Readings in Database Systems

Fifth Edition

edited by
Peter Bailis
Joseph M. Hellerstein
Michael Stonebraker

Readings in Database Systems

Fifth Edition (2015)

edited by Peter Bailis, Joseph M. Hellerstein, and Michael Stonebraker

Contents

Preface

In the ten years since the previous edition of *Readings in Database Systems*, the field of data management has exploded. Database and data-intensive systems today operate over unprecedented volumes of data, fueled in large part by the rise of "Big Data" and massive decreases in the cost of storage and computation. Cloud computing and microarchitectural trends have made distribution and parallelism nearly ubiquitous concerns. Data is collected from an increasing variety of heterogeneous formats and sources in increasing volume, and utilized for an ever increasing range of tasks. As a result, commodity database systems have evolved considerably along several dimensions, from the use of new storage media and processor designs, up through query processing architectures, programming interfaces, and emerging application requirements in both transaction processing and analytics. It is an exciting time, with considerable churn in the marketplace and many new ideas from research.

In this time of rapid change, our update to the traditional "Red Book" is intended to provide both a grounding in the core concepts of the field as well as a commentary on selected trends. Some new technologies bear striking resemblance to predecessors of decades past, and we think it's useful for our readers to be familiar with the primary sources. At the same time, technology trends are necessitating a re-evaluation of almost all dimensions of database systems, and many classic designs are in need of revision. Our goal in this collection is to surface important long-term lessons and foundational designs, and highlight the new ideas we believe are most novel and relevant.

Accordingly, we have chosen a mix of classic, traditional papers from the early database literature as well as papers that have been most influential in recent developments, including transaction processing, query processing, advanced analytics, Web data, and language design. Along with each chapter, we have included a short commentary introducing the papers and describing why we selected each. Each commentary is authored by one of the editors, but all editors provided input; we hope the commentaries do not lack for opinion.

When selecting readings, we sought topics and papers that met a core set of criteria. First, each selection represents a major trend in data management, as evidenced by both research interest and market demand. Second, each selection is canonical or near-canonical; we sought the most representative paper for each topic. Third, each selection is a primary source. There are good surveys on many of the topics in this collection, which we reference in commentaries. However, reading primary sources provides historical context, gives the reader exposure to the thinking that shaped influential solutions, and helps ensure that our readers are well-grounded in the field. Finally, this collection represents our current tastes about what is "most important"; we expect our readers to view this collection with a critical eye.

One major departure from previous editions of the Red Book is the way we have treated the final two sections on Analytics and Data Integration. It's clear in both research and the marketplace that these are two of the biggest problems in data management today. They are also quickly-evolving topics in both research and in practice. Given this state of flux, we found that we had a hard time agreeing on "canonical" readings for these topics. Under the circumstances, we decided to omit official readings but instead offer commentary. This obviously results in a highly biased view of what's happening in the field. So we do not recommend these sections as the kind of "required reading" that the Red Book has traditionally tried to offer. Instead, we are treating these as optional end-matter: "Biased Views on Moving Targets". Readers are cautioned to take these two sections with a grain of salt (even larger that the one used for the rest of the book.)

We are releasing this edition of the Red Book free of charge, with a permissive license on our text that allows unlimited non-commercial re-distribution, in multiple formats. Rather than secure rights to the recommended papers, we have simply provided links to Google Scholar searches that should help the reader locate the relevant papers. We expect this electronic format to allow more frequent editions of the "book." We plan to evolve the collection as appropriate.

A final note: this collection has been alive since 1988, and we expect it to have a long future life. Accordingly, we have added a modicum of "young blood" to the gray beard editors. As appropriate, the editors of this collection may further evolve over time.

Peter Bailis
Joseph M. Hellerstein
Michael Stonebraker

Chapter 1: Background

Introduced by Michael Stonebraker

Selected Readings:

Joseph M. Hellerstein and Michael Stonebraker. What Goes Around Comes Around. *Readings in Database Systems*, 4th Edition (2005).

Joseph M. Hellerstein, Michael Stonebraker, James Hamilton. Architecture of a Database System. *Foundations and Trends in Databases*, 1, 2 (2007).

I am amazed that these two papers were written a mere decade ago! My amazement about the anatomy paper is that the details have changed a lot just a few years later. My amazement about the data model paper is that nobody ever seems to learn anything from history. Lets talk about the data model paper first.

A decade ago, the buzz was all XML. Vendors were intent on adding XML to their relational engines. Industry analysts (and more than a few researchers) were touting XML as "the next big thing". A decade later it is a niche product, and the field has moved on. In my opinion, (as predicted in the paper) it succumbed to a combination of:

- excessive complexity (which nobody could understand)
- complex extensions of relational engines, which did not seem to perform all that well and
- no compelling use case where it was wildly accepted

It is a bit ironic that a prediction was made in the paper that X would win the Turing Award by successfully simplifying XML. That prediction turned out to be totally wrong! The net-net was that relational won and XML lost.

Of course, that has not stopped "newbies" from reinventing the wheel. Now it is JSON, which can be viewed in one of three ways:

- A general purpose hierarchical data format. Anybody who thinks this is a good idea should read the section of the data model paper on IMS.
- A representation for sparse data. Consider attributes about an employee, and suppose we wish to record hobbies data. For each hobby, the data we record will be different and hobbies are fundamentally sparse. This is straightforward to model in a relational DBMS but it leads to very wide, very sparse tables. This is disasterous for disk-based row stores but works fine in column stores. In the former case, JSON is a reasonable encoding format for the "hobbies" column, and several RDBMSs have recently added support for a JSON data type.
- As a mechanism for "schema on read". In effect, the schema is very wide and very sparse, and essentially all users will want some projection of this schema. When reading from a wide, sparse schema, a user can say what he wants to see at run time. Conceptually, this is nothing but a projection operation. Hence, 'schema on read" is just a relational operation on JSON-encoded data.

In summary, JSON is a reasonable choice for sparse data. In this context, I expect it to have a fair amount of "legs". On the other hand, it is a disaster in the making as a general hierarchical data format. I fully expect RDBMSs to subsume JSON as merely a data type (among many) in their systems. In other words, it is a reasonable way to encode spare relational data.

No doubt the next version of the Red Book will trash some new hierarchical format invented by people who stand on the toes of their predecessors, not on their shoulders.

The other data model generating a lot of buzz in the last decade is Map-Reduce, which was purpose-built by Google to support their web crawl data base. A few years later, Google stopped using Map-Reduce for that application, moving instead to Big Table. Now, the rest of the world is seeing what Google figured out earlier; Map-Reduce is not an architecture with any broad scale applicability. Instead the Map-Reduce market has morphed into an HDFS market, and seems poised to become a relational SQL market. For example, Cloudera has recently introduced Impala, which is a SQL engine, built on top of HDFS, not using Map-Reduce.

More recently, there has been another thrust in HDFS land which merit discussion, namely "data lakes". A reasonable use of an HDFS cluster (which by now most enterprises have invested in and want to find something useful for them to do) is as a queue of data files which have been ingested. Over time, the enterprise will figure out which ones are worth spending the effort to clean up (data curation; covered in Chapter 12 of this book). Hence, the data lake is just a "junk drawer" for files in the meantime. Also, we will have more to say about HDFS, Spark and Hadoop in Chapter 5.

In summary, in the last decade nobody seems to have heeded the lessons in "comes around". New data models have been invented, only to morph into SQL on tables. Hierarchical structures have been reinvented with failure as the predicted result. I would not be surprised to see the next decade to be more of the same. People seemed doomed to reinvent the wheel!

With regard to the Anatomy paper; a mere decade later, we can note substantial changes in how DBMSs are constructed. Hence, the details have changed a lot, but the overall architecture described in the paper is still pretty much true. The paper describes how most of the legacy DBMSs (e.g. Oracle, DB2) work, and a decade ago, this was the prevalent implementation. Now, these systems are historical artifacts; not very good at anything. For example, in the data warehouse market column stores have replaced the row stores described in this paper, because they are 1–2 orders of magnitude faster. In the OLTP world, main-memory SQL engines with very lightweight transaction management are fast becoming the norm. These new developments are chronicled in Chapter 4 of this book. It is now hard to find an application area where legacy row stores are competitive. As such, they deserve to be sent to the "home for retired software".

It is hard to imagine that "one size fits all" will ever be the dominant architecture again. Hence, the "elephants" have a bad "innovators dilemma" problem. In the classic book by Clayton Christiansen, he argues that it is difficult for the vendors of legacy technology to morph to new constructs without losing their customer base. However, it is already obvious how the elephants are going to try. For example, SQLServer 14 is at least two engines (Hekaton a main memory OLTP system and conventional SQLServer — a legacy row store) united underneath a common parser. Hence, the Microsoft strategy is clearly to add new engines under their legacy parser, and then support moving data from a tired engine to more modern ones, without disturbing applications. It remains to be seen how successful this will be.

However, the basic architecture of these new systems continues to follow the parsing/optimizer/executor structure described in the paper. Also, the threading model and process structure is as relevant today as a decade ago. As such, the reader should note that the details of concurrency control, crash recovery, optimization, data structures and indexing are in a state of rapid change, but the basic architecture of DBMSs remains intact.

In addition, it will take a long time for these legacy systems to die. In fact, there is still an enormous amount of IMS data in production use. As such, any student of the field is well advised to understand the architecture of the (dominant for a while) systems.

Furthermore, it is possible that aspects of this paper may become more relevant in the future as computing architectures evolve. For example, the impending arrival of NVRAM may provide an opportunity for new architectural concepts, or a reemergence of old ones.

Chapter 2: Traditional RDBMS Systems

Introduced by Michael Stonebraker

Selected Readings:

Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, Vera Watson. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems*, 1(2), 1976, 97-137.

Michael Stonebraker and Lawrence A. Rowe. The design of POSTGRES. *SIGMOD*, 1986.

David J. DeWitt, Shahram Ghandeharizadeh, Donovan Schneider, Allan Bricker, Hui-I Hsiao, Rick Rasmussen. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990, 44-62.

In this section are papers on (arguably) the three most important real DBMS systems. We will discuss them chronologically in this introduction.

The System R project started under the direction of Frank King at IBM Research probably around 1972. By then Ted Codd's pioneering paper was 18 months old, and it was obvious to a lot of people that one should build a prototype to test out his ideas. Unfortunately, Ted was not a permitted to lead this effort, and he went off to consider natural language interfaces to DBMSs. System R quickly decided to implement SQL, which morphed from a clean block structured language in 1972 [34] to a much more complex structure described in the paper here [33]. See [46] for a commentary on the design of the SQL language, written a decade later.

System R was structured into two groups, the "lower half" and the "upper half". They were not totally synchronized, as the lower half implemented links, which were not supported by the upper half. In defense of the decision by the lower half team, it was clear they were competing against IMS, which had this sort of construct, so it was natural to include it. The upper half simply didn't get the optimizer to work for this construct.

The transaction manager is probably the biggest legacy of the project, and it is clearly the work of the late Jim Gray. Much of his design endures to this day in commercial systems. Second place goes to the System R optimizer. The dynamic programming cost-based approach is still the gold standard for optimizer technology.

My biggest complaint about System R is that the team never stopped to clean up SQL. Hence, when the "upper half" was simply glued onto VSAM to form DB2, the language level was left intact. All the annoying features of the language have endured to this day. SQL will be the COBOL of 2020, a language we are stuck with that everybody will complain about.

My second biggest complaint is that System R used a subroutine call interface (now ODBC) to couple a client application to the DBMS. I consider ODBC among the worst interfaces on the planet. To issue a single query, one has to open a data base, open a cursor, bind it to a query and then issue individual fetches for data records. It takes a page of fairly inscrutable code just to run one query. Both Ingres [150] and Chris Date [45] had much cleaner language embeddings. Moreover, Pascal-R [140] and Rigel [135] were also elegant ways to include DBMS functionality in a programming language. Only recently with the advent of Linq [115] and Ruby on Rails [80] are we seeing a resurgence of cleaner language-specific enbeddings.

After System R, Jim Gray went off to Tandem to work on Non-stop SQL and Kapali Eswaren did a relational startup. Most of the remainder of the team remained at IBM and moved on to work on various other projects, include R*.

The second paper concerns Postgres. This project started in 1984 when it was obvious that continuing to prototype using the academic Ingres code base made no sense. A recounting of the history of Postgres appears in [147], and the reader is directed there for a full blow-by-blow recap of the ups and downs in the development process.

However, in my opinion the important legacy of Postgres is its abstract data type (ADT) system. User-defined types and functions have been added to most mainstream relational DBMSs, using the Postgres model. Hence, that design feature endures to this day. The project also experimented with time-travel, but it did not work very well. I think no-overwrite storage will have its day in the sun as faster storage technology alters the economics of data management.

It should also be noted that much of the importance of Postgres should be accredited to the availability of a robust and performant open-source code line. This is an example of the open-source community model of development and maintenance at its best. A pickup team of volunteers took the Berkeley code line in the mid 1990's and has been shepherding its development ever since. Both Postgres and 4BSD Unix [112] were instrumental in making open source code the preferred mechanism for code development.

The Postgres project continued at Berkeley until 1992, when the commercial company Illustra was formed to support a commercial code line. See [147] for a description of the ups and downs experienced by Illustra in the marketplace.

Besides the ADT system and open source distribution model, a key legacy of the Postgres project was a generation of highly trained DBMS implementers, who have gone on to be instrumental in building several other commercial systems

The third system in this section is Gamma, built at Wisconsin between 1984 and 1990. In my opinion, Gamma popularized the shared-nothing partitioned table approach to multi-node data management. Although Teradata had the same ideas in parallel, it was Gamma that popularized the concepts. In addition, prior to Gamma, nobody talked about hash-joins so Gamma should be credited (along with Kitsuregawa Masaru) with coming up with this class of algorithms.

Essentially all data warehouse systems use a Gamma-style architecture. Any thought of using a shared disk or shared memory system have all but disappeared. Unless network latency and bandwidth get to be comparable to disk bandwidth, I expect the current shared-nothing architecture to continue.

Chapter 3: Techniques Everyone Should Know

Introduced by Peter Bailis

Selected Readings:

Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, Thomas G. Price. Access path selection in a relational database management system. *SIGMOD*, 1979.

C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, Peter M. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1), 1992, 94-162.

Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, Irving L. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. , IBM, September, 1975.

Rakesh Agrawal, Michael J. Carey, Miron Livny. Concurrency Control Performance Modeling: Alternatives and Implications. *ACM Transactions on Database Systems*, 12(4), 1987, 609-654.

C. Mohan, Bruce G. Lindsay, Ron Obermarck. Transaction Management in the R* Distributed Database Management System. *ACM Transactions on Database Systems*, 11(4), 1986, 378-396.

In this chapter, we present primary and near-primary sources for several of the most important core concepts in database system design: query planning, concurrency control, database recovery, and distribution. The ideas in this chapter are so fundamental to modern database systems that nearly every mature database system implementation contains them. Three of the papers in this chapter are far and away the canonical references on their respective topics. Moreover, in contrast with the prior chapter, this chapter focuses on broadly applicable techniques and algorithms rather than whole systems.

Query Optimization

Query optimization is important in relational database architecture because it is core to enabling data-independent query processing. Selinger et al.'s foundational paper on System R enables practical query optimization by decomposing the problem into three distinct subproblems: cost estimation, relational equivalences that define a search space, and cost-based search.

The optimizer provides an estimate for the cost of executing each component of the query, measured in terms of I/O and CPU costs. To do so, the optimizer relies on both pre-computed statistics about the contents of each relation (stored in the system catalog) as well as a set of heuristics for determining the cardinality (size) of the query output (e.g., based on estimated predicate selectivity). As an exercise, consider these heuristics in detail: when do they make sense, and on what inputs will they fail? How might they be improved?

Using these cost estimates, the optimizer uses a dynamic programming algorithm to construct a plan for the query. The optimizer defines a set of physical operators that implement a given logical operator (e.g., looking up a tuple using a full 'segment' scan versus an index). Using this set, the optimizer iteratively constructs a "left-deep" tree of operators that in turn uses the cost heuristics to minimize the total amount of estimated work required to run the operators, accounting for "interesting orders" required by upstream consumers. This avoids having to consider all possible orderings of operators but is still exponential in the plan size; as we discuss in Chapter 7, modern query optimizers still struggle with large plans (e.g., many-way joins). Additionally, while the Selinger et al. optimizer performs compilation in advance, other early systems, like Ingres [150] interpreted the query plan – in effect, on a tuple-by-tuple basis.

Like almost all query optimizers, the Selinger et al. optimizer is not actually "optimal" – there is no guarantee that the plan that the optimizer chooses will be the fastest or cheapest. The relational optimizer is closer in spirit to code optimization routines within modern language compilers (i.e., will perform a best-effort search) rather than mathematical optimization routines (i.e., will find the best solution). However, many of today's relational engines adopt the basic methodology from the paper, including the use of binary operators and cost estimation.

Concurrency Control

Our first paper on transactions, from Gray et al., introduces two classic ideas: multi-granularity locking and multiple lock modes. The paper in fact reads as two separate papers.

First, the paper presents the concept of multi-granularity locking. The problem here is simple: given a database with a hierarchical structure, how should we perform mutual exclusion? When should we lock at a coarse granularity (e.g., the whole database) versus a finer granularity (e.g., a single record), and how can we support concurrent access to different portions of the hierarchy at once? While Gray et al.'s hierarchical layout (consisting of databases, areas, files, indexes, and records) differs slightly from that of a modern database system, all but the most rudimentary database locking systems adapt their proposals today.

Second, the paper develops the concept of multiple degrees of isolation. As Gray et al. remind us, a goal of concurrency control is to maintain data that is "consistent" in that it obeys some logical assertions. Classically, database systems used serializable transactions as a means of enforcing consistency: if individual transactions each leave the database in a "consistent" state, then a serializable execution (equivalent to some serial execution of the transactions) will guarantee that all transactions observe a "consistent" state of the database [57]. Gray et al.'s "Degree 3" protocol describes the classic (strict) "two-phase locking" (2PL), which guarantees serializable execution and is a major concept in transaction processing.

However, serializability is often considered too expensive to enforce. To improve performance, database systems often instead execute transactions using non-serializable isolation. In the paper here, holding locks is expensive: waiting for a lock in the case of a conflict takes time, and, in the event of a deadlock, might take forever (or cause aborts). Therefore, as early as 1973, database systems such as IMS and System R began to experiment with non-serializable policies. In a lock-based concurrency control system, these policies are implemented by holding locks for shorter durations. This allows greater concurrency, may lead to fewer deadlocks and system-induced aborts, and, in a distributed setting, may permit greater availability of operation.

