Rethinking Cost and Performance of Database Systems

Daniela Florescu
Oracle Corp.
dana.florescu@oracle.com

Donald Kossmann
28msec Inc. & ETH Zurich
donald.kossmann@28msec.com

ABSTRACT

Traditionally, database systems were optimized in the following way: "Given a set of machines, try to minimize the response time of each request." This paper argues that today, users would like a database system to optimize the opposite question: "Given a response time goal for each request, try to minimize the number of machines (i.e., cost in $)." Furthermore, this paper gives an example that demonstrates that the new optimization problem may result in a totally different system architecture.

1. INTRODUCTION

If you do DB research, then you optimize something. The fun of research is to define your own *something*; that is, define the constraints and the metrics of your optimization problem. If we are honest, however, the database community has pretty much answered and optimized the same question: "How to make a DBMS faster?" The only fun was defining the "scenario" in which to make the DBMS faster. Possible scenarios were new kinds of queries (e.g., analytics), new data types (e.g., images, polygons, or XML), and new architectures (e.g., streaming data). Nevertheless, the y-axes of the graphs of most SIGMOD and VLDB papers are the same and labelled with "[secs]" response time or "[tps]" throughput. Furthermore, the constraints such as strong consistency (i.e., ACID transactions) and the available hardware resources were fixed.

The purpose of this paper is to re-think what *better* means. That is, this paper tries to define how the y-axes of the graphs of future SIGMOD and VLDB papers should be labelled. The paper argues that those labels should change in the future. In a nutshell, rather than using "[secs]" or "[tps]", the y-axes should be labelled with "$". Furthermore, *consistency* should be a parameter that is varied when experimenting with a system. In other words, the purpose of this paper is to redefine the database optimization problem: define new constraints and different metrics to optimize.

This paper is motivated by several trends. Obviously, the most important trend are changing requirements for many applications. Many applications on the Web simply do not require strong consistency as mandated by the ACID paradigm; instead, these applications require scalability to millions of users and they require that no user is ever blocked by any other user. Furthermore, these applications require that all users get an answer within a second: There is no batch processing on the Internet and Web users do not understand why processing a complex query might take longer than processing a simple query. Another trend is the growing complexity of software systems and the distributed, service-oriented architecture of most software systems. Furthermore, there are technology trends such as cloud computing and large data centers with cheap hardware that have changed the assumptions of modern software architectures. The first contribution of this paper is to detail how these trends have changed the DB optimization problem.

A second contribution of this paper is a discussion on how the *new* database optimization problem might impact the architecture of a modern database system and possibly change some of the fixed parameters of traditional database research. As a starting point, this paper describes the architecture of the 28msec application server, an integrated database system, Web server, and virtual machine that runs on the Amazon Web Services platform (AWS).

The remainder of this paper is structured as follows: Section 2 lists the *new* requirements for modern database systems. Section 3 presents the architecture of the 28msec application server as an example for how these new requirements can be met. Section 4 discusses related work. Section 5 contains conclusions.

2. WHAT DO USERS CARE ABOUT?

Essentially, the requirements of users have not changed: They want it *all*. Also, the definition of *all* has not changed much: zero cost, zero response time, infinite throughput, infinite scalability in terms of users supported, linear scalability with the number of machines added, 100 percent predictability of cost and performance, ACID-style transactions, 100 percent availability for read and write requests, and flexibility to customize the system towards individual needs at any point in time with little effort. What has changed are the priorities and the constraints if the users cannot have it all. In a nutshell, the *traditional* database optimization problem can be defined as follows:

> *Given a set of hardware resources and guaranteeing full data consistency (i.e., ACID transactions), minimize the response time of requests and maximize the throughput of requests.*

The *new* database optimization problem can be defined as follows:

Feature	*Trad. DB*	*New DB*
Cost [$]	fixed	minimize
Performance [secs and tps]	optimize	fixed
Scalability [machines]	maximize	fixed
Predictability [$ and secs]	-	fixed
Consistency [%]	fixed	maximize
Flexibility [#variants]	-	maximize

Table 1: Traditional vs. New DB Optimization Problem

> *Given performance requirements of an application (peak throughput; maximum tolerable response times), minimize the required hardware resources and maximize the data consistency.*

Those two problems are not in conflict and, clearly, many techniques to optimize traditional database systems are applicable in the new world. As shown in Sections 3 and 4, however, the subtle differences may have significant impact on the architecture of a system. Table 1 summarizes these differences in the problem formulation; these differences are described in the remainder of this section in a bit more detail.

