A Comparative Evaluation of Transparent Scaling Techniques for Dynamic Content Servers

C. Amza*, A. L. Cox†, W. Zwaenepoel‡

*Department of Electrical and Computer Engineering, University of Toronto, Canada
†Department of Computer Science, Rice University, Houston, USA
‡School of Computer and Communication Sciences, EPFL, Lausanne, Switzerland
amza@eecg.toronto.edu, alc@cs.rice.edu, willy.zwaenepoel@epfl.ch

Abstract

We study several transparent techniques for scaling dynamic content web sites, and we evaluate their relative impact when used in combination. Full transparency implies strong data consistency as perceived by the user, no modifications to existing dynamic content site tiers and no additional programming effort from the user or site administrator upon deployment.

We study strategies for scheduling and load balancing queries on a cluster of replicated database back-ends. We also investigate transparent query caching as a means of enhancing database replication.

Our work shows that, on an experimental platform with up to 8 database replicas, the various techniques work in synergy to improve overall scaling for the e-commerce TPC-W benchmark. We rank the techniques necessary for high performance in order of impact as follows. Key among the strategies are scheduling strategies, such as conflict-aware scheduling, that minimize consistency maintainance overheads. The choice of load balancing strategy is less important. Transparent query result caching increases performance significantly at any given cluster size for a mostly-read workload. Its benefits are limited for write-intensive workloads, where content-aware scheduling is the only scaling option.

1 Introduction

We investigate the relative impact of several techniques for transparently scaling a dynamic content web site on commodity clusters.

Web sites serving dynamic content commonly consist of a front-end web server, an application interpreter, and a back-end database (see figure 1). The site's (dynamic) content is stored in the database. A number of application scripts provide access to that content. The client sends an HTTP request to the web server containing the script's URL and some parameters. The web server invokes the application interpreter to execute the script, which issues SQL queries, one at a time, to the database and formats the results as an HTML page. This page is then returned by the web server to the client as an HTTP response.

Scaling dynamic content sites has received considerable attention in recent years due to the large and growing economic impact of such sites. Most previous research on scaling dynamic content sites through either dynamic content caching [9, 10], or content replication [15, 34, 40] has focused on *specialized*, per-application solutions, which require site administrator or user intervention. In particular, dynamic content caching typically requires specifying page fragments through templates [9] and using ad-hoc rules for invalidating cached results by means of database triggers [10]. Similarly, most existing data replication approaches require the definition of specialized per-application consistency schemes [15, 34, 40] or force the user to handle inconsistent results [20].

To the best of our knowledge, this paper is the first evaluation of a *transparent*, combined query result caching and cluster replication solution for scaling dynamic content web server clusters. Our study draws on recently proposed content-aware scheduling techniques in replicated database clusters [8, 31, 33, 25] and in particular on our own previous work [3, 4] on asynchronous replication with conflict-aware scheduling.

In order to evaluate the relative impact of conflict-aware scheduling versus other scaling optimizations, we study load balancing and scheduling strategies with and without content-awareness and transparent dynamic content caching with automatic invalidations. All techniques provide full application independence; they guarantee strong data consistency to the user and do not require modifications to existing dynamic content site tiers or additional programming

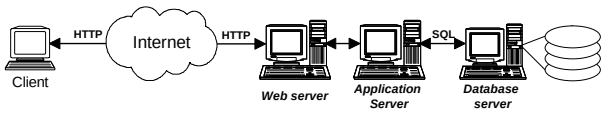

Figure 1. Common Architecture for Dynamic Content Sites

effort for deployment.

In our experimental evaluation we use the TPC-W benchmark [36]. This benchmark is designed to be representative of an e-commerce workload, specifically an online bookstore. It specifies the site's data and the possible interactions with the data. It has three workload mixes with an increasing fraction of writes. The browsing mix (5% writes) reflects a read-heavy workload. The shopping mix with 20% writes is meant to be the most representative workload. The ordering mix (50% writes) represents a write-heavy workload. We have implemented a web site meeting the TPC-W specification, using three popular open source software packages: the Apache web server [1], the PHP web-scripting/application development language [30] and the MySQL database server [26]. Since PHP is implemented as an Apache module, the web server and the application interpreter must coexist on the same machine(s). Our experimental platform consists of a cluster of AMD Athlon 800Mhz processor PCs connected by Fast Ethernet and running FreeBSD. Our largest experimental setup includes 8 database server machines and 8 web server machines.