In the second half of this paper, Gray et al. provide a rudimentary formalization of the behavior of these lock-based policies. Today, they are prevalent; as we discuss in Chapter 6, non-serializable isolation is the default in a majority of commercial and open source RDBMSs, and some RDBMSs do not offer serializability at all. Degree 2 is now typically called Repeatable Read isolation and Degree 1 is now called Read Committed isolation, while Degree 0 is infrequently used [27]. The paper also discusses the important notion of recoverability: policies under which a transaction can be aborted (or "undone") without affecting other transactions. All but Degree 0 transactions satisfy this property.

A wide range of alternative concurrency control mechanisms followed Gray et al.'s pioneering work on lock-based serializability. As hardware, application demands, and access patterns have changed, so have concurrency control subsystems. However, one property of concurrency control remains a near certainty: there is no unilateral "best" mechanism in concurrency control. The optimal strategy is workload-dependent. To illustrate this point, we've included a study from Agrawal, Carey, and Livny. Although dated, this paper's methodology and broad conclusions remain on target. It's a great example of thoughtful, implementation-agnostic performance analysis work that can provide valuable lessons over time.

Methodologically, the ability to perform so-called "back of the envelope" calculations is a valuable skill: quickly estimating a metric of interest using crude arithmetic to arrive at an answer within an order of magnitude of the correct value can save hours or even years of systems implementation and performance analysis. This is a long and useful tradition in database systems, from the "Five Minute Rule" [73] to Google's "Numbers Everyone Should Know" [48]. While some of the lessons drawn from these estimates are transient [69, 66], often the conclusions provide long-term lessons.

However, for analysis of complex systems such as concurrency control, simulation can be a valuable intermediate step between back of the envelope and full-blown systems benchmarking. The Agrawal study is an example of this approach: the authors use a carefully designed system and user model to simulate locking, restart-based, and optimistic concurrency control.

Several aspects of the evaluation are particularly valuable. First, there is a "crossover" point in almost every graph: there aren't clear winners, as the best-performing mechanism depends on the workload and system configuration. In contrast, virtually every performance study without a crossover point is likely to be uninteresting. If a scheme "always wins," the study should contain an analytical analysis, or, ideally, a proof of why this is the case. Second, the authors consider a wide range of system configurations; they investigate and discuss almost all parameters of their model. Third, many of the graphs exhibit non-monotonicity (i.e., don't always go up and to the right); this a product of thrashing and resource limitations. As the authors illustrate, an assumption of infinite resources leads to dramatically different conclusions. A less careful model that made this assumption implicit would be much less useful.

Finally, the study's conclusions are sensible. The primary cost of restart-based methods is "wasted" work in the event of conflicts. When resources are plentiful, speculation makes sense: wasted work is less expensive, and, in the event of infinite resources, it is free. However, in the event of more limited resources, blocking strategies will consume fewer resources and offer better overall performance. Again, there is no unilaterally optimal choice. However, the paper's concluding remarks have proven prescient: computing resources are still scarce, and, in fact, few commodity systems today employ entirely restart-based methods. However, as technology ratios – disk, network, CPU speeds – continue to change, re-visiting this trade-off is valuable.

Database Recovery

Another major problem in transaction processing is maintaining durability: the effects of transaction processing should survive system failures. A near-ubiquitous technique for maintaining durability is to perform logging: during transaction execution, transaction operations are stored on fault-tolerant media (e.g., hard drives or SSDs) in a log. Everyone working in data systems should understand how write-ahead logging works, preferably in some detail.

The canonical algorithm for implementing a "No Force, Steal" WAL-based recovery manager is IBM's ARIES algorithm, the subject of our next paper. (Senior database researchers may tell you that very similar ideas were invented at the same time at places like Tandem and Oracle.) In ARIES, the database need not write dirty pages to disk at commit time ("No Force"), and the database can flush dirty pages to disk at any time ("Steal") [78]; these policies allow high performance and are present in almost every commercial RDBMS offering but in turn add complexity to the database. The basic idea in ARIES is to perform crash recovery in three stages. First, ARIES performs an analysis phase by replaying the log forwards in order to determine which transactions were in progress at the time of the crash. Second, ARIES performs a redo stage by (again) replaying the log and (this time) performing the effects of any transactions that were in progress at the time of the crash. Third, ARIES performs an undo stage by playing the log backwards and undoing the effect of uncommitted transactions. Thus, the key idea in ARIES is to "repeat history" to perform recovery; in fact, the undo phase can execute the same logic that is used to abort a transaction during normal operation.

ARIES should be a fairly simple paper but it is perhaps the most complicated paper in this collection. In graduate database courses, this paper is a rite of passage. However, this material is fundamental, so it is important to understand. Fortunately, Ramakrishnan and Gehrke's undergraduate textbook [127] and a survey paper by Michael Franklin [61] each provide a milder treatment. The full ARIES paper we have included here is complicated significantly by its diversionary discussions of the drawbacks of alternative design decisions along the way. On the first pass, we encourage readers to ignore this material and focus solely on the ARIES approach. The drawbacks of alternatives are important but should be saved for a more careful second or third read. Aside from its organization, the discussion of ARIES protocols is further complicated by discussions of managing internal state like indexes (i.e., nested top actions and logical undo logging — the latter of which is also used in exotic schemes like Escrow transactions [124]) and techniques to minimize downtime during recovery. In practice, it is important for recovery time to appear as short as possible; this is tricky to achieve.

Distribution

Our final paper in this chapter concerns transaction execution in a distributed environment. This topic is especially important today, as an increasing number of databases are distributed – either replicated, with multiple copies of data on different servers, or partitioned, with data items stored on disjoint servers (or both). Despite offering benefits to capacity, durability, and availability, distribution introduces a new set of concerns. Servers may fail and network links may be unreliable. In the absence of failures, network communication may be costly.

We concentrate on one of the core techniques in distributed transaction processing: atomic commitment (AC). Very informally, given a transaction that executes on multiple servers (whether multiple replicas, multiple partitions, or both), AC ensures that the transaction either commits or aborts on all of them. The classic algorithm for achieving AC dates to the mid-1970s and is called Two-Phase Commit (2PC; not to be confused with 2PL above!) [67, 100]. In addition to providing a good overview of 2PC and interactions between the commit protocol and the WAL, the paper here contains two variants of AC that improve its performance. The Presumed Abort variant allows processes to avoid forcing an abort decision to disk or acknowledge aborts, reducing disk utilization and network traffic. The Presumed Commit optimization is similar, optimizing space and network traffic when more transactions commit. Note the complexity of the interactions between the 2PC protocol, local storage, and the local transaction manager; the details are important, and correct implementation of these protocols can be challenging.

The possibility of failures substantially complicates AC (and most problems in distributed computing). For example, in 2PC, what happens if a coordinator and participant both fail after all participants have sent their votes but before the coordinator has heard from the failed participant? The remaining participants will not know whether or to commit or abort the transaction: did the failed participant vote YES or vote NO? The participants cannot safely continue. In fact, *any* implementation of AC may block, or fail to make progress, when operating over an unreliable network [28]. Coupled with a serializable concurrency control mechanism, blocking AC means throughput may stall. As a result, a related set of AC algorithms examined AC under relaxed assumptions regarding both the network (e.g., by assuming a synchronous network) [145] and the information available to servers (e.g., by making use of a "failure detector" that determines when nodes fail) [76].

Finally, many readers may be familiar with the closely related problem of consensus or may have heard of consensus implementations such as the Paxos algorithm. In consensus, any proposal can be chosen, as long as all processes eventually will agree on it. (In contrast, in AC, any individual participant can vote NO, after which all participants must abort.) This makes consensus an "easier" problem than AC [75], but, like AC, any implementation of consensus can also block in certain scenarios [60]. In modern distributed databases, consensus is often used as the basis for replication, to ensure replicas apply updates in the same order, an instance of state-machine replication (see Schneider's tutorial [141]). AC is often used to execute transactions that span multiple partitions. Paxos by Lamport [99] is one of the earliest (and most famous, due in part to a presentation that rivals ARIES in complexity) implementations of consensus. However, the Viewstamped Replication [102] and Raft [125], ZAB [92], and Multi-Paxos [35] algorithms may be more helpful in practice. This is because these algorithms implement a distributed log abstraction (rather than a 'consensus object' as in the original Paxos paper).

Unfortunately, the database and distributed computing communities are somewhat separate. Despite shared interests in replicated data, transfer of ideas between the two were limited for many years. In the era of cloud and Internet-scale data management, this gap has shrunk. For example, Gray and Lamport collaborated in 2006 on Paxos Commit [71], an interesting algorithm combining AC and Lamport's Paxos. There is still much to do in this intersection, and the number of "techniques everyone should know" in this space has grown.

Chapter 4: New DBMS Architectures

Introduced by Michael Stonebraker

Selected Readings:

Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, Stan Zdonik. C-store: A Column-oriented DBMS. *SIGMOD*, 2005.

Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, Mike Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. *SIGMOD*, 2013.

Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, Michael Stonebraker. OLTP Through the Looking Glass, and What We Found There. *SIGMOD*, 2008.

Probably the most important thing that has happened in the DBMS landscape is the death of "one size fits all". Until the early 2000's the traditional disk-based row-store architecture was omni-present. In effect, the commercial vendors had a hammer and everything was a nail.

In the last fifteen years, there have been several major upheavals, which we discuss in turn.

First, the community realized that column stores are dramatically superior to row stores in the data warehouse marketplace. Data warehouses found early acceptance in customer facing retail environments and quickly spread to customer facing data in general. Warehouses recorded historical information on customer transactions. In effect, this is the who-what-why-when-where of each customer interaction.

The conventional wisdom is to structure a data warehouse around a central Fact table in which this transactional information is recorded. Surrounding this are dimension tables which record information that can be factored out of the Fact table. In a retail scenario one has dimension tables for Stores, Customers, Products and Time. The result is a so-called star schema [96]. If stores are grouped into regions, then there may be multiple levels of dimension tables and a snowflake schema results.

The key observation is that Fact tables are generally "fat" and often contain a hundred or more attributes. Obviously, they also "long" since there are many, many facts to record. In general, the queries to a data warehouse are a mix of repeated requests (produce a monthy sales report by store) and "ad hoc" ones. In a retail warehouse, for example, one might want to know what is selling in the Northeast when a snowstorm occurs and what is selling along the Atlantic seaboard during hurricanes.

Moreover, nobody runs a select * query to fetch all of the rows in the Fact table. Instead, they invariably specify an aggregate, which retrieves a half-dozen attributes out of the 100 in the table. The next query retrieves a different set, and there is little-to-no locality among the filtering criteria.

In this use case, it is clear that a column store will move a factor of 16 less data from the disk to main memory than a row store will (6 columns versus 100). Hence, it has an unfair advantage. Furthermore, consider a storage block. In a column store, there is a single attribute on that block, while a row store will have 100. Compression will clearly work better on one attribute than on 100. In addition, row stores have a header on the front of each record (in SQLServer it is apparently 16 bytes). In contrast, column stores are careful to have no such header.

Lastly, a row-based executor has an inner loop whereby a record is examined for validity in the output. Hence, the overhead of the inner loop, which is considerable, is paid per record examined. In contrast, the fundamental operation of a column store is to retrieve a column and pick out the qualifying items. As such, the inner-loop overhead is paid once per column examined and not once per row examined. As such a column executor is way more efficient in CPU time and retrieves way less data from the disk. In most real-world environments, column stores are 50–100 times faster than row stores.

Early column stores included Sybase IQ [108], which appeared in the 1990s, and MonetDB [30]. However, the technology dates to the 1970s [25, 104]. In the 2000's C-Store/Vertica appeared as well-financed startup with a high performance implementation. Over the next decade the entire data warehouse market morphed from a row-store world to a column store world. Arguably, Sybase IQ could have done the same thing somewhat earlier, if Sybase had invested more aggressively in the technology and done a multi-node implementation. The advantages of a column executor are persuasively discussed in [30], although it is "down in the weeds" and hard to read.

The second major sea change was the precipitous decline in main memory prices. At the present time, one can buy a 1Terabyte for perhaps $25,000, and a high performance computing cluster with a few terabytes can be bought for maybe $100K. The key insight is that OLTP data bases are just not that big. One terabyte is a very big OLTP data base, and is a candidate for main memory deployment. As noted in the looking glass paper in this section, one does not want to run a disk-based row store when data fits in main memory the overhead is just way too high.

In effect, the OLTP marketplace is now becoming a main memory DBMS marketplace. Again, traditional disk-based row stores are just not competitive. To work well, new solutions are needed for concurrency control, crash recovery, and multi-threading, and I expect OLTP architectures to evolve over the next few years.

My current best guess is that nobody will use traditional two phase locking. Techniques based on timestamp ordering or multiple versions are likely to prevail. The third paper in this section discusses Hekaton, which implements a state-of-the art MVCC scheme.

Crash recovery must also be dealt with. In general, the solution proposed is replication, and on-line failover, which was pioneered by Tandem two decades ago. The traditional wisdom is to write a log, move the log over the network, and then roll forward at the backup site. This active-passive architecture has been shown in [111] to be a factor of 3 inferior to an active-active scheme where the transactions is simply run at each replica. If one runs an active-active scheme, then one must ensure that transactions are run in the same order at each replica. Unfortunately, MVCC does not do this. This has led to interest in deterministic concurrency control schemes, which are likely to be wildly faster in an end-to-end system that MVCC.

In any case, OLTP is going to move to main memory deployment, and a new class of main memory DBMSs is unfolding to support this use case.

The third phenomenon that has unfolded is the "no SQL" movement. In essence, there are 100 or so DBMSs, which support a variety of data models and have the following two characteristics:

1. "Out of box" experience. They are easy for a programmer to get going and do something productive. RDBMSs, in contrast, are very heavyweight, requiring a schema up front.

2. Support for semi-structured data. If every record can have values for different attributes, then a traditional row store will have very, very wide tuples, and be very sparse, and therefore inefficient.

This is a wake-up call to the commercial vendors to make systems that are easier to use and support semi-structured data types, such as JSON. In general, I expect the No SQL market to merge with the SQL market over time as RDBMSs react to the two points noted above.

The fourth sea change is the emergence of the Hadoop/HDFS/Spark environment, which is discussed in Chapter 6.

Chapter 5: Large-Scale Dataflow Engines

Introduced by Peter Bailis

Selected Readings:

Jeff Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI*, 2004.

Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. *OSDI*, 2008.

Of the many developments in data management over the past decade, MapReduce and subsequent large-scale data processing systems have been among the most disruptive and the most controversial. Cheap commodity storage and rising data volumes led many Internet service vendors to discard conventional database systems and data warehouses and build custom, home-grown engines instead. Google's string of publications on their large-scale systems, including Google File System [62], MapReduce, Chubby [32], and BigTable [37], are perhaps the most famous and influential in the market. In almost all cases, these new, homegrown systems implemented a small subset of the features found in conventional databases, including high-level languages, query optimizers, and efficient execution strategies. However, these systems and the resulting open source Hadoop ecosystem proved highly popular with many developers. This led to considerable investment, marketing, research interest, and development on these platforms, which, today are in flux, but, as an ecosystem, have come to resemble traditional data warehouses—with some important modifications. We reflect on these trends here.

History and Successors

Our first reading is the original Google MapReduce paper from 2004. MapReduce was a library built for simplifying parallel, distributed computation over distributed data at Google's scale—particularly, the batch rebuild of web search indexes from crawled pages. It is unlikely that, at the time, a traditional data warehouse could have handled this workload. However, compared to a conventional data warehouse, MapReduce provides a very low-level interface (two-stage dataflow) that is closely tied to a fault-tolerant execution strategy (intermediate materialization between two-stage dataflow). Equally importantly, MapReduce was designed as a library for parallel programming rather than an end-to-end data warehousing solution; for example, MapReduce delegates storage to Google File System. At the time, members of the database community decried the architecture as simplistic, inefficient, and of limited use [53].

While the original MapReduce paper was released in 2003, there was relatively little additional activity external to Google until 2006, when Yahoo! open-sourced the Hadoop MapReduce implementation. Subsequently, there was an explosion of interest: within a year, a range of projects including Dryad (Microsoft) [89], Hive (Facebook) [156], Pig (Yahoo) [123] were all under development. These systems, which we will call post-MapReduce systems, gained considerable traction with developers—who were largely concentrated in Silicon Valley—as well as serious VC investment. A multitude of research spanning the systems, databases, and networking communities investigated issues including scheduling, straggler mitigation, fault tolerance, UDF query optimization, and alternative programming models [16].

Almost immediately, post-MapReduce systems expanded their interface and functionality to include more sophisticated declarative interfaces, query optimization strategies, and efficient runtimes. Today's post-MapReduce systems have come to implement a growing proportion of the feature set of conventional RDBMSs. The latest generation of data processing engines such as Spark [163], F1 [143], Impala [98], Tez [1], Naiad [119], Flink/Stratosphere [9], AsterixDB [10], and Drill [82] frequently i) expose higher-level query languages such as SQL, ii) more advanced execution strategies, including the ability to process general graphs of operators, and iii) use indexes and other functionality of structured input data sources when possible. In the Hadoop ecosystem, dataflow engines have become the substrate for a suite of higher-level functionality and declarative interfaces, including SQL [15, 156], graph processing [64, 110], and machine learning [63, 146].

There is also increasing interest in stream processing functionality, revisiting many of the concepts pioneered in the database community in the 2000s. A growing commercial and open source ecosystem has developed ”connectors” to various structured and semi-structured data sources, catalog functionality (e.g., HCatalog), and data serving and limited transactional capabilities (e.g., HBase). Much of this functionality, such as the typical query optimizers in these frameworks, is rudimentary compared to many mature commercial databases but is quickly evolving.

DryadLINQ, our second selected reading for this section, is perhaps most interesting for its interface: a set of embedded language bindings for data processing that integrates seamlessly with Microsoft’s .NET LINQ to provide a parallelized collections library. DryadLINQ executes queries via the earlier Dryad system [89], which implemented a runtime for arbitrary dataflow graphs using a replay-based fault tolerance. While DryadLINQ still restricts programmers to a set of side-effect free dataset transformations (including “SQL-like” operations), it presents a considerably higher-level interface than Map Reduce. DryadLINQ’s language integration, lightweight fault tolerance, and basic query optimization techniques proved influential in later dataflow systems, including Apache Spark [163] and Microsoft’s Naiad [119].