Cost. Again, this paper argues that *cost* as measured in $ is the main metric that needs to be optimized. The big question is no longer how fast a database workload can be executed or whether a particular throughput can be achieved; instead, the question is how many machines are necessary to meet the performance requirements of a particular workload. Cloud computing, increased cost for power and cooling, and the popularity of services like Amazon Web Services (AWS) have significantly contributed to this observation. Furthermore, hardware resources are no longer a one-time investment; instead, hardware resources are a significant lineitem on the monthly IT bill. Furthermore, cloud computing and AWS have made this cost a *continuous* metric (rather than *discrete* metric): The more hardware you consume, the more you pay and consumption is measured in a fine-grained way as CPU cycles and bytes of storage needed. Using AWS, costs are metered in *millidollars*. Traditionally, hardware resources have been provisioned in the granularity of machines; as a result, incremental costs are in the order of *kilodollars*.

The need to put cost into the performance metric has long been realized as part of the TPC benchmarks [1]. It has also been endorsed as part of the Mariposa project [12]. Unfortunately, most database research today still ignores this metric and even the TPC benchmarks mix cost and performance into a tps/$ metric. This paper argues for a more extreme approach to make cost alone *the* metric for most experiments. Furthermore, cloud computing services like AWS have made this metric comparable. Everybody can run experiments with various algorithms on AWS and report on "Amazon $" consumed. In the TPC benchmark reports, the $ metric is often questionable and controversial.

Performance (Response Time and Throughput). In most businesses, performance is a *constraint* and not an optimization goal. Google, for instance, is fast enough as it is; improving the response time of Google would not help. There are only few applications for which "faster is better" without any limits: Algorithmic trading in the financial sector is one of these rare examples. In all interactive applications (e.g., Web applications), a response time of a few hundred milliseconds is good enough in order to make users happy. In terms of througput, the requirement is to sustain a particular peak workload. Again, the big question in most IT scenarios is no longer whether it is *possible* to sustain the load: (Almost) everything is possible. Again, the question is at what *cost* it can be done; the "cheaper the better."

Another reason to put less emphasis on performance of a DBMS is that in practice, many performance problems are no longer caused by the DBMS. Modern IT systems are very complex and the DBMS is only one component of many. Again, improving the performance of the DBMS might not help.

Scalability. One basic assumption made today and in the discussion of the last two paragraphs on cost and performance is that every IT problem can be solved by throwing hardware (i.e., money) at it. Therefore, infinite scalability is a *must* in many modern IT systems. Ultimately, every business wants to grow, even if it is small at the beginning: Scalability involves that the IT costs grow linearly with the business and that this growth is unlimited. Unfortunately, this requirement is not met by many traditional database systems for which the cost function is a step function and the scalability is bound.

Predictability. For many businesses, predictability is a *must* in the same way as scalability. In other words, optimizing for the 99 percentile has become more imortant than optimizing for the average or mean. Here, predictability refers to both the predictability of performance and cost. Most database vendors have made a great deal of effort in order to reduce the cost of administration and make the performance of their systems more robust; nevertheless, database administration and provisioning of hardware resources for large-scale database applications is still a black art.

Consistency. ACID is great! SOA is greater!

With ACID, developers need not worry about consistency and can focus on the application logic. SOA (service-oriented architecture) helps developers to structure and more importantly evolve their applications.