Our conclusions are:

- With the appropriate combination of load balancing and scheduling techniques, the TPC-W benchmark scales well with increases in cluster size. We get an 8, 7.4 and 5.0 factor of improvement for the browsing, shopping and ordering mixes, respectively, at 8 database engines.
- Conflict-aware scheduling with its two main ingredients, asynchronous replication and conflict avoidance, has the most beneficial impact.
- Load balancing has a secondary impact. Furthermore, optimizing for locality in connection with load balancing has almost no impact.
- At any given cluster size, transparent query result caching significantly improves performance (by up to a factor of 2), but only for workloads with low write frequency.

The remainder of this paper is structured as follows. Section 2 provides the necessary background on content-aware scheduling and load-balancing techniques. Section 3 introduces our overall cluster design. Section 4 describes the scheduling techniques explored in the paper, and Section 5 describes the load balancing techniques. Section 6 describes our combination of query result caching and clustering. Section 7 describes our benchmark and experimental platform. We experimentally investigate how the different scheduling, load balancing and caching techniques affect scaling in Section 8. Section 9 discusses related work. Section 10 concludes the paper.

2 Background: Content-Aware Scheduling and Load Balancing

In tune with the recent trends towards designing self-managing systems [17], a few dynamic content scaling solutions that provide complete application transparency have been proposed.

Content-aware scheduling techniques for dynamic content sites [3, 4, 8, 31, 33] build on recent database research towards providing scaling and strong consistency at the same time [23, 22]. Typically, while a database replication scheme with asynchronous replication is used internally for scaling, the database queries are scheduled such as to hide *any* inconsistencies from the user, thus providing full application transparency. The query scheduling algorithm works as follows. First, a total ordering is used for applying all updates to the database replicas (e.g., as provided by group communication [22]). The updates are applied asynchronously at each replica in this order. More importantly, a second mechanism involves keeping track of the state of the database replicas and scheduling read-only database queries on fully consistent replicas.

Content-aware load balancing has been studied in the context of scaling *static content* web sites through locality-aware request distribution (LARD) [7, 28]. Intuitively, the need for a re-evaluation of this technique in the context of dynamic content sites arises from the basic differences between static and dynamic content. The latter is typically more CPU intensive, executes a small number of pre-defined scripts with widely different complexities, and, above all, contains updates to the content. This shifts the focus from simple load balancing techniques based on data locality [28] to more complex strategies driven by the need for data consistency maintainance.

3 Dynamic Content Web Site Architecture

We discuss the programming model (in section 3.1) and the design of the cluster dynamic content site (in section 3.2) that form the basis for the implementation of the scheduling and load balancing algorithms discussed in sections 4 and 5, respectively.

3.1 Consistency and Programming Model

The consistency model we use for all our protocols is strong consistency or 1-copy-serializability [6], which makes the system look like one copy to the user. With 1-copy-serializability, conflicting operations of different transactions execute in the same order on all replicas (i.e., the execution of all transactions is equivalent to a serial execution, and that particular serial execution is the same on all replicas).

The user inserts transaction delimiters wherever atomicity is required in the application code. In the absence of transaction delimiters, each single query is considered a transaction and is automatically committed (so called "auto-commit" mode).

Our method requires that all tables accessed in a transaction and their access types (read or write) be known at the beginning of each transaction. During a pre-processing phase, we parse the application scripts to obtain a conservative approximation of this information. The pre-processor inserts a "lock tables" database query at the beginning of each script for all tables accessed and their access type. Although these queries are not actually executed by the databases, they form the basis of a conservative two-phase locking protocol [6] based on conflict classes [29] that avoids deadlocks. We choose a protocol that avoids deadlocks, because the deadlock probability for a replicated database cluster becomes prohibitive in large clusters, due to the extra updates that each node performs on behalf of the other nodes [16].