Impact and Legacy

There are at least three lasting impacts of the MapReduce phenomenon that might not have occurred otherwise. These ideas are – like distributed dataflow itself – not necessarily novel, but the ecosystem of post-MapReduce dataflow and storage systems have broadly increased their impact:

1.) *Schema flexibility.* Perhaps most importantly, traditional data warehouse systems are walled gardens: ingested data is pristine, curated, and has structure. In contrast, MapReduce systems process arbitrarily structured data, whether clean or dirty, curated or not. There is no loading step. This means users can store data first and consider what to do with it later. Coupled with the fact that storage (e.g., in the Hadoop File System) is considerably cheaper than in a traditional data warehouse, users can afford to retain data for longer and longer. This is a major shift from traditional data warehouses and is a key factor behind the rise and gathering of ”Big Data.” A growing number of storage formats (e.g., Avro, Parquet, RCFile) marry semi-structured data and advances in storage such as columnar layouts. In contrast with XML, this newest wave of semi-structured data is even more flexible. As a result, extract-transform-load (ETL) tasks are major workload for post-MapReduce engines. It is difficult to overstate the impact of schema flexibility on the modern practice of data management at all levels, from analyst to programmer and analytics vendor, and we believe it will become even more important in the future. However, this heterogeneity is not free: curating such “data lakes” is expensive (much more than storage) and is a topic we consider in depth in Chapter 12.

2.) *Interface flexibility.* Today, most all users interact with Big Data engines in SQL-like languages. However, these engines also allow users to program using a combination of paradigms. For example, an organization might use imperative code to perform file parsing, SQL to project a column, and machine learning subroutines to cluster the results – all within a single framework. Tight, idiomatic language integration as in DryadLINQ is commonplace, further improving programmability. While traditional database engines historically supported user-defined functions, these new engines’ interfaces make user-defined computations simpler to express and also make it easier to integrate the results of user-defined computations with the results of queries expressed using traditional relational constructs like SQL. Interface flexibility and integration is a strong selling point for data analytics offerings; the ability to combine ETL, analytics, and post-processing in a single system is remarkably convenient to programmers — but not necessarily to users of traditional BI tools, which make use of traditional JDBC interfaces.

3.) *Architectural flexibility.* A common critique of RDBMSs is that their architecture is too tightly coupled: storage, query processing, memory management, transaction processing, and so on are closely intertwined, with a lack of clear interfaces between them in practice. In contrast, as a result of its bottom-up development, the Hadoop ecosystem has effectively built a data warehouse as a series of modules. Today, organizations can write and run programs against the raw file system (e.g., HDFS), any number of dataflow engines (e.g., Spark), using advanced analytics packages (e.g., GraphLab [105], Parameter Server [101]), or via SQL (e.g., Impala [98]). This flexibility adds performance overhead, but the ability to mix and match components and analytics packages is unprecedented at this scale. This architectural flexibility is perhaps most interesting to systems builders and vendors, who have additional degrees of freedom in designing their infrastructure offerings.

To summarize, a dominant theme in today's distributed data management infrastructure is flexibility and heterogeneity: of storage formats, of computation paradigms, and of systems implementations. Of these, storage format heterogeneity is probably the highest impact by an order of magnitude or more, simply because it impacts novices, experts, and architects alike. In contrast, heterogeneity of computation paradigms most impacts experts and architects, while heterogeneity of systems implementations most impacts architects. All three are relevant and exciting developments for database research, with lingering questions regarding market impact and longevity.

Looking Ahead

In a sense, MapReduce was a short-lived, extreme architecture that blew open a design space. The architecture was simple and highly scalable, and its success in the open source domain led many to realize that there was demand for alternative solutions and the principle of flexibility that it embodied (not to mention a market opportunity for cheaper data warehousing solutions based on open source). The resulting interest is still surprising to many and is due to many factors, including community zeitgeist, clever marketing, economics, and technology shifts. It is interesting to consider which differences between these new systems and RDBMSs are fundamental and which are due to engineering improvements.

Today, there is still debate about the appropriate architecture for large-scale data processing. As an example, Rasmussen et al. provide a strong argument for why intermediate fault tolerance is not necessary except in very large (100+ node) clusters [132]. As another example, McSherry et al. have colorfully illustrated that many workloads can be efficiently processed using a single server (or thread!), eliminating the need for distribution at all [113]. Recently, systems such as the GraphLab project [105] suggested that domain-specific systems are necessary for performance; later work, including Grail [58] and GraphX [64], argued this need not be the case. A further wave of recent proposals have also suggested new interfaces and systems for stream processing, graph processing, asynchronous programming, and general-purpose machine learning. Are these specialized systems actually required, or can one analytics engine rule them all? Time will tell, but I perceive a push towards consolidation.

Finally, we would be remiss not to mention Spark, which is only six years old but is increasingly popular with developers and is very well supported both by VC-backed startups (e.g., Databricks) and by established firms such as Cloudera and IBM. While we have included DryadLINQ as an example of a post-MapReduce system due to its historical significance and technical depth, the Spark paper [163], written in the early days of the project, and recent extensions including SparkSQL [15], are worthwhile additional reads. Like Hadoop, Spark rallied major interest at a relatively early stage of maturity. Today, Spark still has a ways to go before its feature set rivals that of a traditional data warehouse. However, its feature set is rapidly growing and expectations of Spark as the successor to MapReduce in the Hadoop ecosystem are high; for example, Cloudera is working to replace MapReduce with Spark in the Hadoop ecosystem [81]. Time will tell whether these expectations are accurate; in the meantime, the gaps between traditional warehouses and post-MapReduce systems are quickly closing, resulting in systems that are as good at data warehousing as traditional systems, but also much more.

Commentary: Michael Stonebraker

26 October 2015

Recently, there has been considerable interest in data analytics as part of the marketing moniker "big data". Historically, this meant business intelligence (BI) analytics and was serviced by BI applications (Cognos, Business Objects, etc.) talking to a relational data warehouse (such as Teradata, Vertica, Red Shift, Greenplum, etc.). More recently it has become associated with "data science". In this context, let's start ten years ago with Map-Reduce, which was purpose-built by Google to support their web crawl data base. Then, the marketing guys took over with the basic argument: "Google is smart; Map-Reduce is Google's next big thing, so it must be good". Cloudera, Hortonworks and Facebook were in the vanguard in hyping Map-Reduce (and its open source look-alike Hadoop). A few years ago, the market was abuzz drinking the Map-Reduce koolaid. About the same time, Google stopped using Map-Reduce for the application that it was purpose-built for, moving instead to Big Table. With a delay of about 5 years, the rest of the world is seeing what Google figured out earlier; Map-Reduce is not an architecture with any broad scale applicability.

In effect Map-Reduce suffers from the following two problems:

1. It is inappropriate as a platform on which to build data warehouse products. There is no interface inside any commercial data warehouse product which looks like Map-Reduce, and for good reason. Hence, DBMSs do not want this sort of platform.
2. It is inappropriate as a platform on which to build distributed applications. Not only is the Map-Reduce interface not flexible enough for distributed applications but also a message passing system that uses the file system is way too slow to be interesting.

Of course, that has not stopped the Map-Reduce vendors. They have simply rebranded their platform to be HDFS (a file system) and have built products based on HDFS that do not include Map-Reduce. For example, Cloudera has recently introduced Impala, which is a SQL engine, not built on Map-Reduce. In point of fact, Impala does not really use HDFS either, choosing to drill through that layer to read and write the underlying local Linux files directly. HortonWorks and Facebook have similar projects underway. As such the Map-Reduce crowd has turned into a SQL crowd and Map-Reduce, as an interface, is history. Of course, HDFS has serious problems when used by a SQL engine, so it is not clear that it will have any legs, but that remains to be seen. In any case, the Map-Reduce-HDFS market will merge with the SQL-data warehouse market; and may the best systems prevail. In summary, Map-Reduce has failed as a distributed systems platform, and vendors are using HDFS as a file system under data warehouse products.

This brings us to Spark. The original argument for Spark is that it is a faster version of Map-Reduce. It is a main memory platform with a fast message passing interface. Hence, it should not suffer from the performance problems of Map-Reduce when used for distributed applications. However, according to Spark's lead author Matei Zaharia, more than 70% of the Spark accesses are through SparkSQL. In effect, Spark is being used as a SQL engine, not as a distributed applications platform! In this context Spark has an identity problem. If it is a SQL platform, then it needs some mechanism for persistence, indexing, sharing of main memory between users, meta data catalogs, etc. to be competitive in the SQL/data warehouse space. It seems likely that Spark will turn into a data warehouse platform, following Hadoop along the same path.

On the other hand, 30% of Spark accesses are not to SparkSQL and are primarily from Scala. Presumably this is a distributed computing load. In this context, Spark is a reasonable distributed computing platform. However, there are a few issues to consider. First, the average data scientist does a mixture of data management and analytics. Higher performance comes from tightly coupling the two. In Spark there is no such coupling, since Spark's data formats are not necessarily common across these two tasks. Second, Spark is main memory-only (at least for now). Scalability requirements will presumably get this fixed over time. As such, it will be interesting to see how Spark evolves off into the future.

In summary, I would like to offer the following takeaways:

- Just because Google thinks something is a good idea does not mean you should adopt it.
- Disbelieve all marketing spin, and figure out what benefit any given product actually has. This should be especially applied to performance claims.
- The community of programmers has a love affair with "the next shiny object". This is likely to create "churn" in your organization, as the "half-life" of shiny objects may be quite short.

Chapter 6: Weak Isolation and Distribution

Introduced by Peter Bailis

Selected Readings:

Atul Adya, Barbara Liskov, and Patrick O'Neil. Generalized Isolation Level Definitions. *ICDE*, 2000.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. *SOSP*, 2007.

Eric Brewer. CAP Twelve Years Later: How the "Rules" Have Changed. *IEEE Computer*, 45, 2 (2012).

Conventional database wisdom dictates that serializable transactions are the canonical solution to the problem of concurrent programming, but this is seldom the case in real-world databases. In practice, database systems instead overwhelmingly implement concurrency control that is non-serializable, exposing users to the possibility that their transactions will not appear to have executed in some serial order. In this chapter, we discuss why the use of this so-called "weak isolation" is so widespread, what these non-serializable isolation modes actually do, and why reasoning about them is so difficult.

Overview and Prevalence

Even in the earliest days of database systems (see Chapter 3), systems builders realized that implementing serializability is expensive. The requirement that transactions appear to execute sequentially has profound consequences for the degree of concurrency a database can achieve. If transactions access disjoint sets of data items in the database, serializability is effectively "free": under these disjoint access patterns, a serializable schedule admits data parallelism. However, if transactions contend on the same data items, in the worst case, the system cannot process them with any parallelism whatsoever. This property is fundamental to serializability and independent of the actual implementation: because transactions cannot safely make progress independently under all workloads (i.e., they must *coordinate*), any implementation of serializability may, in effect, require serial execution. In practice, this means that transactions may need to wait, decreasing throughput while increasing latency. Transaction processing expert Phil Bernstein suggests that serializability typically incurs a three-fold performance penalty on a single-node database compared to one of the most common weak isolation levels called Read Committed [29]. Depending on the implementation, serializability may also lead to more aborts, restarted transactions, and/or deadlocks. In distributed databases, these costs increase because networked communication is expensive, increasing the time required to execute serial critical sections (e.g., holding locks); we have observed multiple order-of-magnitude performance penalties under adverse conditions [20].

As a result, instead of implementing serializability, database system designers instead often implemented weaker models. Under weak isolation, transactions are not guaranteed to observe serializable behavior. Instead, transactions will observe a range of anomalies (or "phenomena"): behaviors that could not have arisen in a serial execution. The exact anomalies depend on which model is provided, but example anomalies include reading intermediate data that another transaction produced, reading aborted data, reading two or more different values for the same item during execution of the same transaction, and "losing" some effects of transactions due to concurrent writes to the same item.

These weak isolation modes are surprisingly prevalent. In a recent survey of eighteen SQL and "NewSQL" databases [18], we found that only three of eighteen provided serializability by default and eight (including Oracle and SAP's flagship offerings) did not offer serializability at all! This state of affairs is further complicated by often inaccurate use of terminology: for example, Oracle's "serializable" isolation guarantee actually provides Snapshot Isolation, a weak isolation mode [59]. There is also a race to to bottom among vendors. Anecdotally, when vendor A, a major player in the transaction processing market, switched its default isolation mode from serializability to Read Committed, vendor B, who still defaulted to serializability, began to lose sales contracts during bake-offs with vendor A. Vendor B's database was clearly slower, so why would the customer choose B instead of A? Unsurprisingly, vendor B now provides Read Committed isolation by default, too.

The Key Challenge: Reasoning about Anomalies

The primary reason why weak isolation is problematic is that, depending on the application, weak isolation anomalies can result in application-level inconsistency: the invariants that each transaction preserves in a serializable execution may no longer hold under weak isolation. For example, if two users attempt to withdraw from a bank account at the same time and their transactions run under a weak isolation mode allowing concurrent writes to the same data item (e.g., the common Read Committed model), the users may successfully withdraw more money than the account contained (i.e., each reads the current amount, each calculates the amount following their withdrawal, then each writes the "new" total to the database). This is not a hypothetical scenario. In a recent, colorful example, an attacker systematically exploited weak isolation behavior in the Flexcoin Bitcoin exchange; by repeatedly and programmatically triggering non-transactional read-modify-write behavior in the Flexcoin application (an vulnerability under Read Committed isolation and, under a more sophisticated access pattern, Snapshot Isolation), the attacker was able to withdraw more Bitcoins than she should have, thus bankrupting the exchange [2].

Perhaps surprisingly, few developers I talk with regarding their use of transactions are even aware that they are running under non-serializable isolation. In fact, in our research, we have found that many open-source ORM-backed applications assume serializable isolation, leading to a range of possible application integrity violations when deployed on commodity database engines [19]. The developers who are aware of weak isolation tend to employ a range of alternative techniques at the application level, including explicitly acquiring locks (e.g., SQL "SELECT FOR UPDATE") and introducing false conflicts (e.g., writing to a dummy key under Snapshot Isolation). This is highly error-prone and negates many of the benefits of the transaction concept.

Unfortunately, specifications for weak isolation are often incomplete, ambiguous, and even inaccurate. These specifications have a long history dating to the 1970s. While they have improved over time, they remain problematic.

The earliest weak isolation modes were specified operationally: as we saw in Chapter 3, popular models like Read Committed were originally invented by modifying the duration for which read locks were held [72]. The definition of Read Committed was: "hold read locks for a short duration, and hold write locks for a long duration."

The ANSI SQL Standard later attempted to provide an implementation-independent description of several weak isolation modes that would apply not only to lock-based mechanisms but also to multi-versioned and optimistic methods as well. However, as Gray et al. describe in [27], the SQL Standard is both ambiguous and underspecified: there are multiple possible interpretations of the English-language description, and the formalization does not capture all behaviors of the lock-based implementations. Additionally, the ANSI SQL Standard does not cover all isolation modes: for example, vendors had already begun shipping production databases providing Snapshot Isolation (and labeling it as serializable!) before Gray et al. defined it in their 1995 paper. (Sadly, as of 2015, the ANSI SQL Standard remains unchanged.)

To complicate matters further, Gray et al.'s 1995 revised formalism is also problematic: it focuses on lock-related semantics and rules out behavior that might be considered safe in a multi-versioned concurrency control system. Accordingly, for his 1999 Ph.D. thesis [6], Atul Adya introduced the best formalism for weak isolation that we have to date. Adya's thesis adapts the formalism of multi-version serialization graphs [28] to the domain of weak isolation and describes anomalies according to restrictions on those graphs. We include Adya's corresponding ICDE 2000 paper, but isolation aficionados should consult the full thesis. Unfortunately, Adya's model is still underspecified in some cases (e.g., what exactly does G0 mean if no reads are involved?), and implementations of these guarantees differ across databases.

Even with a perfect specification, weak isolation is still a real challenge to reason about. To decide whether weak isolation is "safe," programmers must mentally translate their application-level consistency concerns down to low-level read and write behavior [11]. This is ridiculously difficult, even for seasoned concurrency control experts. In fact, one might wonder what benefits of transactions remain if serializability is compromised?

Why is it easier to reason about Read Committed isolation than no isolation at all? Given how many database engines like Oracle run under weak isolation, how does modern society function at all – whether users are booking airline flights, administering hospitals, or performing stock trades? The literature lends few clues, casting serious questions about the success of the transaction concept as deployed in practice today.

The most compelling argument I have encountered for why weak isolation seems to be "okay" in practice is that few applications today experience high degrees of concurrency. Without concurrency, most implementations of weak isolation deliver serializable results. This in turn has led to a fruitful set of research results. Even in a distributed setting, weakly isolated databases deliver "consistent" results: for example, at Facebook, only 0.0004% of results returned from their eventually consistent store were "stale" [106], and others have found similar results [23, 159]. However, while for many applications weak isolation is apparently not problematic, it can be: as our Flexcoin example illustrates, given the possibility of errors, application writers must be vigilant in accounting for (or otherwise explicitly ignoring) concurrency-related anomalies.

Weak Isolation, Distribution, and "NoSQL"

With the rise of Internet-scale services and cloud computing, weak isolation has become even more prevalent. As I mentioned earlier, distribution exacerbates overheads of serializability, and, in the event of partial system failures (e.g., servers crashing), transactions may stall indefinitely. As more and more programmers began to write distributed applications and used distributed databases, these concerns became mainstream.

The past decade saw the introduction of a range of new data stores optimized for the distributed environment, collectively called "NoSQL." The "NoSQL" label is unfortunately overloaded and refers to many aspects of these stores, from lack of literal SQL support to simpler data models (e.g., key-value pairs) and little to no transactional support. Today, as in MapReduce-like systems (Chapter 5), NoSQL stores are adding many these features. However, a notable, fundamental difference is that these NoSQL stores frequently focus on providing better availability of operations via weaker models, with an explicit focus on fault tolerance. (It is somewhat ironic that, while NoSQL stores are commonly associated with the use of non-serializable guarantees, classic RDBMSs do not provide serializability by default either.)

