Declarative Scheduling in Highly Scalable Systems

Tilgner, Christian

DOI: https://doi.org/10.1145/1754239.1754285

Declarative Scheduling in Highly Scalable Systems

Christian Tilgner
Database Technology Research Group
Department of Informatics
University of Zurich, Switzerland
tilgner@ifi.uzh.ch
Advisor: Carl-Christian Kanne
University of Mannheim, Germany

ABSTRACT

In modern architectures based on Web Services or Cloud Computing, a very large number of user requests arrive concurrently and has to be scheduled for execution constrained by correctness criteria, service-level agreements etc. The state of the art is to develop hand-coded schedulers, though this tails great costs, long development times, reduced developer productivity and inflexibility of mapping frequently changing requirements. In this paper, we present our approach for a scheduler component that can be programmed using declarative rules. Instead of handling one request at a time, we propose to treat sets of requests as data collections and to employ database query processing techniques to produce high-quality schedules in an efficient manner. Our declarative scheduler will allow for a more flexible and productive way to define existing scheduling protocols, service level agreements and novel application specific consistency protocols. First results presented here are encouraging and motivate for further investigation.

1. INTRODUCTION AND MOTIVATION

In modern architectures based on Web Services or Cloud Computing, a very large number of user requests arrive concurrently and has to be scheduled for execution. Acceptable schedules are constrained by (1) correctness criteria (e.g. classical serializability), (2) service-level agreements/SLAs (e.g. for premium vs. free customers in Web applications), and (3) performance requirements.

The state of the art is to develop hand-coded schedulers for a given application. This yields fine-tuned schedulers, albeit at a great cost and with long development times. Given the rapid evolution of Web applications, a critical issue for the development of schedulers is developer productivity, which affects time-to-market. Mapping frequently changing requirements, affecting all criteria (1)-(3) above, to modified scheduler code is tedious and error-prone.

The project outlined in this paper aims to investigate *declarative* request scheduling, where the constraints on schedules are formulated as declarative rules. Instead of creating schedulers conforming to those rules by hand, we propose to treat sets of requests as data collections, and to employ database query processing techniques to produce high-quality schedules in an efficient manner.

This approach has two advantages. Firstly, declarative rules are much more concise, easier to understand, and easier to modify than imperative scheduler code. Hence, developer productivity is increased. Secondly, optimization techniques from declarative query processing can be used to improve scheduler performance without affecting the scheduler specification.

The goal of the project is to develop a scheduler component that can be programmed using declarative rules. In a first approach, we look at schedulers as middleware that sits between the requestors and the servers, disable the server's own schedulers as far as possible, and use the schedules produced by our declaratively programmed component. In the long term, we target scheduling in Cloud Computing environments, where dynamic workloads require highly flexible scheduling. For example, reduced consistency criteria may be used during times of high load.

Research questions include, but are not limited to:

- To what extent can existing query languages be used to capture typical constraints on request schedules?
- What is the performance of schedulers based on query processing?
- What should specialized languages for declarative scheduler programming look like?
- How can the performance of declaratively programmed schedulers be improved?

This paper is organized as follows: Related work is presented in Section 2. Section 3 discusses the main idea of the PhD project, our research objectives as well as the architecture of our declarative scheduling solution. The results achieved so far are described in Section 4. Finally, Section 5 outlines ongoing and future work.

2. RELATED WORK

Scheduling takes place in many areas including database management systems (DBMS), networks [14], storage servers [13], web servers [11] etc. When scheduling concurrent requests, constraints like SLAs or correctness have to be ensured.

EDBT 2010, March 22-26, 2010, Lausanne, Switzerland.

The ACTA framework [5] defined a comprehensive logical framework to formally describe constraints on operation schedules. These constraints were formulated as first-order logic sentences and used to reason about correctness and to synthesize new transaction models. Our goal is to use similar logical formulations of constraints, and *execute* them as queries on pending and historical request data to perform declarative scheduling.

