., 1975.

[17] Held, G., and Stonebraker, M. *B*-trees re-examined. Submitted to a technical journal.

[18] IBM Corp. OS ISAM logic. GY28-6618, IBM Corp., White Plains, N.Y., 1966.

[19] Johnson, S.C. YACC, yet another compiler-compiler. UNIX Programmer's Manual, Bell Telephone Labs, Murray Hill, N.J., July 1974.

[20] McDonald, N., and Stonebraker, M. Cupid—The friendly query language. Proc. ACM-Pacific-75, San Francisco, Calif., April 1975, pp. 127–131.

[21] McDonald, N. CUPID: A graphics oriented facility for support of non-programmer interactions with a data base. Ph.D. Th., Dep. of Electrical Eng. and Computer Science, U. of California, Berkeley, Calif., 1975.

[22] Ritchie, D.M., and Thompson, K. The UNIX Time-sharing system. *Comm. ACM 17*, 7 (July 1974), 365–375.

[23] Schoenberg, I. Implementation of integrity constraints in the relational data base management system, INGRES. M.S. Th., Dep. of Electrical Eng. and Computer Science, U. of California, Berkeley, Calif., 1975.

[24] Stonebraker, M. A functional view of data independence. Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

[25] Stonebraker, M., and Wong, E. Access control in a relational data base management system by query modification. Proc. 1974 ACM Nat. Conf., San Diego, Calif., Nov. 1974, pp. 180–187.

[26] Stonebraker, M. High level integrity assurance in relational data base systems. ERI Mem. No. M473, Electronics Res. Lab., U. of California, Berkeley, Calif., Aug. 1974.

[27] Stonebraker, M. Implementation of integrity constraints and views by query modification. Proc. 1975 SIGMOD Workshop on Management of Data, San Jose, Calif., May 1975, pp. 65–78.

[28] Stonebraker, M., and Rubinstein, P. The INGRES protection system. Proc. 1976 ACM National Conf., Houston, Tex., Oct. 1976 (to appear).

[29] Tsichritzis, D. A network framework for relational implementation. Rep. CSRG-51, Computer Systems Res. Group, U. of Toronto, Toronto, Ont., Canada, Feb. 1975.

[30] Wong, E., and Youssefi, K. Decomposition—A strategy for query processing. *ACM Trans. on Database Systems 1*, 3 (Sept. 1976), 223–241 (this issue).

[31] Zook, W., et al. INGRES—Reference manual, 5. ERL Mem. No. M585, Electronics Res. Lab., U. of California, Berkeley, Calif., April 1976.

Received January 1976; revised April 1976

The Collected Works of Michael Stonebraker

D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. F. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. B. Zdonik. 2003a. Aurora: A data stream management system. In *Proc. ACM SIGMOD International Conference on Management of Data*, p. 666. DOI: 10.1145/872757.872855. 225, 228, 229, 230, 232

D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik. 2003b. Aurora: a new model and architecture for data stream management. *VLDB Journal*, 12(2): 120–139. DOI: 10.1007/s00778-003-0095-z. 228, 229, 324

D. J. Abadi, R. Agrawal, A. Ailamaki, M. Balazinska, P. A. Bernstein, M. J. Carey, S. Chaudhuri, J. Dean, A. Doan, M. J. Franklin, J. Gehrke, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, H. V. Jagadish, D. Kossmann, S. Madden, S. Mehrotra, T. Milo, J. F. Naughton, R. Ramakrishnan, V. Markl, C. Olston, B. C. Ooi, C. Ré, D. Suciu, M. Stonebraker, T. Walter, and J. Widom. 2014. The Beckman report on database research. *ACM SIGMOD Record*, 43(3): 61–70. DOI: 10.1145/2694428.2694441. 92

D. Abadi, R. Agrawal, A. Ailamaki, M. Balazinska, P. A. Bernstein, M. J. Carey, S. Chaudhuri, J. Dean, A. Doan, M. J. Franklin, J. Gehrke, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, H. V. Jagadish, D. Kossmann, S. Madden, S. Mehrotra, T. Milo, J. F. Naughton, R. Ramakrishnan, V. Markl, C. Olston, B. C. Ooi, C. Ré, D. Suciu, M. Stonebraker, T. Walter, and J. Widom. 2016. The Beckman report on database research. *Communications of the ACM*, 59(2): 92–99. DOI: 10.1145/2845915. 92

Z. Abedjan, C. G. Akcora, M. Ouzzani, P. Papotti, and M. Stonebraker. 2015a. Temporal rules discovery for web data cleaning. *Proc. VLDB Endowment*, 9(4): 336–347. http://www.vldb.org/pvldb/vol9/p336-abedjan.pdf. 297

Z. Abedjan, J. Morcos, M. N. Gubanov, I. F. Ilyas, M. Stonebraker, P. Papotti, and M. Ouzzani. 2015b. Dataxformer: Leveraging the web for semantic transformations. In *Proc. 7th Biennial Conference on Innovative Data Systems Research*. http://www.cidrdb.org/cidr2015/Papers/CIDR15_Paper31.pdf. 296, 297

Z. Abedjan, X. Chu, D. Deng, R. C. Fernandez, I. F. Ilyas, M. Ouzzani, P. Papotti, M. Stonebraker, and N. Tang. 2016a. Detecting data errors: Where are we and what needs to be done? *Proc. VLDB Endowment*, 9(12): 993–1004. http://www.vldb.org/pvldb/vol9/p993-abedjan.pdf. 298

Z. Abedjan, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, and M. Stonebraker. 2016b. Dataxformer: A robust transformation discovery system. In *Proc. 32nd International Conference on Data Engineering*, pp. 1134–1145. DOI: 10.1109/ICDE.2016.7498319. 296

S. Abiteboul, R. Agrawal, P. A. Bernstein, M. J. Carey, S. Ceri, W. B. Croft, D. J. DeWitt, M. J. Franklin, H. Garcia-Molina, D. Gawlick, J. Gray, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, M. L. Kersten, M. J. Pazzani, M. Lesk, D. Maier, J. F. Naughton, H. Schek, T. K. Sellis, A. Silberschatz, M. Stonebraker, R. T. Snodgrass, J. D. Ullman, G. Weikum, J. Widom, and S. B. Zdonik. 2003. The Lowell database research self assessment. *CoRR*, cs.DB/0310006. http://arxiv.org/abs/cs.DB/0310006. 92

S. Abiteboul, R. Agrawal, P. A. Bernstein, M. J. Carey, S. Ceri, W. B. Croft, D. J. DeWitt, M. J. Franklin, H. Garcia-Molina, D. Gawlick, J. Gray, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, M. L. Kersten, M. J. Pazzani, M. Lesk, D. Maier, J. F. Naughton, H. Schek, T. K. Sellis, A. Silberschatz, M. Stonebraker, R. T. Snodgrass, J. D. Ullman, G. Weikum, J. Widom, and S. B. Zdonik. 2005. The Lowell database research self-assessment. *Communications of the ACM*, 48(5): 111–118. DOI: 10.1145/1060710.1060718. 92

R. Agrawal, A. Ailamaki, P. A. Bernstein, E. A. Brewer, M. J. Carey, S. Chaudhuri, A. Doan, D. Florescu, M. J. Franklin, H. Garcia-Molina, J. Gehrke, L. Gruenwald, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, H. F. Korth, D. Kossmann, S. Madden, R. Magoulas, B. C. Ooi, T. O'Reilly, R. Ramakrishnan, S. Sarawagi, M. Stonebraker, A. S. Szalay, and G. Weikum. 2008. The Claremont report on database research. *ACM SIGMOD Record*, 37(3): 9–19. DOI: 10.1145/1462571.1462573. 92

R. Agrawal, A. Ailamaki, P. A. Bernstein, E. A. Brewer, M. J. Carey, S. Chaudhuri, A. Doan, D. Florescu, M. J. Franklin, H. Garcia-Molina, J. Gehrke, L. Gruenwald, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, H. F. Korth, D. Kossmann, S. Madden, R. Magoulas, B. C. Ooi, T. O'Reilly, R. Ramakrishnan, S. Sarawagi, M. Stonebraker, A. S. Szalay, and G. Weikum. 2009. The Claremont report on database research. *Communications of the ACM*, 52(6): 56–65. DOI: 10.1145/1516046.1516062. 92

A. Aiken, J. Chen, M. Lin, M. Spalding, M. Stonebraker, and A. Woodruff. 1995. The Tioga-2 database visualization environment. In *Proc. Workshop on Database Issues for Data Visualization*, pp. 181–207. DOI: 10.1007/3-540-62221-7_15.

A. Aiken, J. Chen, M. Stonebraker, and A. Woodruff. 1996. Tioga-2: A direct manipulation database visualization environment. In *Proc. 12th International Conference on Data Engineering*, pp. 208–217. DOI: 10.1109/ICDE.1996.492109.

E. Allman and M. Stonebraker. 1982. Observations on the evolution of a software system. *IEEE Computer*, 15(6): 27–32. DOI: 10.1109/MC.1982.1654047.

E. Allman, M. Stonebraker, and G. Held. 1976. Embedding a relational data sublanguage in a general purpose programming language. In *Proc. SIGPLAN Conference on Data: Abstraction, Definition and Structure*, pp. 25–35. DOI: 10.1145/800237.807115. 195

J. T. Anderson and M. Stonebraker. 1994. SEQUOIA 2000 metadata schema for satellite images. *ACM SIGMOD Record*, 23(4): 42–48. DOI: 10.1145/190627.190642.

A. Arasu, M. Cherniack, E. F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. 2004. Linear road: A stream data management benchmark. In *Proc. 30th International Conference on Very Large Data Bases*, pp. 480–491. http://www.vldb.org/conf/2004/RS12P1.pdf. 326

T. Atwoode, J. Dash, J. Stein, M. Stonebraker, and M. E. S. Loomis. 1994. Objects and databases (panel). In *Proc. 9th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 371–372. DOI: 10.1145/191080.191138.

H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E. F. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. B. Zdonik. 2004. Retrospective on Aurora. *VLDB Journal*, 13(4): 370–383. DOI: 10.1007/s00778-004-0133-5. 228, 229

M. Balazinska, H. Balakrishnan, and M. Stonebraker. 2004b. Load management and high availability in the Medusa distributed stream processing system. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 929–930. DOI: 10.1145/1007568.1007701. 325

M. Balazinska, H. Balakrishnan, and M. Stonebraker. 2004a. Contract-based load management in federated distributed systems. In *Proc. 1st USENIX Symposium on Networked Systems Design and Implementation*. http://www.usenix.org/events/nsdi04/tech/balazinska.html. 228, 230

M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. 2005. Fault-tolerance in the Borealis distributed stream processing system. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 13–24. DOI: 10.1145/1066157.1066160. 228, 230, 234, 325

M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. 2008. Fault-tolerance in the borealis distributed stream processing system. *ACM Transactions on Database Systems*, 33(1): 3:1–3:44. DOI: 10.1145/1331904.1331907.

D. Barbará, J. A. Blakeley, D. H. Fishman, D. B. Lomet, and M. Stonebraker. 1994. The impact of database research on industrial products (panel summary). *ACM SIGMOD Record*, 23(3): 35–40. DOI: 10.1145/187436.187455.

V. Barr and M. Stonebraker. 2015a. A valuable lesson, and whither hadoop? *Communications of the ACM*, 58(1): 18–19. DOI: 10.1145/2686591. 50

V. Barr and M. Stonebraker. 2015b. How men can help women in cs; winning 'computing's nobel prize'. *Communications of the ACM*, 58(11): 10–11. DOI: 10.1145/2820419.

V. Barr, M. Stonebraker, R. C. Fernandez, D. Deng, and M. L. Brodie. 2017. How we teach cs2all, and what to do about database decay. *Communications of the ACM*, 60(1): 10–11. http://dl.acm.org/citation.cfm?id=3014349.

L. Battle, M. Stonebraker, and R. Chang. 2013. Dynamic reduction of query result sets for interactive visualizaton. In *Proc. 2013 IEEE International Conference on Big Data*, pp. 1–8. DOI: 10.1109/BigData.2013.6691708.

L. Battle, R. Chang, and M. Stonebraker. 2016. Dynamic prefetching of data tiles for interactive visualization. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 1363–1375. DOI: 10.1145/2882903.2882919.

R. Berman and M. Stonebraker. 1977. GEO-OUEL: a system for the manipulation and display of geographic data. In *Proc. 4th Annual Conference Computer Graphics and Interactive Techniques*, pp. 186–191. DOI: 10.1145/563858.563892.

P. A. Bernstein, U. Dayal, D. J. DeWitt, D. Gawlick, J. Gray, M. Jarke, B. G. Lindsay, P. C. Lockemann, D. Maier, E. J. Neuhold, A. Reuter, L. A. Rowe, H. Schek, J. W. Schmidt, M. Schrefl, and M. Stonebraker. 1989. Future directions in DBMS research—the Laguna Beach participants. *ACM SIGMOD Record*, 18(1): 17–26. 92

P. A. Bernstein, M. L. Brodie, S. Ceri, D. J. DeWitt, M. J. Franklin, H. Garcia-Molina, J. Gray, G. Held, J. M. Hellerstein, H. V. Jagadish, M. Lesk, D. Maier, J. F. Naughton, H. Pirahesh, M. Stonebraker, and J. D. Ullman. 1998a. The Asilomar report on database research. *ACM SIGMOD Record*, 27(4): 74–80. DOI: 10.1145/306101.306137. 92

P. A. Bernstein, M. L. Brodie, S. Ceri, D. J. DeWitt, M. J. Franklin, H. Garcia-Molina, J. Gray, G. Held, J. M. Hellerstein, H. V. Jagadish, M. Lesk, D. Maier, J. F. Naughton, H. Pirahesh, M. Stonebraker, and J. D. Ullman. 1998b. The Asilomar report on database research. *CoRR*, cs.DB/9811013. http://arxiv.org/abs/cs.DB/9811013. 92

A. Bhide and M. Stonebraker. 1987. Performance issues in high performance transaction processing architectures. In *Proc. 2nd International Workshop High Performance Transaction Systems*, pp. 277–300. DOI: 10.1007/3-540-51085-0_51. 91

A. Bhide and M. Stonebraker. 1988. A performance comparison of two architectures for fast transaction processing. In *Proc. 4th International Conference on Data Engineering*, pp. 536–545. DOI: 10.1109/ICDE.1988.105501. 91

M. L. Brodie and M. Stonebraker. 1993. Darwin: On the incremental migration of legacy information systems. Technical Report TR-0222-10-92-165, GTE Laboratories Incorporated.

M. L. Brodie and M. Stonebraker. 1995a. *Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach*. Morgan Kaufmann. 91

M. L. Brodie and M. Stonebraker. 1995b. *Legacy Information Systems Migration: Gateways, Interfaces, and the Incremental Approach*. Morgan Kaufmann.

M. L. Brodie, R. M. Michael Stonebraker, and J. Pei. 2018. The case for the co-evolution of applications and data. In *New England Database Days*.

P. Brown and M. Stonebraker. 1995. Bigsur: A system for the management of earth science data. In *Proc. 21th International Conference on Very Large Data Bases*, pp. 720–728. http://www.vldb.org/conf/1995/P720.pdf.

M. J. Carey and M. Stonebraker. 1984. The performance of concurrency control algorithms for database management systems. In *Proc. 10th International Conference on Very Large Data Bases*, pp. 107–118. http://www.vldb.org/conf/1984/P107.pdf. 91, 200

D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik. 2002. Monitoring streams—A new class of data management applications. In *Proc. 28th International Conference on Very Large Data Bases*, pp. 215–226. DOI: 10.1016/B978-155860869-6/50027-5. 228, 229, 324

D. Carney, U. Çetintemel, A. Rasin, S. B. Zdonik, M. Cherniack, and M. Stonebraker. 2003. Operator scheduling in a data stream manager. In *Proc. 29th International Conference on Very Large Data Bases*, pp. 838–849. http://www.vldb.org/conf/2003/papers/S25P02.pdf. 228, 229

U. Çetintemel, J. Du, T. Kraska, S. Madden, D. Maier, J. Meehan, A. Pavlo, M. Stonebraker, E. Sutherland, N. Tatbul, K. Tufte, H. Wang, and S. B. Zdonik. 2014. S-store: A streaming NewSQL system for big velocity applications. *Proc. VLDB Endowment*, 7(13): 1633–1636. http://www.vldb.org/pvldb/vol7/p1633-cetintemel.pdf. 234, 251

U. Çetintemel, D. J. Abadi, Y. Ahmad, H. Balakrishnan, M. Balazinska, M. Cherniack, J. Hwang, S. Madden, A. Maskey, A. Rasin, E. Ryvkina, M. Stonebraker, N. Tatbul, Y. Xing, and S. Zdonik. 2016. The Aurora and Borealis stream processing engines. In M. N. Garofalakis, J. Gehrke, and R. Rastogi, editors, *Data Stream Management—Processing High-Speed Data Streams*, pp. 337–359. Springer. ISBN 978-3-540-28607-3. DOI: 10.1007/978-3-540-28608-0_17.

R. Chandra, A. Segev, and M. Stonebraker. 1994. Implementing calendars and temporal rules in next generation databases. In *Proc. 10th International Conference on Data Engineering*, pp. 264–273. DOI: 10.1109/ICDE.1994.283040. 91

S. Chaudhuri, A. K. Chandra, U. Dayal, J. Gray, M. Stonebraker, G. Wiederhold, and M. Y. Vardi. 1996. Database research: Lead, follow, or get out of the way?—panel abstract. In *Proc. 12th International Conference on Data Engineering*, p. 190.

P. Chen, V. Gadepally, and M. Stonebraker. 2016. The BigDAWG monitoring framework. In *Proc. 2016 IEEE High Performance Extreme Computing Conference*, pp. 1–6. DOI: 10.1109/HPEC.2016.7761642. 373

Y. Chi, C. R. Mechoso, M. Stonebraker, K. Sklower, R. Troy, R. R. Muntz, and E. Mesrobian. 1997. ESMDIS: earth system model data information system. In *Proc. 9th International Conference on Scientific and Statistical Database Management*, pp. 116–118. DOI: 10.1109/SSDM.1997.621169.

P. Cudré-Mauroux, H. Kimura, K. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. J. DeWitt, B. Heath, D. Maier, S. Madden, J. M. Patel, M. Stonebraker, and S. B. Zdonik. 2009. A demonstration of SciDB: A science-oriented DBMS. *Proc. VLDB Endowment*, 2(2): 1534–1537. DOI: 10.14778/1687553.1687584.

J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. B. Zdonik. 2013. Anti-caching: A new approach to database management system architecture. *Proc. VLDB Endowment*, 6(14): 1942–1953. http://www.vldb.org/pvldb/vol6/p1942-debrabant.pdf.

J. DeBrabant, J. Arulraj, A. Pavlo, M. Stonebraker, S. B. Zdonik, and S. Dulloor. 2014. A prolegomenon on OLTP database systems for non-volatile memory. In *Proc. 5th International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, pp. 57–63. http://www.adms-conf.org/2014/adms14_debrabant.pdf.