As an example of these NoSQL stores, we include a paper on the Dynamo system, from Amazon, presented at SOSP 2007. Dynamo was introduced to provide highly available and low latency operation for Amazon's shopping cart. The paper is technically interesting as it combines several techniques, including quorum replication, Merkle tree anti-entropy, consistent hashing, and version vectors. The system is entirely non-transactional, does not provide any kind of atomic operation (e.g., compare and swap), and relies on the application writer to reconcile divergent updates. In the limit, any node can update any item (under hinted handoff).

By using a merge function, Dynamo adopts an "optimistic replication" policy: accept writes first, reconcile divergent versions later [138, 70]. On the one hand, presenting a set of divergent versions to the user is more friendly than simply discarding some concurrent updates, as in Read Committed isolation. On the other hand, programmers must reason about merge functions. This raises many questions: what is a suitable merge for an application? How do we avoid throwing away committed data? What if an operation should not have been performed concurrently in the first place? Some open source Dynamo clones, like Apache Cassandra, do not provide merge operators and simply choose a "winning" write based on a numerical timestamp. Others, like Basho Riak, have adopted "libraries" of automatically mergeable datatypes like counters, called Commutative Replicated Data Types [142].

Dynamo also does not make promises about recency of reads. Instead, it guarantees that, if writes stop, eventually all replicas of a data item will contain the same set of writes. This eventual consistency is a remarkably weak guarantee: technically, an eventually consistent datastore can return stale (or even garbage) data for an indefinite amount of time [22]. In practice, data store deployments frequently return recent data [159, 23], but, nevertheless, users must reason about non-serializable behavior. Moreover, in practice, many stores offer intermediate forms of isolation called "session guarantees" that ensure that users read their own writes (but not the writes of other users); interestingly, these techniques were developed in the early 1990s as part of the Bayou project on mobile computing and have recently come to prominence again [154, 153].

Trade-offs and the CAP Theorem

We have also included Brewer's 12 year retrospective on the CAP Theorem. Originally formulated following Brewer's time building Inktomi, one of the first scalable search engines, Brewer's CAP Theorem pithily describes trade-offs between the requirement for coordination (or "availability") and strong guarantees like serializability. While earlier results described this trade-off [91, 47], CAP became a rallying cry for mid-2000s developers and has considerable impact. Brewer's article briefly discusses performance implications of CAP, as well as the possibility of maintaining some consistency criteria without relying on coordination.

Programmability and Practice

As we have seen, weak isolation is a real challenge: its performance and availability benefits mean it is extremely popular in deployments despite the fact that we have little understanding of its behavior. Even with a perfect specification, existing formulations of weak isolation would still be a extremely difficult to reason about. To decide whether weak isolation is "safe," programmers must mentally translate their application-level consistency concerns down to low-level read and write behavior [11]. This is ridiculously difficult, even for seasoned concurrency control experts.

As a result, I believe there is a serious opportunity to investigate semantics that are not subject to the performance and availability overheads of serializability but are more intuitive, usable, and programmable than existing guarantees. Weak isolation has historically been highly challenging to reason about, but this need not be the case. We and others have found that several high-value use cases, including index and view maintenance, constraint maintenance, and distributed aggregation, frequently do not actually require coordination for "correct" behavior; thus, for these use cases, serializability is overkill [17, 21, 136, 142]. That is, by providing databases with additional knowledge about their applications, database users can have their cake and eat it too. Further identifying and exploiting these use cases is an area ripe for research.

Conclusions

In summary, weak isolation is prevalent due to its many benefits: less coordination, higher performance, and greater availability. However, its semantics, risks, and usage are poorly understood, even in an academic context. This is particularly baffling given the amount of research devoted to serializable transaction processing, which is considered by many to be a "solved problem." Weak isolation is arguably even more deserving of such a thorough treatment. As I have highlighted, many challenges remain: how do modern systems even work, and how should users program weak isolation? For now, I offer the following take-aways:

- Non-serializable isolation is prevalent in practice (in both classical RDBMSs and recent NoSQL upstarts) due to its concurrency-related benefits.
- Despite this prevalence, many existing formulations of non-serializable isolation are poorly specified and difficult to use.
- Research into new forms of weak isolation show how to preserve meaningful semantics and improve programmability without the expense of serializability.

Chapter 7: Query Optimization

Introduced by Joe Hellerstein

Selected Readings:

Goetz Graefe and William J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. *ICDE*, 1993.

Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. *SIGMOD*, 2000.

Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, Miso Cilimdzic. Robust Query Processing Through Progressive Optimization. *SIGMOD*, 2004.

Query optimization is one of the signature components of database technology—the bridge that connects declarative languages to efficient execution. Query optimizers have a reputation as one of the hardest parts of a DBMS to implement well, so it's no surprise they remain a clear differentiator for mature commercial DBMSs. The best of the open-source relational database optimizers are limited by comparison, and some have relatively naive optimizers that only work for the simplest of queries.

It's important to remember that no query optimizer is truly producing "optimal" plans. First, they all use estimation techniques to guess at real plan costs, and it's well known that errors in these estimation techniques can balloon—in some circumstances being as bad as random guesses [88]. Second, optimizers use heuristics to limit the search space of plans they choose, since the problem is NP-hard [86]. One assumption that's gotten significant attention recently is the traditional use of 2-table join operators; this has been shown to be theoretically inferior to new multi-way join algorithms in certain cases [121].

Despite these caveats, relational query optimization has proven successful, and has enabled relational database systems to serve a wide range of bread-and-butter use cases fairly well in practice. Database vendors have invested many years into getting their optimizers to perform reliably on a range of use cases. Users have learned to live with limitations on the number of joins. Optimizers still, for the most part, make declarative SQL queries a far better choice than imperative code for most uses.

In addition to being hard to build and tune, serious query optimizers also have a tendency to grow increasingly complex over time as they evolve to handle richer workloads and more corner cases. The research literature on database query optimization is practically a field unto itself, full of technical details—many of which have been discussed in the literature by researchers at mature vendors like IBM and Microsoft who work closely with product groups. For this book, we focus on the big picture: the main architectures that have been considered for query optimization and how have they been reevaluated over time.

Volcano/Cascades

We begin with the state of the art. There are two reference architectures for query optimization from the early days of database research that cover most of the serious optimizer implementations today. The first is Selinger et al.'s System R optimizer described in Chapter 3. System R's optimizer is textbook material, implemented in many commercial systems; every database researcher is expected to understand it in detail. The second is the architecture that Goetz Graefe and his collaborators refined across a series of research projects: Exodus, Volcano, and Cascades. Graefe's work is not covered as frequently in the research literature or the textbooks as the System R work, but it is widely used in practice, notably in Microsoft SQL Server, but purportedly in a number of other commercial systems as well. Graefe's papers on the topic have something of an insider's flavor—targeted for people who know and care about implementing query optimizers. We chose the Volcano paper for this book as the most approachable representative of the work, but aficionados should also read the Cascades paper [65]—not only does it raise and address a number of detailed deficiencies of Volcano, but it's the latest (and hence standard) reference for the approach. Recently, two open-source Cascades-style optimizers have emerged: Greenplum's Orca optimizer is now part of the Greenplum open source, and Apache Calcite is an optimizer that can be used with multiple backend query executors and languages, including LINQ.

Graefe's optimizer architecture is notable for two main reasons. First, it was expressly designed to be extensible. Volcano deserves credit for being quite forward-looking—long predating MapReduce and the big data stacks—in exploring the idea that dataflow could be useful for a wide range of data-intensive applications. As a result, the Graefe optimizers are not just for compiling SQL into a plan of dataflow iterators. They can be parameterized for other input languages and execution targets; this is a highly relevant topic in recent years with the rise of specialized data models and languages on the one hand (see Chapter 2 and 9), and specialized execution engines on the other (Chapter 5). The second innovation in these optimizers was the use of a top-down or goal-oriented search strategy for finding the cheapest plan in the space of possible plans. This design choice is connected to the extensibility API in Graefe's designs, but that is not intrinsic: the Starburst system showed how to do extensibility for Selinger's bottom-up algorithm [103]. This "top-down" vs "bottom-up" debate for query optimization has advocates on both sides, but no clear winner; a similar top-down/bottom-up debate came out to be more or less a tie in the recursive query processing literature as well [128]. Aficionados will be interested to note that these two bodies of literature–recursive query processing and query optimizer search–were connected directly in the Evita Raced optimizer, which implemented both top-down and bottom-up optimizer search by using recursive queries as the language for implementing an optimizer [43].

Adaptive Query Processing

By the late 1990's, a handful of trends suggested that the overall architecture of query optimization deserved a significant rethink. These trends included:

- Continuous queries over streaming data.
- Interactive approaches to data exploration like Online Aggregation.
- Queries over data sources that are outside the DBMS and do not provide reliable statistics or performance.
- Unpredictable and dynamic execution environments, including elastic and multitenant settings and widely distributed systems like sensor networks.
- Opaque data and user-defined functions in queries, where statistics can only be estimated by observing behavior.

In addition, there was ongoing practical concern about the theoretical fact that plan cost estimation was often erratic for multi-operator queries [88]. As a result of these trends, interest emerged in adaptive techniques for processing queries, where execution plans could change mid-query. We present two complementary points in the design space for adaptive query processing; there is a long survey with a more comprehensive overview [52].

Eddies

The work on eddies, represented by our second paper, pushed hard on the issue of adaptivity: if query "re-planning" has to occur mid-execution, why not remove the architectural distinction between planning and execution entirely? In the eddies approach, the optimizer is encapsulated as a dataflow operator that is itself interposed along other dataflow edges. It can monitor the rates of dataflow along those edges, so it has dynamic knowledge of their behavior, with whatever history it cares to record. With that ongoing flow of information, it can dynamically control the rest of the aspects of query planning via dataflow routing: the order of commutative operators is determined by the order tuples are routed through operators (the focus of the first eddies paper that we include here) the choice of physical operators (e.g. join algorithms, index selection) is determined by routing tuples among multiple alternative, potentially redundant physical operators in the flow [129, 51] the scheduling of operators is determined by buffering inputs and deciding which output to deliver to next [131]. As an extension, multiple queries can be scheduled by interposing on their flows and sharing common operators [109]. Eddies intercept the ongoing dataflow of query operators while they are in flight, pipelining data from their inputs to their output. For this reason it's important that eddy routing be implemented efficiently; Deshpande developed implementation enhancements along these lines [50]. The advantage of this pipelined approach is that eddies can adaptively change strategies in the middle of executing a pipelined operator like a join, which is useful if a query operator is either very long-lived (as in a streaming system) or a very poor choice that should be abandoned long before it runs to completion. Interestingly, the original Ingres optimizer also had the ability to make certain query optimization decisions on a per-tuple basis [161].

Progressive Optimization

The third paper in this section from IBM represents a much more evolutionary approach, which extends a System R style optimizer with adaptivity features; this general technique was pioneered by Kabra and DeWitt [93] but receives a more complete treatment here. Where eddies focused on intra-operator reoptimization (while data is "in motion"), this work focuses on inter-operator reoptimization (when data is "at rest"). Some of the traditional relational operators including sorting and most hash-joins are blocking: they consume their entire input before producing any output. This presents an opportunity after input is consumed to compare observed statistics to optimizer predictions, and reoptimize the "remainder" of the query plan using traditional query optimization technique. The downside of this approach is that it does no reoptimization while operators are consuming their inputs, making it inappropriate for continuous queries over streams, for pipelining operators like symmetric hash join [160] or for long-running relational queries that have poorly-chosen operators in the initial parts of the plan – e.g. when data is being accessed from data sources outside the DBMS that do not provide useful statistics [116, 157].

It's worth noting that these two architectures for adaptivity could in principle coexist: an eddy is "just" a dataflow operator, meaning that a traditional optimizer can generate a query plan with an eddy connecting a set of streaming operators, and also do reoptimization at blocking points in the dataflow in the manner of our third paper.

Discussion

This brings us to a discussion of current trends in dataflow architectures, especially in the open source big data stack. Google MapReduce set back by a decade the conversation about adaptivity of data in motion, by baking blocking operators into the execution model as a fault-tolerance mechanism. It was nearly impossible to have a reasoned conversation about optimizing dataflow pipelines in the mid-to-late 2000's because it was inconsistent with the Google/Hadoop fault tolerance model. In the last few years the discussion about execution frameworks for big data has suddenly opened up wide, with a quickly-growing variety of dataflow and query systems being deployed that have more similarities than differences (Tenzing, F1, Dremel, DryadLINQ, Naiad, Spark, Impala, Tez, Drill, Flink, etc.) Note that all of the motivating issues for adaptive optimization listed above are very topical in today's big data discussion, but not well treated.

More generally, I would say that the "big data" community in both research and open source has been far too slow to focus on query optimization, to the detriment of both the current systems and the query optimization field. To begin with, the "hand-planned" MapReduce programming model remained a topic of conversation for far longer than it should have. It took a long time for the Hadoop and systems research communities to accept that a declarative language like SQL or LINQ is a good general-purpose interface, even while maintaining low-level MapReduce-style dataflow programming as a special-case "fast path". More puzzling is the fact that even when the community started building SQL interfaces like Hive, query optimization remained a little-discussed and poorly-implemented topic. Maybe it's because query optimizers are harder to build well than query executors. Or maybe it was fallout from the historical quality divide between commercial and open source databases. MySQL was the open source de facto reference for "database technology" for the preceding decade, with a naive heuristic optimizer. Perhaps as a result, many (most?) open source big data developers didn't understand—or trust—query optimizer technology.

In any case, this tide is turning in the big data community. Declarative queries have returned as the primary interface to big data, and there are efforts underway in essentially all the projects to start building at least a 1980's-era optimizer. Given the list of issues I mention above, I'm confident we'll also see more innovative query optimization approaches deployed in new systems over the coming years.

Chapter 8: Interactive Analytics

Introduced by Joe Hellerstein

Selected Readings:

Venky Harinarayan, Anand Rajaraman, Jeffrey D. Ullman. Implementing Data Cubes Efficiently. *SIGMOD*, 1996.

Yihong Zhao, Prasad M. Deshpande, Jeffrey F. Naughton. An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. *SIGMOD*, 1997.

Joseph M. Hellerstein, Ron Avnur, Vijayshankar Raman. Informix under CONTROL: Online Query Processing. *Data Mining and Knowledge Discovery*, 4(4), 2000, 281-314.

Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, Ion Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. *EuroSys*, 2013.

For decades, most database workloads have been partitioned into two categories: (1) many small "transaction processing" queries that do lookups and updates on a small number of items in a large database, and (2) fewer big "analytic" queries that summarize large volumes of data for analysis. This section is concerned with ideas for accelerating the second category of queriesparticularly to answer them at interactive speeds, and allow for summarization, exploration and visualization of data.

Over the years there has been a great deal of buzzword bingo in industry to capture some or all of this latter workload, from "Decision Support Systems" (DSS) to "Online Analytic Processing" (OLAP) to "Business Intelligence" (BI) to "Dashboards" and more generally just "Analytics". Billions of dollars of revenue have been associated with these labels over time, so marketers and industry analysts worked hard over the years to define, distinguish and sometimes try to subvert them. By now it's a bit of a mess of nomenclature. The interested reader can examine Wikipedia and assess conventional wisdom on what these buzzwords came to mean and how they might be different; be warned that it will not be a scientifically satisfying exercise.

Here, I will try to keep things simple and somewhat technically grounded.

Human cognition cannot process large amounts of raw data. In order for a human to make sense of data, the data has to be "distilled" in some way to a relatively small set of records or visual marks. Typically this is done by partitioning the data and running simple arithmetic aggregation functions on the partitions think of SQL's "GROUP BY" functionality as a canonical pattern[1]. Subsequently the data typically needs to be visualized for users to relate it to their task at hand.

The primary challenge we address in this chapter is to make large-scale grouping/aggregation queries run at interactive speeds—even in cases where it is not feasible to iterate through all the data associated with the query.

How do we make a query run in less time than it takes to look at the data? There is really only one answer: we answer the query without looking at (all) the data. Two variants of this idea emerge:

Precomputation: If we know something about the query workload in advance, we can distill the data in various ways to allow us to support quick answers (either accurate or approximate) to certain queries. The simplest version of this idea is to precompute the answers to a set of queries, and only support those queries. We discuss more sophisticated answers below. Sampling: If we cannot anticipate the queries well in advance, our only choice is to look at a subset of the data at query time. This amounts to sampling from the data, and approximating the true answer based on the sample.

The papers in this section focus on one or both of these approaches.

Our first two papers address what the database

[1]Database-savvy folks take GROUP BY and aggregation for granted. In statistical programming packages (e.g., R's plyr library, or Python's pandas), this is apparently a relatively new issue, referred to as the "Split-Apply-Combine Strategy". A wrinkle in that context is the need to support both array and table notation.

community dubbed "data cubes" [DataCubes]. Data cubes were originally supported by standalone query/visualization tools called On Line Analytic Processing (OLAP) systems. The name is due to relational pioneer Ted Codd, who was hired as a consultant to promote an early OLAP vendor called Essbase (subsequently bought by Oracle). This was not one of Codd's more scholarly endeavors.

Early OLAP tools used a pure "precomputation" approach. They ingested a table, and computed and stored the answers to a family of GROUP BY queries over that table: each query grouped on a different subset of the columns, and computed summary aggregates over the non-grouped numerical columns. For example, in a table of car sales, it might show total sales by Make, total sales by Model, total sales by Region, and total sales by combinations of any 2 or 3 of those attributes. A graphical user interface allowed users to navigate the resulting space of group-by queries interactivelythis space of queries is what became known as a data cube[2]. Originally, OLAP systems were pitched as standalone "Multidimensional Databases" that were fundamentally different than relational databases. However, Jim Gray and a consortium of authors in the relational industry explained how the notion of a data cube can fit in the relational context [68], and the concept subsequently migrated into the SQL standard as a single query construct: "CUBE BY". There is also a standard alternative to SQL called MDX that is less verbose for OLAP purposes. Some of the terminology from data cubes has become common parlance—in particular, the idea of "drilling down" into details and "rolling up" to coarser summaries.

A naive relational precomputation approach for precomputing a full data cube does not scale well. For a table with k potential grouping columns, such an approach would have to run and store the results for 2^k GROUP BY queries, one for each subset of the columns. Each query would require a full pass of the table.