Another area of research are middleware approaches. A lot of research has been done in this area focusing on improving user scalability and performance of DBMSs. This is necessary because the user scalability of DBMSs is limited, which is why they are not perfectly applicable for highly scalable systems. Since QoS plays an increasing role, there are middleware approaches trying to add support for QoS on top of standard DBMSs because they do not provide an effective differentiation between transactions, e.g., transactions from users with different priorities [20]. Such middleware approaches are mostly using queues and external schedulers scheduling enqueued requests according to defined SLAs. Scheduling with consideration of SLAs is even more complex and complicated in highly scalable systems. Due to space limitations, only the most closely related middleware approaches are described in the following.

Schroeder et al. present an *external queue management system* (EQMS) which schedules request enqueued in an external queue. They also adjust the multiprogramming level (MPL) of the underlying DBMS [20] and use external priorization [21] to ensure QoS targets. The database replication middleware *Ganymed* intends to provide load balancing as well as user scalability, the latter without sacrificing consistency. It also purposes to improve the performance of read operations. Therefore, Ganymed is using an algorithm differentiating between update and read-only transactions [19]. Krompass et al. present a service level objective-aware *workload management system* (WLMS) capable of handling online transaction processing (OLTP) and business intelligence (BI) workloads [16]. They focus on classifying queries, fulfilling QoS as well as recognizing and automatic handling of problem queries. Individual requests are penalized, depending on the SLA and the current degree of SLA conformance. *Clustered JDBC* (C-JDBC), a Java middleware framework for database clustering, implements the Redundant Arrays of Inexpensive Databases (RAIDb) concept and offers a single database view to client applications [4]. Its focus lies on offering a high availability and performance scalability. In [7], Elnikety et al. present a *gatekeeper proxy* (GP) providing external admission control and request scheduling to improve performance. Bhatti and Friedrich propose *WebQoS*, a system for supporting server QoS [2]. Their solution uses admission control and scheduling based on several scheduling policies to improve performance and to support distinct service levels. *QShuffler* is a query scheduler that focuses on the problem of scheduling large batches of queries in BI settings such as report generation [1]. Its goal is to minimize the total completion time of workloads.

Table 1 summarizes whether the mentioned approaches cover the following aspects: improving/ensuring performance (**P**), support of **QoS**, ability of a declarative (**D**) definition of scheduling protocols, flexibility (**F**) by the possibility of changing protocols instead of using hard coded scheduling protocols and focusing on achieving a higher user scalability (**HS**).

Table 1: Related Approaches (P-Performance, QoS-Quality of Service, D-Declarativity, F-Flexibility, HS-High Scalability)

Approach	P	QoS	D	F	HS
EQMS	+	+	-	-	-
Ganymed	+	-	-	-	+
WLMS	+	+	-	-	-
C-JDBC	+	-	-	-	+
GP	+	-	-	-	-
WebQoS	+	+	-	+	-
QShuffler	+	-	-	-	-

Database vendors also realized the importance of QoS. They developed commercial products enabling QoS and improving performance. Such tools are, for instance, the IBM DB2 Query Patroller [6] or the Oracle Resource Manager [18].

Our work is different from all these approaches and tools in that they all focus only on certain aspects such as QoS or performance. None of these approaches covers a declarative specification of applied scheduling algorithms. Instead, they use fixed, imperatively implemented algorithms. Hence, applications following these approaches are not flexible enough to react to changing requirements or evolving business processes. Changes of business processes can require modifications of applied scheduling algorithms and, thus, cause expensive reimplementation. This must be solved in a better and more flexible way.

In this paper, we present a declarative solution where scheduling protocols can be defined declaratively which allows for easier and more productive development as well as higher flexibility.

In addition, none of the mentioned approaches considers application-specific consistency, and no one allows for adaptable relaxed consistency. In practice, it turned out that relaxed consistency is necessary for highly scalable systems [8]. As described by the CAP theorem [10], it is not possible to preserve scalability and availability while achieving the same level of consistency like DBMSs (i.e. ACID transactions) [3]. Techniques like strict or strong consistency and database-style transactions do not scale at Internet level and are rarely needed in modern large-scale distributed systems anyway [3][22][12]. For most parts of modern highly scalable web applications, e.g., hotel or flight reservation systems, or Internet shops like Amazon relaxed consistency is sufficient. Thus, [22] and [9] claim for new consistency levels which are different from the SQL isolation levels used by DBMSs.