D. Deng, R. C. Fernandez, Z. Abedjan, S. Wang, M. Stonebraker, A. K. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, and N. Tang. 2017a. The data civilizer system. In *Proc. 8th Biennial Conference on Innovative Data Systems Research*. http://cidrdb.org/cidr2017/papers/p44-deng-cidr17.pdf. 293

D. Deng, A. Kim, S. Madden, and M. Stonebraker. 2017b. SILKMOTH: an efficient method for finding related sets with maximum matching constraints. *CoRR*, abs/1704.04738. http://arxiv.org/abs/1704.04738.

D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. A. Wood. 1984. Implementation techniques for main memory database systems. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 1–8. DOI: 10.1145/602259.602261. 111

D. J. DeWitt and M. Stonebraker. January 2008. MapReduce: A major step backwards. *The Database Column*. http://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.html. Accessed April 8, 2018. 50, 114, 136, 184, 209

D. J. DeWitt, I. F. Ilyas, J. F. Naughton, and M. Stonebraker. 2013. We are drowning in a sea of least publishable units (lpus). In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 921–922. DOI: 10.1145/2463676.2465345.

P. Dobbins, T. Dohzen, C. Grant, J. Hammer, M. Jones, D. Oliver, M. Pamuk, J. Shin, and M. Stonebraker. 2007. Morpheus 2.0: A data transformation management system. In *Proc. 3rd International Workshop on Database Interoperability*.

T. Dohzen, M. Pamuk, J. Hammer, and M. Stonebraker. 2006. Data integration through transform reuse in the Morpheus project. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 736–738. DOI: 10.1145/1142473.1142571.

J. Dozier, M. Stonebraker, and J. Frew. 1994. Sequoia 2000: A next-generation information system for the study of global change. In *Proc. 13th IEEE Symposium Mass Storage Systems*, pp. 47–56. DOI: 10.1109/MASS.1994.373028.

J. Duggan and M. Stonebraker. 2014. Incremental elasticity for array databases. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 409–420. DOI: 10.1145/2588555.2588569.

J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. B. Zdonik. 2015a. The BigDAWG polystore system. *ACM SIGMOD Record*, 44(2): 11–16. DOI: 10.1145/2814710.2814713. 284

J. Duggan, O. Papaemmanouil, L. Battle, and M. Stonebraker. 2015b. Skew-aware join optimization for array databases. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 123–135. DOI: 10.1145/2723372.2723709.

A. Dziedzic, J. Duggan, A. J. Elmore, V. Gadepally, and M. Stonebraker. 2015. BigDAWG: a polystore for diverse interactive applications. *Data Systems for Interactive Analysis Workshop*.

A. Dziedzic, A. J. Elmore, and M. Stonebraker. 2016. Data transformation and migration in polystores. In *Proc. 2016 IEEE High Performance Extreme Computing Conference*, pp. 1–6. DOI: 10.1109/HPEC.2016.7761594. 372

A. J. Elmore, J. Duggan, M. Stonebraker, M. Balazinska, U. Çetintemel, V. Gadepally, J. Heer, B. Howe, J. Kepner, T. Kraska, S. Madden, D. Maier, T. G. Mattson, S. Papadopoulos, J. Parkhurst, N. Tatbul, M. Vartak, and S. Zdonik. 2015. A demonstration of the BigDAWG polystore system. *Proc. VLDB Endowment*, 8(12): 1908–1911. http://www.vldb.org/pvldb/vol8/p1908-Elmore.pdf. 287, 371

R. S. Epstein and M. Stonebraker. 1980. Analysis of distributed data base processing strategies. In *Proc. 6th International Conference on Very Data Bases*, pp. 92–101.

R. S. Epstein, M. Stonebraker, and E. Wong. 1978. Distributed query processing in a relational data base system. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 169–180. DOI: 10.1145/509252.509292. 198

R. C. Fernandez, Z. Abedjan, S. Madden, and M. Stonebraker. 2016. Towards large-scale data discovery: position paper. In *Proc. 3rd International Workshop on Exploratory Search in Databases and the Web*, pp. 3–5. DOI: 10.1145/2948674.2948675.

R. C. Fernandez, D. Deng, E. Mansour, A. A. Qahtan, W. Tao, Z. Abedjan, A. K. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, M. Stonebraker, and N. Tang. 2017b. A demo of the data civilizer system. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 1639–1642. DOI: 10.1145/3035918.3058740.

R. C. Fernandez, D. Deng, E. Mansour, A. A. Qahtan, W. Tao, Z. Abedjan, A. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, M. Stonebraker, and N. Tang. 2017a. A demo of the data civilizer system. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 1636–1642. 293

R. C. Fernandez, Z. Abedjan, F. Koko, G. Yuan, S. Madden, and M. Stonebraker. 2018a. Aurum: A data discovery system. In *Proc. 34th International Conference on Data Engineering*, pp. 1001–1012.

R. C. Fernandez, E. Mansour, A. Qahtan, A. Elmagarmid, I. Ilyas, S. Madden, M. Ouzzani, M. Stonebraker, and N. Tang. 2018b. Seeping semantics: Linking datasets using word embeddings for data discovery. In *Proc. 34th International Conference on Data Engineering*, pp. 989–1000.

V. Gadepally, P. Chen, J. Duggan, A. J. Elmore, B. Haynes, J. Kepner, S. Madden, T. Mattson, and M. Stonebraker. 2016a. The BigDAWG polystore system and architecture. In *Proc. 2016 IEEE High Performance Extreme Computing Conference*, pp. 1–6. DOI: 10.1109/HPEC.2016.7761636. 287, 373

V. Gadepally, P. Chen, J. Duggan, A. J. Elmore, B. Haynes, J. Kepner, S. Madden, T. Mattson, and M. Stonebraker. 2016b. The BigDAWG polystore system and architecture. *CoRR*, abs/1609.07548. http://arxiv.org/abs/1609.07548.

V. Gadepally, P. Chen, J. Duggan, A. Elmore, B. Haynes, J. Kepnera, S. Madden, T. Mattson, and M. Stonebraker. 2016c. The BigDAWG polystore system and architecture. In *Proc. 2016 IEEE High Performance Extreme Computing Conference*, pp. 1–6. DOI: 10.1109/HPEC.2016.7761636.

V. Gadepally, J. Duggan, A. J. Elmore, J. Kepner, S. Madden, T. Mattson, and M. Stonebraker. 2016d. The BigDAWG architecture. *CoRR*, abs/1602.08791. http://arxiv.org/abs/1602.08791.

A. Go, M. Stonebraker, and C. Williams. 1975. An approach to implementing a geo-data system. In *Proc. Workshop on Data Bases for Interactive Design*, pp. 67–77.

J. Gray, H. Schek, M. Stonebraker, and J. D. Ullman. 2003. The Lowell report. In *Proc. ACM SIGMOD International Conference on Management of Data*, p. 680. DOI: 10.1145/872757.872873. 92

M. N. Gubanov and M. Stonebraker. 2013. Bootstraping synonym resolution at web scale. In *Proc. DIMACS/CCICADA Workshop on Big Data Integration*.

M. N. Gubanov and M. Stonebraker. 2014. Large-scale semantic profile extraction. In *Proc. 17th International Conference on Extending Database Technology*, pp. 644–647. DOI: 10.5441/002/edbt.2014.64.

M. N. Gubanov, M. Stonebraker, and D. Bruckner. 2014. Text and structured data fusion in data tamer at scale. In *Proc. 30th International Conference on Data Engineering*, pp. 1258–1261. DOI: 10.1109/ICDE.2014.6816755.

A. M. Gupta, V. Gadepally, and M. Stonebraker. 2016. Cross-engine query execution in federated database systems. In *Proc. 2016 IEEE High Performance Extreme Computing Conference*, pp. 1–6. DOI: 10.1109/HPEC.2016.7761648. 373

A. Guttman and M. Stonebraker. 1982. Using a relational database management system for computer aided design data. *Quarterly Bulletin IEEE Technical Committee on Data Engineering*, 5(2): 21–28. http://sites.computer.org/debull/82JUN-CD.pdf. 201

J. Hammer, M. Stonebraker, and O. Topsakal. 2005. THALIA: test harness for the assessment of legacy information integration approaches. In *Proc. 21st International Conference on Data Engineering*, pp. 485–486. DOI: 10.1109/ICDE.2005.140.

R. Harding, D. V. Aken, A. Pavlo, and M. Stonebraker. 2017. An evaluation of distributed concurrency control. *Proc. VLDB Endowment*, 10(5): 553–564. DOI: 10.14778/3055540.3055548.

S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. 2008. OLTP through the looking glass, and what we found there. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 981–992. DOI: 10.1145/1376616.1376713. 152, 246, 251, 346

P. B. Hawthorn and M. Stonebraker. 1979. Performance analysis of a relational data base management system. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 1–12. DOI: 10.1145/582095.582097.

G. Held and M. Stonebraker. 1975. Storage structures and access methods in the relational data base management system INGRES. In *Proc. ACM Pacific 75—Data: Its Use, Organization and Management*, pp. 26–33. 194

G. Held and M. Stonebraker. 1978. B-trees re-examined. *Communications of the ACM*, 21(2): 139–143. DOI: 10.1145/359340.359348. 90, 197

G. Held, M. Stonebraker, and E. Wong. 1975. INGRES: A relational data base system. In *National Computer Conference*, pp. 409–416. DOI: 10.1145/1499949.1500029. 102, 397

J. M. Hellerstein and M. Stonebraker. 1993. Predicate migration: Optimizing queries with expensive predicates. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 267–276. DOI: 10.1145/170035.170078.

J. M. Hellerstein and M. Stonebraker. 2005. *Readings in Database Systems*, 4. MIT Press. ISBN 978-0-262-69314-1. http://mitpress.mit.edu/books/readings-database-systems.

J. M. Hellerstein, M. Stonebraker, and R. Caccia. 1999. Independent, open enterprise data integration. *Quarterly Bulletin IEEE Technical Committee on Data Engineering*, 22(1): 43–49. http://sites.computer.org/debull/99mar/cohera.ps.

J. M. Hellerstein, M. Stonebraker, and J. R. Hamilton. 2007. Architecture of a database system. *Foundations and Trends in Databases*, 1(2): 141–259. DOI: 10.1561/1900000002.

W. Hong and M. Stonebraker. 1991. Optimization of parallel query execution plans in XPRS. In *Proc. 1st International Conference on Parallel and Distributed Information Systems*, pp. 218–225. DOI: 10.1109/PDIS.1991.183106.

W. Hong and M. Stonebraker. 1993. Optimization of parallel query execution plans in XPRS. *Distributed and Parallel Databases*, 1(1): 9–32. DOI: 10.1007/BF01277518.

J. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. B. Zdonik. 2005. High-availability algorithms for distributed stream processing. In *Proc. 21st International Conference on Data Engineering*, pp. 779–790. DOI: 10.1109/ICDE.2005.72. 228, 230, 325

A. Jhingran and M. Stonebraker. 1990. Alternatives in complex object representation: A performance perspective. In *Proc. 6th International Conference on Data Engineering*, pp. 94–102. DOI: 10.1109/ICDE.1990.113458.

A. Jindal, P. Rawlani, E. Wu, S. Madden, A. Deshpande, and M. Stonebraker. 2014. VERTEXICA: your relational friend for graph analytics! *Proc. VLDB Endowment*, 7(13): 1669–1672. http://www.vldb.org/pvldb/vol7/p1669-jindal.pdf.

R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endowment*, 1(2): 1496–1499. DOI: 10.14778/1454159.1454211. 247, 249, 341

R. H. Katz, J. K. Ousterhout, D. A. Patterson, and M. Stonebraker. 1988. A project on high performance I/O subsystems. *Quarterly Bulletin IEEE Technical Committee on Data Engineering*, 11(1): 40–47. http://sites.computer.org/debull/88MAR-CD.pdf.

J. T. Kohl, C. Staelin, and M. Stonebraker. 1993a. Highlight: Using a log-structured file system for tertiary storage management. In *Proc. of the Usenix Winter 1993 Technical Conference*, pp. 435–448.

J. T. Kohl, M. Stonebraker, and C. Staelin. 1993b. Highlight: a file system for tertiary storage. In *Proc. 12th IEEE Symposium Mass Storage Systems*, pp. 157–161. DOI: 10.1109/MASS.1993.289765.

C. P. Kolovson and M. Stonebraker. 1989. Indexing techniques for historical databases. In *Proc. 5th International Conference on Data Engineering*, pp. 127–137. DOI: 10.1109/ICDE.1989.47208.

C. P. Kolovson and M. Stonebraker. 1991. Segment indexes: Dynamic indexing techniques for multi-dimensional interval data. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 138–147. DOI: 10.1145/115790.115807.

R. A. Kowalski, D. B. Lenat, E. Soloway, M. Stonebraker, and A. Walker. 1988. Knowledge management—panel report. In *Proc. 2nd International Conference on Expert Database Systems*, pp. 63–69.

A. Kumar and M. Stonebraker. 1987a. The effect of join selectivities on optimal nesting order. *ACM SIGMOD Record*, 16(1): 28–41. DOI: 10.1145/24820.24822.

A. Kumar and M. Stonebraker. 1987b. Performance evaluation of an operating system transaction manager. In *Proc. 13th International Conference on Very Large Data Bases*, pp. 473–481. http://www.vldb.org/conf/1987/P473.pdf.

A. Kumar and M. Stonebraker. 1988. Semantics based transaction management techniques for replicated data. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 117–125. DOI: 10.1145/50202.50215.

A. Kumar and M. Stonebraker. 1989. Performance considerations for an operating system transaction manager. *IEEE Transactions on Software Engineering*, 15(6): 705–714. DOI: 10.1109/32.24724.

R. Kung, E. N. Hanson, Y. E. Ioannidis, T. K. Sellis, L. D. Shapiro, and M. Stonebraker. 1984. Heuristic search in data base systems. In *Proc. 1st International Workshop on Expert Database Systems*, pp. 537–548.

C. A. Lynch and M. Stonebraker. 1988. Extended user-defined indexing with application to textual databases. In *Proc. 14th International Conference on Very Large Data Bases*, pp. 306–317. http://www.vldb.org/conf/1988/P306.pdf.

N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. 2014. Rethinking main memory OLTP recovery. In *Proc. 30th International Conference on Data Engineering*, pp. 604–615. DOI: 10.1109/ICDE.2014.6816685.

E. Mansour, D. Deng, A. Qahtan, R. C. Fernandez, Wenbo, Z. Abedjan, A. Elmagarmid, I. Ilyas, S. Madden, M. Ouzzani, M. Stonebraker, and N. Tang. 2018. Building data civilizer pipelines with an advanced workflow engine. In *Proc. 34th International Conference on Data Engineering*, pp. 1593–1596.

T. Mattson, D. A. Bader, J. W. Berry, A. Buluç, J. Dongarra, C. Faloutsos, J. Feo, J. R. Gilbert, J. Gonzalez, B. Hendrickson, J. Kepner, C. E. Leiserson, A. Lumsdaine, D. A. Padua, S. Poole, S. P. Reinhardt, M. Stonebraker, S. Wallach, and A. Yoo. 2013. Standards for graph algorithm primitives. In *Proc. 2013 IEEE High Performance Extreme Computing Conference*, pp. 1–2. DOI: 10.1109/HPEC.2013.6670338.

T. Mattson, D. A. Bader, J. W. Berry, A. Buluç, J. J. Dongarra, C. Faloutsos, J. Feo, J. R. Gilbert, J. Gonzalez, B. Hendrickson, J. Kepner, C. E. Leiserson, A. Lumsdaine, D. A. Padua, S. W. Poole, S. P. Reinhardt, M. Stonebraker, S. Wallach, and A. Yoo. 2014. Standards for graph algorithm primitives. *CoRR*, abs/1408.0393. DOI: 10.1109/HPEC.2013.6670338.

N. H. McDonald and M. Stonebraker. 1975. CUPID - the friendly query language. In *Proc. ACM Pacific 75—Data: Its Use, Organization and Management*, pp. 127–131.

J. Meehan, N. Tatbul, S. B. Zdonik, C. Aslantas, U. Çetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, M. Stonebraker, K. Tufte, and H. Wang. 2015a. S-store: Streaming meets transaction processing. *CoRR*, abs/1503.01143. DOI: 10.14778/2831360.2831367. 234

J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Çetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, M. Stonebraker, K. Tufte, and H. Wang. 2015b. S-store: Streaming meets transaction processing. *Proc. VLDB Endowment*, 8(13): 2134–2145. DOI: 10.14778/2831360.2831367. 234, 288, 331, 374

J. Morcos, Z. Abedjan, I. F. Ilyas, M. Ouzzani, P. Papotti, and M. Stonebraker. 2015. Dataxformer: An interactive data transformation tool. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 883–888. DOI: 10.1145/2723372.2735366. 296

B. Muthuswamy, L. Kerschberg, C. Zaniolo, M. Stonebraker, D. S. P. Jr., and M. Jarke. 1985. Architectures for expert-DBMS (panel). In *Proc. 1985 ACM Annual Conference on the Range of Computing: Mid-80's*, pp. 424–426. DOI: 10.1145/320435.320555.

K. O'Brien, V. Gadepally, J. Duggan, A. Dziedzic, A. J. Elmore, J. Kepner, S. Madden, T. Mattson, Z. She, and M. Stonebraker. 2017. BigDAWG polystore release and demonstration. *CoRR*, abs/1701.05799. http://arxiv.org/abs/1701.05799.

V. E. Ogle and M. Stonebraker. 1995. Chabot: Retrieval from a relational database of images. *IEEE Computer*, 28(9): 40–48. DOI: 10.1109/2.410150.

M. A. Olson, W. Hong, M. Ubell, and M. Stonebraker. 1996. Query processing in a parallel object-relational database system. *Quarterly Bulletin IEEE Technical Committee on Data Engineering*, 19(4): 3–10. http://sites.computer.org/debull/96DEC-CD.pdf.

C. Olston, M. Stonebraker, A. Aiken, and J. M. Hellerstein. 1998a. VIQING: visual interactive querying. In *Proc. 1998 IEEE Symposium on Visual Languages*, pp. 162–169. DOI: 10.1109/VL.1998.706159.

C. Olston, A. Woodruff, A. Aiken, M. Chu, V. Ercegovac, M. Lin, M. Spalding, and M. Stonebraker. 1998b. Datasplash. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 550–552. DOI: 10.1145/276304.276377.

J. Ong, D. Fogg, and M. Stonebraker. 1984. Implementation of data abstraction in the relational database system ingres. *ACM SIGMOD Record*, 14(1): 1–14. DOI: 10.1145/984540.984541. 201, 202, 206

R. Overmyer and M. Stonebraker. 1982. Implementation of a time expert in a data base system. *ACM SIGMOD Record*, 12(3): 51–60. DOI: 10.1145/984505.984509.

A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. 2009. A comparison of approaches to large-scale data analysis. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 165–178. DOI: 10.1145/1559845.1559865.