Unfortunately, ACID and SOA are like water and oil: they do not mix well. ACID requires full control of all data management activities carried out by a transaction, whereas SOA mandates autonomy of all services involved in a transaction. While standards such as XA have helped to support distributed ACID-style transactions in a service-oriented infrastructure, supporting ACID transactions in large-scale distributed systems still remains painful. Fortunately, ACID transactions and strong consistency are rarely needed, as observed in a recent keynote by W. Vogels [16] and a paper by P. Helland [8]. Even for mission critical operations, a lower-level of consistency is often sufficient. [3] argues that the level of consistency should be flexible and that there should be a trade-off between the cost and consistency of the data. Furthermore, both [16] and [3] argue for a different definition of consistency levels along the lines of [15], as used for the design of large-scale distributed systems, rather than SQL isolation levels, as implemented by today's generation of commercial database systems. Again, this perspective is mandated by the importance of SOA as a design principle.

The requirement to provide 100 percent read and write availability for all users has also overshadowed the importance of the ACID paradigm as the gold standard for data consistency. In large Web

applications (e.g., Amazon, eBay, expedia, etc.), no user is ever allowed to be blocked; in particular, no user is allowed to be blocked by the actions of another user. Again, this requirement takes priority over the goal to achieve strong consistency. As a result, it is better to design a system so that it deals with and helps resolve inconsistencies, rather than having a system that prevents inconsistencies under all circumstances. "Shit happens" even with ACID and disregarding this observation limits the capabilities of a system. As a result, consistency is an optimization goal in modern IT systems in order to minimize the cost of resolving inconsistencies and not a constraint as in traditional database systems.

Flexibility. Flexibility is the ability to customize a software system to the individual needs of a customer. Flexibility makes it also easier, faster and cheaper to make new releases of a system. Flexibility is, thus, measured in the number of variants of a system that are deployed. Flexibility is particularly important for applications that have a diverse user base; typical examples are enterprise applications such as SAP R/3 or Oracle Finance. As applications become more complex, flexibility as a design goal has become increasingly important. A recent indicator for the importance of this requirement is the success of the Salesforce AppExchange technology. Traditional OLTP systems such as the credit/debit systems of the 1970s did not have this requirement so that flexibilty has not been a pressing optimization goal for traditional database management systems. Obviously, providing good performance and scalability is much easier if the system need not be flexible.

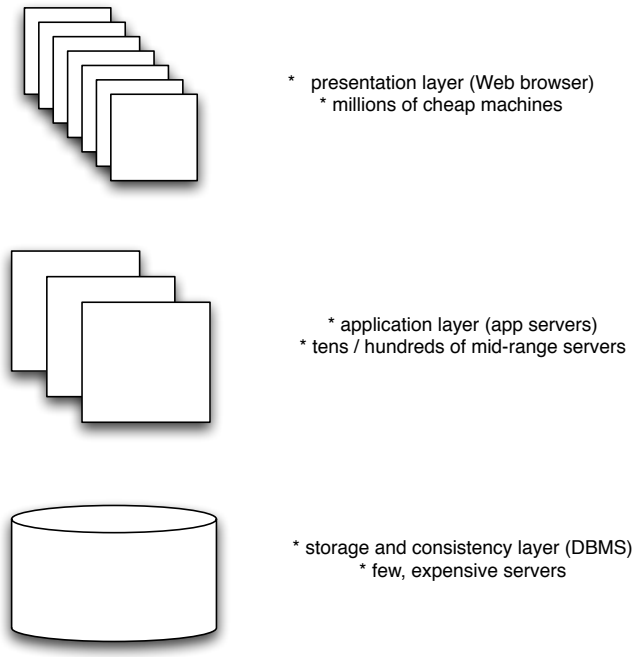

Figure 1: Classic Database Architecture

3. DOES IT MATTER?

This section revisits the *classic* database architecture that was designed to optimize the *traditional* DB problem (Table 1). Furthermore, this section describes a new, different database architecture that was designed to optimize the *new* DB problem.