Most approaches try to ensure high consistency. But there is also a great deal of work on relaxing consistency. For instance, [3] presents protocols for weaker levels of consistency. [15] proposes Consistency Rationing, a concept that allows to switch between only two consistency guarantees automatically at runtime. But they do not allow a declarative definition of application specific consistency models which will be possible using our declarative scheduling solution. This way, we would be more efficient and productive than implementing application specific consistency by hand, as Amazon, Yahoo, Ebay, Google and Flickr do.

3. RESEARCH APPROACH

In this section, we introduce our approach for schedul-

ing in highly scalable systems, a declarative programmable scheduler. The underlying idea, the research objectives, the scheduler architecture as well as evaluation methods will be explained in the following subsections.

3.1 Method

This subsection presents the main idea of our approach, gives an overview of the existing kinds of requests and describes which languages could be used for our scheduler.

Depending on the kind of scheduler, requests are of different nature. Requests can be (1) atomic read/write database statements, (2) regular database statements comprising simple queries or more complex statements, (3) web service invocations such as XML-based SOAP requests, (4) requests to storage servers to store respectively access files etc.

For all these kinds of requests, the *general problem* of the request scheduler is how to determine the execution order of a large number of concurrent requests efficiently enforcing several constraints. Such constraints are e.g. SLAs as well as correctness which can be described by the two-phase locking (2PL) protocol. The more concurrent requests exist, the more complex and problematic this problem is.

Up to now, request schedulers mostly use fixed scheduling algorithms. Their procedural implementation is complex and difficult to understand. Hence, applications following these approaches are not flexible enough to react to changing requirements or evolving business processes. Changes of business processes can require modifications of applied scheduling algorithms and, thus, cause expensive reimplementation.

To facilitate the problem of determining the execution order of many concurrent requests, our idea is to *treat requests as regular data*. Requests can, thus, be inserted into a DBMS. This allows the usage of query processing to access the stored requests to identify an appropriate execution order. Since SLAs and correctness constraints do not solely relate to pending requests, but also to previously executed requests, both request types have to be stored.

We decided to develop a scheduling solution which uses a declarative language to specify scheduling protocols. The usage of a declarative language allows for a high-level description of applied scheduling protocols. The selection of the actual scheduling algorithm is now performed by the scheduling system. Hence, programming scheduling protocols can be learned more easily and requires less effort than an imperative implementation. Our declarative approach requires a less extensive description and fewer lines of code of scheduling protocols, facilitating higher developer productivity. Another advantage of using a declarative language is the higher flexibility. Our declarative scheduler will (a) be capable of defining traditional consistency protocols (e.g. variants of 2PL), (b) allow for the definition of SLAs, and (c) facilitate the specification of new application-specific consistency protocols. There are several declarative candidate languages for our scheduler, such as SQL, Datalog, XSLT or XQuery. After we have defined all necessary issues, the most suitable declarative language has to be evaluated. Issues that have to be considered are, for example ordering capabilities and applied data models that have to comply with the request data models. By such a scheduler language, we can define scheduling protocols which realize certain correctness or SLA rules. These rules can be specified, e.g. using the ACTA framework [5].

3.2 Initial Research Objectives

The research objectives we see in this PhD project are described here. In our approach, we see requests as regular data. The main idea is to solve the problem of determining the execution order of concurrent requests like querying data using declarative query processing.

Our objective is to investigate to what extent declarative query processing is suited for request scheduling in highly scalable systems. We strive for a declarative programmable scheduler allowing for a more flexible and powerful method to schedule requests. In order to achieve this objective, we have to address the following research objectives. We will:

1. evaluate existing declarative query languages and their capabilities to specify scheduling protocols.
2. design and implement an architecture of a declarative scheduler.
3. evaluate the performance of the declarative scheduler and compare it to existing systems.
4. design a specialized language and system based on the experiences gained from objectives 1-3.

3.3 Architecture

In this subsection, we discuss possible scheduler architectures and describe the architecture we chose. Furthermore, we explain the operating mode of our declarative scheduler.