H. Pirk, S. Madden, and M. Stonebraker. 2015. By their fruits shall ye know them: A data analyst's perspective on massively parallel system design. In *Proc. 11th Workshop on Data Management on New Hardware*, pp. 5:1–5:6. DOI: 10.1145/2771937.2771944.

G. Planthaber, M. Stonebraker, and J. Frew. 2012. Earthdb: scalable analysis of MODIS data using SciDB. In *Proc. 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, pp. 11–19. DOI: 10.1145/2447481.2447483.

S. Potamianos and M. Stonebraker. 1996. The POSTGRES rules system. In *Active Database Systems: Triggers and Rules For Advanced Database Processing*, pp. 43–61. Morgan Kaufmann. 91, 168

C. Ré, D. Agrawal, M. Balazinska, M. Cafarella, M. Jordan, T. Kraska, and R. Ramakrishnan. 2015. Machine learning and databases: The sound of things to come or a cacophony of hype? In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 283–284.

D. R. Ries and M. Stonebraker. 1977a. Effects of locking granularity in a database management system. *ACM Transactions on Database Systems*, 2(3): 233–246. DOI: 10.1145/320557.320566. 91, 198

D. R. Ries and M. Stonebraker. 1977b. A study of the effects of locking granularity in a data base management system (abstract). In *Proc. ACM SIGMOD International Conference on Management of Data*, p. 121. DOI: 10.1145/509404.509422. 91

D. R. Ries and M. Stonebraker. 1979. Locking granularity revisited. *ACM Transactions on Database Systems*, 4(2): 210–227. http://doi.acm.org/10.1145/320071.320078. DOI: 10.1145/320071.320078. 91

L. A. Rowe and M. Stonebraker. 1981. Architecture of future data base systems. *ACM SIGMOD Record*, 11(1): 30–44. DOI: 10.1145/984471.984473.

L. A. Rowe and M. Stonebraker. 1986. The commercial INGRES epilogue. In M. Stonebraker, editor, *The INGRES Papers: Anatomy of a Relational Database System*, pp. 63–82. Addison-Wesley.

L. A. Rowe and M. Stonebraker. 1987. The POSTGRES data model. In *Proc. 13th International Conference on Very Large Data Bases*, pp. 83–96. http://www.vldb.org/conf/1987/P083.pdf. 258

L. A. Rowe and M. Stonebraker. 1990. The POSTGRES data model. In A. F. Cardenas and D. McLeod, editors, *Research Foundations in Object-Oriented and Semantic Database Systems*, pp. 91–110. Prentice Hall.

S. Sarawagi and M. Stonebraker. 1994. Efficient organization of large multidimensional arrays. In *Proc. 10th International Conference on Data Engineering*, pp. 328–336. DOI: 10.1109/ICDE.1994.283048.

S. Sarawagi and M. Stonebraker. 1996. Reordering query execution in tertiary memory databases. In *Proc. 22th International Conference on Very Large Data Bases*. http://www.vldb.org/conf/1996/P156.pdf.

G. A. Schloss and M. Stonebraker. 1990. Highly redundant management of distributed data. In *Proc. Workshop on the Management of Replicated Data*, pp. 91–92.

A. Seering, P. Cudré-Mauroux, S. Madden, and M. Stonebraker. 2012. Efficient versioning for scientific array databases. In *Proc. 28th International Conference on Data Engineering*, pp. 1013–1024. DOI: 10.1109/ICDE.2012.102.

L. J. Seligman, N. J. Belkin, E. J. Neuhold, M. Stonebraker, and G. Wiederhold. 1995. Metrics for accessing heterogeneous data: Is there any hope? (panel). In *Proc. 21th International Conference on Very Large Data Bases*, p. 633. http://www.vldb.org/conf/1995/P633.pdf.

M. I. Seltzer and M. Stonebraker. 1990. Transaction support in read optimizied and write optimized file systems. In *Proc. 16th International Conference on Very Large Data Bases*, pp. 174–185. http://www.vldb.org/conf/1990/P174.pdf.

M. I. Seltzer and M. Stonebraker. 1991. Read optimized file system designs: A performance evaluation. In *Proc. 7th International Conference on Data Engineering*, pp. 602–611. DOI: 10.1109/ICDE.1991.131509.

M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker. 2016. Clay: Fine-grained adaptive partitioning for general database schemas. *Proc. VLDB Endowment*, 10(4): 445–456. DOI: 10.14778/3025111.3025125.

J. Sidell, P. M. Aoki, A. Sah, C. Staelin, M. Stonebraker, and A. Yu. 1996. Data replication in mariposa. In *Proc. 12th International Conference on Data Engineering*, pp. 485–494. DOI: 10.1109/ICDE.1996.492198.

A. Silberschatz, M. Stonebraker, and J. D. Ullman. 1990. Database systems: Achievements and opportunities—the "Lagunita" report of the NSF invitational workshop on the future of database system research held in Palo Alto, CA, February 22–23, 1990. *ACM SIGMOD Record*, 19(4): 6–22. DOI: 10.1145/122058.122059. 92

A. Silberschatz, M. Stonebraker, and J. D. Ullman. 1991. Database systems: Achievements and opportunities. *Communications of the ACM*, 34(10): 110–120. DOI: 10.1145/125223.125272.

A. Silberschatz, M. Stonebraker, and J. D. Ullman. 1996. Database research: Achievements and opportunities into the 21st century. *ACM SIGMOD Record*, 25(1): 52–63. DOI: 10.1145/381854.381886.

D. Skeen and M. Stonebraker. 1981. A formal model of crash recovery in a distributed system. In *Proc. 5th Berkeley Workshop on Distributed Data Management and Computer Networks*, pp. 129–142.

D. Skeen and M. Stonebraker. 1983. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, 9(3): 219–228. DOI: 10.1109/TSE.1983.236608. 199

M. Stonebraker and R. Cattell. 2011. 10 rules for scalable performance in "simple operation" datastores. *Communications of the ACM*, 54(6): 72–80. DOI: 10.1145/1953122.1953144.

M. Stonebraker and U. Çetintemel. 2005. "One size fits all": An idea whose time has come and gone (abstract). In *Proc. 21st International Conference on Data Engineering*, pp. 2–11. DOI: 10.1109/ICDE.2005.1. 50, 92, 103, 131, 152, 367, 401

M. Stonebraker and D. J. DeWitt. 2008. A tribute to Jim Gray. *Communications of the ACM*, 51(11): 54–57. DOI: 10.1145/1400214.1400230.

M. Stonebraker and A. Guttman. 1984. Using a relational database management system for computer aided design data—an update. *Quarterly Bulletin IEEE Technical Committee on Data Engineering*, 7(2): 56–60. http://sites.computer.org/debull/84JUN-CD.pdf.

M. Stonebraker and A. Guttman. 1984. R-trees: a dynamic index structure for spatial searching. In *Proc. of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD '84)*, pp. 47–57. ACM, New York. DOI:10.1145/602259.602266. 201

M. Stonebraker and M. A. Hearst. 1988. Future trends in expert data base systems. In *Proc. 2nd International Conference on Expert Database Systems*, pp. 3–20. 395

M. Stonebraker and G. Held. 1975. Networks, hierarchies and relations in data base management systems. In *Proc. ACM Pacific 75—Data: Its Use, Organization and Management*, pp. 1–9.

M. Stonebraker and J. M. Hellerstein, editors. 1998. *Readings in Database Systems*, 3. Morgan Kaufmann.

M. Stonebraker and J. M. Hellerstein. 2001. Content integration for e-business. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 552–560. DOI: 10.1145/375663.375739.

M. Stonebraker and J. Hong. 2009. Saying good-bye to DBMSS, designing effective interfaces. *Communications of the ACM*, 52(9): 12–13. DOI: 10.1145/1562164.1562169.

M. Stonebraker and J. Hong. 2012. Researchers' big data crisis; understanding design and functionality. *Communications of the ACM*, 55(2): 10–11. DOI: 10.1145/2076450.2076453.

M. Stonebraker and J. Kalash. 1982. TIMBER: A sophisticated relation browser (invited paper). In *Proc. 8th International Conference on Very Data Bases*, pp. 1–10. http://www.vldb.org/conf/1982/P001.pdf.

M. Stonebraker and K. Keller. 1980. Embedding expert knowledge and hypothetical data bases into a data base system. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 58–66. DOI: 10.1145/582250.582261. 200

M. Stonebraker and G. Kemnitz. 1991. The Postgres next generation database management system. *Communications of the ACM*, 34(10): 78–92. DOI: 10.1145/125223.125262. 168, 206, 213

M. Stonebraker and A. Kumar. 1986. Operating system support for data management. *Quarterly Bulletin IEEE Technical Committee on Data Engineering*, 9(3): 43–50. http://sites.computer.org/debull/86SEP-CD.pdf. 47

M. Stonebraker and D. Moore. 1996. *Object-Relational DBMSs: The Next Great Wave*. Morgan Kaufmann. 111

M. Stonebraker and E. J. Neuhold. 1977. A distributed database version of INGRES. In *Proc. 2nd Berkeley Workshop on Distributed Data Management and Computer Networks*, pp. 19–36. 109, 198, 199

M. Stonebraker and M. A. Olson. 1993. Large object support in POSTGRES. In *Proc. 9th International Conference on Data Engineering*, pp. 355–362. DOI: 10.1109/ICDE.1993.344046.

M. Stonebraker and J. Robertson. 2013. Big data is “buzzword du jour;” CS academics “have the best job”. *Communications of the ACM*, 56(9): 10–11. DOI: 10.1145/2500468.2500471.

M. Stonebraker and L. A. Rowe. 1977. Observations on data manipulation languages and their embedding in general purpose programming languages. In *Proc. 3rd International Conference on Very Data Bases*, pp. 128–143.

M. Stonebraker and L. A. Rowe. 1984. Database portals: A new application program interface. In *Proc. 10th International Conference on Very Large Data Bases*, pp. 3–13. http://www.vldb.org/conf/1984/P003.pdf.

M. Stonebraker and L. A. Rowe. 1986. The design of Postgres. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 340–355. DOI: 10.1145/16894.16888. 149, 203, 206

M. Stonebraker and P. Rubinstein. 1976. The INGRES protection system. In *Proc. 1976 ACM Annual Conference*, pp. 80–84. DOI: 10.1145/800191.805536. 398

M. Stonebraker and G. A. Schloss. 1990. Distributed RAID—A new multiple copy algorithm. In *Proc. 6th International Conference on Data Engineering*, pp. 430–437. DOI: 10.1109/ICDE.1990.113496.

M. Stonebraker and A. Weisberg. 2013. The VoltDB main memory DBMS. *Quarterly Bulletin IEEE Technical Committee on Data Engineering*, 36(2): 21–27. http://sites.computer.org/debull/A13june/VoltDB1.pdf.

M. Stonebraker and E. Wong. 1974b. Access control in a relational data base management system by query modification. In *Proc. 1974 ACM Annual Conference, Volume 1*, pp. 180–186. DOI: 10.1145/800182.810400. 45

M. Stonebraker, P. Rubinstein, R. Conway, D. Strip, H. R. Hartson, D. K. Hsiao, and E. B. Fernandez. 1976a. SIGBDP (paper session). In *Proc. 1976 ACM Annual Conference*, p. 79. DOI: 10.1145/800191.805535.

M. Stonebraker, E. Wong, P. Kreps, and G. Held. 1976b. The design and implementation of INGRES. *ACM Transactions on Database Systems*, 1(3): 189–222. DOI: 10.1145/320473.320476. 47, 148, 398

M. Stonebraker, R. R. Johnson, and S. Rosenberg. 1982a. A rules system for a relational data base management system. In *Proc. 2nd International Conference on Databases: Improving Database Usability and Responsiveness*, pp. 323–335. 91, 202

M. Stonebraker, J. Woodfill, J. Ranstrom, M. C. Murphy, J. Kalash, M. J. Carey, and K. Arnold. 1982b. Performance analysis of distributed data base systems. *Quarterly Bulletin IEEE Technical Committee on Data Engineering*, 5(4): 58–65. http://sites.computer.org/debull/82DEC-CD.pdf.

M. Stonebraker, W. B. Rubenstein, and A. Guttman. 1983a. Application of abstract data types and abstract indices to CAD data bases. In *Engineering Design Applications*, pp. 107–113.

M. Stonebraker, H. Stettner, N. Lynn, J. Kalash, and A. Guttman. 1983b. Document processing in a relational database system. *ACM Transactions on Information Systems*, 1(2): 143–158. DOI: 10.1145/357431.357433.

M. Stonebraker, J. Woodfill, and E. Andersen. 1983c. Implementation of rules in relational data base systems. *Quarterly Bulletin IEEE Technical Committee on Data Engineering*, 6(4): 65–74. http://sites.computer.org/debull/83DEC-CD.pdf. 91, 202

M. Stonebraker, J. Woodfill, J. Ranstrom, J. Kalash, K. Arnold, and E. Andersen. 1983d. Performance analysis of distributed data base systems. In *Proc. 2nd Symposium on Reliable Distributed Systems*, pp. 135–138.

M. Stonebraker, J. Woodfill, J. Ranstrom, M. C. Murphy, M. Meyer, and E. Allman. 1983e. Performance enhancements to a relational database system. *ACM Transactions on Database Systems*, 8(2): 167–185. DOI: 10.1145/319983.319984.

M. Stonebraker, E. Anderson, E. N. Hanson, and W. B. Rubenstein. 1984a. Quel as a data type. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 208–214. DOI: 10.1145/602259.602287. 208

M. Stonebraker, J. Woodfill, J. Ranstrom, J. Kalash, K. Arnold, and E. Andersen. 1984b. Performance analysis of distributed data base systems. *Performance Evaluation*, 4(3): 220. DOI: 10.1016/0166-5316(84)90036-1.

M. Stonebraker, D. DuBourdieux, and W. Edwards. 1985. Problems in supporting data base transactions in an operating system transaction manager. *Operating Systems Review*, 19(1): 6–14. DOI: 10.1145/1041490.1041491.

M. Stonebraker, T. K. Sellis, and E. N. Hanson. 1986. An analysis of rule indexing implementations in data base systems. In *Proc. 1st International Conference on Expert Database Systems*, pp. 465–476. 91

M. Stonebraker, J. Anton, and E. N. Hanson. 1987a. Extending a database system with procedures. *ACM Transactions on Database Systems*, 12(3): 350–376. DOI: 10.1145/27629.27631.

M. Stonebraker, J. Anton, and M. Hirohama. 1987b. Extendability in POSTGRES. *Quarterly Bulletin IEEE Technical Committee on Data Engineering*, 10(2): 16–23. http://sites.computer.org/debull/87JUN-CD.pdf.

M. Stonebraker, E. N. Hanson, and C. Hong. 1987c. The design of the postgres rules system. In *Proc. 3th International Conference on Data Engineering*, pp. 365–374. DOI: 10.1109/ICDE.1987.7272402. 91

M. Stonebraker, E. N. Hanson, and S. Potamianos. 1988a. The POSTGRES rule manager. *IEEE Transactions on Software Engineering*, 14(7): 897–907. DOI: 10.1109/32.42733. 91, 168

M. Stonebraker, R. H. Katz, D. A. Patterson, and J. K. Ousterhout. 1988b. The design of XPRS. In *Proc. 14th International Conference on Very Large Data Bases*, pp. 318–330. http://www.vldb.org/conf/1988/P318.pdf.

M. Stonebraker, M. A. Hearst, and S. Potamianos. 1989. A commentary on the POSTGRES rule system. *ACM SIGMOD Record*, 18(3): 5–11. DOI: 10.1145/71031.71032. 91, 168, 395

M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. 1990a. On rules, procedures, caching and views in data base systems. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 281–290. DOI: 10.1145/93597.98737.

M. Stonebraker, L. A. Rowe, and M. Hirohama. 1990b. The implementation of postgres. *IEEE Transactions on Knowledge and Data Engineering*, 2(1): 125–142. DOI: 10.1109/69.50912. 47, 168

M. Stonebraker, L. A. Rowe, B. G. Lindsay, J. Gray, M. J. Carey, and D. Beech. 1990e. Third generation data base system manifesto—The committee for advanced DBMS function. In *Proc. ACM SIGMOD International Conference on Management of Data*, p. 396.

M. Stonebraker, L. A. Rowe, B. G. Lindsay, J. Gray, M. J. Carey, M. L. Brodie, P. A. Bernstein, and D. Beech. 1990c. Third-generation database system manifesto—The committee for advanced DBMS function. *ACM SIGMOD Record*, 19(3): 31–44. DOI: 10.1145/101077.390001. 91

M. Stonebraker, L. A. Rowe, B. G. Lindsay, J. Gray, M. J. Carey, M. L. Brodie, P. A. Bernstein, and D. Beech. 1990d. Third-generation database system manifesto—The committee for advanced DBMS function. In *Proc. IFIP TC2/WG 2.6 Working Conference on Object-Oriented Databases: Analysis, Design & Construction*, pp. 495–511. 91

M. Stonebraker, R. Agrawal, U. Dayal, E. J. Neuhold, and A. Reuter. 1993a. DBMS research at a crossroads: The Vienna update. In *Proc. 19th International Conference on Very Large Data Bases*, pp. 688–692. http://www.vldb.org/conf/1993/P688.pdf.

M. Stonebraker, J. Chen, N. Nathan, C. Paxson, A. Su, and J. Wu. 1993b. Tioga: A database-oriented visualization tool. In *Proceedings IEEE Conference Visualization*, pp. 86–93. DOI: 10.1109/VISUAL.1993.398855. 393

M. Stonebraker, J. Chen, N. Nathan, C. Paxson, and J. Wu. 1993c. Tioga: Providing data management support for scientific visualization applications. In *Proc. 19th International Conference on Very Large Data Bases*, pp. 25–38. http://www.vldb.org/conf/1993/P025.pdf. 393

M. Stonebraker, J. Frew, and J. Dozier. 1993d. The SEQUOIA 2000 project. In *Proc. 3rd International Symposium Advances in Spatial Databases*, pp. 397–412. DOI: 10.1007/3-540-56869-7_22.

M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. 1993e. The Sequoia 2000 benchmark. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 2–11. DOI: 10.1145/170035.170038.

M. Stonebraker, P. M. Aoki, R. Devine, W. Litwin, and M. A. Olson. 1994a. Mariposa: A new architecture for distributed data. In *Proc. 10th International Conference on Data Engineering*, pp. 54–65. DOI: 10.1109/ICDE.1994.283004. 401

M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, A. Sah, and C. Staelin. 1994b. An economic paradigm for query processing and data migration in Mariposa. In *Proc. 3rd International Conference on Parallel and Distributed Information Systems*, pp. 58–67. DOI: 10.1109/PDIS.1994.331732.

M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. 1996. Mariposa: A wide-area distributed database system. *VLDB Journal*, 5(1): 48–63. DOI: 10.1007/s007780050015.