3.1 Traditional Databases Revisited

Figure 1 shows the *classic* three-tier architecture in order to build database applications. This architecture was pioneered by SAP and variants of this architecture are still the state-of-the-art in order to build database applications. All requests are initiated by users at the presentation layer, e.g., using a Web browser. The application logic is encoded in the middle tier; the middle tier also involves functionality such as the Web server. All data management is carried out at the lowest tier using a DBMS (e.g., IBMD DB2, Microsoft SQL Server, MySQL, Oracle etc.).

The beauty of this architecture is that it serves extremely well the requirements of traditional database systems (Table 1). The main constraint of keeping data consistent is achieved in the bottom tier, inside the database server. The main optimization goal of minimizing response times and maximizing throughput is achieved by a series of techniques that have been developed over the last four decades in all tiers: caching, indexing and data partitioning, to name just a few. Furthermore, scalability is achieved on the two top tiers: at those tiers, the architecture of Figure 1 scales almost infinitely. Scalability, however, is limited at the bottom tier.

Unfortunately, the three-tier architecture of Figure 1 is not a good fit for the *new* requirements of Table 1. Most importantly, this architecture is expensive. Typically, significant investments are needed for the bottom tier, thereby involving expensive hardware (rather than cheap hardware). The hardware must be provisioned for *expected peak performance*, which can be orders of magnitudes higher than the *real average performance*. As a result, a great deal of hardware is wasted and it is difficult to re-use those unused hardware resources for other purposes. Furthermore, maintaining the DB infrastructure is expensive and does not get automatically cheaper, like anything else. Finally, the DB software itself can be a cost factor: Most of the functionality is not needed by an application so that clients are forced to pay for unneeded features. This situation is improving with open source database systems such as PostGres and MySQL; nevertheless, many organizations are still locked into expensive commercial database systems.

The classic three-tier DB architecture and the way it is implemented today also does not meet the predictability requirements. With many concurrent clients accessing the same database, it is virtually impossible to understand what is happening at the database level. Also, today's generation of database products have just not been designed for predictability. Furthermore, as mentioned above, scalability is limited in this architecture to the top two tiers.

Flexibility is another design goal that is not supported well by traditional database architectures. The reason is that, in the state-of-the-art, different technologies are used at all three tiers. SQL and the relational model are used at the bottom tier; OO is used in the middle tier; and XML/HTML with some client-side scripting are used at the top tier. As a result, a change in functionality (i.e., customization) must be implemented at all three tiers, thereby using three different technologies. Such customizations are expensive to implement and difficult to test.

It is interesting to re-iterate two design principles of the classic three-tier architecture for database applications. The first principle is *control*. At the bottom tier, the DBMS controls all hardware resources and all accesses to the data. The second design principle is typically referred to as *query shipping* [6]. Query shipping means that as much functionality as possible is pushed to the bot-

tom tier; e.g., as stored procedures or exploiting other features of modern database systems (e.g., new data types and query capabilities). On the positive side, both of these design principles have helped to achieve the two most important design goals of traditional database systems: consistency and performance. Query shipping was also motivated by the business model of most DBMS vendors: New DBMS licenses could only be sold by providing more functionality. On the negative side, these two design principles have turned DBMSes into monolithic monsters, getting bigger and bigger, never smaller. This trend has hurt predictability, flexibility, and scalability. Furthermore, these two design principles have made IT systems expensive because expensive hardware is needed in order to sustain the load at the bottom tier and because the complexity of modern DBMS has its price. Not surprisingly, therefore, the architecture presented in the next subsection has exactly the inverse design principles.

3.2 A New Architecture

Figure 2 shows an architecture designed for the *new* database optimization problem. Comparing Figures 2 and 1, both architectures look similar at first glance, but the differences are significant. The key ideas of the architecture of Figure 2 can be summarized as follows:

- Consistency is not handled at the storage layer, but at the application layer. At the bottom layer, there is only a large-scale distributed storage such as the one provided by Amazon S3.
- Consistency in the middle-tier is achieved by distributed protocols such as those devised in [15]. That is, there is no entity that controls all accesses to the storage. Instead, all application servers that access data agree on *conventions* on how to read and update data stored in the bottom layer.
- Cheap hardware can be used in *all* layers. All layers are designed to scale to thousands, if not millions of machines. The whole architecture is designed such that any node can fail at any time. At the bottom-tier, fault tolerance is achieved by replicating data and by giving poor consistency guarantees only (i.e., eventual consistency). In the upper layers, machines are stateless so that the worst that can happen if a machine fails is that work of the currently active transactions is lost.
- Essentially, the traditional DBMS has disappeared from the architecture of Figure 2. The main DBMS functionality, i.e., transaction and query processing, has been integrated into the application layer.
- It is possible to run both the application layer and the presentation layer on an end user's machine. Again, processes at both layers are stateless and the architecture has been designed so that processes at both of these layers can fail at any time.

This architecture has been implemented by 28msec as part of its Sausalito product[1]. Sausalito integrates a virtual machine, data management system, data stream processor, queue, and Web server in a single portable platform. This sounds like yet another monster, even more than a DBMS, but in fact it is not. At the moment, the footprint of the whole platform is about 140 mega-bytes, mostly because it has a leaner and more focused functionality.

[1]More information can be found at http://www.28msec.com.

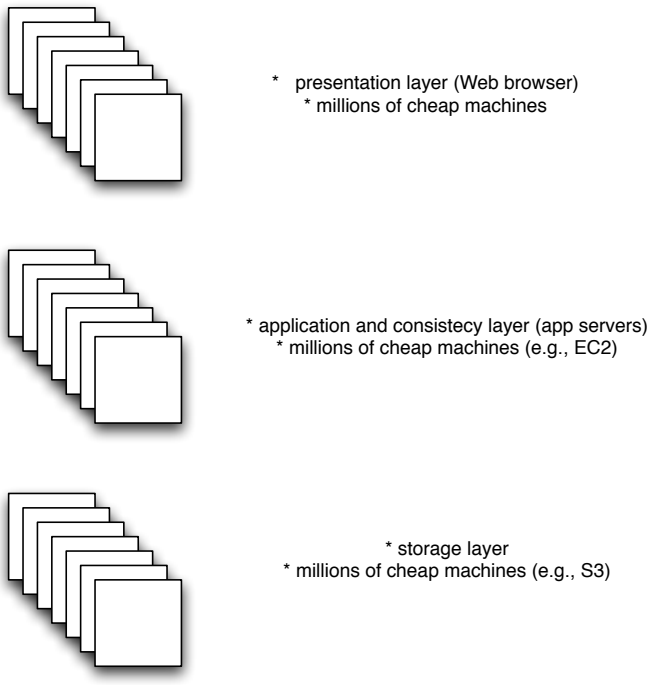

Figure 2: 28msec Architecture

With Sausalito, all data and (pre-compiled) application code is stored as blobs using Amazon S3. Each HTTP request (e.g., a click of a user in a Web browser or a Web Service call from an application) is processed in the following way: Using Amazon EC2 and its scheduling and load-balancing service, an available EC2 server is selected to process that request. This EC2 server loads from S3 the pre-compiled code which handles this request. This pre-compiled code is then interpreted on the EC2 server by the Sausalito runtime system, thereby possibly involving accesses to objects in the database which are also stored as blobs in S3. All EC2 servers are stateless and can fail at any time. If the load increases (decreases), EC2 servers can be added (dropped, respectively). Amazon S3 uses cheap hardware at the storage layer and achieves availabilty by replicating all blobs multiple times across different data centers. Synchronoziation of concurrent accesses to the same data (from multiple EC2 servers) is effected using the protocols described in [3].

As a programming language to implement all application logic and database access, Sausalito supports XQuery (including updates and scripting extensions). Sausalito uses XQuery because it supports all Web standards nicely (in particular, REST and Web services), integrates database queries, and is sufficient to build full-fledged Web-based applications. For the same purpose, Google's AppEngine uses Python with an embedded SQL-like dialect for database access. Microsoft relies on the .NET programming languages with LINQ.