We strove for a solution which prevents us from wasting time with implementation effort such as changing database internals. Instead, we preferred a realization which permits to gain experience in a short time. Furthermore, we aspired to a database independent solution which allows the application of different DBMSs.

To meet these objectives, we opted for a middleware scheduler. In the chosen approach, clients connect to the middleware scheduler instead of connecting directly to the server. The external scheduler decides on the order in which these requests are sent to the server. We call this kind of scheduling *external scheduling*. *Internal scheduling* means that clients send requests directly to a server which does the scheduling itself. Using a middleware solution has the advantage that the scheduler can be designed independently from the underlying storage solution. But it also raises challenges. External scheduling is more difficult compared to internal scheduling because less database information is available externally, e.g. information about the database state. Furthermore, the external scheduler has to consider consistency aspects respectively simulate locking. Therefore, a way has to be found to identify which database data is accessed by a request. This will require application knowledge, i.e. knowledge about the kind of statements. [17] gives a possible solution to this problem.

Another possible solution would be to extend an open-source DBMS, such as PostgreSQL, with declarative scheduling facilities. But this alternative has the disadvantage that it is bound to only one specific DBMS which does not match our requirements.

In general, in our middleware architecture illustrated in Figure 1, the clients do not connect to the server directly. Instead, they connect to the scheduler which schedules the requests, sends them to the server and returns their results to the clients. To schedule the requests, we make use of

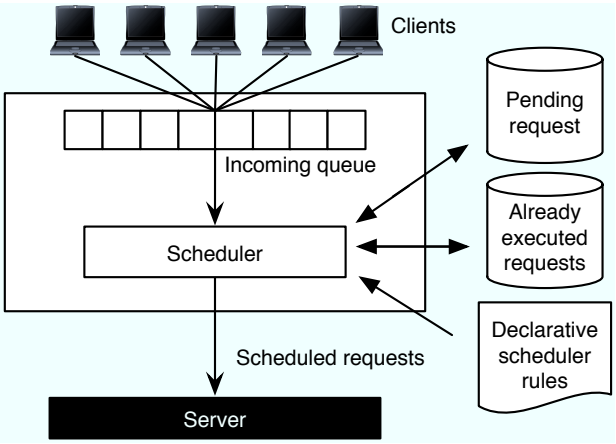

Figure 1: Scheduler architecture overview

a DBMS, store the requests in a database and use query processing to select and identify all executable requests.

In detail, when clients connect to the external scheduler, a control instance creates a separate client worker for each connected client. Each client worker communicates with its client. If the client worker receives a request from its client, the request is, in a first step, buffered in an incoming queue. Periodically, the scheduler gets triggered, e.g. by one of the client workers. The trigger condition can be configured (dynamically). The best condition has to be evaluated experimentally. Possible conditions are, e.g. a lapse of time, a certain fill level of the incoming queue or a hybrid version. When the scheduler gets triggered, it empties the incoming queue and moves all requests into the pending request database as a batch job. In the third step, the scheduler runs a query on the requests stored in the pending request database that realizes certain scheduling rules. Thereby, the scheduler accesses a second database, called history database, in which all relevant prior executed requests are stored. From this history database, all necessary information about the current database state etc. can be obtained. The result of the query is an ordered schedule of the next requests qualified for execution. These requests are then inserted into the history database and deleted from the pending request database. All qualified requests are now sent to the server and, if possible, executed as a batch job, whereby we expect a performance improvement. Finally, the results are returned to the clients.

To be able to to measure the real declarative scheduling overhead, we will design the scheduler to be able to run in a non-scheduling mode. In this mode, the scheduler forwards the requests to the server without scheduling. This way, the server undertakes the task of doing request scheduling.

3.4 Evaluation

Here, we describe how we plan to evaluate our approach. With our declarative scheduler, we want to offer a less complex and more productive method to define scheduling protocols compared to the implementation by hand done so far.

Since we strive for a scheduler with good performance, we will experimentally evaluate how close we can approximate to existing schedulers, e.g. database schedulers.

To evaluate the aspects complexity and productivity, we will carry out a study and instruct people to implement existing and new scheduling protocols declaratively and imperatively. Afterwards, we will compare the function points as well as lines of code of both approaches.