M. Stonebraker, P. Brown, and M. Herbach. 1998a. Interoperability, distributed applications and distributed databases: The virtual table interface. *Quarterly Bulletin IEEE Technical Committee on Data Engineering*, 21(3): 25–33. http://sites.computer.org/debull/98sept/informix.ps.

M. Stonebraker, P. Brown, and D. Moore. 1998b. *Object-Relational DBMSs*, 2. Morgan Kaufmann.

M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. 2005a. C-store: A column-oriented DBMS. In *Proc. 31st International Conference on Very Large Data Bases*, pp. 553–564. http://www.vldb2005.org/program/paper/thu/p553-stonebraker.pdf. 104, 132, 151, 238, 242, 258, 333, 335, 402

M. Stonebraker, U. Çetintemel, and S. B. Zdonik. 2005b. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4): 42–47. DOI: 10.1145/1107499.1107504. 282

M. Stonebraker, C. Bear, U. Çetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. B. Zdonik. 2007a. One size fits all? Part 2: Benchmarking studies. In *Proc. 3rd Biennial Conference on Innovative Data Systems Research*, pp. 173–184. http://www.cidrdb.org/cidr2007/papers/cidr07p20.pdf. 103, 282

M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. 2007b. The end of an architectural era (it's time for a complete rewrite). In *Proc. 33rd International Conference on Very Large Data Bases*, pp. 1150–1160. http://www.vldb.org/conf/2007/papers/industrial/p1150-stonebraker.pdf. 247, 341, 344

M. Stonebraker, J. Becla, D. J. DeWitt, K. Lim, D. Maier, O. Ratzesberger, and S. B. Zdonik. 2009. Requirements for science data bases and SciDB. In *Proc. 4th Biennial Conference on Innovative Data Systems Research*. http://www-db.cs.wisc.edu/cidr/cidr2009/Paper_26.pdf. 257

M. Stonebraker, D. J. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. 2010. Mapreduce and parallel DBMSS: friends or foes? *Communications of the ACM*, 53(1): 64–71. DOI: 10.1145/1629175.1629197. 50, 136, 251

M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. 2011. The architecture of SciDB. In *Proc. 23rd International Conference on Scientific and Statistical Database Management*, pp. 1–16. DOI: 10.1007/978-3-642-22351-8_1.

M. Stonebraker, A. Ailamaki, J. Kepner, and A. S. Szalay. 2012. The future of scientific data bases. In *Proc. 28th International Conference on Data Engineering*, pp. 7–8. DOI: 10.1109/ICDE.2012.151.

M. Stonebraker, P. Brown, D. Zhang, and J. Becla. 2013a. SciDB: A database management system for applications with complex analytics. *Computing in Science and Engineering*, 15(3): 54–62. DOI: 10.1109/MCSE.2013.19.

M. Stonebraker, D. Bruckner, I. F. Ilyas, G. Beskales, M. Cherniack, S. B. Zdonik, A. Pagan, and S. Xu. 2013b. Data curation at scale: The data tamer system. In *Proc. 6th Biennial Conference on Innovative Data Systems Research*. http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper28.pdf. 105, 150, 269, 297, 357, 358

M. Stonebraker, J. Duggan, L. Battle, and O. Papaemmanouil. 2013c. SciDB DBMS research at M.I.T. *Quarterly Bulletin IEEE Technical Committee on Data Engineering*, 36(4): 21–30. http://sites.computer.org/debull/A13dec/p21.pdf.

M. Stonebraker, S. Madden, and P. Dubey. 2013d. Intel “big data” science and technology center vision and execution plan. *ACM SIGMOD Record*, 42(1): 44–49. DOI: 10.1145/2481528.2481537.

M. Stonebraker, A. Pavlo, R. Taft, and M. L. Brodie. 2014. Enterprise database applications and the cloud: A difficult road ahead. In *Proc. 7th IEEE International Conference on Cloud Computing*, pp. 1–6. DOI: 10.1109/IC2E.2014.97.

M. Stonebraker, D. Deng, and M. L. Brodie. 2016. Database decay and how to avoid it. In *Proc. 2016 IEEE International Conference on Big Data*, pp. 7–16. DOI: 10.1109/BigData.2016.7840584.

M. Stonebraker, D. Deng, and M. L. Brodie. 2017. Application-database co-evolution: A new design and development paradigm. In *North East Database Day*, pp. 1–3.

M. Stonebraker. 1972a. Retrieval efficiency using combined indexes. In *Proc. 1972 ACM-SIGFIDET Workshop on Data Description, Access and Control*, pp. 243–256.

M. Stonebraker. 1972b. A simplification of forrester’s model of an urban area. *IEEE Transactions on Systems, Man, and Cybernetics*, 2(4): 468–472. DOI: 10.1109/TSMC.1972.4309156.

M. Stonebraker. 1974a. The choice of partial inversions and combined indices. *International Journal Parallel Programming*, 3(2): 167–188. DOI: 10.1007/BF00976642.

M. Stonebraker. 1974b. A functional view of data independence. In *Proc. 1974 ACM SIGMOD Workshop on Data Description, Access and Control*, pp. 63–81. DOI: 10.1145/800296.811505. 404, 405

M. Stonebraker. 1975. Getting started in INGRES—A tutorial, Memorandum No. ERL-M518, Electronics Research Laboratory, College of Engineering, UC Berkeley. 196

M. Stonebraker. 1975. Implementation of integrity constraints and views by query modification. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 65–78. DOI: 10.1145/500080.500091. 45, 90

M. Stonebraker. 1976a. Proposal for a network INGRES. In *Proc. 1st Berkeley Workshop on Distributed Data Management and Computer Networks*, p. 132.

M. Stonebraker. 1976b. The data base management system INGRES. In *Proc. 1st Berkeley Workshop on Distributed Data Management and Computer Networks*, p. 336. 195

M. Stonebraker. 1976c. A comparison of the use of links and secondary indices in a relational data base system. In *Proc. 2nd International Conference on Software Engineering*, pp. 527–531. http://dl.acm.org/citation.cfm?id=807727.

M. Stonebraker. 1978. Concurrency control and consistency of multiple copies of data in distributed INGRES. In *Proc. 3rd Berkeley Workshop on Distributed Data Management and Computer Networks*, pp. 235–258. 90, 398

M. Stonebraker. May 1979a. Muffin: A distributed database machine. Technical Report ERL Technical Report UCB/ERL M79/28, University of California at Berkeley. 151

M. Stonebraker. 1979b. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Transactions on Software Engineering*, 5(3): 188–194. DOI: 10.1109/TSE.1979.234180. 398

M. Stonebraker. 1980. Retrospection on a database system. *ACM Transactions on Database Systems*, 5(2): 225–240. DOI: 10.1145/320141.320158.

M. Stonebraker. 1981a. Operating system support for database management. *Communications of the ACM*, 24(7): 412–418. DOI: 10.1145/358699.358703.

M. Stonebraker. 1981b. Chairman's column. *ACM SIGMOD Record*, 11(3): i–iv.

M. Stonebraker. 1981c. Chaiman's column. *ACM SIGMOD Record*, 11(4): 2–4.

M. Stonebraker. 1981d. Chairman's column. *ACM SIGMOD Record*, 12(1): 1–3.

M. Stonebraker. 1981e. In memory of Kevin Whitney. *ACM SIGMOD Record*, 12(1): 7.

M. Stonebraker. 1981f. Chairman's column. *ACM SIGMOD Record*, 11(1): 1–4.

M. Stonebraker. 1981g. Hypothetical data bases as views. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 224–229. DOI: 10.1145/582318.582352.

M. Stonebraker. 1982a. Chairman's column. *ACM SIGMOD Record*, 12(3): 2–4.

M. Stonebraker. 1982b. Letter to Peter Denning (two VLDB conferences). *ACM SIGMOD Record*, 12(3): 6–7.

M. Stonebraker. 1982c. Chairman's column. *ACM SIGMOD Record*, 12(4): a–c.

M. Stonebraker. 1982d. Chairman's column. *ACM SIGMOD Record*, 13(1): 2–3&4.

M. Stonebraker. 1982e. Adding semantic knowledge to a relational database system. In M. L. Brodie, M. John, and S. J. W., editors, *On Conceptual Modelling*, pp. 333–352. Springer. DOI: 10.1007/978-1-4612-5196-5_12.

M. Stonebraker. 1982f. A database perspective. In M. L. Brodie, M. John, and S. J. W., editors, *On Conceptual Modelling*, pp. 457–458. Springer. DOI: 10.1007/978-1-4612-5196-5_18.

M. Stonebraker. 1983a. DBMS and AI: is there any common point of view? In *Proc. ACM SIGMOD International Conference on Management of Data*, p. 134. DOI: 10.1145/582192.582215. 201, 205

M. Stonebraker. April 1983b. Chairman's column. *ACM SIGMOD Record*, 13(3): 1–3.

M. Stonebraker. January 1983c. Chairman's column. *ACM SIGMOD Record*, 13(2): 1–3.

M. Stonebraker. 1984. Virtual memory transaction management. *Operating Systems Review*, 18(2): 8–16. DOI: 10.1145/850755.850757. 203

M. Stonebraker. 1985a. Triggers and inference in data base systems. In *Proc. 1985 ACM Annual Conference on the Range of Computing: Mid-80's Perspective*, p. 426. DOI: 10.1145/320435.323372.

M. Stonebraker. 1985b. Triggers and inference in database systems. In M. L. Brodie and J. Mylopoulos, editors, *On Knowledge Base Management Systems*, pp. 297–314. Springer. 202

M. Stonebraker. 1985c. Expert database systems/bases de données et systèmes experts. In *Journées Bases de Données Avancés*.

M. Stonebraker. 1985d. The case for shared nothing. In *Proc. International Workshop on High-Performance Transaction Systems*, p. 0. 91

M. Stonebraker. 1985e. Tips on benchmarking data base systems. *Quarterly Bulletin IEEE Technical Committee on Data Engineering*, 8(1): 10–18. http://sites.computer.org/debull/85MAR-CD.pdf.

M. Stonebraker, editor. 1986a. *The INGRES Papers: Anatomy of a Relational Database System*. Addison-Wesley.

M. Stonebraker. 1986b. Inclusion of new types in relational data base systems. In *Proc. 2nd International Conference on Data Engineering*, pp. 262–269. DOI: 10.1109/ICDE.1986.7266230. 88, 202, 258

M. Stonebraker. 1986c. Object management in Postgres using procedures. In *Proc. International Workshop on Object-Oriented Database Systems*, pp. 66–72. http://dl.acm.org/citation.cfm?id=318840. 45, 88, 399

M. Stonebraker. 1986d. The case for shared nothing. *Quarterly Bulletin IEEE Technical Committee on Data Engineering*, 9(1): 4–9. http://sites.computer.org/debull/86MAR-CD.pdf. 91, 216

M. Stonebraker. 1986e. Design of relational systems (introduction to section 1). In M. Stonebraker, editor, *The INGRES Papers: Anatomy of a Relational Database System*, pp. 1–3. Addison-Wesley.

M. Stonebraker. 1986f. Supporting studies on relational systems (introduction to section 2). In M. Stonebraker, editor, *The INGRES Papers*, pp. 83–85. Addison-Wesley.

M. Stonebraker. 1986g. Distributed database systems (introduction to section 3). In M. Stonebraker, editor, *The INGRES Papers: Anatomy of a Relational Database System*, pp. 183–186. Addison-Wesley.

M. Stonebraker. 1986h. The design and implementation of distributed INGRES. In M. Stonebraker, editor, *The INGRES Papers: Anatomy of a Relational Database System*, pp. 187–196. Addison-Wesley.

M. Stonebraker. 1986i. User interfaces for database systems (introduction to section 4). In M. Stonebraker, editor, *The INGRES Papers: Anatomy of a Relational Database System*, pp. 243–245. Addison-Wesley.

M. Stonebraker. 1986j. Extended semantics for the relational model (introduction to section 5). In M. Stonebraker, editor, *The INGRES Papers: Anatomy of a Relational Database System*, pp. 313–316. Addison-Wesley.

M. Stonebraker. 1986k. Database design (introduction to section 6). In M. Stonebraker, editor, *The INGRES Papers: Anatomy of a Relational Database System*, pp. 393–394. Addison-Wesley.

M. Stonebraker. 1986l. Object management in a relational data base system. In *Digest of Papers - COMPCON*, pp. 336–341.

M. Stonebraker. 1987. The design of the POSTGRES storage system. In *Proc. 13th International Conference on Very Large Data Bases*, pp. 289–300. http://www.vldb.org/conf/1987/P289.pdf. 168, 214, 258

M. Stonebraker, editor. 1988a. *Readings in Database Systems*. Morgan Kaufmann.

M. Stonebraker. 1988b. Future trends in data base systems. In *Proc. 4th International Conference on Data Engineering*, pp. 222–231. DOI: 10.1109/ICDE.1988.105464.

M. Stonebraker. 1989a. The case for partial indexes. *ACM SIGMOD Record*, 18(4): 4–11. DOI: 10.1145/74120.74121.

M. Stonebraker. 1989b. Future trends in database systems. *IEEE Transactions on Knowledge and Data Engineering*, 1(1): 33–44. DOI: 10.1109/69.43402.

M. Stonebraker. 1990a. The third-generation database manifesto: A brief retrospection. In *Proc. IFIP TC2/WG 2.6 Working Conference on Object-Oriented Databases: Analysis, Design & Construction*, pp. 71–72.

M. Stonebraker. 1990b. Architecture of future data base systems. *Quarterly Bulletin IEEE Technical Committee on Data Engineering*, 13(4): 18–23. http://sites.computer.org/debull/90DEC-CD.pdf.

M. Stonebraker. 1990c. Data base research at Berkeley. *ACM SIGMOD Record*, 19(4): 113–118. DOI: 10.1145/122058.122072.

M. Stonebraker. 1990d. Introduction to the special issue on database prototype systems. *IEEE Transactions on Knowledge and Data Engineering*, 2(1): 1–3. DOI: 10.1109/TKDE.1990.10000.

M. Stonebraker. 1990e. The Postgres DBMS. In *Proc. ACM SIGMOD International Conference on Management of Data*, p. 394.

M. Stonebraker. 1991a. Managing persistent objects in a multi-level store. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 2–11. DOI: 10.1145/115790.115791.

M. Stonebraker. 1991b. Object management in Postgres using procedures. In K. R. Dittrich, U. Dayal, and A. P. Buchmann, editors, *On Object-Oriented Database System*, pp. 53–64. Springer. DOI: http://10.1007/978-3-642-84374-7_5.

M. Stonebraker. 1971. The reduction of large scale Markov models for random chains. Ph.D. Dissertation. University of Michigan, Ann Arbor, MI. AAI7123885. 43

M. Stonebraker. 1992a. The integration of rule systems and database systems. *IEEE Transactions on Knowledge and Data Engineering*, 4(5): 415–423. DOI: 10.1109/69.166984. 91

M. Stonebraker, editor. 1992b. *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*. ACM Press.

M. Stonebraker. 1993a. The SEQUOIA 2000 project. *Quarterly Bulletin IEEE Technical Committee on Data Engineering*, 16(1): 24–28. http://sites.computer.org/debull/93MAR-CD.pdf.

M. Stonebraker. 1993b. Are we polishing a round ball? (panel abstract). In *Proc. 9th International Conference on Data Engineering*, p. 606.

M. Stonebraker. 1993c. The miro DBMS. In *Proc. ACM SIGMOD International Conference on Management of Data*, p. 439. DOI: 10.1145/170035.170124. 314

M. Stonebraker. 1994a. SEQUOIA 2000: A reflection of the first three years. In *Proc. 7th International Working Conference on Scientific and Statistical Database Management*, pp. 108–116. DOI: 10.1109/SSDM.1994.336956.

M. Stonebraker, editor. 1994b. *Readings in Database Systems*, 2. Morgan Kaufmann.

M. Stonebraker. 1994c. Legacy systems—the Achilles heel of downsizing (panel). In *Proc. 3rd International Conference on Parallel and Distributed Information Systems*, p. 108.

M. Stonebraker. 1994d. In memory of Bob Kooi (1951-1993). *ACM SIGMOD Record*, 23(1): 3. DOI: 10.1145/181550.181551.

M. Stonebraker. 1995. An overview of the Sequoia 2000 project. *Digital Technical Journal*, 7(3). http://www.hpl.hp.com/hpjournal/dtj/vol7num3/vol7num3art3.pdf. 215, 255

M. Stonebraker. 1998. Are we working on the right problems? (panel). In *Proc. ACM SIGMOD International Conference on Management of Data*, p. 496. DOI: 10.1145/276304.276348.

M. Stonebraker. 2002. Too much middleware. *ACM SIGMOD Record*, 31(1): 97–106. DOI: 10.1145/507338.507362. 91

M. Stonebraker. 2003. Visionary: A next generation visualization system for databases. In *Proc. ACM SIGMOD International Conference on Management of Data*, p. 635. http://www.acm.org/sigmod/sigmod03/eproceedings/papers/ind00.pdf.

M. Stonebraker. 2004. Outrageous ideas and/or thoughts while shaving. In *Proc. 20th International Conference on Data Engineering*, p. 869. DOI: 10.1109/ICDE.2004.1320096.

M. Stonebraker. 2008a. Why did Jim Gray win the Turing Award? *ACM SIGMOD Record*, 37(2): 33–34. DOI: 10.1145/1379387.1379398.

M. Stonebraker. 2008b. Technical perspective—one size fits all: An idea whose time has come and gone. *Communications of the ACM*, 51(12): 76. DOI: 10.1145/1409360.1409379. 92

M. Stonebraker. 2009a. Stream processing. In L. Liu and M. T. Özsu, editors. *Encyclopedia of Database Systems*, pp. 2837–2838. Springer. DOI: 10.1007/978-0-387-39940-9_371.

M. Stonebraker. 2009b. A new direction for tpc? In *Proc. 1st TPC Technology Conference on Performance Evaluation and Benchmarking*, pp. 11–17. DOI: 10.1007/978-3-642-10424-4_2.

M. Stonebraker. 2010a. SQL databases v. nosql databases. *Communications of the ACM*, 53(4): 10–11. DOI: 10.1145/1721654.1721659. 50

M. Stonebraker. 2010b. In search of database consistency. *Communications of the ACM*, 53(10): 8–9. DOI: 10.1145/1831407.1831411.

M. Stonebraker. 2011a. Stonebraker on data warehouses. *Communications of the ACM*, 54(5): 10–11. DOI: 10.1145/1941487.1941491.

M. Stonebraker. 2011b. Stonebraker on nosql and enterprises. *Communications of the ACM*, 54(8): 10–11. DOI: 10.1145/1978542.1978546. 50

M. Stonebraker. 2012a. SciDB: An open-source DBMS for scientific data. *ERCIM News*, 2012(89). http://ercim-news.ercim.eu/en89/special/scidb-an-open-source-dbms-for-scientific-data.

M. Stonebraker. 2012b. New opportunities for new SQL. *Communications of the ACM*, 55(11): 10–11. DOI: 10.1145/2366316.2366319.