Obviously, there is little hope that any system (including Sausalito) that implements the architecture of Figure 2 is able to match

the state-of-the-art in terms of performance and consistency. With regard to consistency, it is not possible to implement strong consistency, high availability and scalability at the same time, according to Brewer's CAP theorem [7]. With regard to performance, state-of-the-art DBMSes have been effectively optimized for fourty years. The architecture of Figure 2, however, works well for the other features of Table 1. As shown with Sausalito, it can be implemented in a cost-effective way by using cheap hardware everywhere. Furthermore, it was designed for scalability. Flexibility is achieved by a simplified platform and by using only one programming and data model, rather than different models at all levels. As shown in [3] for many workloads, predictable cost and performance can be achieved, too.

From the discussion of this section, it should have become clear that the architecture of Figure 2 was devised with the opposite design principles as compared to the architecture of Figure 1. Rather than controlling all reads and writes to the data, there are conventions that rule access to data in a distributed and loosely coupled way; these conventions are implemented as distributed protocols [15, 3]. Instead of moving functionality to the data, the storage layer has a minimal "get" and "put" interface. All other functionality is implemented at the application layer. One particular aspect is security: Security must be implemented at the application level, thereby encrypting all data stored in the storage layer and controlling the dissemination of keys to access that data. We expect that implementing security is not more difficult in the architecture of Figure 2 than in the architecture of Figure 1 because security must be considered at the application and presentation layers anyway.

4. RELATED WORK

Obviously, we are not the first to question modern DBMS architectures. All ideas presented in this paper have been floating around for a while and the purpose of this position paper is to try to put the pieces of the big picture together. Obviously, the recent hype about cloud computing and Amazon Web Services (e.g., S3 and EC2) and the Google AppEngine services have cleared the path towards re-thinking the cost of modern information systems. The impact of SOA design principles on database application architectures has been addressed recently in talks by Vogels [16] and Helland [8]. Furthermore, there have been a number of relevant techniques that take a different spin towards database optimization such as Eddies [2] and query optimization for the expected and worst case [4].

In the late 1990s, people realized that big application systems such as SAP use a DBMS only as a glorified file system with a built-in persistent B-tree [5]. At the time, the reaction of the DBMS vendors was to ask SAP how they could *extend* their products in order to better meet SAP's requirements. The right question to ask would have been how to repackage DBMS functionality in order to better integrate into the application architecture. Unfortunately, this question was never asked.

The most relevant recent work on database architecture is Stonebraker et al.'s observation that "one size does not fit all" [14, 13]. That work showed the current short-comings of state-of-the-art database management systems in several important application areas: stream processing, decision support (i.e., OLAP), and transaction processing (i.e., OLTP). Comparing that work with the observations presented in this paper, there is a great deal of commonality. First, both lines of work agree that state-of-the-art database systems do not seem to be optimal for anything anymore. Second and more importantly, both lines of work argue that DB functionality can be *repackaged* effectively and in a lean way in order to better match the needs of applications. Third, while Stonebraker et al.'s work is focussed on performance and achieving orders of magnitude performance improvements, the same arguments can actually be applied to *cost* which is the focus of our work. Fourth, the architectural principles discussed in Section 3.2 do not contradict in any way with the technology proposed in [14, 13]. In some sense, our work builds on top of the results of Stonebraker et al. and is not at all in conflict with that work; Stonebraker et al.'s work is geared towards how to build modern data management systems (e.g., column store, on-the-fly data processing, and simplified synchronization) whereas the purpose of this paper is to present the bigger picture and how to integrate that technology into the whole application and technology stack.

There has also been a great deal of work on improving the scalability of database management for the Web. Most significantly, there has been work on caching and materialized Web views; e.g., [10, 17, 11]. The goal of all that work is to increase the scalability of modern Web-based systems by off-loading the database. Furthermore, as mentioned earlier, cost (measured in $) has played a role in the TPC benchmarks. It has also been used in a number of frameworks in order to improve performance and quality of service in a distributed information system. Prominent examples are the Mariposa system [12] and the "Quality Contracts (QC)" framework [9].