Our first paper by Harinarayan, Rajaraman and Ullman reduces this space: it chooses a judicious subset of queries in the cube that are worthy of precomputation; it then uses the results of those queries to compute the results to any other query in the cube. This paper is one of the most-cited papers in the area, in part because it was early in observing that the structure of the data cube problem is a set-containment lattice. This lattice structure underlies their solution, and recurs in many other papers on data cubes (including our next paper), as well as on certain data mining algorithms like Association Rules (a.k.a. Frequent Itemsets) [7]. Everyone working in the OLAP space should have read this paper.

Our second paper by Zhao, Deshpande and Naughton focuses on the actual computation of results in the cube. The paper uses an "array-based" approach: that is, it assumes the data is stored in an Essbase-like sparse array structure, rather than a relational table structure, and presents a very fast algorithm that exploits that structure. However, it makes the surprising observation that even for relational tables, it is worthwhile to convert tables to arrays in order to run this algorithm, rather than to run a (far less efficient) traditional relational algorithm. This substantially widens the design space for query engines. The implication is that you can decouple your data model from the internal model of your query engine. So a special-purpose "engine" (Multidimensional OLAP in this case) may add value by being embedded in a more general-purpose engine (Relational in this case). Some years after the OLAP wars, Stonebraker started arguing that "one size doesn't fit all" for database engines, and hence that specialized database engines (not unlike Essbase) are indeed important [149]. This paper is an example of how that line of reasoning sometimes plays out: clever specialized techniques get developed, and if they're good enough they can pay off in more general contexts as well. Innovating on both sides of that line—specialization and generalization—has led to good research results over the years. Meanwhile, anyone building a query engine should keep in mind the possibility that the internal representations of data and operations can be a superset of the representations of the API.

Related to this issue is the fact that analytic databases have become much more efficient in the last decade due to in-database compression, and the march of Moore's Law. Stonebraker has asserted to me that column stores make OLAP accelerators irrelevant. This is an interesting argument to consider, though hasn't been validated by the market. Vendors still build cubing engines, and BI tools commonly implement them as accelerators on top of relational databases and Hadoop. Certainly the caching techniques of our first paper remain relevant. But the live query processing tradeoffs

[2]Note that this idea did not originate in databases. In statistics, and later in spreadsheets, there is an old, well-known idea of a contingency table or cross tabulation (crosstab).

between high-performance analytic database techniques and data cubing techniques may deserve a revisit.

Our third paper on "online aggregation" starts exploring from the opposite side of the territory from OLAP, attempting to handle ad-hoc queries quickly without precomputation by producing incrementally refining approximate answers. The paper was inspired by the kind of triage that people perform every day in gathering evidence to make decisions; rather than prespecifying hard deadlines, we often make qualitative decisions about when to stop evaluating and to act. Specific data-centric examples include the "early returns" provided in election coverage, or the multiresolution delivery of images over low-bandwidth connectionsin both cases we have a good enough sense of what is likely to happen long before the process completed.

Online aggregation typically makes use of sampling to achieve incrementally refining results. This is not the first (or last!) use of database sampling to provide approximate query answers. (Frank Olken's thesis [122] is a good early required reference for database sampling.) But online aggregation helped kick off an ongoing sequence of work on approximate query processing that has persisted over time, and is of particular interest in the current era of Big Data and structure-on-use.

We include the first paper on online aggregation here. To appreciate the paper, it's important to remember that databases at the time had long operated under a mythology of "correctness" that is a bit hard to appreciate in today's research environment. Up until approximately the 21st century, computers were viewed by the general populace—and the business community—as engines of accurate, deterministic calculation. Phrases like "Garbage In, Garbage Out" were invented to remind users to put "correct" data into the computer, so it could do its job and produce "correct" outputs. In general, computers weren't expected to produce "sloppy" approximate results.

So the first battle being fought in this paper is the idea that the complete accuracy in large-scale analytics queries is unimportant, and that users should be able to balance accuracy and running time in a flexible way. This line of thinking quickly leads to three research directions that need to work in harmony: fast query processing, statistical approximation, and user interface design. The inter-dependencies of these three themes make for an interesting design space that researchers and products are still exploring today.

The initial paper we include here explores how to embed this functionality in a traditional DBMS. It also provides statistical estimators for standard SQL aggregates over samples, and shows how stratified sampling can be achieved using standard B-trees via "index striding", to enable different groups in a GROUP BY query to be sampled at different rates. Subsequent papers in the area have explored integrating online aggregation with many of the other standard issues in query processing, many of which are surprisingly tricky: joins, parallelism, subqueries, and more recently the specifics of recent Big Data systems like MapReduce and Spark.

Both IBM and Informix pursued commercial efforts for online aggregation in the late 1990s, and Microsoft also had a research agenda in approximate query processing as well. None of these efforts came to market. One reason for this at the time was the hidebound idea that "database customers won't tolerate wrong answers"[3]. A more compelling reason related to the coupling of user interface with query engine and approximation. At that time, many of the BI vendors were independent of the database vendors. As a result, the database vendors didn't "own" the end-user experience in general, and could not deliver the online aggregation functionality directly to users via standard APIs. For example, traditional query cursor APIs do not allow for multiple approximations of the same query, nor do they support confidence intervals associated with aggregate columns. The way the market was structured at the time did not support aggressive new technology spanning both the back-end and front-end.

Many of these factors have changed today, and online aggregation is getting a fresh look in research and in industry. The first motivation, not surprisingly, is the interest in Big Data. Big Data is not only large in volume, but also has wide "variety" of formats and uses which means that it may not be parsed and understood until users want to analyze. For exploratory analytics on Big Data, the combination of large volumes and schema-on-use makes precomputation unattractive or impossible. But sampling on-the-fly remains cheap and useful.

In addition, the structure of the industry and its interfaces has changed since the 1990s. From the bottom up, query engine standards today often emerge and evolve through open source development, and the winning projects (e.g., Hadoop and Spark) become close

[3]This was particularly ironic given that the sampling support provided by some of the vendors was biased (by sampling blocks instead of tuples).

enough to monopolies that their APIs can dictate client design. At the same time from the top down, hosted data visualization products in the cloud are often vertically integrated: the front-end experience is the primary concern, and is driven by a (often special-purpose) back-end implementation without concern for standardization. In both cases, it's possible to deliver a unique feature like online aggregation through the stack from engine to applications.

In that context we present one of the more widely-read recent papers in the area, on BlinkDB. The system makes use of what Olken calls "materialized sample views": precomputed samples over base tables, stored to speed up approximate query answering. Like the early OLAP papers, BlinkDB makes the case that only a few GROUP BY clauses need to be precomputed to get good performance on (stable) workloads. Similar arguments are made by the authors of the early AQUA project on approximate queries [5], but they focused on precomputed synopses ("sketches") rather than materialized sample views as their underlying approximation mechanism. The BlinkDB paper also makes the case for stratification in its views to capture small groups, reminiscent of the Index Striding in the online aggregation paper. BlinkDB has received interest in industry, and the Spark team has recently proposed augmenting its precomputed samples with sampling on the fly—a sensible mixture of techniques to achieve online aggregation as efficiently as possible. Recent commercial BI tools like ZoomData seem to use online aggregation as well (they call it "query sharpening").

With all this discussion of online aggregation, it's worth taking a snapshot of current market realities. In the 25 years since it was widely introduced, OLAP-style precomputation has underpinned what is now a multi-billion dollar BI industry. By contrast, approximation at the user interface is practically non-existent. So for those of you keeping score at home based on revenue generation, the simple solution of precomputation is the current winner by a mile. It's still an open question when and if approximation will become a bread-and-butter technique in practice. At a technical level, the fundamental benefits of sampling seem inevitably useful, and the technology trends around data growth and exploratory analytics make it compelling in the Big Data market. But today this is still a technology that is before its time.

A final algorithmic note: approximate queries via sketches are in fact very widely used by engineers and data scientists in the field today as building blocks for analysis. Outside of the systems work covered here, well-read database students should be familiar with techniques like CountMin sketches, HyperLogLog sketches, Bloom filters, and so on. A comprehensive survey of the field can be found in [44]; implementations of various sketches can be found in a number of languages online, including as user-defined functions in the MADlib library mentioned in Chapter 11.

Chapter 9: Languages

Introduced by Joe Hellerstein

Selected Readings:

Joachim W. Schmidt. Some High Level Language Constructs for Data of Type Relation. *ACM Transactions on Database Systems*, 2(3), 1977, 247-261.

Arvind Arasu, Shivnath Babu, Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 15(2), 2006, 121-142.

Peter Alvaro, Neil Conway, Joseph M. Hellerstein, William R. Marczak. Consistency Analysis in Bloom: A CALM and Collected Approach. *CIDR*, 2011.

From reading database papers, you might expect that typical database users are data analysts, business decision-makers, or IT staff. In practice, the majority of database users are software engineers, who build database-backed applications that are used further up the stack. Although SQL was originally designed with non-technical users in mind, it is rare for people to interact directly with a database via a language like SQL unless they are coding up a database-backed application.

So if database systems are mostly just APIs for software development, what do they offer programmers? Like most good software, database systems offer powerful abstractions.

Two stand out:

1. The transaction model gives programmers the abstraction of a single-process, sequential machine that never fails mid-task. This protects programmers from a gigantic cliff of complexity—namely, the innate parallelism of modern computing. A single compute rack today has thousands of cores running in parallel across dozens of machines that can fail independently. Yet application programmers can still blithely write sequential code as if it were 1965 and they were loading a deck of punch-cards into a mainframe to be executed individually from start to finish.

2. Declarative query languages like SQL provide programmers with abstractions for manipulating sets of data. Famously, declarative languages shield programmers from thinking about *how* to access data items, and instead let them focus on *what* data items to return. This data independence also shields application programmers from changes in the organization of underlying databases, and shields database administrators from getting involved in the design and maintenance of applications.

Just how useful have these abstractions been over time? How useful are they today?

1. As a programming construct, serializable transactions have been very influential. It is easy for programmers to bracket their code with BEGIN and COMMIT/ROLLBACK. Unfortunately, as we discussed in Chapter 6, transactions are expensive, and are often compromised. "Relaxed" transactional semantics break the serial abstraction for users and expose application logic to the potential for races and/or unpredictable exceptions. If application developers want to account for this, they have to manage the complexity of concurrency, failure, and distributed state. A common response to the lack of transactions is to aspire to "eventual consistency" [154], as we discuss in the weak isolation section. But as we discussed in Chapter 6, this still shifts all the correctness burdens to the application developer. In my opinion, this situation represents a major crisis in modern software development.

2. Declarative query languages have also been a success—certainly an improvement over the navigational languages that preceded them, which led to spaghetti code that needed to be rewritten every time you reorganized the database. Unfortunately, query languages are quite different from the imperative languages that programmers usually use. Query languages consume and pro-

duce simple unordered "collection types" (sets, relations, streams); programming languages typically carry out ordered execution of instructions, often over complex structured data types (trees, hashtables, etc.). Programmers of database applications are forced to bridge this so-called "impedance mismatch" between programs and database queries. This has been a hassle for database programmers since the earliest days of relational databases.

Database Language Embeddings: Pascal/R

The first paper in this section presents a classical example of tackling the second problem: helping imperative programmers with the impedance mismatch. The paper begins by defining operations for what we might now recognize (40-odd years later!) as familiar collection types: the "dictionary" type in Python, the "map" type in Java or Ruby, etc. The paper then patiently takes us through the possibilities and pitfalls of various language constructs that seem to recur in applications across the decades. A key theme is a desire for differentiating between enumeration (for generating output) and quantification (for checking properties)—the latter can often be optimized if you are explicit. In the end, the paper proposes a declarative, SQL-like sublanguage for Relation types that is embedded into Pascal. The result is relatively natural and not unlike some of the better interfaces today.

Although this approach seems natural now, the topic took decades to gain popular attention. Along the way, database "connectivity" APIs like ODBC and JDBC arose as a crutch for C/C++ and Java—they allowed users to push queries to the DBMS and iterate through results, but the type systems remained separate, and bridging from SQL types to host language types was unpleasant. Perhaps the best modern evolution of ideas like Pascal/R is Microsoft's LINQ library, which provides language-embedded collection types and functions so application developers can write query-like code over a variety of backend databases and other collections (XML documents, spreadsheets, etc.) We included a taste of LINQ syntax in the DryadLINQ paper in Chapter 5.

In the 2000's, web applications like social media, online forums, interactive chat, photo sharing and product catalogs were implemented and reimplemented over relational database backends. Modern scripting languages for web programming were a bit more convenient than Pascal, and typically included decent collection types. In that environment, application developers eventually saw recognized patterns in their code and codified them into what are now called Object-Relational Mappings (ORMs). Ruby on Rails was one of the most influential ORMs to begin with, though by now there are many others. Every popular application programming language has at least one, and there are variations in features and philosophy. The interested reader is referred to Wikipedia's "List of object-relational mapping software" wiki.

ORMs do a few handy things for the web programmer. First they provide language-native primitives for working with collections much like Pascal/R. Second they can enable updates to in-memory language objects to be transparently reflected in the database-backed state. They often offer some language-native syntax for familiar database design concepts like entities, relationships, keys and foreign keys. Finally, some ORMs including Rails offer nice tools for tracking the way that database schemas evolve over time to reflect changes in the application code ("migrations" in Rails terminology).

This is an area where the database research community and industry should pay more attention: these are our users! There are some surprising—and occasionally disconcerting—disconnects between ORMs and databases [19]. The author of Rails, for example, is a colorful character named David Heinemeier Hansson ("DHH") who believes in "opinionated software" (that reflects his opinions, of course). He was quoted saying the following:

> I don't want my database to be clever! ...I consider stored procedures and constraints vile and reckless destroyers of coherence. No, Mr. Database, you can not have my business logic. Your procedural ambitions will bear no fruit and you'll have to pry that logic from my dead, cold object-oriented hands . . . I want a single layer of cleverness: My domain model.

This unwillingness to trust the DBMS leads to many problems in Rails applications. Applications written against ORMs are often very slow—the ORMs themselves don't do much to optimize the way that queries are generated. Instead, Rails programmers often need to learn to program "differently" to encourage Rails to generate efficient SQL—similar to the discussion in the Pascal/R paper, they need to learn to avoid looping and table-at-a-time iteration. A typical evolution in a Rails app is to write it naively, observe slow performance, study the SQL logs it generates, and rewrite the app to convince the ORM to generate "better" SQL. Recent work by Cheung and colleagues explores the idea that program synthesis techniques can automatically generate these optimizations [38]; it's an interesting direction, and time will tell how much complexity it can automate away. The separation between database and applications can also have negative effects for correctness. For example, Bailis recently showed [19] how a host of existing open-source Rails applications are susceptible to integrity violations due to improper enforcement within the application (instead of the database).

Despite some blind spots, ORMs have generally been an important practical leap forward in programmability of database-backed applications, and a validation of ideas that go back as far as Pascal/R. Some good ideas take time to catch on.

Stream Queries: CQL

Our second paper on CQL is a different flavor of language work—it's a query language design paper. It presents the design of a new declarative query language for a data model of streams. The paper is interesting for a few reasons. First, it is a clean, readable, and relatively modern example of query language design. Every few years a group of people emerges with yet another data model and query language: examples include Objects and OQL, XML and XQuery, or RDF and SPARQL. Most of these exercises begin with an assertion that "X changes everything" for some data model X, leading to the presentation of a new query language that often seems familiar and yet strangely different than SQL. CQL is a refreshing example of language design because it does the opposite: it highlights that fact that streaming data, viewed through the right lens, actually changes very little. CQL evolves SQL just enough to isolate the key distinctions between querying "resting" tables and "moving" streams. This leaves us with a crisp understanding of what's really different, semantically, when you have to talk about streams; many other current streaming languages are quite a bit more ad hoc and messy than CQL.

In addition to this paper being a nice exemplar of thoughtful query language design, it also represents a research area that received a lot of attention in the database literature, and remains intriguing in practice. The first generation of streaming data research systems from the early 2000's [3, 120, 118, 36] did not have major uptake either as open source or in the variety of startups that grew out of those systems. However the topic of stream queries has been gaining interest again in industry in recent years, with open source systems like SparkStreaming, Storm and Heron seeing uptake, and companies like Google espousing the importance of continuous dataflow as a new reality of modern services [8]. We may yet see stream query systems occupy more than their current small niche in financial services.

Another reason CQL is interesting is that streams are something of a middle ground between databases and "events". Databases store and retrieve collection types; event systems transmit and handle discrete events. Once you view your events as data, then event programming and stream programming look quite similar. Given that event programming is a widely-used programming model in certain domains (e.g. Javascript for user interfaces, Erlang for distributed systems), there should be a relatively small impedance mismatch between an event programming language like Javascript and a data stream system. An interesting example of this approach is the Rx (Reactive extensions) language, which is a streaming addition to LINQ that makes programming event streams feel like writing functional query plans; or as its author Erik Meijer puts it, "your mouse is a database" [114].

Programming Correct Applications without Transactions: Bloom

The third paper on Bloom connects to a number of the points above; it has a relational model of state at the application level, and a notion of network channels that relates to CQL streams. But the primary goal is to help programmers manage the loss of the first abstraction at the beginning of this chapter introduction; the one I described as a major crisis. A big question for modern developers is this: can you find a correct distributed implementation for your program without using transactions or other other expensive schemes to control orders of operation?

Bloom's answer to this question is to give programmers a "disorderly" programming language: one that discourages them from accidentally using ordering.

Bloom's default data structures are relations; its basic programming constructs are logical rules that can run in any order. In short, it's a general-purpose language that is similar to a relational query language. For the same reason that SQL queries can be optimized and parallelized without changing output, simple Bloom programs have a well-defined (consistent) result independent of the order of execution. The exception to this intuition comes with lines of Bloom code that are "non-monotonic", testing for a property that can oscillate between true and false as time passes (e.g. "NOT EXISTS x" or "HAVING COUNT() = x".) These rules are sensitive to execution and message ordering, and need to be "protected" by coordination mechanisms.

The CALM theorem formalizes this notion, answering the question above definitively: you can find a consistent, distributed, coordination-free implementation for your program if and only if its specification is monotonic [84, 14]. The Bloom paper also illustrates how a compiler can use CALM in practice to pinpoint the need for coordination in Bloom programs. CALM analysis can also be applied to data stream languages in systems like Storm with the help of programmer annotations [12]. A survey of the theoretical results in this area is given in [13].