M. Stonebraker. 2013. We are under attack; by the least publishable unit. In *Proc. 6th Biennial Conference on Innovative Data Systems Research*. http://www.cidrdb.org/cidr2013/Talks/CIDR13_Gongshow16.ppt. 273

M. Stonebraker. 2015a. Turing lecture. In *Proc. Federated Computing Research Conference*, pp. 2–2. DOI: 10.1145/2820468.2820471.

M. Stonebraker. 2015b. What it's like to win the Turing Award. *Communications of the ACM*, 58(11): 11. xxxi, xxxiii

M. Stonebraker. 2015c. The Case for Polystores. ACM SIGMOD Blog, http://wp.sigmod.org/?p=1629. 370, 371

M. Stonebraker. 2016. The land sharks are on the squawk box. *Communications of the ACM*, 59(2): 74–83. DOI: 10.1145/2869958. 50, 129, 139, 260, 319

M. Stonebraker. 2018. My top ten fears about the DBMS field. In *Proc. 34th International Conference on Data Engineering*, pp. 24–28.

M. Sullivan and M. Stonebraker. 1991. Using write protected data structures to improve software fault tolerance in highly available database management systems. In *Proc. 17th International Conference on Very Large Data Bases*, pp. 171–180. http://www.vldb.org/conf/1991/P171.pdf.

R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. 2014a. E-store: Fine-grained elastic partitioning for distributed transaction processing. *Proc. VLDB Endowment*, 8(3): 245–256. http://www.vldb.org/pvldb/vol8/p245-taft.pdf. 188, 251

R. Taft, M. Vartak, N. R. Satish, N. Sundaram, S. Madden, and M. Stonebraker. 2014b. Genbase: a complex analytics genomics benchmark. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 177–188. DOI: 10.1145/2588555.2595633.

R. Taft, W. Lang, J. Duggan, A. J. Elmore, M. Stonebraker, and D. J. DeWitt. 2016. Step: Scalable tenant placement for managing database-as-a-service deployments. In *Proc. 7th ACM Symposium on Cloud Computing*, pp. 388–400. DOI: 10.1145/2987550.2987575.

R. Taft, N. El-Sayed, M. Serafini, Y. Lu, A. Aboulnaga, M. Stonebraker, R. Mayerhofer, and F. Andrade. 2018. P-Store: an elastic database system with predictive provisioning. In *Proc. ACM SIGMOD International Conference on Management of Data*. 188

W. Tao, D. Deng, and M. Stonebraker. 2017. Approximate string joins with abbreviations. *Proc. VLDB Endowment*, 11(1): 53–65.

N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. 2003. Load shedding in a data stream manager. In *Proc. 29th International Conference on Very Large Data Bases*, pp. 309–320. http://www.vldb.org/conf/2003/papers/S10P03.pdf. 228, 229

N. Tatbul, S. Zdonik, J. Meehan, C. Aslantas, M. Stonebraker, K. Tufte, C. Giossi, and H. Quach. 2015. Handling shared, mutable state in stream processing with correctness guarantees. *Quarterly Bulletin IEEE Technical Committee on Data Engineering*, 38(4): 94–104. http://sites.computer.org/debull/A15dec/p94.pdf.

T. J. Teorey, J. W. DeHeus, R. Gerritsen, H. L. Morgan, J. F. Spitzer, and M. Stonebraker. 1976. SIGMOD (paper session). In *Proc. 1976 ACM Annual Conference*, p. 275. DOI: 10.1145/800191.805596.

M. S. Tuttle, S. H. Brown, K. E. Campbell, J. S. Carter, K. Keck, M. J. Lincoln, S. J. Nelson, and M. Stonebraker. 2001a. The semantic web as "perfection seeking": A view from drug terminology. In *Proc. 1st Semantic Web Working Symposium*, pp. 5–16. http://www.semanticweb.org/SWWS/program/full/paper49.pdf.

M. S. Tuttle, S. H. Brown, K. E. Campbell, J. S. Carter, K. Keck, M. J. Lincoln, S. J. Nelson, and M. Stonebraker. 2001b. The semantic web as "perfection seeking": A view from drug terminology. In I. F. Cruz, S. Decker, J. Euzenat, and D. L. McGuinness, editors, *The Emerging Semantic Web, Selected Papers from the 1st Semantic Web Working Symposium*, volume 75 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

J. Widom, A. Bosworth, B. Lindsey, M. Stonebraker, and D. Suciu. 2000. Of XML and databases (panel session): Where's the beef? In *Proc. ACM SIGMOD International Conference on Management of Data*, p. 576. DOI: 10.1145/335191.335476.

M. W. Wilkins, R. Berlin, T. Payne, and G. Wiederhold. 1985. Relational and entity-relationship model databases and specialized design files in vlsi design. In *Proc. 22nd ACM/IEEE Design Automation Conference*, pp. 410–416.

J. Woodfill and M. Stonebraker. 1983. An implementation of hypothetical relations. In *Proc. 9th International Conference on Very Data Bases*, pp. 157–166. http://www.vldb.org/conf/1983/P157.pdf.

A. Woodruff and M. Stonebraker. 1995. Buffering of intermediate results in dataflow diagrams. In *Proc. IEEE Symposium on Visual Languages*, p. 187. DOI: 10.1109/VL.1995.520808.

A. Woodruff and M. Stonebraker. 1997. Supporting fine-grained data lineage in a database visualization environment. In *Proc. 13th International Conference on Data Engineering*, pp. 91–102. DOI: 10.1109/ICDE.1997.581742.

A. Woodruff, P. Wisnovsky, C. Taylor, M. Stonebraker, C. Paxson, J. Chen, and A. Aiken. 1994. Zooming and tunneling in Tioga: Supporting navigation in multimedia space. In *Proc. IEEE Symposium on Visual Languages*, pp. 191–193. DOI: 10.1109/VL.1994.363622.

A. Woodruff, A. Su, M. Stonebraker, C. Paxson, J. Chen, A. Aiken, P. Wisnovsky, and C. Taylor. 1995. Navigation and coordination primitives for multidimensional visual browsers. In *Proc. IFIP WG 2.6 3rd Working Conference Visual Database Systems*, pp. 360–371. DOI: 10.1007/978-0-387-34905-3_23.

A. Woodruff, J. A. Landay, and M. Stonebraker. 1998a. Goal-directed zoom. In *CHI '98 Conference Summary on Human Factors in Computing Systems*, pp. 305–306. DOI: 10.1145/286498.286781.

A. Woodruff, J. A. Landay, and M. Stonebraker. 1998b. Constant density visualizations of non-uniform distributions of data. In *Proc. 11th Annual ACM Symposium on User Interface Software and Technology*, pp. 19–28. DOI: 10.1145/288392.288397.

A. Woodruff, J. A. Landay, and M. Stonebraker. 1998c. Constant information density in zoomable interfaces. In *Proc. Working Conference on Advanced Visual Interfaces*, pp. 57–65. DOI: 10.1145/948496.948505.

A. Woodruff, J. A. Landay, and M. Stonebraker. 1999. VIDA: (visual information density adjuster). In *CHI '99 Extended Abstracts on Human Factors in Computing Systems*, pp. 19–20. DOI: 10.1145/632716.632730.

A. Woodruff, C. Olston, A. Aiken, M. Chu, V. Ercegovac, M. Lin, M. Spalding, and M. Stonebraker. 2001. Datasplash: A direct manipulation environment for programming semantic zoom visualizations of tabular data. *Journal of Visual Languages and Computing*, 12(5): 551–571. DOI: 10.1006/jvlc.2001.0219.

E. Wu, S. Madden, and M. Stonebraker. 2012. A demonstration of dbwipes: Clean as you query. *Proc. VLDB Endowment*, 5(12): 1894–1897. DOI: 10.14778/2367502.2367531.

E. Wu, S. Madden, and M. Stonebraker. 2013. Subzero: A fine-grained lineage system for scientific databases. In *Proc. 29th International Conference on Data Engineering*, pp. 865–876. DOI: 10.1109/ICDE.2013.6544881.

X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. 2014. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endowment*, 8(3): 209–220. http://www.vldb.org/pvldb/vol8/p209-yu.pdf.

K. Yu, V. Gadepally, and M. Stonebraker. 2017. Database engine integration and performance analysis of the BigDAWG polystore system. *High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017. DOI: 10.1109/HPEC.2017.8091081. 376

S. B. Zdonik, M. Stonebraker, M. Cherniack, U. Çetintemel, M. Balazinska, and H. Balakrishnan. 2003. The aurora and medusa projects. *Quarterly Bulletin IEEE Technical Committee on Data Engineering*, 26(1): 3–10. http://sites.computer.org/debull/A03mar/zdonik.ps. 228, 324

References

D. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. 2005. The design of the Borealis stream processing engine. *Proc. of the 2nd Biennial Conference on Innovative Data Systems Research* (*CIDR*'05), Asilomar, CA, January. 228

Z. Abedjan, L. Golab, and F. Naumann. August 2015. Profiling relational data: a survey. *The VLDB Journal,* 24(4): 557–581. DOI: DOI: 10.1007/s00778-015-0389-y. 297

ACM. 2015a. Announcement: Michael Stonebraker, Pioneer in Database Systems Architecture, Receives 2014 ACM Turing Award. http://amturing.acm.org/award_winners/stonebraker_1172121.cfm. Accessed February 5, 2018.

ACM. March 2015b. Press Release: MIT's Stonebraker Brought Relational Database Systems from Concept to Commercial Success, Set the Research Agenda for the Multibillion-Dollar Database Field for Decades. http://sigmodrecord.org/publications/sigmodRecord/1503/pdfs/04_announcements_Stonebraker.pdf. Accessed February 5, 2018.

ACM. 2016. A.M. Turing Award Citation and Biography. http://amturing.acm.org/award_winners/stonebraker_1172121.cfm. Accessed September 24, 2018. xxxi

Y. Ahmad, B. Berg, U. Çetintemel, M. Humphrey, J. Hwang, A. Jhingran, A. Maskey, O. Papaemmanouil, A. Rasin, N. Tatbul, W. Xing, Y. Xing, and S. Zdonik. June 2005. Distributed operation in the Borealis Stream Processing Engine. Demonstration, *ACM SIGMOD International Conference on Management of Data (SIGMOD'05)*. Baltimore, MD. Best Demonstration Award. 230, 325

M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. 1976. System R: relational approach to database management. *ACM Transactions on Database Systems*, 1(2): 97–137. DOI: 10.1145/320455.320457. 397

P. Bailis, E. Gan, S. Madden, D. Narayanan, K. Rong, and S. Suri. 2017. Macrobase: Prioritizing attention in fast data. *Proc. of the 2017 ACM International Conference on Management of Data*. ACM. DOI: 10.1145/3035918.3035928. 374

Berkeley Software Distribution. n.d. In Wikipedia. http://en.wikipedia.org/wiki/Berkeley_Software_Distribution. Last accessed March 1, 2018. 109

G. Beskales, I.F. Ilyas, L. Golab, and A. Galiullin. 2013. On the relative trust between inconsistent data and inaccurate constraints. *Proc. of the IEEE International Conference on Data Engineering*, *ICDE 2013*, pp. 541–552. Australia. DOI: 10.1109/ICDE.2013.6544854. 270

L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley. 2017. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics http://netlib.org/scalapack/slug/index.html. Last accessed December 31, 2017. 258

D. Bitton, D. J. DeWitt, and C. Turbyfill. 1983. Benchmarking database systems—a systematic approach. Computer Sciences Technical Report #526, University of Wisconsin. http://minds.wisconsin.edu/handle/1793/58490, 111

P. A. Boncz, M. L. Kersten, and S. Manegold. December 2008. Breaking the memory wall in MonetDB. *Communications of the ACM*, 51(12): 77–85. DOI: 10.1145/1409360.1409380. 151

M. L. Brodie. June 2015. Understanding data science: an emerging discipline for data-intensive discovery. In S. Cutt, editor, *Getting Data Right: Tackling the Challenges of Big Data Volume and Variety*. O'Reilly Media, Sebastopol, CA. 291

Brown University, Department of Computer Science. Fall 2002. Next generation stream-based applications. *Conduit Magazine*, 11(2). https://cs.brown.edu/about/conduit/conduit_v11n2.pdf. Last accessed May 14, 2018. 322

BSD licenses. n.d. In Wikipedia. http://en.wikipedia.org/wiki/BSD_licenses. Last accessed March 1, 2018. 109

M. Cafarella and C. Ré. April 2018. The last decade of database research and its blindingly bright future. or Database Research: A love song. DAWN Project, Stanford University. http://dawn.cs.stanford.edu/2018/04/11/db-community/. 6

M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. 1994. Shoring up persistent applications. *Proc. of the 1994 ACM SIGMOD international conference on Management of data (SIGMOD '94)*, 383–394. DOI: 10.1145/191839.191915. 152

M. J. Carey, D. J. Dewitt, M. J. Franklin, N.E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. 1994. Shoring up persistent applications. In *Proc. of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD '94)*, pp. 383–394. DOI: 10.1145/191839.191915. 336

M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. E. Cody, R. Fagin, M. Flickner, A. W. Luniewski, W. Niblack, and D. Petkovic. 1995. Towards heterogeneous multimedia information systems: The garlic approach. In *Research Issues in Data Engineering, 1995: Distributed Object Management, Proceedings*, pp. 124–131. IEEE. DOI: 10.1109/RIDE.1995.378736. 284

CERN. http://home.cern/about/computing. Last accessed December 31, 2017.

D. D. Chamberlin and R. F. Boyce. 1974. SEQUEL: A structured English query language. In *Proc. of the 1974 ACM SIGFIDET (now SIGMOD) Workshop on Data Description,* *Access and Control (SIGFIDET '74)*, pp. 249–264. ACM, New York. DOI: 10.1145/800296.811515. 404, 407

D. D. Chamberlin, M. M. Astrahan, K. P. Eswaran, P. P. Griffiths, R. A. Lorie, J. W. Mehl, P. Reisner, and B. W. Wade. 1976. SEQUEL 2: a unified approach to data definition, manipulation, and control. *IBM Journal of Research and Development*, 20(6): 560–575. DOI: 10.1147/rd.206.0560. 398

S. Chandrasekaran, O, Cooper, A. Deshpande, M.J. Franklin, J.M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. 2003. TelegraphCQ: Continuous dataflow processing for an uncertain world. *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*, pp. 668–668. ACM, New York. DOI :10.1145/872757.872857. 231

J. Chen, D.J. DeWitt, F. Tian, and Y. Wang. 2000. NiagaraCQ: A scalable continuous query system for Internet databases. *Proc. of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*, pp. 379–390. ACM, New York. DOI 10.1145/342009.335432. 231

M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. 2003. Scalable distributed stream processing. *Proc. of the First Biennial Conference on Innovative Database Systems* (*CIDR*'03), Asilomar, CA, January. 228

C. M. Christensen. 1997. *The Innovator's Dilemma: When New Technologies Cause Great Firms to Fail*. Harvard Business School Press, Boston, MA. 100

X. Chu, I. F. Ilyas, and P. Papotti. 2013a. Holistic data cleaning: Putting violations into context. *Proc. of the IEEE International Conference on Data Engineering*, *ICDE 2013*, pp. 458–469. Australia. DOI: 10.1109/ICDE.2013.6544847 270, 297

X. Chu, I. F. Ilyas, and P. Papotti. 2013b. Discovering denial constraints. *Proc. of the VLDB Endowment*, *PVLDB* 6(13): 1498–1509. DOI: 10.14778/2536258.2536262. 270

X. Chu, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, N. Tang, and Y. Ye. 2015. Katara: A data cleaning system powered by knowledge bases and crowdsourcing. In *Proc. of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*, pp. 1247–1261. ACM, New York. DOI: 10.1145/2723372.2749431. 297

P. J. A. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice. 2009. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Research* 38.6: 1767–1771. DOI: 10.1093/nar/gkp1137. 374

E. F. Codd. June 1970. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6): 377–387. DOI: 10.1145/362384.362685. 42, 98, 166, 397, 404, 405, 407

M. Collins. 2016. Thomson Reuters uses Tamr to deliver better connected content at a fraction of the time and cost of legacy approaches. Tamr blog, July 28. https://www.tamr.com/video/thomson-reuters-uses-tamr-deliver-better-connected-content-fraction-time-cost-legacy-approaches/. Last accessed January 24, 2018. 275

G. Copeland and D. Maier. 1984. Making smalltalk a database system. *Proc. of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD '84)*, pp. 316–325. ACM, New York. DOI: 10.1145/602259.602300. 111

C. Cranor, T. Johnson, V. Shkapenyuk, and O. Spatscheck. 2003. Gigascope: A stream database for network applications. *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*, pp. 647–651. ACM, New York. DOI :10.1145/872757.872838. 231

A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Cetintemel, and S. Zdonik. 2015. Tupleware: "Big Data, Big Analytics, Small Clusters." *CIDR*. DOI: 10.1.1.696.32. 374

M. Dallachiesa, A. Ebaid, A. Eldawi, A. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. 2013. NADEEF, a commodity data cleaning system. *Proc. of the 2013 ACM SIGMOD Conference on Management of Data*, pp. 541–552. New York. http://dx.doi.org/10.1145/2463676.2465327. 270, 297

T. Dasu and J. M. Loh. 2012. Statistical distortion: Consequences of data cleaning. *PVLDB*, 5(11): 1674–1683. DOI: 10.14778/2350229.2350279. 297

C. J. Date and E. F. Codd. 1975. The relational and network approaches: Comparison of the application programming interfaces. In *Proc. of the 1974 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control: Data Models: Data-Structure-Set Versus Relational (SIGFIDET '74)*, pp. 83–113. ACM, New York. DOI: 10.1145/800297.811534. 405

D. J. DeWitt. 1979a. Direct a multiprocessor organization for supporting relational database management systems. *IEEE Transactions of Computers,* 28(6), 395–406. DOI: 10.1109/TC.1979.1675379. 109

D. J. DeWitt. 1979b. Query execution in DIRECT. In *Proc. of the 1979 ACM SIGMOD International Conference on Management of Data (SIGMOD '79)*, pp. 13–22. ACM, New York. DOI: 10.1145/582095.582098. 109

D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. 1986. GAMMA—a high performance dataflow database machine. *Proc. of the 12th International Conference on Very Large Data Bases (VLDB '86)*, W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, editors, pp. 228–237. Morgan Kaufmann Publishers Inc., San Francisco, CA. 111

D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. March 1990. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering,* 2(1): 44–62. DOI: 10.1109/69.50905. 151, 400

D. DeWitt and J. Gray. June 1992. Parallel database systems: the future of high performance database systems. *Communications of the ACM,* 35(6): 85–98. DOI: 10.1145/129888.129894. 199

D. J. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. 2013. Split query processing in polybase. *Proc. of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*, pp. 1255–1266. ACM, New York. 284

C. Diaconu, C. Freedman, E. Ismert, P-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proc. of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*, pp. 1243–1254. ACM, New York. DOI: 10.1145/2463676.2463710.