5. CONCLUSION

The purpose of a database system is to help developers to write applications, deploy their applications and run their applications. In the 1960s, most of the database applications were simple debit/credit transactions: database systems were a huge help and almost solved the whole problem: It was well affordable to spend a large portion of the IT budget on database software and administration. In the meantime, applications have grown and databases (and middleware) only solve a relatively small fraction of the problem. As a result, database systems are creating often more pain than they actually help because database systems are just one of many components, yet highly demanding and often dictating the whole application architecture. In a nutshell, the assumptions and the goals of the usage of database systems have changed over the last fourty years, when the first generation of database systems was built.

The purpose of this paper is to rethink the assumptions and goals and, thus, redefine the *DB problem*. The paper tried to do that for Web-based applications such as, e.g., an online bookstore or a car-pooling service. The result was a new problem statement, and not surprisingly a new architecture, and a different packaging of database functionality. Obviously, these results are biased by the particular application scenarios of Web-based, interactive applications. Other applications such as data warehousing and decision support may result in different results. Furthermore, the proposed new database application architecture is not mature: While it seems that big players like Google and start-ups like 28msec are moving to such an architecture, there is still a great deal of work needed before there are products which are ready for prime time. Nevertheless, the key message of this paper stays the same; citing Goetz Graefe (in the mid 1990s): "Database vendors should not be building a Ferrari - they should be building a Ford Taurus."

Acknowledgments. We would like to thank Paul Hofmann (SAP) for discussions and many helpful comments on this paper.

6. REFERENCES

[1] Anon et al. A measure of transaction processing power. *Datamation*, 1984.

[2] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. of ACM SIGMOD*, pages 261–272, Jun 2000.

[3] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *Proc. of ACM SIGMOD*, pages 251–264, Jun 2008.

[4] F. Chu, J. Halpern, and J. Gehrke. Least expected cost query optimization: What can we expect? In *Proc. of ACM PODS*, pages 293–302, Jun 2002.

[5] J. Doppelhammer, T. Höppler, A. Kemper, and D. Kossmann. Database performance in the real world: TPC-D and SAP R/3. In *Proc. of ACM SIGMOD*, pages 219–230, Jun 1997.

[6] M. Franklin, B. Jonsson, and D. Kossmann. Performance tradeoffs for client-server query processing. In *Proc. of ACM SIGMOD*, pages 149–160, Jun 1996.

[7] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services. *SIGACT News*, 33(2):51–59, 2002.

[8] P. Helland. Life beyond distributed transactions: An apostate's opinion. In *Proc. of CIDR Conf.*, pages 132–141, Jan 2007.

[9] A. Labrinidis, H. Qu, and J. Xu. Quality contracts for real-time enterpises. In *Business Intelligence for the Real-Time Enterprises*, pages 143–156, August 2007.

[10] A. Labrinidis and N. Roussoupoulos. Webview materialization. In *Proc. of ACM SIGMOD*, pages 367–378, Jun 2000.

[11] Q. Luo, S. Krshnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. Lindsay, and J. Naughton. Middle-tier database caching for e-business. In *Proc. of ACM SIGMOD*, pages 600–611, Jun 2002.

[12] M. Stonebraker, P. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *VLDB Journal*, 5(1):48–63, 1996.

[13] M. Stonebraker, C. Bear, U. Cetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. Zdonik. One size fits all? Part 2: Benchmarking studies. In *Proc. of CIDR Conf.*, pages 173–184, Jan 2007.

[14] M. Stonebraker, S. Madden, D. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *Proc. of VLDB Conf.*, pages 1150–1160, Sep 2007.

[15] A. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, Upper Saddle River, NJ, 2002.

[16] W. Vogels. Data access patterns in the Amazon.com technology platform. In *Proc. of VLDB*, page 1, Sep 2007.

[17] K. Yagoub, D. Florescu, V. Issarny, and P. Valduriez. Caching strategies for data-intensive web sites. In *Proc. of VLDB*, pages 188–199, Sep 2000.