To test the scalability of our declarative scheduler, we will run it in a multi-user test bed and compare the results with existing solutions.

4. EXAMPLE

We illustrate our approach to scheduling with a small example and preliminary experimental results. This example shall not be understood as final result. Its purpose is to show how to describe scheduling protocols declaratively. Therefore, we chose the well-known strong 2PL (SS2PL) protocol.

In our experimental scenario, the server from Figure 1 is a commercial DBMS, and we want to schedule database statements. As the scheduler language, we chose SQL. The workload consists of small OLTP-style transactions. Our goal is to find out what a classical scheduling protocol (SS2PL in this case) looks like when formulated using SQL, and how a simple query processing-based declarative scheduler performs against the native, lock-based scheduler of the DBMS. We chose this scenario for its simplicity and ease of exposition, not because we think it is the best application area for declarative scheduling.

4.1 Method

We first tried to identify a lower bound for the overhead of the native scheduler. We measured throughput in multi-user mode under isolation level serializable, with a varying number of concurrently active clients. In a separate run, we also logged the produced schedule. We then reran this schedule with a single concurrent transaction, and locking disabled as much as possible (see below). The difference in run-time is a lower bound for the scheduling overhead of the native scheduler. We also measured the run-time of a query that, given a table of pending requests and a table of previously executed requests, computes those pending requests that can be executed according to the SS2PL protocol [23], guaranteeing serializability. This gives us an estimate on the overhead of declarative scheduling.

4.2 Native Scheduler Overhead

4.2.1 Setup

We ran the experiments on a machine with a 2.8GHz single-core CPU and 2GB memory. Our workload consisted of transactions with 20 SELECT and 20 UPDATE statements against a single table of 100000 rows. Each statement affected exactly one random row, with a uniform probability for each row. We used a database instance that fitted in the database buffer and averaged results over multiple runs. We generated a workload and checked how many transactions were committed under isolation level serializable in 240s for concurrently active client counts between 1 and 600. We then measured how much time the same workload took in single-user mode. This meant that we acquired an exclusive lock on the table to reduce locking overhead and processed the same statement sequence in a single transaction.

4.2.2 Results

Figure 2 shows the ratio of the execution times of the

multi-user and single-user mode, with single-user performance shown as 100 percent.

At 300 clients, 550055 statements have been executed within 240s in multi-user mode, and the same request sequence was processed in 194s in single-user mode. This amounts to a scheduling overhead of 46s. With an increasing amount of clients, scheduling overhead increases and the number of processed statements goes down. At 500 clients, only 48267 statements have been executed within 240s in multi-user mode. This statement sequence took 15s to process in single-user mode resulting in a scheduling overhead of 225s.

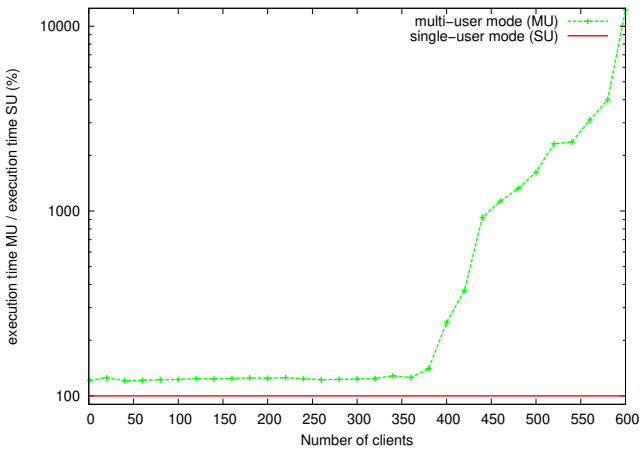

Figure 2: Comparison of execution times of single-user and multi-user mode

4.3 Declarative Scheduling Overhead

We established an upper bound for the performance of a declarative scheduler without having a finished implementation, as follows.