K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. November 1976. The notions of consistency and predicate locks in a database system. *Communications of the ACM,* 19(11): 624–633. DOI: 10.1145/360363.360369. 114

W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. April 2012. Towards certain fixes with editing rules and master data. *The VLDB Journal*, 21(2): 213–238. DOI: 10.1007/s00778-011-0253-7. 297

D. Fogg. September 1982. Implementation of domain abstraction in the relational database system INGRES. Master of Science Report, Dept. of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA. 201

T. Flory, A. Robbin, and M. David. May 1988. Creating SIPP longitudinal analysis files using a relational database management system. CDE Working Paper No. 88-32, Institute for Research on Poverty, University of Wisconsin-Madison, Madison, WI. 197

V. Gadepally, J. Kepner, W. Arcand, D. Bestor, B. Bergeron, C. Byun, L. Edwards, M. Hubbell, P. Michaleas, J. Mullen, A. Prout, A. Rosa, C. Yee, and A. Reuther. 2015. D4M: Bringing associative arrays to database engines. *High Performance Extreme Computing Conference (HPEC)*. IEEE, 2015. DOI: 10.1109/HPEC.2015.7322472. 370

V. Gadepally, K. O'Brien, A. Dziedzic, A. Elmore, J. Kepner, S. Madden, T. Mattson, J. Rogers, Z. She, and M. Stonebraker. September 2017. BigDAWG Version 0.1. *IEEE High Performance Extreme*. DOI: 10.1109/HPEC.2017.8091077. 288, 369

J. Gantz and D. Reinsel. 2013. The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East—United States, IDC, February. 5

L. Gerhardt, C. H. Faham, and Y. Yao. 2015. Accelerating scientific analysis with SciDB. *Journal of Physics: Conference Series*, 664(7). 268

B. Grad. 2007. Oral history of Michael Stonebraker, Transcription. Recorded: August 23, 2007. Computer History Museum, Moultonborough, NH. http://archive.computerhistory.org/resources/access/text/2012/12/102635858-05-01-acc.pdf. Last accessed April 8, 2018. 42, 43, 44, 98

A. Guttman. 1984. R-trees: a dynamic index structure for spatial searching. In *Proc. of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD '84)*, pp. 47–57. ACM, New York. DOI: 10.1145/602259.602266. 205

L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. 1989. Extensible query processing in starburst. In *Proc. of the 1989 ACM SIGMOD International Conference on Management of Data (SIGMOD '89)*, pp. 377–388. ACM, New York. DOI: 10.1145/67544.66962. 399

D. Halperin, V. Teixeira de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker. 2014. Demonstration of the Myria big data management service. *Proc. of the 2014 ACM SIGMOD International Conference* *on Management of Data (SIGMOD '14)*, p. 881–884. ACM, New York. DOI: 10.1145/2588555.2594530. 284, 370

B. Haynes, A. Cheung, and M. Balazinska. 2016. PipeGen: Data pipe generator for hybrid analytics. *Proc. of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*, M. K. Aguilera, B. Cooper, and Y. Diao, editors, pp. 470–483. ACM, New York. DOI: 10.1145/2987550.2987567. 287

M. A. Hearst. 2009. *Search user interfaces*. Cambridge University Press, New York. 394

J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. 1995. Generalized search trees for database systems. In *Proc. of the 21th International Conference on Very Large Data Bases (VLDB '95)*, pp. 562–573. Morgan Kaufmann Publishers Inc., San Francisco, CA. http://dl.acm.org/citation.cfm?id=645921.673145. 210

J. M. Hellerstein, E. Koutsoupias, D. P. Miranker, C. H. Papadimitriou, V. Samoladas. 2002. On a model of indexability and its bounds for range queries, *Journal of the ACM (JACM)*, 49.1: 35–55. DOI: 10.1145/505241.505244. 210

IBM. 1997. Special Issue on IBM's S/390 Parallel Sysplex Cluster. *IBM Systems Journal*, 36(2). 400

S. Idreos, F. Groffen, N. Nes, S. Manegold, S. K. Mullender, and M. L. Kersten. 2012. MonetDB: two decades of research in column-oriented database architectures. *IEEE Data Engineering Bulletin*, 35(1): 40–45. 258

N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. Zdonik. 2008. Towards a streaming SQL standard. *Proc. VLDB Endowment*, pp. 1379–1390. August 1–2. DOI: 10.14778/1454159.1454179. 229

A. E. W. Johnson, T. J. Pollard, L. Shen, L. H. Lehman, M. Feng, M. Ghassemi, B. E. Moody, P. Szolovits, L. A. G. Celi, and R. G. Mark. 2016. MIMIC-III, a freely accessible critical care database. *Scientific Data* 3: 160035 DOI: 10.1038/sdata.2016.35. 370

V. Josifovski, P. Schwarz, L. Haas, and E. Lin. 2002. Garlic: a new flavor of federated query processing for DB2. In *Proc. of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02)*, pp. 524–532. ACM, New York. DOI: 10.1145/564691.564751. 401

J. W. Josten, C. Mohan, I. Narang, and J. Z. Teng. 1997. DB2's use of the coupling facility for data sharing. *IBM Systems Journal*, 36(2): 327–351. DOI: 10.1147/sj.362.0327. 400

S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. 2011. Wrangler: Interactive visual specification of data transformation scripts. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)*, pp. 3363–3372. ACM, New York. DOI: 10.1145/1978942.1979444. 297

R. Katz. editor. June 1982. Special issue on design data management. *IEEE Database Engineering Newsletter*, 5(2). 200

J. Kepner, V. Gadepally, D. Hutchison, H. Jensen, T. Mattson, S. Samsi, and A. Reuther. 2016. Associative array model of SQL, NoSQL, and NewSQL Databases. *IEEE High*

Performance Extreme Computing Conference (HPEC) 2016, Waltham, MA, September 13–15. DOI: 10.1109/HPEC.2016.7761647. 289

V. Kevin and M. Whitney. 1974. Relational data management implementation techniques. In *Proc. of the 1974 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control (SIGFIDET '74)*, pp. 321–350. ACM, New York. DOI: 10.1145/800296.811519 404

Z. Khayyat, I.F. Ilyas, A. Jindal, S. Madden, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, and S. Yin. 2015. Bigdansing: A system for big data cleansing. In *Proc. of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15),* pp. 1215–1230. ACM, New York. DOI: 10.1145/2723372.2747646. 297

R. Kimball and M. Ross. 2013. *The Data Warehouse Toolkit*. John Wiley & Sons, Inc. https://www.kimballgroup.com/data-warehouse-business-intelligence-resources/books/. Last accessed March 2, 2018. 337

M. Kornacker, C. Mohan, and J.M. Hellerstein. 1997. Concurrency and recovery in generalized search trees. In *Proc. of the 1997 ACM SIGMOD International Conference on Management of Data (SIGMOD '97)*, pp. 62–72. ACM, New York. DOI: 10.1145/253260.253272. 210

A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. August 2012. The Vertica Analytic Database: C-Store 7 years later. *Proc. VLDB Endowment*, 5(12): 1790–1801. DOI: 10.14778/2367502.2367518. 333, 336

L. Lamport. 2001. Paxos Made Simple. http://lamport.azurewebsites.net/pubs/paxos-simple.pdf. Last accessed December 31, 2017. 258

D. Laney. 2001. 3D data management: controlling data volume, variety and velocity. META Group Research, February 6. https://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf. Last accessed April 22, 2018. 357

P-A. Larson, C. Clinciu, E.N. Hanson, A. Oks, S.L. Price, S. Rangarajan, A. Surna, and Q. Zhou. 2011. SQL server column store indexes. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*, pp. 1177–1184. ACM, New York. DOI: 10.1145/1989323.1989448.

J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis, and M. J. Carey. 2014. MISO: Souping up big data query processing with a multistore system. *Proc. of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*, pp. 1591–1602. ACM, New York. DOI: 10.1145/2588555.2588568. 284

B. G. Lindsay. 1987. A retrospective of R*: a distributed database management system. In *Proc. of the IEEE*,75(5): 668–673. DOI: 10.1109/PROC.1987.13780. 400

B. Liskov and S.N. Zilles. 1974. Programming with abstract data types. *SIGPLAN Notices,* 9(4): 50–59. DOI: 10.1145/942572.807045. 88

S. Marcin and A. Csillaghy. 2016. Running scientific algorithms as array database operators: Bringing the processing power to the data. *2016 IEEE International Conference on Big Data*. pp. 3187–3193. DOI: 10.1109/BigData.2016.7840974. 350

T. Mattson, V. Gadepally, Z. She, A. Dziedzic, and J. Parkhurst. 2017. Demonstrating the BigDAWG polystore system for ocean metagenomic analysis. *CIDR'17* Chaminade, CA. http://cidrdb.org/cidr2017/papers/p120-mattson-cidr17.pdf. 288, 374

J. Meehan, C. Aslantas, S. Zdonik, N. Tatbul, and J. Du. 2017. Data ingestion for the connected world. *Conference on Innovative Data Systems Research (CIDR'17)*, Chaminade, CA, January. 376

A. Metaxides, W. B. Helgeson, R. E. Seth, G. C. Bryson, M. A. Coane, D. G. Dodd, C. P. Earnest, R. W. Engles, L. N. Harper, P. A. Hartley, D. J. Hopkin, J. D. Joyce, S. C. Knapp, J. R. Lucking, J. M. Muro, M. P. Persily, M. A. Ramm, J. F. Russell, R. F. Schubert, J. R. Sidlo, M. M. Smith, and G. T. Werner. April 1971. *Data Base Task Group Report to the CODASYL Programming Language Committee*. ACM, New York. 43

C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. 1992. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1), 94–162. DOI: 10.1145/128765.128770. 402

R. Motwani, J. Widom, A. Arasu B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. 2003. Query processing, approximation, and resource management in a data stream management system. *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, January. 229, 231

A. Oloso, K-S Kuo, T. Clune, P. Brown, A. Poliakov, H. Yu. 2016. Implementing connected component labeling as a user defined operator for SciDB. *Proc. of 2016 IEEE International Conference on Big Data (Big Data)*. Washington, DC. DOI: 10.1109/BigData.2016.7840945. 263, 350

M. A. Olson. 1993. The design and implementation of the inversion file system. *USENIX* Winter. http://www.usenix.org/conference/usenix-winter-1993-conference/presentation/design-and-implementation-inversion-file-syste. Last accessed January 22, 2018. 215

J. C. Ong. 1982. Implementation of abstract data types in the relational database system INGRES, Master of Science Report, Dept. of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, September 1982. 201

A. Palmer. 2013. Culture matters: Facebook CIO talks about how well Vertica, Facebook people mesh. *Koa Labs Blog*, December 20. http://koablog.wordpress.com/2013/12/20/culture-matters-facebook-cio-talks-about-how-well-vertica-facebook-people-mesh. Last accessed March 14, 2018. 132, 133

A. Palmer. 2015a. The simple truth: happy people, healthy company. *Tamr Blog*, March 23. http://www.tamr.com/the-simple-truth-happy-people-healthy-company/. Last accessed March 14, 2018. 138

A. Palmer. 2015b. Where the red book meets the unicorn, *Xconomy*, June 22. http://www.xconomy.com/boston/2015/06/22/where-the-red-book-meets-the-unicorn/ Last accessed March 14, 2018. 130

A. Pavlo and M. Aslett. September 2016. What's really new with NewSQL? *ACM SIGMOD Record*, 45(2): 45–55. DOI: DOI: 10.1145/3003665.3003674. 246

G. Press. 2016. Cleaning big data: most time-consuming, least enjoyable data science task, survey says. *Forbes*, May 23. https://www.forbes.com/sites/gilpress/2016/03/23/data-preparation-most-time-consuming-least-enjoyable-data-science-task-survey-says/#79e14e326f63. 357

N. Prokoshyna, J. Szlichta, F. Chiang, R. J. Miller, and D. Srivastava. 2015. Combining quantitative and logical data cleaning. *PVLDB*, 9(4): 300–311. DOI: 10.14778/2856318.2856325. 297

E. Ryvkina, A. S. Maskey, M. Cherniack, and S. Zdonik. 2006. Revision processing in a stream processing engine: a high-level design. *Proc. of the 22nd International Conference on Data Engineering* (*ICDE*'06), pp. 141–. Atlanta, GA, April. IEEE Computer Society, Washington, DC. DOI: 10.1109/ICDE.2006.130. 228

C. Saracco and D. Haderle. 2013. The history and growth of IBM's DB2. *IEEE Annals of the History of Computing*, 35(2): 54–66. DOI: 10.1109/MAHC.2012.55. 398

N. Savage. May 2015. Forging relationships. *Communications of the ACM*, 58(6): 22–23. DOI: 10.1145/2754956.

M. C. Schatz and B. Langmead. 2013. The DNA data deluge. *IEEE Spectrum Magazine*. https://spectrum.ieee.org/biomedical/devices/the-dna-data-deluge. 354

Z. She, S. Ravishankar, and J. Duggan. 2016. BigDAWG polystore query optimization through semantic equivalences. *High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016. DOI: :10.1109/HPEC.2016.7761584. 373

SIGFIDET panel discussion. 1974. In *Proc. of the 1974 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control: Data Models: Data-Structure-Set Versus Relational (SIGFIDET '74)*, pp. 121–144. ACM, New York. DOI: 10.1145/800297.811534. 404

R. Snodgrass. December 1982. Monitoring distributed systems: a relational approach. Ph.D. Dissertation, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 197

A. Szalay. June 2008. The Sloan digital sky survey and beyond. *ACM SIGMOD Record*, 37(2): 61–66. 255

Tamr. 2017. Tamr awarded patent for enterprise-scale data unification system. Tamr blog. February 9 2017. https://www.tamr.com/tamr-awarded-patent-enterprise-scale-data-unification-system-2/. Last accessed January 24, 2018. 275

R. Tan, R. Chirkova, V. Gadepally, and T. Mattson. 2017. Enabling query processing across heterogeneous data models: A survey. *IEEE Big Data Workshop: Methods to Manage Heterogeneous Big Data and Polystore Databases*, Boston, MA. DOI: 10.1109/BigData.2017.8258302. 284, 376

N. Tatbul and S. Zdonik. 2006. Window-aware Load Shedding for Aggregation Queries over Data Streams. In *Proc. of the 32nd International Conference on Very Large Databases (VLDB'06)*, Seoul, Korea. 228, 229

N. Tatbul, U. Çetintemel, and S. Zdonik. 2007. “Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing." *International Conference on Very Large Data Bases (VLDB’07)*, Vienna, Austria. 228, 229

R. P. van de Riet. 1986. Expert database systems. In *Future Generation Computer Systems*, 2(3): 191–199, DOI: 10.1016/0167-739X(86)90015-4. 407

M. Vartak, S. Rahman, S. Madden, A. Parameswaran, and N. Polyzotis. September 2015. Seedb: Efficient data-driven visualization recommendations to support visual analytics. *PVLDB*, 8(13): 2182–2193. DOI: 10.14778/2831360.2831371. 297

B. Wallace. June 9, 1986. Data base tool links to remote sites. *Network World*. http://books.google.com/books?id=aBwEAAAAMBAJ&pg=PA49&lpg=PA49&dq=ingres+star&source=bl&ots=FSMIR4thMj&sig=S1fzaaOT5CHRq4cwbLFEQp4UYCs&hl=en&sa=X&ved=0ahUKEwjJ1J_NttvZAhUG82MKHco2CfAQ6AEIYzAP#v=onepage&q=ingres%20star&f=false. Last accessed March 14, 2018.305

J. Wang and N. J. Tang. 2014. Towards dependable data repairing with fixing rules. In *Proc. of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD ’14*), pp. 457–468. ACM, New York. DOI: 10.1145/2588555.2610494. 297

E. Wong and K. Youssefi. September 1976. Decomposition—a strategy for query processing. *ACM Transactions on Database Systems*, 1(3): 223–241. DOI: 10.1145/320473.320479. 196

E. Wu and S. Madden. 2013. Scorpion: Explaining away outliers in aggregate queries. *PVLDB*, 6(8): 553–564. DOI: 10.14778/2536354.2536356. 297

Y. Xing, S. Zdonik, and J.-H. Hwang. April 2005. Dynamic load distribution in the Borealis Stream Processor. *Proc. of the 21st International Conference on Data Engineering* (*ICDE*’05), Tokyo, Japan. DOI: 10.1109/ICDE.2005.53. 228, 230, 325

Index

Page numbers with ‘f’ indicate figures; page numbers with ‘n’ indicate footnotes.

Biographies

Editor

Michael L. Brodie

Michael L. Brodie has over 45 years of experience in research and industrial practice in databases, distributed systems, integration, artificial intelligence, and multidisciplinary problem-solving. Dr. Brodie is a research scientist at the Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology; advises startups; serves on advisory boards of national and international research organizations; and is an adjunct professor at the National University of Ireland, Galway and at the University of Technology, Sydney. As Chief Scientist of IT at Verizon for over 20 years, he was responsible for advanced technologies, architectures, and methodologies for IT strategies and for guiding industrial-scale deployments of emerging technologies. He has served on several National Academy of Science committees. Current interests include Big Data, Data Science, and Information Systems evolution. Dr. Brodie holds a Ph.D. in databases from the University of Toronto and a Doctor of Science (*honoris causa*) from the National University of Ireland. Visit www.Michaelbrodie.com for further information.

Authors

Daniel J. Abadi

Daniel J. Abadi is the Darnell-Kanal Professor of Computer Science at the University of Maryland, College Park. He performs research on database system architecture and implementation, especially at the intersection of scalable and distributed systems. He is best known for the development of the storage and query execution engines of the C-Store (column-oriented database) prototype, which was commercialized by Vertica and eventually acquired by Hewlett-Packard, and for his HadoopDB research on fault-tolerant scalable analytical database systems, which was commercialized by Hadapt and acquired by Teradata in 2014. Abadi has been a recipient of a Churchill Scholarship, a NSF CAREER Award, a Sloan Research Fellowship, a VLDB Best Paper Award, a VLDB 10-year Best Paper Award, the 2008 SIGMOD Jim Gray Doctoral Dissertation Award, the 2013–2014 Yale Provost's Teaching Prize, and the 2013 VLDB Early Career Researcher Award. He received his Ph.D. in 2008 from MIT. He blogs at DBMS Musings (http://dbmsmusings.blogspot.com) and Tweets at @daniel_abadi.