There has been a flurry of related language work on avoiding coordination: a number of papers propose using associative, commutative, idempotent operations [83, 142, 42]; these are inherently monotonic. Another set of work examines alternative correctness criteria, e.g., ensuring only specific invariants over database state [20] or using alternative program analysis to deliver serializable outcomes without impelementing traditional read-write concurrency [137]. The area is still new; papers have different models (e.g. some with transactional boundaries and some not) and often don't agree on definitions of "consistency" or "coordination". (CALM defines consistency in terms of globally deterministic outcomes, coordination as messaging that is required regardless of data partitioning or replication [14].) It's important to get more clarity and ideas here—if programmers can't have transactions then they need help at the app-development layer.

Bloom also serves as an example of a recurring theme in database research: general-purpose declarative languages (a.k.a. "logic programming"). Datalog is the standard example, and has a long and controversial history in database research. A favorite topic of database theoreticians in the 1980's, Datalog met ferocious backlash from systems researchers of the day as being irrelevant in practice [152]. More recently it has gotten some attention from (younger) researchers in databases and other applied fields [74]—for example, Nicira's software-defined networking stack (acquired by VMWare for a cool billion dollars) uses a Datalog language for network forwarding state [97]. There is a spectrum between using declarative sublanguages for accessing database state, and very aggressive uses of declarative programming like Bloom for specifying application logic. Time will tell how this declarative-imperative boundary shifts for programmers in various contexts, including infrastructure, applications, web clients and mobile devices.

Chapter 10: Web Data

Introduced by Peter Bailis

Selected Readings:

Sergey Brin and Larry Page. The Anatomy of a Large-scale Hypertextual Web Search Engine. *WWW*, 1998.

Eric A. Brewer. Combining Systems and Databases: A Search Engine Retrospective. *Readings in Database Systems, Fourth Edition*, 2005.

Michael J. Cafarella, Alon Halevy, Daisy Zhe Wang, Eugene Wu, Yang Zhang. WebTables: Exploring the Power of Tables on the Web. *VLDB*, 2008.

Since the previous edition of this collection, the World Wide Web has unequivocally laid any lingering questions regarding its longevity and global impact to rest. Several multi-Billion-user services including Google and Facebook have become central to modern life in the first world, while Internet- and Web-related technology has permeated both business and personal interactions. The Web is undoubtedly here to stayat least for the foreseeable future.

Web data systems bring a new set of challenges, including high scale, data heterogeneity, and a complex and evolving set of user interaction modes. Classical relational database system designs did not have the Web workload in mind, and are not the technology of choice in this context. Rather, Web data management requires a melange of techniques spanning information retrieval, database internals, data integration, and distributed systems. In this section, we include three papers that highlight technical solutions to problems inherent in Web data management.

Our first two papers describe the internals of search engine and indexing technology. Our first paper, from Larry Page and Sergey Brin, Google co-founders, describes the internals of an early prototype of Google. The paper is interesting both from a historical perspective as well as a technical one. The first Web indices, such as Yahoo!, consisted of human-curated "directories". While directory curation proved useful, directories were difficult to scale and required considerable human power to maintain. As a result, a number of search engines, including Google but also Inktomi, co-created by Eric Brewer, author of the second paper, sought automated approaches. The design of these engines is conceptually straightforward: a set of crawlers downloads copies of web data and builds (and maintains) read-only indices that are used to compute a relevance scoring function. Queries, in turn, are serviced by a front-end web service that reads from the indices and presents an ordered set of results, ranked by scoring function.

The implementation and realization of these engines is complex. For example, scoring algorithms are highly tuned, and their implementation is considered a trade secret even within search engines today: Web authors have a large incentive to manipulate the scoring function to their advantage. The PageRank algorithm described in the Google paper (and detailed in [126]) is an famous example of a scoring function, and measures the "influence" of each page measured according to the hyperlink graph. Both papers describe how a combination of mostly unspecified attributes is used for scoring in practice, including "anchor text" (providing context on the source of a link) and other forms of metadata. The algorithmic foundations of these techniques, such as keyword indexing date, date to the 1950s [107], while others, such as TFxIDF ranking and inverted indices, date to the 1960s [139]. Many of the key systems innovations in building Internet search engines came in scaling them and in handling dirty, heterogenous data sources.

While the high-level details of these papers are helpful in understanding how modern search engines operate, these papers are also interesting for their commentary on the process of building a production Web search engine. A central message in each is that Web services must account for variety; the Google authors describe how assumptions made in typical information retrieval techniques may no longer hold in a Web context (e.g., the "Bill Clinton sucks" web page). Web sources change at varying rates, necessitating prioritized crawling for maintaining fresh indices. Brewer also highlights the importance of fault tolerance and availability of operation, echoing his experience in the field building Inktomi (which also led to the development of concepts including *harvest* and *yield* [31] and the CAP Theorem; see Chapter 7). Brewer outlines the difficulty in building a search engine using commodity database engines (e.g., Informix was 10x slower than Inktomi's custom solution). However, he notes that the principles of database system design, including "top-down" design, data independence, and a declarative query engine, are valuable in this context—if appropriately adapted.

Today, Web search engines are considered mature technology. However, competing services continually improve search experience by adding additional functionality. Today's search engines are much more than information retrieval engines for textual data web pages; the content of the first two papers is a small subset of the internals of a service like Google or Baidu. These services provide a range of functionality, including targeted advertisement, image search, navigation, shopping, and mobile search. There is undoubtedly bleed-over in retrieval, entity resolution, and indexing techniques between these domains, but each requires domain-specific adaptation.

As an example of a new type of search enabled by massive Web data, we include a paper from the WebTables project led by Alon Halevy at Google. WebTables allows users to query and understand relationships between data stored in HTML tables. HTML tables are inherently varied in structure due to a lack of fixed schema. However, aggregating enough of them at Web scale and performing some lightweight automated data integration enables some interesting queries (e.g., a table of influenza outbreak locations can be combined with a table containing data about city populations). Mining the schema of these tables, determining their structure and veracity (e.g., only 1% of the tables in the paper corpus were, in fact, relations), and efficiently inferring their relationships is difficult. The paper we have included describes techniques for building an attribute correlation statistics database (AcsDB) to answer queries about the table metadata, enabling novel functionality including schema auto-complete. The WebTables project continues today in various forms, including Google Table Search and integration with Google's core search technology; an update on the project can be found in [24]. The ability to produce structured search results is desirable in several non-traditional domains, including mobile, contextual, and audio-based search.

The WebTables paper in particular highlights the power of working with Web data at scale. In a 2009 article, Halevy and colleagues describe the "Unreasonable Effectiveness of Data," effectively arguing that, with sufficient amount of data, enough latent structure is captured to make modeling simpler: relatively simple data mining techniques often beat more mathematically sophisticated statistical models [79]. This argument stresses the potential for unlocking hidden structure by sheer volume of data and computation, whether mining schema correlations or performing machine translation between languages. With a big enough haystack, needles become large. Even examining 1% of the tables in the web corpus, the VLDB 2009 paper studies 154M distinct relations, a corpus that was "five orders of magnitude larger than the largest one [previously] considered."

The barrier for performing analysis of massive datasets and system architectures outside of these companies is decreasing, due to cheap commodity storage and cloud computing resources. However, it is difficult to replicate the feedback loop between users (e.g., spammers) and algorithms (e.g., search ranking algorithms). Internet companies are uniquely positioned to pioneer systems designs that account for this feedback loop. As database technologies power additional interactive domains, we believe this paradigm will become even more important. That is, the database market and interesting database workloads may benefit from similar analyses. For example, it would be interesting to perform a similar analysis on hosted database platforms such as Amazon Redshift and Microsoft SQL Azure, enabling a variety of functionality including index auto-tuning, adaptive query optimization, schema discovery from unstructured data, query autocomplete, and visualization recommendations.

Chapter 11: A Biased Take on a Moving Target: Complex Analytics

by Michael Stonebraker

In the past 5-10 years, new analytic workloads have emerged that are more complex than the typical business intelligence (BI) use case. For example, internet advertisers might want to know "How do women who bought an Apple computer in the last four days differ statistically from women who purchased a Ford pickup truck in the same time period?" The next question might be: "Among all our ads, which one is the most profitable to show to the female Ford buyers based on their click-through likelihood?" These are the questions asked by today's data scientists, and represent a very different use case from the traditional SQL analytics run by business intelligence specialists. It is widely assumed that data science will completely replace business intelligence over the next decade or two, since it represents a more sophisticated approach to mining data warehouses for new insights. As such, this document focuses on the needs of data scientists.

I will start this section with a description of what I see as the job description of a data scientist. After cleaning and wrangling his data, which currently consumes the vast majority of his time and which is discussed in the section on data integration, he generally performs the following iteration:

```
Until (tired) {
   Data management operation(s);
   Analytic operation(s);
}
```

In other words, he has an iterative discovery process, whereby he isolates a data set of interest and then performs some analytic operation on it. This often suggests either a different data set to try the same analytic on or a different analytic on the same data set. By and large what distinguishes data science from business intelligence is that the analytics are predictive modeling, machine learning, regressions, ... and not SQL analytics.

In general, there is a pipeline of computations that constitutes the analytics. For example, Tamr has a module which performs entity consolidation (deduplication) on a collection of records, say N of them, at scale. To avoid the N ** 2 complexity of brute force algorithms, Tamr identifies a collection of "features", divides them into ranges that are unlikely to co-occur, computes (perhaps multiple) "bins" for each record based on these ranges, reshuffles the data in parallel so it is partitioned by bin number, deduplicates each bin, merges the results, and finally constructs composite records out of the various clusters of duplicates. This pipeline is partly SQL-oriented (partitioning) and partly array-oriented analytics. Tamr seems to be typical of data science workloads in that it is a pipeline with half a dozen steps.

Some analytic pipelines are "one-shots" which are run once on a batch of records. However, most production applications are incremental in nature. For example, Tamr is run on an initial batch of input records and then periodically a new "delta" must be processed as new or changed input records arrive. There are two approaches to incremental operation. If deltas are processed as "mini batches" at periodic intervals of (say) one day, one can add the next delta to the previously processed batch and rerun the entire pipeline on all the data each time the input changes. Such a strategy will be very wasteful of computing resources. Instead, we will focus on the case where incremental algorithms must be run after an initial batch processing operation. Such incremental algorithms require intermediate states of the analysis to be saved to persistent storage at each interation. Although the Tamr pipeline is of length 6 or so, each step must be saved to persistent storage to support incremental operation. Since saving state is a data management operation, this make the analytics pipeline of length one.

The ultimate "real time" solution is to run incremental analytics continuously services by a streaming platform such as discussed in the section on new DBMS technology. Depending on the arrival rate of new records, either solution may be preferred.

Most complex analytics are array-oriented, i.e. they are a collection of linear algebra operations defined on arrays. Some analytics are graph oriented, such as social network analysis. It is clear that arrays can be simulated on table-based systems and that graphs can be simulated on either table systems or array systems. As such, later in this document, we discuss how many different architectures are needed for this used case.

Some problems deal with dense arrays, which are ones where almost all cells have a value. For example, an array of closing stock prices over time for all securities on the NYSE will be dense, since every stock has a closing price for each trading day. On the other hand, some problems are sparse. For example, a social networking use case represented as a matrix would have a cell value for every pair of persons that were associated in some way. Clearly, this matrix will be very sparse. Analytics on sparse arrays are quite different from analytics on dense arrays.

In this section we will discuss such workloads at scale. If one wants to perform such pipelines on "small data" then any solution will work fine.

The goal of a data science platform is to support this iterative discovery process. We begin with a sad truth. Most data science platforms are file-based and have nothing to do with DBMSs. The preponderance of analytic codes are run in R, MatLab, SPSS, SAS and operate on file data. In addition, many Spark users are reading data from files. An exemplar of this state of affairs is the NERSC high performance computing (HPC) system at Lawrence Berkeley Labs. This machine is used essentially exclusively for complex analytics; however, we were unable to get the Vertica DBMS to run at all, because of configuration restrictions. In addition, most "big science" projects build an entire software stack from the bare metal on up. It is plausible that this state of affairs will continue, and DBMSs will not become a player in this market. However, there are some hopeful signs such as the fact that genetic data is starting to be deployed on DBMSs, for example the 1000 Genomes Project [144] is based on SciDB.

In my opinion, file system technology suffers from several big disadvantages. First metadata (calibration, time, etc.) is often not captured or is encoded in the name of the file, and is therefore not searchable. Second, sophisticated data processing to do the data management piece of the data science workload is not available and must be written (somehow). Third, file data is difficult to share data among colleagues. I know of several projects which export their data along with their parsing program. The recipient may be unable to recompile this accessor program or it generates an error. In the rest of this discussion, I will assume that data scientists over time wish to use DBMS technology. Hence, there will be no further discussion of file-based solutions.

With this backdrop, we show in Table 1 a classification of data science platforms. To perform the data management portion, one needs a DBMS, according to our assumption above. This DBMS can have one of two flavors. First, it can be record-oriented as in a relational row store or a NoSQL engine or column-oriented as in most data warehouse systems. In these cases, the DBMS data structure is not focused on the needs of analytics, which are essentially all array-oriented, so a more natural choice would be an array DBMS. The latter case has the advantage that no conversion from a record or column structure is required to perform analytics. Hence, an array structure will have an innate advantage in performance. In addition, an array-oriented storage structure is multi-dimensional in nature, as opposed to table structures which are usually one-dimensional. Again, this is likely to result in higher performance.

The second dimension concerns the coupling between the analytics and the DBMS. On the one hand, they can be independent, and one can run a query, copying the result to a different address space where the analytics are run. At the end of the analytics pipeline (often of length one), the result can be saved back to persistent storage. This will result in lots of data churn between the DBMS and the analytics. On the other hand, one can run analytics as user-defined functions in the same address space as the DBMS. Obviously the tight coupling alternative will lower data churn and should result in superior performance.

In this light, there are four cells, as noted in Table 1. In the lower left corner, Map-Reduce used to be the exemplar; more recently Spark has eclipsed Map-Reduce as the platform with the most interest. There is no persistence mechanism in Spark, which depends on RedShift or H-Base, or ... for this purpose. Hence, in Spark a user runs a query in some DBMS to generate a data set, which is loaded into Spark, where analytics are performed. The DBMSs supported by Spark are all record or column-oriented, so a conversion to array representation is required for the analytics.

A notable example in the lower right hand corner is MADLIB [85], which is a user-defined function library supported by the RDBMS Greenplum. Other vendors have more recently started supporting other alternatives; for example Vertica supports user-defined functions in R. In the upper right hand corner are array systems with built-in analytics such as SciDB [155], TileDB [56] or Rasdaman [26].

In the rest of this document, we discuss performance implications. First, one would expect performance to improve as one moves from lower left to upper right in Table 1. Second, most complex analytics reduce to

	Loosely coupled	Tightly coupled
Array representation		SciDB, TileDB, Rasdaman
Table respresentation	Spark + HBase	MADLib, Vertica + R

Table 1: A Classification of Data Science Platforms

a small collection of "inner loop" operations, such as matrix multiply, singular-value decomposition and QR decomposition. All are computationally intensive, typically floating point codes. It is accepted by most that hardware-specific pipelining can make nearly an order of magnitude difference in performance on these sorts of codes. As such, libraries such as BLAS, LAPACK, and ScaLAPACK, which call the hardware-optimized Intel MKL library, will be wildly faster than codes which don't use hardware optimization. Of course, hardware optimization will make a big difference on dense array calculations, where the majority of the effort is in floating point computation. It will be less significance on sparse arrays, where indexing issues may dominate the computation time.

Third, codes that provide approximate answers are way faster than ones that produce exact answers. If you can deal with an approximate answer, then you will save mountains of time.

Fourth, High Performance Computing (HPC) hardware are generally configured to support large batch jobs. As such, they are often structured as a computation server connected to a storage server by networking, whereby a program must pre-allocation disk space in a computation server cache for its storage needs. This is obviously at odds with a DBMS, which expects to be continuously running as a service. Hence, be aware that you may have trouble with DBMS systems on HPC environments. An interesting area of exploration is whether HPC machines can deal with both interactive and batch workloads simultaneously without sacrificing performance.

Fifth, scalable data science codes invariably run on multiple nodes in a computer network and are often network-bound [55]. In this case, you must pay careful attention to networking costs and TCP-IP may not be a good choice. In general MPI is a higher performance alternative.

Sixth, most analytics codes that we have tested fail to scale to large data set sizes, either because they run out of main memory or because they generate temporaries that are too large. Make sure you test any platform you would consider running on the data set sizes you expect in production!

Seventh, the structure of your analytics pipeline is crucial. If your pipeline is on length one, then tight coupling is almost certainly a good idea. On the other hand, if the pipeline is on length 10, loose coupling will perform almost as well. In incremental operation, expect pipelines of length one.

In general, all solutions we know of have scalability and performance problems. Moreover, most of the exemplars noted above are rapidly moving targets, so performance and scalability will undoubtedly improve. In summary, it will be interesting to see which cells in Table 1 have legs and which ones don't. The commercial marketplace will be the ultimate arbitrer!

In my opinion, complex analytics is current in its "wild west" phase, and we hope that the next edition of the red book can identify a collection of core seminal papers. In the meantime, there is substantial research to be performed. Specifically, we would encourage more benchmarking in this space in order to identify flaws in existing platforms and to spur further research and development, especially benchmarks that look at end-to-end tasks involving both data management tasks and analytics. This space is moving fast, so the benchmark results will likely be transient. That's probably a good thing: we're in a phase where the various projects should be learning from each other.

There is currently a lot of interest in custom parallel algorithms for core analytics tasks like convex optimization; some of it from the database community. It will be interesting to see if these algorithms can be incorporated into analytic DBMSs, since they don't typically follow a traditional dataflow execution style. An exemplar here is Hogwild! [133], which achieves very fast performance by allowing lock-free parallelism in shared memory. Google Downpour [49] and Microsoft's Project Adam [39] both adapt this basic idea to a distributed context for deep learning.

Another area where exploration is warranted is out-of-memory algorithms. For example, Spark requires your data structures to fit into the combined amount of main memory present on the machines in your cluster. Such solutions will be brittle, and will almost certainly have scalability problems.

Furthermore, an interesting topic is the desirable approach to graph analytics. One can either build special purpose graph analytics, such as GraphX [64] or GraphLab [105] and connect them to some sort of DBMS. Alternately, one can simulate such codes with either array analytics, as espoused in D4M [95] or table analytics, as suggested in [90]. Again, may the solution space bloom, and the commercial market place be the arbiter!

