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

Readings in Database Systems

Fifth Edition

edited by
Peter Bailis
Joseph M. Hellerstein
Michael Stonebraker

Contents

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-