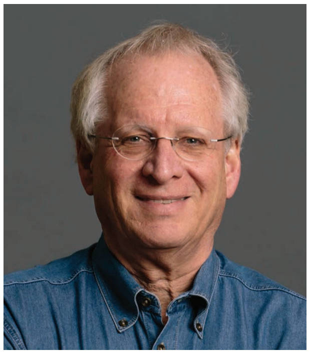
Magdalena Balazinska

Magdalena Balazinska is a professor in the Paul G. Allen School of Computer Science and Engineering at the University of Washington and is the director of the University's eScience Institute. She's also director of the IGERT PhD Program in Big Data and Data Science and the associated Advanced Data Science PhD Option. Her research interests are in database management systems with a current focus on data management for data science, big data systems, and cloud computing. Magdalena holds a Ph.D. from the Massachusetts Institute of Technology (2006). She is a Microsoft Research New Faculty Fellow (2007) and received the inaugural VLDB Women in Database Research Award (2016), an ACM SIGMOD Test-of-Time Award (2017), an NSF CAREER Award (2009), a 10-year most influential paper award (2010), a Google Research Award (2011), an HP Labs Research Innovation Award (2009 and 2010), a Rogel Faculty Support Award (2006), a Microsoft Research Graduate Fellowship (2003–2005), and multiple best-paper awards.

Nikolaus Bates-Haus

Nikolaus Bates-Haus is Technical Lead at Tamr Inc., an enterprise-scale data unification company, where he assembled the original engineering team and led the development of the first generation of the product. Prior to joining Tamr, Nik was Lead Architect and Director of Engineering at Endeca (acquired by Oracle in 2011), where he led development of the MDEX analytical database engine, a schema-on-read column store designed for large-scale parallel query evaluation. Previously, Nik worked in data integration, machine learning, parallel computation, and real-time processing at Torrent Systems, Thinking Machines, and Philips Research North America. Nik holds an M.S. in Computer Science from Columbia University and a B.A. in Mathematics/Computer Science from Wesleyan University. Tamr is Nik's seventh startup.

Philip A. Bernstein



Philip A. Bernstein is a Distinguished Scientist at Microsoft Research, where he has worked for over 20 years. He is also an Affiliate Professor of Computer Science at the University of Washington. Over the last 20 years, he has been a product architect at Microsoft and Digital Equipment Corp., a professor at Harvard University and Wang Institute of Graduate Studies, and a VP Software at Sequoia Systems. He has published over 150 papers and 2 books on the theory and implementation of database systems, especially on transaction processing and data integration, and has contributed to a variety of database products. He is an ACM Fellow, an AAAS Fellow, a winner of ACM SIGMOD's Codd Innovations Award, a member of the Washington State Academy of Sciences, and a member of the U.S. National Academy of Engineering. He received a B.S. from Cornell and M.Sc. and Ph.D. degrees from the University of Toronto.

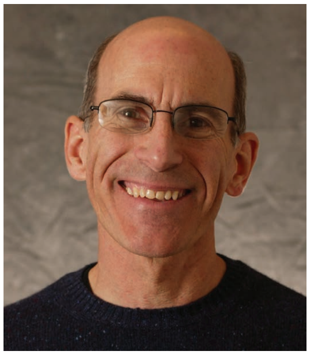
Janice L. Brown

Janice L. Brown is president and founder of Janice Brown & Associates, Inc., a communications consulting firm. She uses strategic communications to help entrepreneurs and visionary thinkers launch technology companies, products, and ventures, as well as sell their products and ideas. She has been involved in three ventures (so far) with 2014 Turing Award-winner Michael Stonebraker: Vertica Systems, Tamr, and the Intel Science and Technology Center for Big Data. Her background includes positions at several public relations and advertising agencies, and product PR positions at two large technology companies. Her work for the Open Software Foundation won the PRSA's Silver Anvil Award, the "Oscar" of the PR industry. Brown has a B.A. from Simmons College. Visit www.janicebrown.com.

Paul Brown

Paul Brown first met Mike Stonebraker in early 1992 at Brewed Awakening coffee shop on Euclid Avenue in Berkeley, CA. Mike and John Forrest were interviewing Paul to take over the job Mike Olson had just left. Paul had a latte. Mike had tea. Since then, Paul has worked for two of Mike's startups: Illustra Information Technologies and SciDB / Paradigm4. He was co-author with Mike of a book and a number of research papers. Paul has worked for a series of DBMS companies all starting with the letter "I": Ingres, Illustra, Informix, and IBM. Alliterative ennui setting in, Paul joined Paradigm4 as SciDB's Chief Architect. He has since moved on to work for Teradata. Paul likes dogs, DBMSs, and (void *). He hopes he might have just picked up sufficient gravitas in this industry to pull off the beard.

Paul Butterworth



Paul Butterworth served as Chief Systems Architect at Ingres from 1980–1990. He is currently co-founder and Chief Technology Officer (CTO) at VANTIQ, Inc. His past roles include Executive Vice President, Engineering at You Technology Inc., and co-founder and CTO of Emotive Communications, where he conceived and designed the Emotive Cloud Platform for enterprise mobile computing. Before that, Paul was an architect at Oracle and a founder & CTO at AmberPoint, where he directed the technical strategy for the AmberPoint SOA governance products. Prior to AmberPoint, Paul was a Distinguished Engineer and Chief Technologist for the Developer Tools Group at Sun Microsystems and a founder, Chief Architect, and Senior Vice President of Forte Software. Paul holds undergraduate and graduate degrees in Computer Science from UC Irvine.

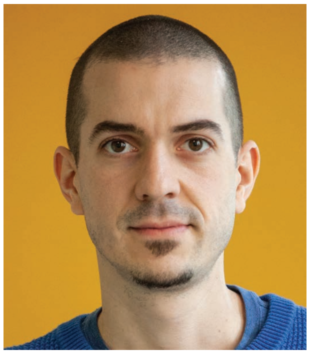
Michael J. Carey

Michael J. Carey received his B.S. and M.S. from Carnegie-Mellon University and his Ph.D. from the University of California, Berkeley, in 1979, 1981, and 1983, respectively. He is currently a Bren Professor of Information and Computer Sciences at the University of California, Irvine (UCI) and a consulting architect at Couchbase, Inc. Before joining UCI in 2008, Mike worked at BEA Systems for seven years and led the development of BEA's AquaLogic Data Services Platform product for virtual data integration. He also spent a dozen years teaching at the University of Wisconsin-Madison, five years at the IBM Almaden Research Center working on object-relational databases, and a year and a half at Propel Software, an e-commerce platform startup, during the infamous 2000–2001 Internet bubble. He is an ACM Fellow, an IEEE Fellow, a member of the National Academy of Engineering, and a recipient of the ACM SIGMOD E.F. Codd Innovations Award. His current interests center on data-intensive computing and scalable data management (a.k.a. Big Data).

Fred Carter

Fred Carter , a software architect in a variety of software areas, worked at Ingres Corporation in several senior positions, including Principal Scientist/Chief Architect. He is currently a principal architect at VANTIQ, Inc. Prior to VANTIQ, Fred was the runtime architect for AmberPoint, which was subsequently purchased by Oracle. At Oracle, he continued in that role, moving the AmberPoint system to a cloud-based, application performance monitoring service. Past roles included architect for EAI products at Forte (continuing at Sun Microsystems) and technical leadership positions at Oracle, where he designed distributed object services for interactive TV, online services, and content management, and chaired the Technical Committee for the Object Definition Alliance to foster standardization in the area of network-based multimedia systems. Fred has an undergraduate degree in Computer Science from Northwestern University and received his M.S. in Computer Science from UC Berkeley.

Raul Castro Fernandez



Raul Castro Fernandez is a postdoc at MIT, working with Samuel Madden and Michael Stonebraker on data discovery—how to help people find relevant data among databases, data lakes, and the cloud. Raul built Aurum, a data discovery system, to identify relevant data sets among structured data. Among other research lines, he is looking at how to incorporate unstructured data sources, such as PDFs and emails. More generally, he is interested in data-related problems, from efficient data processing to machine learning engineering. Before MIT, Raul completed his Ph.D. at Imperial College London, where he focused on designing new abstractions and building systems for large-scale data processing.

Ugur Çetintemel

Ugur Çetintemel is a professor in the department of Computer Science at Brown University. His research is on the design and engineering of high-performance, user-friendly data management and processing systems that allow users to analyze large data sets interactively. Ugur chaired SIGMOD '09 and served on the editorial boards of *VLDB Journal*, *Distributed and Parallel Databases*, and *SIGMOD Record*. He is the recipient of a National Science Foundation Career Award and an IEEE 10-year test of time award in Data Engineering, among others. Ugur was a co-founder and a senior architect of StreamBase, a company that specializes in high-performance data processing. He was also a Brown Manning Assistant Professor and has been serving as the Chair of the Computer Science Department at Brown since July 2014.

Xuedong Chen

Xuedong Chen is currently an Amazon.com Web Services software developer in Andover, Massachusetts. From 2002–2007 he was a Ph.D. candidate at UMass Boston, advised by Pat and Betty O'Neil. He, along with Pat O'Neil and others, were co-authors with Mike Stonebraker.

Mitch Cherniack

Mitch Cherniack is an Associate Professor at Brandeis University. He is a previous winner of an NSF Career Award and co-founder of Vertica Systems and StreamBase Systems. His research in Database Systems has focused on query optimization, streaming data systems, and column-based database architectures. Mitch received his Ph.D. from Brown University in 1999, an M.S. from Concordia University in 1992, and a B.Ed. from McGill University in 1984.

David J. DeWitt

David J. DeWitt joined the Computer Sciences Department at the University of Wisconsin in September 1976 after receiving his Ph.D. from the University of Michigan. He served as department chair from July 1999 to July 2004. He held the title of John P. Morgridge Professor of Computer Sciences when he retired from the University of Wisconsin in 2008. In 2008, he joined Microsoft as a Technical Fellow to establish and manage the Jim Gray Systems Lab in Madison. In 2016, he moved to Boston to join the MIT Computer Science and AI Laboratory as an Adjunct Professor. Professor DeWitt is a member of the National Academy of Engineering (1998), a fellow of the American Academy of Arts and Sciences (2007), and an ACM Fellow (1995). He received the 1995 Ted Codd SIGMOD Innovations Award. His pioneering contributions to the field of scalable database systems for "big data" were recognized by ACM with the 2009 Software Systems Award.

Aaron J. Elmore

Aaron J. Elmore is an assistant professor in the Department of Computer Science and the College of the University of Chicago. Aaron was previously a postdoctoral associate at MIT working with Mike Stonebraker and Sam Madden. Aaron's thesis on *Elasticity Primitives for Database-as-a-Service* was completed at the University of California, Santa Barbara under the supervision of Divy Agrawal and Amr El Abbadi. Prior to receiving a Ph.D., Aaron spent several years in industry and completed an M.S. at the University of Chicago.

Miguel Ferreira

Miguel Ferreira is an alumnus of MIT. He was coauthor of the paper, "Integrating Compression and Execution in Column-Oriented Database Systems," while working with Samuel Madden and Daniel Abadi, and "C-store: A Column-Oriented DBMS," with Mike Stonebraker, Daniel Abadi, and others.

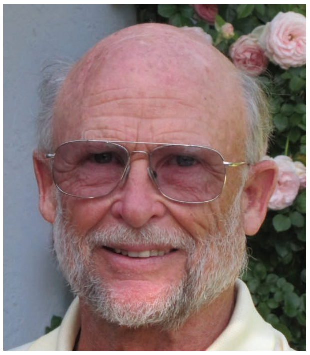
Vijay Gadepally

Vijay Gadepally is a senior member of the technical staff at the Massachusetts Institute of Technology (MIT) Lincoln Laboratory and works closely with the Computer Science and Artificial Intelligence Laboratory (CSAIL). Vijay holds an M.Sc. and Ph.D. in Electrical and Computer Engineering from The Ohio State University and a B.Tech in Electrical Engineering from the Indian Institute of Technology, Kanpur. In 2011, Vijay received an Outstanding Graduate Student Award at The Ohio State University. In 2016, Vijay received the MIT Lincoln Laboratory's Early Career Technical Achievement Award and in 2017 was named to AFCEA's inaugural 40 under 40 list. Vijay's research interests are in high-performance computing, machine learning, graph algorithms, and high-performance databases.

Nabil Hachem

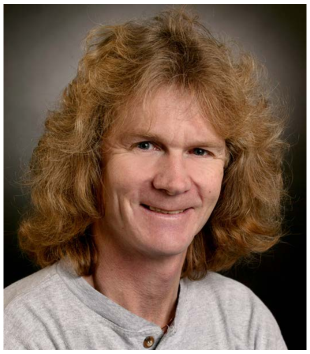
Nabil Hachem is currently Vice President, Head of Data Architecture, Technology, and Standards at MassMutual. He was formerly Global Head of Data Engineering at Novartis Institute for Biomedical Research, Inc. He also held senior data engineering posts at Vertica Systems, Inc., Infinity Pharmaceuticals, Upromise Inc., Fidelity Investments Corp., and Ask Jeeves Inc. Nabil began his career as an electrical engineer and operations department manager for a data telecommunications firm in Lebanon. In addition to his commercial career, Nabil taught computer science at Worcester Polytechnic Institute. He co-authored dozens of papers on scientific databases, file structures, and join algorithms, among others. Nabil received a degree in Electrical Engineering from the American University of Beirut and earned his Ph.D. in Computer Engineering from Syracuse University.

Don Haderle



Don Haderle joined IBM in 1968 as a software developer and retired in 2005 as the software executive operating as Chief Technology Officer (CTO) for Information Management. He consulted with venture capitalists and advised startups. He currently sits on technical advisory boards for a number of companies and consults independently. Considered the father of commercial high-performance, industrial-strength relational database systems, he was the technical leader and chief architect of DB2 from 1977–1998. He led DB2's overall architecture and development, making key personal contributions to and holding fundamental patents in all key elements, including: logging primitives, memory management, transaction fail-save and recovery techniques, query processing, data integrity, sorting, and indexing. As CTO, Haderle collaborated with researchers to incubate new product directions for the information management industry. Don was appointed an IBM Fellow in 1989 and Vice President of Advanced Technology in 1991; named an ACM Fellow in 2000; and elected to the National Academy of Engineering in 2008. He is a graduate of UC Berkeley (B.A., Economics, 1967).

James Hamilton



James Hamilton is Vice President and Distinguished Engineer on the Amazon Web Services team, where he focuses on infrastructure efficiency, reliability, and scaling. He has spent more than 20 years working on high-scale services, database management systems, and compilers. Prior to joining AWS, James was architect on the Microsoft Data Center Futures team and the Windows Live Platform Services team. He was General Manager of the Microsoft Exchange Hosted Services team and has led many of the SQL Server engineering teams through numerous releases. Before joining Microsoft, James was Lead Architect on the IBM DB2 UDB team. He holds a B.Sc. inComputer Science from the University of Victoria and a Master's in Math, Computer Science from the University of Waterloo.

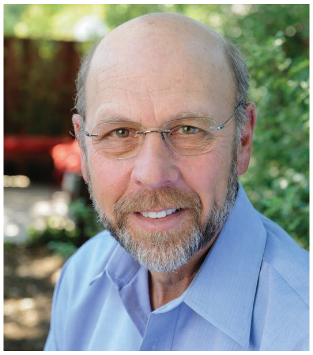
Stavros Harizopoulos

Stavros Harizopoulos is currently a Software Engineer at Facebook, where he leads initiatives on Realtime Analytics. Before that, he was a Principal Engineer at AWS Redshift, a petabyte-scale columnar Data Warehouse in the cloud, where he was leading efforts on performance and scalability. In 2011, he co-founded Amiato, a fully managed real-time ETL cloud service, which was later acquired by Amazon. In the past, Stavros has held research-scientist positions at HP Labs and MIT CSAIL, working on characterizing the energy efficiency of database servers, as well as dissecting the performance characteristics of modern in-memory and column-store databases. He is a Carnegie Mellon Ph.D. and a Y Combinator alumnus.

Marti Hearst

Marti Hearst is a professor in the School of Information and the EECS Department at UC Berkeley. She was formerly a member of the research staff at Xerox PARC and received her Ph.D. from the CS Division at UC Berkeley. Her primary research interests are user interfaces for search engines, information visualization, natural language processing, and improving education. Her book *Search User Interfaces* was the first of its kind in academics. Prof. Hearst was named a Fellow of the ACM in 2013 and a member of the CHI Academy in 2017, and is president of the Association for Computational Linguistics. She has received four student-initiated Excellence in Teaching Awards.

Jerry Held



Jerry Held has been a successful Silicon Valley entrepreneur, executive, and investor for over 40 years. He has managed all growth stages of companies, from conception to multi-billion-dollar global enterprise. He is currently chairman of Tamr and Madaket Health and serves on the boards of NetApp, Informatica, and Copia Global. His past board service includes roles as executive chairman of Vertica Systems and MemSQL and lead independent director of Business Objects. Previously, Dr. Held was "CEO-in-residence" at venture capital firm Kleiner Perkins Caufield & Byers. He was senior vice president of Oracle Corporation's server product division and a member of the executive team that grew Tandem Computers from pre-revenue to multi-billion-dollar company. Among many other roles, he led pioneering work in fault-tolerant, shared-nothing, and scale-out relational database systems. He received his Ph.D. in Computer Science from the University of California, Berkeley, where he led the initial development of the Ingres relational database management system.

Pat Helland

Pat Helland has been building databases, transaction systems, distributed systems, messaging systems, multiprocessor hardware, and scalable cloud systems since 1978. At Tandem Computers, he was Chief Architect of the transaction engine for NonStop SQL. At Microsoft, he architected Microsoft Transaction Server, Distributed Transaction Coordinator, SQL Service Broker, and evolved the Cosmos big data infrastructure to include optimizing database features as well as petabyt-scale transactionally correct event processing. While at Amazon, Pat contributed to the design of the Dynamo eventually consistent store and also the Product Catalog. Pat attended the University of California, Irvine from 1973–1976 and was in the inaugural UC Irvine Information and Computer Science Hall of Fame. Pat chairs the Dean's Leadership Council of the Donald Bren School of Information and Computer Sciences (ICS), UC Irvine.

Joseph M. Hellerstein

Joseph M. Hellerstein is the Jim Gray Professor of Computer Science at the University of California, Berkeley, whose work focuses on data-centric systems and the way they drive computing. He is an ACM Fellow, an Alfred P. Sloan Research Fellow, and the recipient of three ACM-SIGMOD "Test of Time" awards for his research. In 2010, *Fortune Magazine* included him in their list of 50 smartest people in technology, and MIT's *Technology Review* magazine included his work on their TR10 list of the 10 technologies "most likely to change our world." Hellerstein is the co-founder and Chief Strategy Officer of Trifacta, a software vendor providing intelligent interactive solutions to the messy problem of wrangling data. He serves on the technical advisory boards of a number of computing and Internet companies including Dell EMC, SurveyMonkey, Captricity, and Datometry, and previously served as the Director of Intel Research, Berkeley.

Wei Hong

Wei Hong is an engineering director in Google's Data Infrastructure and Analysis (DIA) group, responsible for the streaming data processing area including building and maintaining the infrastructure for some of Google's most revenue-critical data pipelines in Ads and Commerce. Prior to joining Google, he cofounded and led three startup companies: Illustra and Cohera with Mike Stonebraker in database systems and Arch Rock in Internet of Things. He also held senior engineering leadership positions at Informix, PeopleSoft, Cisco, and Nest. He was a senior researcher at Intel Research Berkeley working on sensor networks and streaming database systems and won an ACM SIGMOD Test of Time Award. He is a co-inventor of 80 patents. He received his Ph.D. from UC Berkeley and hos ME, BE, and BS from Tsinghua University.