Lastly, many analytics codes use MPI for communication, whereas DBMSs invariably use TCP-IP. Also, parallel dense analytic packages, such as ScaLAPACK, organize data into a block-cyclic organization across nodes in a computing cluster [40]. I know of no DBMS that supports block-cyclic partitioning. Removing this impedance mismatch between analytic packages and DBMSs is an interesting research area, one that is targeted by the Intel-sponsored ISTC on Big Data [151].

Commentary: Joe Hellerstein

6 December 2015

I have a rather different take on this area than Mike, both from a business perspective and in terms of research opportunities. At base, I recommend a "big tent" approach to this area. DB folk have much to contribute, but we'll do far better if we play well with others.

Let's look at the industry. First off, advanced analytics of the sort we're discussing here will not replace BI as Mike suggests. The BI industry is healthy and growing. More fundamentally, as noted statistician John Tukey pointed out in his foundational work on Exploratory Data Analysis,[4] a chart is often much more valuable than a complex statistical model. Respect the chart!

That said, the advanced analytics and data science market is indeed growing and poised for change. But unlike the BI market, this is not a category where database technology currently plays a significant role. The incumbent in this space is SAS, a company that makes multiple billions of dollars in revenue each year, and is decidedly not a database company. When VCs look at companies in this space, they're looking for "the next SAS". SAS users are not database users. And the users of open-source alternatives like R are also not database users. If you assume as Mike does that "data scientists will want to use DBMS technology" — particularly a monolithic "analytic DBMS" — you're swimming upstream in a strong current.

For a more natural approach to the advanced analytics market, ask yourself this: what is a serious threat to SAS? Who could take a significant bite out of the cash that enterprises currently spend there? Here are some starting points to consider:

1. **Open source stats programming**: This includes R and the Python data science ecosystem (NumPy, SciKit-Learn, iPython Notebook). These solutions don't currently don't scale well, but efforts are underway aggressively to address those limitations. This ecosystem could evolve more quickly than SAS.
2. **Tight couplings to big data platforms**. When the data is big enough, performance requirements may "drag" users to a new platform — namely a platform that already hosts the big data in their organization. Hence the interest in "DataFrame" interfaces to platforms like Spark/MLLib, PivotalR/MADlib, and Vertica dplyr. Note that the advanced analytics community is highly biased toward open source. The cloud is also an interesting platform here, and not one where SAS has an advantage.
3. **Analytic Services**. By this I mean interactive online services that use analytic methods at their core: recommender systems, real-time fraud detection, predictive equipment maintenance and so on. This space has aggressive system requirements for response times, request scaling, fault tolerance and ongoing evolution that products like SAS don't address. Today, these services are largely built with custom code. This doesn't scale across industries — most companies can't recruit developers that can work at this level. So there is ostensibly an opportunity here in commoditizing this technology for the majority of use cases. But it's early days for this market — it remains to be seen whether analytics service platforms can be made simple enough for commodity deployment. If the tech evolves, then cloud-based services may have significant opportunities for disruption here as well.

On the research front, I think it's critical to think outside the database box, and collaborate aggressively. To me this almost goes without saying. Nearly every subfield in computing is working on big data analytics in some fashion, and smart people from a variety of areas are quickly learning their own lessons about data and scale. We can have fun playing with these folks, or we can ignore them to our detriment.

So where can database research have a big impact in this space? Some possiblities that look good to me include these:

1. **New approaches to Scalability**. We have successfully

[4]Tukey, John. Exploratory Data Analysis. Pearson, 1977.

shown that parallel dataflow — think MADlib, MLlib or the work of Ordonez at Teradata[5] — can take you a long way toward implementing scalable analytics *without* doing violence at the system architecture level. That's useful to know. Moving forward, can we do something that is usefully faster and more scalable than parallel dataflow over partitioned data? Is that necessary? Hogwild! has generated some of the biggest excitement here; note that it's work that spans the DB and ML communities.

2. **Distributed infrastructure for analytic services**. As I mentioned above, analytic services are an interesting opportunity for innovation. The system infrastructure issues on this front are fairly wide open. What are the main components of architectures for analytics services? How are they stitched together? What kind of data consistency is required across the components? So-called Parameter Servers are a topic of interest right now, but only address a piece of the puzzle.[6] There has been some initial work on online serving, evolution and deployment of models.[7] I hope there will be more.

3. **Analytic lifecycle and metadata management**. This is an area where I agree with Mike. Analytics is often a people-intensive exercise, involving data exploration and transformation in addition to core statistical modeling. Along the way, a good deal of context needs to be managed to understand how models and data products get developed across a range of tools and systems. The database commmunity has perspectives on this area that are highly relevant, including workflow management, data lineage and materialized view maintenance. VisTrails is an example of research in this space that is being used in practice.[8] This is an area of pressing need in industry as well — especially work that takes into account the real-world diversity of analytics tools and systems in the field.

[5]e.g., Ordonez, C. Integrating K-means clustering with a relational DBMS using SQL. TKDE 18(2) 2006. Also Ordonez, C. Statistical Model Computation with UDFs. TKDE 22(12), 2010.

[6]Ho, Q., et al. More effective distributed ML via a stale synchronous parallel parameter server. NIPS 2013.

[7]Crankshaw, D, et al. The missing piece in complex analytics: Low latency, scalable model management and serving with Velox. CIDR 2015. See also Schleier-Smith, J. An Architecture for Agile Machine Learning in Real-Time Applications. KDD 2015.

[8]See http://www.vistrails.org.

Chapter 12: A Biased Take on a Moving Target: Data Integration

by Michael Stonebraker

I will start this treatise with a history of two major themes in data integration. In my opinion, the topic began with the major retailers in the 1990s consolidating their sales data into a data warehouse. To do this they needed to extract data from in-store sales systems, transform it into a predefined common representation (think of this as a global schema), and then load it into a data warehouse. This data warehouse kept historical sales data for a couple of years and was used by the buyers in the organization to rotate stock. In other words, a buyer would figure out that pet rocks are "out" and barbie dolls are "in." Hence, he would tie up the barbie doll factory with a big order and move the pet rocks up front and discount them to get rid of them. A typical retail data warehouse paid for itself within a year through better buying and stock rotation decisions. In the late 1990s and early 2000's there was a giant "pile on" as essentially all enterprises followed the lead of retailers and organized their customer-facing data into a data warehouse.

A new industry was spawned to support the loading of data warehouses, called extract, transform, and load (ETL) systems. The basic methodology was:

a) Construct a global schema in advance.

b) Send a programmer out to the owner of each data source and have him figure out how to do the extraction. Historically, writing such "connectors" was a lot of work because of arcane formats. Hopefully, this will become less problematic in the future with more open source and standardized formats.

c) Have him write transformations, often in a scripting language, and any necessary cleaning routines

d) Have him write a script to load the data into a data warehouse

It became apparent that this methodology scales to perhaps a dozen data sources, because of three big issues:

1. A global schema is needed up front. About this time, there was a push in many enterprises to write an enterprise-wide schema for all company objects. A team was charged with doing this and would work on it for a couple of years. At the end of this time, their result was two years out of date, and was declared a failure. Hence, an upfront global schema is incredibly difficult to construct for a broad domain. This limits the plausible scope of data warehouses.

2. Too much manual labor. A skilled programmer is required to perform most of the steps in the methodology for each data source.

3. Data integration and cleaning is fundamentally difficult. The typical data warehouse project in the 1990's was a factor of two over budget and a factor of two late. The problem was that planners underestimated the difficulty of the data integration challenge. There are two big issues. First, data is dirty. A rule of thumb is that 10% of your data is incorrect. This results from using nicknames for people or products, having stale addresses for suppliers, having incorrect ages for people, etc. The second is that deduplication is hard. One has to decide if Mike Stonebraker and M.R. Stonebraker are the same entity or different ones. Equally challenging is two restaurants at the same address. They might be in a food court, one might have replaced the other in a stand-alone location or this might be a data error. It is expensive to figure out ground truth in such situations.

In spite of these issues, data warehouses have been a huge success for customer facing data, and are in use by most major enterprises. In this use case, the pain of assembling composite data is justified by the better decision making that results. I hardly ever hear enterprises complaining about the operational cost of their data warehouse. What I hear instead is an incessant desire by business analysts for more data sources, whether these be public data off the web or other enterprise data. For example, the average large enterprise has about 5000 operational data stores, and only a few are in the data warehouse.

As a quick example, I visited a major beer manufacturer a while ago. He had a typical data warehouse of sales of his products by brand, by distributor, by time period, etc. I told the analysts that El Nino was widely predicted to occur that winter and it would be wetter than normal on the west coast and warmer than normal in the Northeast. I then asked if beer sales are correlated to temperature or precipitation. They replied "I wish we could answer that question, but weather data is not in our warehouse". Supporting data source scalability is very difficult using ETL technology.

Fast forward to the 2000's, and the new buzzword is *master data management (MDM).* The idea behind MDM is to standardize the enterprise representation of important entities such as customers, employees, sales, purchases, suppliers, etc. Then carefully curate a master data set for each entity type and get everyone in the enterprise to use this master. This sometimes goes by the mantra "golden records". In effect, the former ETL vendors are now selling MDM, as a broader scope offering. In my opinion, MDM is way over-hyped.

Let me start with "Who can be against standards?" Certainly not me. However, MDM has the following problems, which I will illustrate by vignette.

When I worked for Informix 15 years ago, the new CEO asked the Human Resources VP at an early staff meeting "How many employees do we have?" She returned the next week with the answer "I don't know and there is no way to find out?" Informix operated in 58 countries, each with its own labor laws, definition of an employee, etc. There was no golden record for employees. Hence, the only way to answer the CEOs question would be to perform data integration on these 58 data sources to resolve the semantic issues. Getting 58 country managers to agree to do this would be challenging, made more difficult by the fact that Informix did not even own all the organizations involved. The new CEO quickly realized the futility of this exercise.

So why would a company allow this situation to occur? The answer is simple: business agility. Informix set up country operations on a regular basis, and wanted the sales team up and running quickly. Inevitably they would hire a country manager and tell him to "make it happen". Sometimes it was a distributor or other independent entity. If they had said "here are the MDM golden records you need to conform to", then the country manager or distributor would spend months trying to reconcile his needs with the MDM system in place. In other words, MDM is the opposite of business agility. Obviously every enterprise needs to strike a balance between standards and agility.

A second vignette concerns a large manufacturing enterprise. They are decentralized into business units for business agility reasons. Each business unit has its own purchasing system to specify the terms and conditions under which the business unit interacts with its suppliers. There are some 300 of these systems. There is an obvious return on investment to consolidate these systems. After all it is less code to maintain and the enterprise can presumably get better-consolidated terms than each business unit can individually. So why are there so many purchasing systems? Acquisitions. This enterprise grew largely by acquisition. Each acquisition became a new business unit, and came with its own data systems, contracts in place, etc. It is often simply not feasible to merge all these data systems into the parent's IT infrastructure. In summary, acquisitions screw up MDM.

So what is entailed in data integration (or data curation)? It is the following steps:

1. *Ingest.* A data source must be located and captured. This requires parsing whatever data structure is used for storage.
2. *Transform.* For example, Euros to dollars or airport code to city name.
3. *Clean.* Data errors must be found and rectified.
4. *Schema integration.* Your wages is my salary.
5. *Consolidate (deduplication).* Mike Stonebraker and M.R. Stonebraker must be consolidated into a single record.

The ETL vendors do this at high cost and with low scalability. The MDM vendors have a similar profile. So there is a big unmet need. Data curation at scale is the "800 pound gorilla in the corner." So what are the research challenges in this area?

Let's go through the steps one by one.

Ingest is simply a matter of parsing data sources. Others have written such "connectors", and they are generally expensive to construct. An interesting challenge would be to semi-automatically generate connectors.

Data transformations have also been extensively researched, mostly in the last decade or so. Scripting/visualization facilities to specify transforms have been studied in [130, 54]. Data Wrangler [94] appears to be the state of the art in this area, and the interested reader is encouraged to take a look. In addition, there are a bunch of commercial offerings that offer transformation services for a fee (e.g. address to (lat,long) or company to canonical company representation). In addition, work on finding transformations of interest from the public web is reported in [4].

Data cleaning has been studied using a variety of techniques. [41] applied functional dependencies to discover erroneous data and suggest automatic repairs. Outlier detection (which may correspond to errors) has been studied in many contexts [87]. [162, 158] are query systems to discover interesting patterns in the data. Such patterns may correspond to errors. [148] have studied the utilization of crowd sourcing and expert sourcing to correct errors, once they have been identified. Lastly, there are a variety of commercial services that will clean common domains, such as addresses of persons and dates. In my opinion, data cleaning research MUST utilize real-world data. Finding faults that have been injected into other-wise clean data just is not believable. Unfortunately, real world "dirty" data is quite hard to come by.

Schema matching has been extensively worked on for at least 20 years. The interested reader should consult [117, 77, 134] for the state of the art in this area.

Entity consolidation is a problem of finding records in a high dimensional space (all of the attributes about an entity – typically 25 or more) that are close together. Effectively this is a clustering problem in 25 space. This is an N ** 2 problem that will have a very long running time at scale. Hence, approximate algorithms are clearly the way to proceed here. A survey of techniques appears in [87].

In my opinion, the real problem is an end-to-end system. Data curation entails all of these steps, which must be seamlessly integrated, and enterprise-wide systems must perform curation at scale. An interesting end-to-end approach that appears to scale well is the Data Tamer system [148]. On the other hand, data curation problems also occur at the department level, where an individual contributor wants to integrate a handful of data sources, and the Data Wrangler system noted above appears to be an interesting approach. There are commercial companies supporting both of these systems, so regular enhancements should be forthcoming.

Hopefully, the next edition of the Red Book will have a collection of seminal papers in this area to replace this (self-serving) call to action. In my opinion, this is one of the most important topics that enterprises are dealing with. My one caution is "the rubber has to meet the road". If you want to work in this area, you have got to try your ideas on real world enterprise data. Constructing artificial data, injecting faults into it, and then finding these faults is simply not believable. If you have ideas in this area, I would recommend building an end-to-end system. In this way, you make sure that you are solving an important problem, rather than just a "point problem" which real world users may or may not be interested in.

Commentary: Joe Hellerstein

6 December 2015

I agree with Mike's assessment here in general, but wanted to add my perspective on the space, relating to the "department level" problem he mentions in passing.

Based on experience with users across a wide range of organizations, we've seen that data transformation is increasingly a user-centric task, and depends critically upon the user experience: the interfaces and languages for interactively assessing and manipulating data.

In many of today's settings, the right outcome from data transformation depends heavily on context. To understand if data is dirty, you have to know what it is "supposed" to look like. To transform data for use, you need to understand what it is being used for. Even in a single organization, the context of how data is going to be used and what it needs to be like varies across people and across time. Add this to Mike's critique of the idea of a "golden master"–it simply doesn't make sense for many modern use cases, especially in analytical contexts.

Instead, you need to design tools for the people who best understand the data and the use case, and enable them to do their own data profiling and transformation in an agile, exploratory manner. Computer scientists tend to design for technical users–IT professionals and data scientists. But in most organizations, the users who understand the data and the context are closer to the "business" than the IT department. They are often less technically skilled as well. Rather than reach for traditional ETL tools, they tend to fall back on manipulating data in spreadsheets and desktop database packages, neither of which are well suited for assessing data quality or doing bulk transformation. For large datasets they "code in Microsoft Word": they describe their desired workflow in a natural language spec, wait for IT to get around to writing the code for them, and then when they get the results they typically realize they don't quite work. At which point their need for the data has often changed anyhow. No surprise that people often assert that 50-80% of their time is spent in "preparing the data." (As a footnote, in my experience modern "data scientists" tend to wrangle data via ad-hoc scripts in Python, R or SAS DataStep, and are shockingly lax about code quality and revision control for these scripts. As a result they're often worse off over time than the old-school ETL users!)

Business users reach for graphical tools for good reason: they want to understand the data as it is being transformed, and assess whether it is getting closer to a form that's useful for their business problem. As a result, the unattended algorithms from the database research literature have done too little to address the key issues in this space. I believe the most relevant solutions will be based on interfaces that enable people to understand the state of their data intuitively, and collaborate with algorithms to get the data better purposed for use.

This presents a significant opportunity for innovation. Data transformation is a perfect Petri Dish for exploring the general topic of visualizing and interacting with data. This is an area where ideas from Databases, HCI and Machine Learning can be brought together, to create interactive collaborations between algorithms and people that solve practical, context-specific problems with data. Backing this up we need interactive data systems that can do things like provide instantaneous data profiles (various aggregates) over the results of ad-hoc transformation steps, and speculate ahead of users in real time to suggest multiple alternative transformations that could be useful.[9] Some of the topics from the Interactive Analytics chapter are relevant here, particularly for big data sets. I've been happy to see more work on visualization and interaction in the database community in recent years; this is a great application area for that work.

[9]Heer, J., Hellerstein, J.M. and Kandel, S. ”Predictive Interaction for Data Transformation.” CIDR 2015.

List of All Readings

Background

Joseph M. Hellerstein and Michael Stonebraker. What Goes Around Comes Around. *Readings in Database Systems*, 4th Edition (2005).

Joseph M. Hellerstein, Michael Stonebraker, James Hamilton. Architecture of a Database System. *Foundations and Trends in Databases*, 1, 2 (2007).

Traditional RDBMS Systems

Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, Vera Watson. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems*, 1(2), 1976, 97-137.

Michael Stonebraker and Lawrence A. Rowe. The design of POSTGRES. *SIGMOD*, 1986.

David J. DeWitt, Shahram Ghandeharizadeh, Donovan Schneider, Allan Bricker, Hui-I Hsiao, Rick Rasmussen. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990, 44-62.

Techniques Everyone Should Know

Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, Thomas G. Price. Access path selection in a relational database management system. *SIGMOD*, 1979.

C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, Peter M. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1), 1992, 94-162.

Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, Irving L. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. , IBM, September, 1975.

Rakesh Agrawal, Michael J. Carey, Miron Livny. Concurrency Control Performance Modeling: Alternatives and Implications. *ACM Transactions on Database Systems*, 12(4), 1987, 609-654.

C. Mohan, Bruce G. Lindsay, Ron Obermarck. Transaction Management in the R* Distributed Database Management System. *ACM Transactions on Database Systems*, 11(4), 1986, 378-396.