4.3.1 Query

We formulated the strong 2PL protocol in SQL, as illustrated in Listing 1. Under SS2PL, all locks a transaction has acquired are held until the transaction has terminated[23]. The database schema we used consists of a *requests* table containing all pending requests, a *history* table storing relevant prior executed requests and a ready-to-execute (rte) table, each having the attributes listed in Table 2. The results of our SQL query are precisely those pending requests which can safely be executed without violating the logical constraints formulated in SS2PL. Whereas, we assume that each transaction accesses an object only once.

Table 2: Attributes of requests, history and rte table

Attribute	Decription
ID	Consecutive request number
TA	Transaction number
INTRATA	Request number within a transaction
Operation	Operation type (read/write/abort/commit)
Object	Object number

We measured the time (averaged over multiple runs) required for reading the statements from the incoming queue, inserting them into the pending request database, executing the query representing our scheduling protocol, deleting the qualified statements from the pending request database and inserting the qualified requests into the history database. We also counted the number of returned requests.

Listing 1: SQL SS2PL query

```
WITH RLockedObjects AS
(SELECT a.object, a.ta, a.Operation
FROM history a
WHERE NOT EXISTS
(SELECT * FROM history b
WHERE (a.ta=b.ta AND a.object=b.object
AND b.operation='w') OR (a.ta=b.ta
AND (b.operation='a' OR b.operation='c')))),

WLockedObjects AS
(SELECT DISTINCT a.object, a.ta, a.operation
FROM history a LEFT JOIN
(SELECT ta FROM history
WHERE operation='a'
OR operation='c') AS finishedTAs
ON a.ta = finishedTAs.ta
WHERE a.operation='w'
AND finishedTAs.ta IS Null),

OperationsOnWLockedObjects AS
(SELECT r.ta, r.intrata
FROM requests r, WLockedObjects wlo
WHERE r.object=wlo.object AND r.ta<>wlo.ta),

OperationsOnRLockedObjects AS
(SELECT wOpsOnRLObj.ta, wOpsOnRLObj.intrata
FROM requests wOpsOnRLObj, RLockedObjects rl
WHERE wOpsOnRLObj.object=rl.object
AND wOpsOnRLObj.operation='w'
AND wOpsOnRLObj.ta<>rl.ta),

OpsOnSameObjAsPriorSelectOps AS
(SELECT r2.ta, r2.intrata
FROM requests r2, requests r1
WHERE r2.object=r1.object AND r2.ta>r1.ta
AND ((r1.operation='w')
OR (r2.operation='w'))),

QualifiedSS2PLOps AS
((SELECT ta, intrata FROM requests)
EXCEPT (
(SELECT * FROM OperationsOnWLockedObjects)
UNION ALL
(SELECT * FROM OpsOnSameObjAsPriorSelectOps)
UNION ALL
(SELECT * FROM OperationsOnRLockedObjects)))

SELECT r2.*
FROM requests r2, QualifiedSS2PLOps ss2PL
WHERE r2.ta=ss2PL.ta
AND   r2.intrata=ss2PL.intrata
```

4.3.2 Results

Total execution time was 358ms for 300 concurrent clients/active transactions, and 545ms for 500 clients. In this case, the history table was filled with half of the requests of the corresponding workload from Sec. 4.2 above, without requests of committed transactions. Averaging over several different points in the workload, the number of tuples returned by the query is about half of the number of concurrent clients. Hence, we would have to run the scheduler

550055/150=3668 times to schedule the workload for 300 clients, for a total of 3668*358ms=1314s. For the smaller statement count of 500 clients, we only need 48267/250=193 scheduler runs, for a total overhead of 193*545ms=106s.

4.4 Discussion

Although we consider flexibility and developer productivity a primary goal for declarative scheduling, we can see that for classical correctness constraints on read/write operations, the declarative scheduling concept can perform better than regular schedulers, albeit in parameter ranges that are not their primary requirements (e.g. beyond a few concurrent transactions). For 500 concurrent clients, the set-at-a-time approach of using the query processor to schedule requests is faster than a native scheduler. Considering that only a few lines of SQL and a straightforward, non-optimized scheduler implementation were used, we consider this a surprising result. For more complex constraints and workloads/request types, declarative scheduling seems a promising approach.