John Hugg

John Hugg has had a deep love for problems relating to data. He's worked at three database product startups and worked on database problems within larger organizations as well. Although John dabbled in statistics in graduate school, Dr. Stonebraker lured him back to databases using the nascent VoltDB project. Working with the very special VoltDB team was an unmatched opportunity to learn and be challenged. John received an M.S in 2007 and a B.S. in 2005 from Tufts University.

Ihab Ilyas

Ihab Ilyas is a professor in the Cheriton School of Computer Science at the University of Waterloo, where his main research focuses on the areas of big data and database systems, with special interest in data quality and integration, managing uncertain data, rank-aware query processing, and information extraction. Ihab is also a co-founder of Tamr, a startup focusing on large-scale data integration and cleaning. He is a recipient of the Ontario Early Researcher Award (2009), a Cheriton Faculty Fellowship (2013), an NSERC Discovery Accelerator Award (2014), and a Google Faculty Award (2014), and he is an ACM Distinguished Scientist. Ihab is an elected member of the VLDB Endowment board of trustees, elected SIGMOD vice chair, and an associate editor of *ACM Transactions on Database Systems* (TODS). He holds a Ph.D. in Computer Science from Purdue University and a B.Sc. and an M.Sc. from Alexandria University.

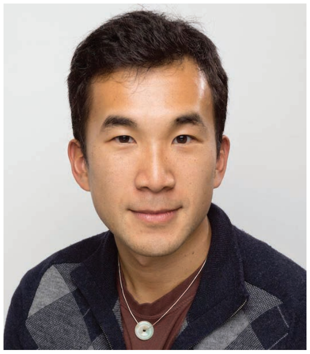
Jason Kinchen

Jason Kinchen , Paradigm4's V.P. of Engineering, is a software professional with over 30 years' experience in delivering highly complex products to life science, automotive, aerospace, and other engineering markets. He is an expert in leading technical teams in all facets of a project life cycle from feasibility analysis to requirements to functional design to delivery and enhancement, and experienced in developing quality-driven processes improving the software development life cycle and driving strategic planning. Jason is an avid cyclist and a Red Cross disaster action team volunteer.

Moshe Tov Kreps

Moshe Tov Kreps (formerly known as Peter Kreps) is a former researcher at the University of California at Berkeley and the Lawrence Berkeley National Laboratory. He was coauthor, with Mike Stonebraker, Eugene Wong, and Gerald Held, of the seminal paper, "The Design and Implementation of INGRES," published in the ACM Transactions on Database Systems in September 1976.

Edmond Lau



Edmond Lau is the co-founder of Co Leadership, where his mission is to transform engineers into leaders. He runs leadership experiences, multi-week programs, and online courses to bridge people from where they are to the lives and careers they dream of. He's the author of *The Effective Engineer*, the now the de facto onboarding guide for many engineering teams. He's spent his career leading engineering teams across Silicon Valley at Quip, Quora, Google, and Ooyala. As a leadership coach, Edmond also works directly with CTO's, directors, managers, and other emerging leaders to unlock what's possible for them. Edmond has been featured in the *New York Times*, *Forbes*, *Time*, *Slate*, *Inc*., *Fortune*, and *Wired*. He blogs at coleadership.com, has a website (www.theeffectiveengineer.com), and tweets at @edmondlau.

Shilpa Lawande

Shilpa Lawande is CEO and co-founder of postscript .us, an AI startup on a mission to free doctors from clinical paperwork. Previously, she was VP/GM HPE Big Data Platform, including its flagship Vertica Analytics Platform. Shilpa was a founding engineer at Vertica and led its Engineering and Customer Success teams from startup through the company's acquisition by HP. Shilpa has several patents and books on data warehousing to her name, and was named to the 2012 Mass High Tech Women to Watch list and Rev Boston 20 in 2015. Shilpa serves as an advisor at Tamr, and as mentor/volunteer at two educational initiatives, Year Up (Boston) and CSPathshala (India). Shilpa has a M.S. in Computer Science from the University of Wisconsin-Madison and a B.S in Computer Science and Engineering from the Indian Institute of Technology, Mumbai.

Amerson Lin

Amerson Lin received his B.S. and M.Eng both in Computer Science at MIT, the latter in 2005. He returned to Singapore to serve in the military and government before returning to the world of software. He was a consultant at Pivotal and then a business development lead at Palantir in both Singapore and the U.S. Amerson currently runs his own Insurtech startup—Gigacover—which delivers digital insurance to Southeast Asia.

Samuel Madden

Samuel Madden is a professor of Electrical Engineering and Computer Science in MIT's Computer Science and Artificial Intelligence Laboratory. His research interests include databases, distributed computing, and networking. He is known for his work on sensor networks, column-oriented database, high-performance transaction processing, and cloud databases. Madden received his Ph.D. in 2003 from the University of California at Berkeley, where he worked on the TinyDB system for data collection from sensor networks. Madden was named one of Technology Review's Top 35 Under 35 (2005), and is the recipient of several awards, including an NSF CAREER Award (2004), a Sloan Foundation Fellowship (2007), VLDB best paper awards (2004, 2007), and a MobiCom 2006 best paper award. He also received "test of time" awards in SIGMOD 2013 and 2017 (for his work on Acquisitional Query Processing in SIGMOD 2003 and on Fault Tolerance in the Borealis system in SIGMOD 2007), and a ten-year best paper award in VLDB 2015 (for his work on the C-Store system).

Tim Mattson

Tim Mattson is a parallel programmer. He earned his Ph.D. in Chemistry from the University of California, Santa Cruz for his work in molecular scattering theory. Since 1993, Tim has been with Intel Corporation, where he has worked on High Performance Computing: both software (OpenMP, OpenCL, RCCE, and OCR) and hardware/software co-design (ASCI Red, 80-core TFLOP chip, and the 48 core SCC). Tim's academic collaborations include work on the fundamental design patterns of parallel programming, the BigDAWG polystore system, the TileDB array storage manager, and building blocks for graphs "in the language of linear algebra" (the GraphBLAS). Currently, he leads a team of researchers at Intel working on technologies that help application programmers write highly optimized code that runs on future parallel systems. Outside of computing, Tim fills his time with coastal sea kayaking. He is an ACA-certified kayaking coach (level 5, advanced open ocean) and instructor trainer (level three, basic coastal).

Felix Naumann

Felix Naumann studied Mathematics, Economics, and Computer Science at the University of Technology in Berlin. He completed his Ph.D. thesis on “Quality-driven Query Answering” in 2000. In 2001 and 2002, he worked at the IBM Almaden Research Center on topics of data integration. From 2003–2006, he was assistant professor for information integration at the Humboldt-University of Berlin. Since then, he has held the chair for information systems at the Hasso Plattner Institute at the University of Potsdam in Germany. He is Editor-in-Chief of *Information Systems*, and his research interests are in data profiling, data cleansing, and text mining.

Mike Olson

Mike Olson co-founded Cloudera in 2008 and served as its CEO until 2013 when he took on his current role of chief strategy officer (CSO). As CSO, Mike is responsible for Cloudera’s product strategy, open-source leadership, engineering alignment, and direct engagement with customers. Prior to Cloudera, Mike was CEO of Sleepycat Software, makers of Berkeley DB, the open-source embedded database engine. Mike spent two years at Oracle Corporation as Vice President for Embedded Technologies after Oracle’s acquisition of Sleepycat in 2006. Prior to joining Sleepycat, Mike held technical and business positions at database vendors Britton Lee, Illustra Information Technologies, and Informix Software. Mike has a B.S. and an M.S. in Computer Science from the University of California, Berkeley. Mike tweets at @mikeolson.

Elizabeth O’Neil

Elizabeth O’Neil (Betty) is a Professor of Computer Science at the University of Massachusetts, Boston. Her focus is research, teaching, and software development in database engines: performance analysis, transactions, XML support, Unicode support, buffering methods. In addition to her work for UMass Boston, she was, among other pursuits, a long-term (1977–1996) part-time Senior Scientist for Bolt, Beranek, and Newman, Inc., and during two sabbaticals was a full-time consultant for Microsoft Corporation. She is the owner of two patents owned by Microsoft.

Patrick O'Neil

Patrick O'Neil is Professor Emeritus at the University of Massachusetts, Boston. His research has focused on database system cost-performance, transaction isolation, data warehousing, variations of bitmap indexing, and multi-dimensional databases/OLAP. In addition to his research, teaching, and service activities, he is the coauthor—with his wife Elizabeth (Betty)—of a database management textbook, and has been active in developing database performance benchmarks and corporate database consulting. He holds several patents.

Mourad Ouzzani

Mourad Ouzzani is a principal scientist with the Qatar Computing Research Institute, HBKU. Before joining QCRI, he was a research associate professor at Purdue University. His current research interests include data integration, data cleaning, and building large-scale systems to enable science and engineering. He is the lead PI of Rayyan, a system for supporting the creation of systematic reviews, which had more than 11,000 users as of March 2017. He has extensively published in top-tier venues including SIGMOD, PVLDB, ICDE, and TKDE. He received Purdue University Seed for Success Awards in 2009 and 2012. He received his Ph.D. from Virginia Tech and his M.S. and B.S. from USTHB, Algeria.

Andy Palmer

Andy Palmer is co-founder and CEO of Tamr, Inc., the enterprise-scale data unification company that he founded with fellow serial entrepreneur and 2014 Turing Award winner Michael Stonebraker, Ph.D., and others. Previously, Palmer was co-founder and founding CEO of Vertica Systems (also with Mike Stonebraker), a pioneering analytics database company (acquired by HP). He founded Koa Labs, a seed fund supporting the Boston/Cambridge entrepreneurial ecosystem, is a founder-partner at The Founder Collective, and holds a research affiliate position at MIT CSAIL. During his career as an entrepreneur, Palmer has served as Founder, founding investor, BoD member, or advisor to more than 60 startup companies in technology, healthcare, and the life sciences. He also served as Global Head of Software and Data Engineering at Novartis Institutes for BioMedical Research (NIBR) and as a member of the start-up team and Chief Information and Administrative Officer at Infinity Pharmaceuticals (NASDAQ: INFI). Previously, he held positions at innovative technology companies Bowstreet, pcOrder.com, and Trilogy. He holds a BA from Bowdoin (1988) and an MBA from the Tuck School of Business at Dartmouth (1994).

Andy Pavlo

Andy Pavlo is an assistant professor of Databaseology in the Computer Science Department at Carnegie Mellon University. He also used to raise clams. Andy received a Ph,D, in 2013 and an M.Sc. in 2009, both from Brown University, and an M.Sc. in 2006 and a B.Sc., both from Rochester Institute of Technology.

Alex Poliakov

Alex Poliakov has over a decade of experience developing distributed database internals. At Paradigm4, he helps set the vision for the SciDB product and leads a team of Customer Solutions experts who help researchers in scientific and commercial applications make optimal use of SciDB to create new insights, products, and services for their companies. Alex previously worked at Netezza, after graduating from MIT's Course 6. Alex is into flying drones and producing drone videos.

Alexander Rasin

Alexander Rasin is an Associate Professor in the College of Computing and Digital Media (CDM) at DePaul University. He received his Ph.D. and M.Sc. in Computer Science from Brown University, Providence, RI. He is a co-Director of Data Systems and Optimization Lab at CDM and his primary research interest is in database forensics and cybersecurity applications of forensic analysis. Dr. Rasin's other research projects focus on building and tuning performance of domain-specific data management systems—currently in the areas of computer-aided diagnosis and software analytics. Several of his current research projects are supported by NSF.

Jennie Rogers

Jennie Rogers is the Lisa Wissner-Slivka and Benjamin Slivka Junior Professor in Computer Science and an Assistant Professor at Northwestern University. Before that she was a postdoctoral associate in the Database Group at MIT CSAIL where she worked with Mike Stonebraker and Sam Madden. She received her Ph.D. from Brown University under the guidance of Ugur Çetintemel. Her research interests include the management of science data, federated databases, cloud computing, and database performance modeling. Her Erdös number is 3.

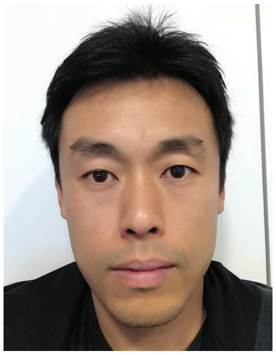
Lawrence A. Rowe

Lawrence A. Rowe is an Emeritus Professor of Electrical Engineering and Computer Science at U.C. Berkeley. His research interests are software systems and applications. His group developed the Berkeley Lecture Webcasting System that produced 30 course lecture webcasts each week viewed by over 500,000 people per month. His publications received three "best paper" and two "test of time" awards. He is an investor/advisor in The Batchery a Berkeley-based seed-stage incubator. Rowe is an ACM Fellow, a co-recipient of the 2002 U.C. Technology Leadership Council Award for IT Innovation, the recipient of the 2007 U.C. Irvine Donald Bren School of ICS Distinguished Alumni Award, the 2009 recipient of the ACM SIGMM Technical Achievement Award, and a co-recipient of the Inaugural ACM SIGMOD Systems Award for the development of modern object-relational DBMS. Larry and his wife Jean produce and sell award-winning premium wines using Napa Valley grapes under the Greyscale Wines brand.

Kriti Sen Sharma

Kriti Sen Sharma is a Customer Solutions Architect at Paradigm4. He works on projects spanning multiple domains (genomics, imaging, wearables, finance, etc.). Using his skills in collaborative problem-solving, algorithm development, and programming, he builds end-to-end applications that address customers' big-data needs and enable them to gain business insights rapidly. Kriti is an avid blogger and also loves biking and hiking. Kriti received a Ph.D. in 2013 and an M.Sc. in 2009, both from Virginia Polytechnic Institute and State University, and an a B.Tech. from Indian Institute of Technology, Kharagpur, in 2005.

Nan Tang



Nan Tang is a senior scientist at Qatar Computing Research Institute, HBKU, Qatar Foundation, Qatar. He received his Ph.D. from the Chinese University of Hong Kong in 2007. He worked as a research staff member at CWI, the Netherlands, from 2008–2010. He was a research fellow at University of Edinburgh from 2010–2012. His current research interests include data curation, data visualization, and intelligent and immersive data analytics.

Jo Tango

Jo Tango founded Kepha Partners. He has invested in the e-commerce, search engine, Internet ad network, wireless, supply chain software, storage, database, security, on-line payments, and data center virtualization spaces. He has been a founding investor in many Stonebraker companies: Goby (acquired by NAVTEQ), Paradigm4, StreamBase Systems (acquired by TIBCO), Vertica Systems (acquired by Hewlett-Packard), and VoltDB. Jo previously was at Highland Capital Partners for nearly nine years, where he was a General Partner. He also spent five years with Bain & Company, where he was based in Singapore, Hong Kong, and Boston, and focused on technology and startup projects. Jo attended Yale University (B.A., *summa cum laude* and Phi Beta Kappa) and Harvard Business School (M.B.A., Baker Scholar). He writes a personal blog at jtangoVC.com.

Nesime Tatbul

Nesime Tatbul is a senior research scientist at the Intel Science and Technology Center at MIT CSAIL. Before joining Intel Labs, she was a faculty member at the Computer Science Department of ETH Zurich. She received her B.S. and M.S. in Computer Engineering from the Middle East Technical University (METU) and her M.S. and Ph.D. in Computer Science from Brown University. Her primary research area is database systems. She is the recipient of an IBM Faculty Award in 2008, a Best System Demonstration Award at SIGMOD 2005, and the Best Poster and the Grand Challenge awards at DEBS 2011. She has served on the organization and program committees for various conferences including SIGMOD (as an industrial program co-chair in 2014 and a group leader in 2011), VLDB, and ICDE (as a PC track chair for Streams, Sensor Networks, and Complex Event Processing in 2013).

Nga Tran

Nga Tran is currently the Director of Engineering in the server development team at Vertica, where she has worked for the last 14 years. Previously, she was a Ph.D. candidate at Brandeis University, where she participated in research that contributed to Mike Stonebraker's research.

Marianne Winslett

Marianne Winslett has been a professor in the Department of Computer Science at the University of Illinois since 1987, and served as the Director of Illinois's research center in Singapore, the Advanced Digital Sciences Center, from 2009–2013. Her research interests lie in information management and security, from the infrastructure level on up to the application level. She is an ACM Fellow and the recipient of a Presidential Young Investigator Award from the U.S. National Science Foundation. She is the former Vice-Chair of ACM SIGMOD and the former co-Editor-in-Chief of *ACM Transactions on the Web*, and has served on the editorial boards of *ACM Transactions on Database Systems*, *IEEE*

Transactions on Knowledge and Data Engineering, *ACM Transactions on Information and System Security*, *The Very Large Data Bases Journal*, and *ACM Transactions on the Web*. She has received two best paper awards for research on managing regulatory compliance data (VLDB, SSS), one best paper award for research on analyzing browser extensions to detect security vulnerabilities (USENIX Security), and one for keyword search (ICDE). Her Ph.D. is from Stanford University.

Eugene Wong

Eugene Wong is Professor Emeritus at the University of California, Berkeley. His distinguished career includes contributions to academia, business, and public service. As Department Chair of EECS, he led the department through its greatest period of growth and into one of the highest ranked departments in its field. In 2004, the Wireless Foundation was established in Cory Hall upon completion of the Eugene and Joan C. Wong Center for Communications Research. He authored or co-authored over 100 scholarly articles and published 4 books, mentored students, and supervised over 20 dissertations. In 1980, he co-founded (with Michael Stonebraker and Lawrence A. Rowe) the INGRES Corporation. He was the Associate Director of the Office of Science and Technology Policy, under George H. Bush; from 1994–1996, he was Vice President for Research and Development for Hong Kong University of Science and Technology. He received the ACM Software System Award in 1988 for his work on INGRES, and was awarded the 2005 IEEE Founders Medal, with the apt citation: “For leadership in national and international engineering research and technology policy, for pioneering contributions in relational databases.”

Stan Zdonik

Stan Zdonik is a tenured professor of Computer Science at Brown University and a noted researcher in database management systems. Much of his work involves applying data management techniques to novel database architectures, to enable new applications. He is co-developer of the Aurora and Borealis stream processing engines, C-Store column store DBMS, and H-Store NewSQL DBMS, and has contributed to other systems including SciDB and the BigDAWG polystore system. He co-founded (with Michael Stonebraker) two startup companies: StreamBase Systems and Vertica Systems. Earlier, while at Bolt Beranek and Newman Inc., Dr. Zdonik worked on the Prophet System, a data management tool for pharmacologists. He has more than 150 peer-reviewed papers in the database field and was named an ACM Fellow in 2006. Dr. Zdonik has a B.S in Computer Science and one in Industrial Management, an M.S. in Computer Science, and the degree of Electrical Engineer, all from MIT, where he went on to receive his Ph.D. in database management under Prof. Michael Hammer.