3.2 Cluster Components

We consider a cluster architecture for a dynamic content site, in which a set of collaborating schedulers distribute incoming requests to a cluster of database replicas and deliver the responses back to the application server (see Figure 2). The web server interacts directly only with the schedulers. For each client interaction, the web server only interacts with a single scheduler. These interactions are synchronous: for each query, the web server blocks until it receives a response from the scheduler.

The schedulers use a set of database proxies, one at each database engine, to communicate with the databases, and a sequencer to assign a unique sequence number to each transaction. The sequence numbers are subsequently used to enforce a total order of conflicting operations at each database replica.

3.2.1 Operation

The web server sends the queries embedded in a script, one at a time, to one of the schedulers. Subsequently, the scheduler sends read-only (SELECT) queries to a single database machine, while the execution of write queries (INSERT, UPDATE, DELETE) is replicated on all machines. The transaction delimiters and lock operations are sent to all databases.

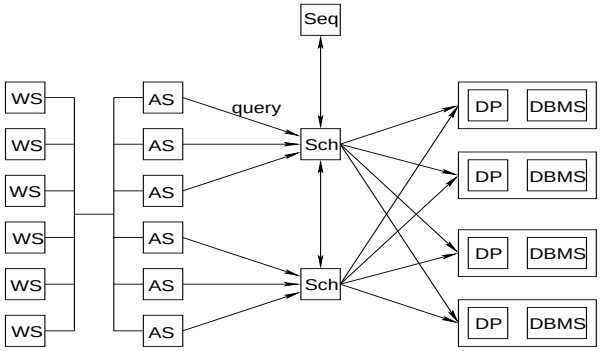

Figure 2. Cluster Architecture Design

The scheduler sends each query to the database proxy, tagged with the sequence number obtained at the beginning of its enclosing transaction. Read queries in auto-commit mode are not assigned a sequence number. The database proxy performs any necessary query queuing for queries arriving not in sequence number order and for consistent conflict ordering (see section 3.2.2), and passes the query to the database. After the database executes the query, it returns the results to its database proxy, which forwards them to the scheduler. The scheduler updates its *state* and forwards the results to the web/application server. At any given time, the *scheduler state* contains: load information, the current availability and the current status of each database backend in terms of the operations that have been applied. From this information, the scheduler can infer which databases are up-to-date, allowing for intelligent scheduling decisions for queries.

Both the scheduler and database proxy layers are transparent. To the web/application servers, a scheduler looks like a database engine. At the other end, each database engine interacts with its database proxy as if it were a regular web/application server. As a result, we can use any off-the-shelf web server (e.g., Apache), application server (e.g., PHP) and database (e.g., MySQL) without modification. Moreover, the system is easy to configure and reconfigures itself automatically in case of failures. Schedulers and database proxies read a configuration file at startup, and set up connections accordingly.

3.2.2 Implementation

The key to good scaling for the entire system lies in the scheduler design. The scheduler code is lightweight such that it can support a large number of web server front-ends

and database back-ends. The schedulers do not execute any complex operations, they do not hold any locks, or maintain any queues. Conflict resolution and conflict queue handling for consistent replicated execution is done at the database proxies. The schedulers and the database proxies are implemented as an event-driven loop, which multiplexes requests and responses over a pool of open connections to and from the web server and database tiers.

In content-aware schemes, each scheduler parses and tags each dispatched query with information such as the query type, estimated query load, tables accessed, and whether or not the query belongs to a previously initiated transaction. Furthermore, the scheduler maintains as part of its internal state, a backlog for the state of all replicated operations (i.e., transaction delimiters, write and lock operations) in the active transactions that it currently handles. The backlog is updated whenever a write-type query is received, when it is sent to the database engines or when a reply is received from one of the database engines.

The scheduler also keeps the current load of each database (see section 5). This data structure is updated with feedback from the database proxy upon each reply. The feedback loop is necessary because each scheduler sees only its own request traffic, and not the traffic going through other schedulers. Hence, the load balancing information it keeps is only an estimate of the real database load.

Finally, for fault tolerance purposes, the schedulers maintain persistent state for all write queries of past transactions that have not been committed or aborted at all databases. More details on fault tolerance are provided in our previous paper [3].