5. SUMMARY AND FUTURE RESEARCH DIRECTIONS

In the proposed PhD project, we investigate the question to what extent requests can be scheduled declaratively. In declarative scheduling, requests are seen as regular data and thus can be stored in a database. Scheduling constraints are formulated as queries which are executed against the database to identify executable requests and their execution order. The results of a first approach presented here are encouraging and motivate for further investigation.

Our next steps will focus on the search or development of a suitable declarative scheduler language which is more succinct than SQL. Afterwards, we will implement a declarative scheduler that implements this declarative language instead of taking pre-scheduled workloads as done in our naive approach. Also, different workloads with more complex statements have to be analyzed.

Later, we will investigate the *consistency aspect*. We will explore how to describe existing consistency protocols declaratively and how to extend our declarative language to define new application specific consistency protocols. One possibility is an *adaptive consistency scheduler* which varies the applied consistency protocols based on metadata and business application requirements as claimed in [8].

6. REFERENCES

[1] M. Ahmad, A. Aboulnaga, S. Babu, and K. Munagala. Modeling and exploiting query interactions in database systems. In *Proceedings of CIKM*, pages 183–192, 2008.

[2] N. Bhatti and R. Friedrich. Web server support for tiered services. *Network, IEEE*, 13(5):64–71, Sep/Oct 1999.

[3] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on s3. In *Proceedings of SIGMOD*, pages 251–264, 2008.

[4] E. Cecchet, J. Marguerite, and W. Zwaenepole. C-jdbc: flexible database clustering middleware. In *Proceedings of USENIX ATEC*, pages 26–26, 2004.

[5] P. K. Chrysanthis and K. Ramamritham. Acta: a framework for specifying and reasoning about transaction structure and behavior. In *Proceedings of SIGMOD*, pages 194–203, 1990.

[6] DB2 Query Patroller. http://www-01.ibm.com/software/data/db2/querypatroller/.

[7] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *Proceedings of WWW*, pages 276–286, 2004.

[8] S. Finkelstein, R. Brendle, and D. Jacobs. Principles for inconsistency. *Proceedings of CIDR*, 2009.

[9] D. Florescu and D. Kossmann. Rethinking cost and performance of database systems. *SIGMOD Rec.*, 38(1):43–48, 2009.

[10] 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.

[11] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. Size-based scheduling to improve web performance. *ACM Trans. Comput. Syst.*, 21(2):207–233, 2003.

[12] P. Helland and D. Campbell. Building on quicksand. *Proceedings of CIDR*, 2009.

[13] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *Proceedings of SIGMETRICS '04/Performance '04*, pages 37–48, 2004.

[14] O. Kipersztok and J. C. Patterson. Intelligent fuzzy control to augment scheduling capabilities of network queuing systems. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing of IPPS*, pages 239–258, 1995.

[15] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: Pay only when it matters. *Proceedings of VLDB conference*, pages 253–264, 2009.

[16] S. Krompass, H. Kuno, U. Dayal, and A. Kemper. Dynamic workload management for very large data warehouses: juggling feathers and bowling balls. In *Proceedings of VLDB conference*, pages 1105–1115, 2007.

[17] D. Lomet and M. F. Mokbel. Locking key ranges with unbundled transaction services. In *Proceedings of VLDB conference*, 2009.

[18] Oracle Database Resource Manager. http://www.oracle.com/technology/deploy/-availability/htdocs/rm_overview.html.

[19] C. Plattner and G. Alonso. Ganymed: scalable replication for transactional web applications. In *Proceedings of Middleware*, pages 155–174, 2004.

[20] B. Schroeder, M. Harchol-Balter, A. Iyengar, and E. Nahum. Achieving class-based qos for transactional workloads. In *Proceedings of ICDE*, 2006.

[21] B. Schroeder, M. Harchol-Balter, A. Iyengar, E. Nahum, and A. Wierman. How to determine a good multi-programming level for external scheduling. In *Proceedings of ICDE*, 2006.

[22] W. Vogels. Data access patterns in the amazon.com technology. In *Proceedings of VLDB conference*, page 1, 2007.

[23] G. Weikum and G. Vossen. *Transactional Information Systems*. Morgan Kaufmann Publishers, 2002.