New DBMS Architectures

Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, Stan Zdonik. C-store: A Column-oriented DBMS. *SIGMOD*, 2005.

Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, Mike Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. *SIGMOD*, 2013.

Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, Michael Stonebraker. OLTP Through the Looking Glass, and What We Found There. *SIGMOD*, 2008.

Large-Scale Dataflow Engines

Jeff Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI*, 2004.

Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu. DryadLINQ: A System for General-Purpose Distributed Data-

Parallel Computing Using a High-Level Language. *OSDI*, 2008.

Weak Isolation and Distribution

Atul Adya, Barbara Liskov, and Patrick O'Neil. Generalized Isolation Level Definitions. *ICDE*, 2000.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. *SOSP*, 2007.

Eric Brewer. CAP Twelve Years Later: How the "Rules" Have Changed. *IEEE Computer*, 45, 2 (2012).

Query Optimization

Goetz Graefe and William J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. *ICDE*, 1993.

Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. *SIGMOD*, 2000.

Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, Miso Cilimdzic. Robust Query Processing Through Progressive Optimization. *SIGMOD*, 2004.

Interactive Analytics

Venky Harinarayan, Anand Rajaraman, Jeffrey D. Ullman. Implementing Data Cubes Efficiently. *SIGMOD*, 1996.

Yihong Zhao, Prasad M. Deshpande, Jeffrey F. Naughton. An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. *SIGMOD*, 1997.

Joseph M. Hellerstein, Ron Avnur, Vijayshankar Raman. Informix under CONTROL: Online Query Processing. *Data Mining and Knowledge Discovery*, 4(4), 2000, 281-314.

Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, Ion Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. *EuroSys*, 2013.

Languages

Joachim W. Schmidt. Some High Level Language Constructs for Data of Type Relation. *ACM Transactions on Database Systems*, 2(3), 1977, 247-261.

Arvind Arasu, Shivnath Babu, Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 15(2), 2006, 121-142.

Peter Alvaro, Neil Conway, Joseph M. Hellerstein, William R. Marczak. Consistency Analysis in Bloom: A CALM and Collected Approach. *CIDR*, 2011.

Web Data

Sergey Brin and Larry Page. The Anatomy of a Large-scale Hypertextual Web Search Engine. *WWW*, 1998.

Eric A. Brewer. Combining Systems and Databases: A Search Engine Retrospective. *Readings in Database Systems, Fourth Edition*, 2005.

Michael J. Cafarella, Alon Halevy, Daisy Zhe Wang, Eugene Wu, Yang Zhang. WebTables: Exploring the Power of Tables on the Web. *VLDB*, 2008.

References

[1] Apache Tez. `https://tez.apache.org/`.

[2] Flexcoin: The Bitcoin Bank, 2014. `http://www.flexcoin.com/`; originally via Emin Gün Sirer.

[3] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB JournalThe International Journal on Very Large Data Bases*, 12(2):120–139, 2003.

[4] Z. Abedjan, J. Morcos, M. Gubanov, I. F. Ilyas, M. Stonebraker, P. Papotti, and M. Ouzzani. Dataxformer: Leveraging the web for semantic transformations. In *CIDR*, 2015.

[5] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The Aqua approximate query answering system. In *SIGMOD*, 1999.

[6] A. Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, MIT, 1999.

[7] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, 1993.

[8] T. Akidau et al. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. In *VLDB*, 2015.

[9] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, et al. The Stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[10] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, et al. Asterixdb: A scalable, open source bdms. In *VLDB*, 2014.

[11] P. Alvaro, P. Bailis, N. Conway, and J. M. Hellerstein. Consistency without borders. In *SoCC*, 2013.

[12] P. Alvaro, N. Conway, J. M. Hellerstein, and D. Maier. Blazes: Coordination analysis for distributed programs. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 52–63. IEEE, 2014.

[13] T. J. Ameloot. Declarative networking: Recent theoretical work on coordination, correctness, and declarative semantics. *ACM SIGMOD Record*, 43(2):5–16, 2014.

[14] T. J. Ameloot, F. Neven, and J. Van den Bussche. Relational transducers for declarative networking. *Journal of the ACM (JACM)*, 60(2):15, 2013.

[15] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark SQL: Relational data processing in spark. In *SIGMOD*, 2015.

[16] S. Babu and H. Herodotou. Massively parallel databases and MapReduce systems. *Foundations and Trends in Databases*, 5(1):1–104, 2013.

[17] P. Bailis. *Coordination avoidance in distributed databases*. PhD thesis, University of California at Berkeley, 2015.

[18] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly Available Transactions: Virtues and limitations. In *VLDB*, 2014.

[19] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Feral Concurrency Control: An empirical investigation of modern application integrity. In *SIGMOD*, 2015.

[20] P. Bailis, A. Fekete, M. J. Franklin, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Coordination avoidance in database systems. In *VLDB*, 2015.

[21] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Scalable atomic visibility with RAMP transactions. In *SIGMOD*, 2014.

[22] P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *ACM Queue*, 11(3), 2013.

[23] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. Probabilistically Bounded Staleness for practical partial quorums. In *VLDB*, 2012.

[24] S. Balakrishnan, A. Halevy, B. Harb, H. Lee, J. Madhavan, A. Rostamizadeh, W. Shen, K. Wilder, F. Wu, and C. Yu. Applying webtables in practice. In *CIDR*, 2015.

[25] D. S. Batory. On searching transposed files. *ACM Transactions on Database Systems (TODS)*, 4(4), Dec. 1979.

[26] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system rasdaman. In *SIGMOD*, 1998.

[27] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.

[28] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-Wesley New York, 1987.

[29] P. A. Bernstein and S. Das. Rethinking eventual consistency. In *SIGMOD*, 2013.

[30] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, 2005.

[31] E. Brewer et al. Lessons from giant-scale services. *Internet Computing, IEEE*, 5(4):46–55, 2001.

[32] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[33] D. D. Chamberlin. Early history of sql. *Annals of the History of Computing, IEEE*, 34(4):78–82, 2012.

[34] D. D. Chamberlin and R. F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264. ACM, 1974.

[35] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC*, 2007.

[36] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, et al. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[37] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[38] A. Cheung, O. Arden, S. Madden, A. Solar-Lezama, and A. C. Myers. StatusQuo: Making familiar abstractions perform using program analysis. In *CIDR*, 2013.

[39] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, 2014.

[40] J. Choi et al. ScaLAPACK: A portable linear algebra library for distributed memory computers—design issues and performance. In *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*, pages 95–106. Springer, 1996.

[41] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *ICDE*, 2013.

[42] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Transactions on Computer Systems (TOCS)*, 32(4):10, 2015.

[43] T. Condie, D. Chu, J. M. Hellerstein, and P. Maniatis. Evita raced: metacompilation for declarative networks. *Proceedings of the VLDB Endowment*, 1(1):1153–1165, 2008.

[44] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1–3):1–294, 2012.

[45] C. J. Date. An architecture for high-level language database extensions. In *SIGMOD*, 1976.

[46] C. J. Date. A critique of the SQL database language. *ACM SIGMOD Record*, 14(3), Nov. 1984.

[47] S. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM CSUR*, 17(3):341–370, 1985.

[48] J. Dean. Designs, lessons and advice from building large distributed systems (keynote). In *LADIS*, 2009.

[49] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[50] A. Deshpande. An initial study of overheads of eddies. *ACM SIGMOD Record*, 33(1):44–49, 2004.

[51] A. Deshpande and J. M. Hellerstein. Lifting the burden of history from adaptive query processing. In *VLDB*, 2004.

[52] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[53] D. DeWitt and M. Stonebraker. Mapreduce: A major step backwards. *The Database Column*, 2008.

[54] T. Dohzen, M. Pamuk, S.-W. Seong, J. Hammer, and M. Stonebraker. Data integration through transform reuse in the morpheus project. In *SIGMOD*, 2006.

[55] J. Duggan and M. Stonebraker. Incremental elasticity for array databases. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 409–420. ACM, 2014.

[56] A. Elmore, J. Duggan, M. Stonebraker, M. Balazinska, U. Cetintemel, V. Gadepally, J. Heer, B. Howe, J. Kepner, T. Kraska, et al. A demonstration of the BigDAWG polystore system. In *VLDB*, 2015.

[57] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[58] J. Fan, A. Gerald, S. Raj, and J. M. Patel. The case against specialized graph analytics engines. In *CIDR*, 2015.

[59] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM TODS*, 30(2):492–528, June 2005.

[60] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[61] M. J. Franklin. Concurrency control and recovery. *The Computer Science and Engineering Handbook*, pages 1–58–1077, 1997.

[62] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP*, 2003.

[63] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *ICDE*, 2011.

[64] J. E. Gonzales, R. S. Xin, D. Crankshaw, A. Dave, M. J. Franklin, and I. Stoica. Graphx: Unifying data-parallel and graph-parallel analytics. In *OSDI*, 2014.

[65] G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[66] G. Graefe. The five-minute rule twenty years later, and how flash memory changes the rules. In *DaMoN*, 2007.

[67] J. Gray. Notes on data base operating systems. In *Operating Systems: An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer Berlin Heidelberg, 1978.

[68] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[69] J. Gray and G. Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *ACM SIGMOD Record*, 26(4):63–68, 1997.

[70] J. Gray, P. Helland, P. ONeil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD*, 1996.

[71] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, Mar. 2006.

[72] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. Granularity of locks and degrees of consistency in a shared data base. Technical report, IBM, 1976.

[73] J. Gray and F. Putzolu. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for cpu time. In *SIGMOD*, 1987.

[74] T. J. Green, S. S. Huang, B. T. Loo, and W. Zhou. Datalog and recursive query processing. *Foundations and Trends in Databases*, 5(2):105–195, 2013.

[75] R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In *WDAG*, 1995.

[76] R. Guerraoui, M. Larrea, and A. Schiper. Non blocking atomic commitment with an unreliable failure detector. In *SRDS*, 1995.

[77] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *VLDB*, 1997.

[78] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.

[79] A. Halevy, P. Norvig, and F. Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, Mar. 2009.

[80] D. H. Hansson et al. Ruby on rails. `http://www.rubyonrails.org`.

[81] D. Harris. Forbes: Why Cloudera is saying ’Goodbye, MapReduce’ and ’Hello, Spark’, 2015. `http://fortune.com/2015/09/09/cloudera-spark-mapreduce/`.

[82] M. Hausenblas and J. Nadeau. Apache Drill: Interactive ad-hoc analysis at scale. *Big Data*, 1(2):100–104, 2013.

[83] P. Helland and D. Campbell. Building on quicksand. In *CIDR*, 2009.

[84] J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *ACM SIGMOD Record*, 39(1):5–19, 2010.

[85] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al. The MADlib analytics library: or MAD skills, the SQL. In *VLDB*, 2012.

[86] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems (TODS)*, 9(3):482–502, 1984.

[87] I. F. Ilyas and X. Chu. Trends in cleaning relational data: Consistency and deduplication. *Foundations and Trends in Databases*, 5(4):281–393, 2012.

[88] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *SIGMOD*, 1991.

[89] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[90] A. Jindal, P. Rawlani, E. Wu, S. Madden, A. Deshpande, and M. Stonebraker. Vertexica: your relational friend for graph analytics! In *VLDB*, 2014.

[91] P. R. Johnson and R. H. Thomas. Rfc 667: The maintenance of duplicate databases. Technical report, 1 1975.

[92] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *DSN*, 2011.

[93] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 1998.

[94] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *CHI*, 2011.

[95] J. Kepner et al. Dynamic distributed dimensional data model (D4M) database and computation system. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 5349–5352. IEEE, 2012.

[96] R. Kimball and M. Ross. *The data warehouse toolkit: the complete guide to dimensional modeling*. John Wiley & Sons, 2011.

[97] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, et al. Network virtualization in multi-tenant datacenters. In *USENIX NSDI*, 2014.

[98] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, et al. Impala: A modern, open-source sql engine for hadoop. In *CIDR*, 2015.

[99] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[100] B. Lampson and H. Sturgis. Crash recovery in a distributed data storage system. Technical report, 1979.

[101] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.

[102] B. Liskov and J. Cowling. Viewstamped replication revisited. Technical report, MIT, 2012.

[103] G. M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *SIGMOD*, 1988.

[104] R. Lorie and A. Symonds. A relational access method for interactive applications. *Courant Computer Science Symposia, Vol. 6: Data Base Systems*, 1971.

[105] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. In *VLDB*, 2012.

[106] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential consistency: measuring and understanding consistency at Facebook. In *SOSP*, 2015.

[107] H. P. Luhn. Auto-encoding of documents for information retrieval systems. *Modern Trends in Documentation*, pages 45–58, 1959.

[108] R. MacNicol and B. French. Sybase iq multiplex-designed for analytics. In *VLDB*, 2004.

[109] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.

[110] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[111] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory OLTP recovery. In *ICDE*, 2014.

[112] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The design and implementation of the 4.4 BSD operating system*. Pearson Education, 1996.

[113] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at what COST? In *HotOS*, 2015.

[114] E. Meijer. Your mouse is a database. *Queue*, 10(3):20, 2012.

[115] E. Meijer, B. Beckman, and G. Bierman. Linq: reconciling object, relations and XML in the .NET framework. In *SIGMOD*, 2006.

[116] J. Melton, J. E. Michels, V. Josifovski, K. Kulkarni, and P. Schwarz. Sql/med: a status report. *ACM SIGMOD Record*, 31(3):81–89, 2002.

[117] R. J. Miller, M. A. Hernández, L. M. Haas, L.-L. Yan, C. H. Ho, R. Fagin, and L. Popa. The clio project: managing heterogeneity. *SIGMOD Record*, 30(1):78–83, 2001.

[118] 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 in a data stream management system. In *CIDR*, 2003.

[119] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP*, 2013.

[120] J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, et al. The niagara internet query system. *IEEE Data Eng. Bull.*, 24(2):27–33, 2001.

[121] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms:[extended abstract]. In *Proceedings of the 31st symposium on Principles of Database Systems*, pages 37–48. ACM, 2012.

[122] F. Olken. *Random sampling from databases*. PhD thesis, University of California at Berkeley, 1993.

[123] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[124] P. E. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems*, 11(4):405–430, 1986.

[125] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, 2014.

[126] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: bringing order to the web. Technical report, Stanford InfoLab, 1999. SIDL-WP-1999-0120.

[127] R. Ramakrishnan and J. Gehrke. *Database management systems*. McGraw Hill, 2000.

[128] R. Ramakrishnan and S. Sudarshan. Top-down vs. bottom-up revisited. In *Proceedings of the International Logic Programming Symposium*, pages 321–336, 1991.

[129] V. Raman, A. Deshpande, and J. M. Hellerstein. Using state modules for adaptive query processing. In *ICDE*. IEEE, 2003.

[130] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, 2001.

[131] V. Raman and J. M. Hellerstein. Partial results for online query processing. In *SIGMOD*, pages 275–286. ACM, 2002.

[132] A. Rasmussen, V. T. Lam, M. Conley, G. Porter, R. Kapoor, and A. Vahdat. Themis: An i/o-efficient mapreduce. In *SoCC*, 2012.

[133] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[134] M. T. Roth and P. M. Schwarz. Don't scrap it, wrap it! a wrapper architecture for legacy data sources. In *VLDB*, 1997.

[135] L. A. Rowe and K. A. Shoens. Data abstraction, views and updates in RIGEL. In *SIGMOD*, 1979.

[136] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *SIGMOD*, 2015.

[137] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *SIGMOD*, 2015.

[138] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1), Mar. 2005.

[139] G. Salton and M. E. Lesk. Computer evaluation of indexing and text processing. *Journal of the ACM (JACM)*, 15(1):8–36, 1968.

[140] J. W. Schmidt. Some high level language constructs for data of type relation. *ACM Trans. Database Syst.*, 2(3), Sept. 1977.

[141] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[142] M. Shapiro, N. Preguica, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types. INRIA TR 7506, 2011.

[143] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, et al. F1: A distributed sql database that scales. In *VLDB*, 2013.

[144] N. Siva. 1000 genomes project. *Nature biotechnology*, 26(3):256–256, 2008.

[145] D. Skeen. Nonblocking commit protocols. In *SIGMOD*, 1981.

[146] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. Gonzalez, M. J. Franklin, M. Jordan, T. Kraska, et al. Mli: An api for distributed machine learning. In *ICDM*, 2013.

[147] M. Stonebraker. The land sharks are on the squawk box. *Communications of the ACM*. To appear.

[148] M. Stonebraker, D. Bruckner, I. F. Ilyas, G. Beskales, M. Cherniack, S. B. Zdonik, A. Pagan, and S. Xu. Data curation at scale: The data tamer system. In *CIDR*, 2013.

[149] M. Stonebraker and U. Çetintemel. "one size fits all": an idea whose time has come and gone. In *ICDE*, 2005.

[150] M. Stonebraker, G. Held, E. Wong, and P. Kreps. The design and implementation of ingres. *ACM Transactions on Database Systems (TODS)*, 1(3):189–222, 1976.

[151] M. Stonebraker, S. Madden, and P. Dubey. Intel big data science and technology center vision and execution plan. *ACM SIGMOD Record*, 42(1):44–49, 2013.

[152] M. Stonebraker and E. Neuhold. The laguna beach report. Technical Report 1, International Institute of Computer Science, 1989.

[153] D. Terry. Replicated data consistency explained through baseball. *Communications of the ACM*, 56(12):82–89, 2013.

[154] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, et al. Session guarantees for weakly consistent replicated data. In *PDIS*, 1994.

[155] The SciDB Development Team. Overview of SciDB: large scale array storage, processing and analysis. In *SIGMOD*, 2010.

[156] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a map-reduce framework. In *VLDB*, 2009.

[157] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost-based query scrambling for initial delays. *ACM SIGMOD Record*, 27(2):130–141, 1998.

[158] M. Vartak, S. Madden, A. Parameswaran, and N. Polyzotis. Seedb: automatically generating query visualizations. In *VLDB*, 2014.

[159] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In *CIDR*, 2011.

[160] A. N. Wilschut and P. M. Apers. Dataflow query execution in a parallel main-memory environment. In *Parallel and Distributed Information Systems, 1991., Proceedings of the First International Conference on*, pages 68–77. IEEE, 1991.

[161] E. Wong and K. Youssefi. Decompositiona strategy for query processing. *ACM Transactions on Database Systems (TODS)*, 1(3):223–241, 1976.

[162] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. In *VLDB*, 2013.

[163] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.