Each database proxy delivers conflicting queries to the database engine on its machine, in the order of their pre-assigned sequence number. The database proxy keeps queues of conflicting operations and also attempts to send queries out-of-order across conflict classes to maximize concurrency. Queries from the same script are issued in-order. Queries belonging to an ongoing transaction are prioritized. This means that, once a query has finished at the database, we first check whether a query from one of the transactions that have started to execute is ready to be sent. The database proxy may also limit congestion on its database machine by holding back queries once a load threshold has been reached on its database (see section 5.3).

4 Scheduling Algorithms

4.1 Synchronous Replication

This is a basic replication scheme with in-order execution of all queries and synchronous execution of writes on all replicas. No query parsing for obtaining each query's tables is necessary. The scheduler waits for completion of every write-type query on all database back-ends, before returning the answer to the web server. The schedulers and database proxies only pass through queries, and keep track of the completion of operations.

4.2 Asynchronous Replication with Consistency

This is a content-aware technique where the scheduler parses each query to obtain its tables. Lock requests are sent to all replicas where the database proxy executes them locally on its own per-table data structures in order of their pre-assigned sequence numbers. The locks are used by the database proxy to keep track of conflicts and enforce a total order of conflicting operations, and are not passed to the database. This allows any writes within a transaction to execute *asynchronously* at each replica, without the need for a 2-phase commit protocol [6, 38] between replicas.

The scheduler returns the response to the web server as soon as a lock, write or commit operation executes at *any* replica. This means that, at any given time, the same script can generate several outstanding queries. Read queries in auto-commit mode proceed without locking after previous update transactions from their assigned proxy's queues on the same tables have committed. These single read queries are common in browse-oriented dynamic content workloads. They are typically much more complex than update transactions, hence will induce asynchronous system behavior because they produce different loads and different conflict types on different replicas.

4.3 Conflict-Aware Scheduling of Read Queries

Conflict-aware scheduling is an enhancement of asynchronous replication. For each read query, the scheduler first determines the set of up-to-date replicas without conflicts, that have completed the previous writes in the same transaction. Conflicts are determined on a per-table basis. The scheduler then selects the least loaded replica from this set as the replica to receive the read query.

This optimization requires that the scheduler, in addition to parsing queries, maintains in its backlogs the completion status of locks and outstanding writes in a transaction, for all database replicas.

5 Load Balancing Strategies

5.1 Generic Load Balancing Schemes

The simplest load balancing policy assigns the requests in **Round Robin (RR)** order to back-ends. Slightly more complex, **Shortest Queue First (SQF)** is based on weighted round-robin, a common load balancing scheme in

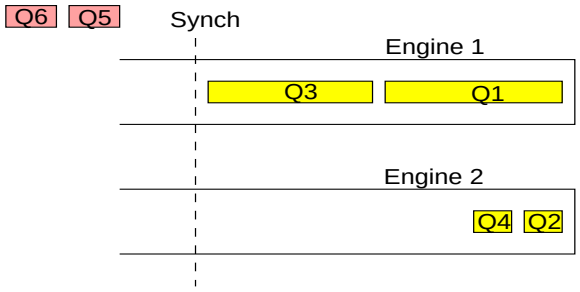

Figure 3. Load balancing using number of queries vs. execution time of queries

static-content cluster servers [18, 11]. **SQF** uses the number of outstanding queries to a particular back-end as an estimate of the load on that back-end. The incoming requests are distributed in round-robin fashion, weighted by the estimate of the load on the different back-ends.

5.2 Content-Aware Load Balancing Schemes

With **Shortest Execution Length First (SELF)**, we measure off-line the execution time of each query on an unloaded (idle) machine. At run-time, the scheduler estimates the load on a particular back-end as the sum of the (a priori measured) execution times of all queries outstanding at that back-end. As opposed to SQF, which treats each query as equal, SELF tries to take into account the widely varying execution times for different query types.

The wide range of query execution times makes SELF a better load balancing strategy for e-commerce workloads. We illustrate this using the example in figure 3. Assume that the SQF scheduler has placed queries Q1, Q2, Q3, Q4 on the two database machines. Furthermore, assume all queries are read-only and access a common table t1. A transaction containing updates to table t1 (Q5 and Q6) follows. The two machines have optimal load balance with respect to SQF (i.e., the same queue length). However, the total database engine load is clearly not balanced in this situation, due to the large variability in query complexities. Even worse, if this technique is used in conjunction with a synchronous scheduler, all subsequent operations (i.e., Q5 and Q6) have to wait for the machine with the longest query times to finish.

Locality-Aware Request Distribution (LARD) was developed and shown to be successful for load balancing static content requests in a cluster [28]. The goal of LARD is to combine good load balance and high locality for increased hit rates in the data caches of each back-end. In our implementation of LARD, the scheduler keeps, for each machine, a history of queries that have executed previously at that machine and the tables that those queries accessed. When a new query arrives, accessing a certain set of tables, the scheduler computes the set of back-ends that have recently accessed the maximum number of those tables. It selects the least loaded machine from that set, unless its load is over a certain threshold. If the selected machine is overloaded, the scheduler sends the query to the least loaded machine.

5.3 Admission Control

Admission control is a potential addition to any load balancing algorithm, in which there is a limit set on the load of outstanding queries at a particular back-end. This limit is specified in terms of the number of queries for SQF and in terms of execution time for SELF (and can be either for LARD). Limiting the load (i.e., admission control) has the effect of smoothing out load peaks due to either bursts of request traffic or load balancing imperfections which could otherwise cause overload conditions on the database server. If the load for its back-end is over the limit, the database proxy holds on to the queries until the load on its back-end drops below the limit.

6 Replicated Clustering and Caching Combination

Each scheduler caches the query results for all read queries that pass through it. If an incoming read query matches an entry in its cache, the scheduler returns the cached query result to the web/application server front-end. If there is no match, a new cache entry is allocated, and the query is sent to the database. The database returns the results of the query to the cache, where they are inserted and then forwarded to the application server front-end. On an update, insert or delete query, the cache performs the necessary invalidations and forwards the write query to the database.

The cache is fully transactional by virtue of being integrated with the scheduler algorithm. Specifically, the cache forwards “lock tables” annotations at the start of each transaction to the databases just as in the usual scheduler protocol. This ensures serialization of conflicts even if now read operations may be serviced by scheduler caches.

The scheduler cache supports two transparent invalidation schemes (table and column based). In these schemes, for each cached query response, the query’s dependencies are recorded in terms of database tables or in terms of database columns. In the case of table-based invalidation, each table object contains references to the cache entries that are dependent on this table. In the case of column-based invalidation, each table object contains a number of column objects, each one corresponding to a column in the corresponding table. Each of these column objects contains

references to the cache entries depending on this column. When an update, insert or delete query is received, the cache invalidates all cache entries dependent on either the affected tables or the affected columns and forwards the query to the database. To keep the size of the cache manageable, the cache implements an LRU replacement strategy.

Similarly to maintaining consistency between each cache and its underlying data, consistency between the different scheduler caches is maintained through an invalidation protocol. When an update, insert or delete query arrives at a particular scheduler, it is first forwarded to the back-end. The query is then parsed to determine its dependencies. An invalidation message containing these dependencies is sent to all the other schedulers. The scheduler to which the query was initially sent then waits for the invalidations to be acknowledged by all the other caches and for the response from the database to come back. At that point, it sends the response back to the client.

7 TPC-W Benchmark

The TPC-W benchmark from the Transaction Processing Council [36] is a transactional web benchmark designed for evaluating e-commerce systems. Several interactions are used to simulate the activity of a retail store. The database size is determined by the number of items in the inventory and the size of the customer population. We use 100K items and 2.8 million customers which results in a database of about 4 GB.

The inventory images, totaling 1.8 GB, are resident on the web server. We implemented the 14 different interactions specified in the TPC-W benchmark specification. Of the 14 scripts, 6 are read-only, while 8 cause the database to be updated. Read-write interactions include user registration, updates of the shopping cart, two order-placement interactions, two involving order inquiry and display, and two involving administrative tasks. We use the same distribution of script execution as specified in TPC-W. The complexity of the interactions varies widely, with interactions taking between 20 ms and 700 ms on an unloaded machine. Read-only interactions consist mostly of complex read queries in auto-commit mode, up to 30 times more heavyweight than read-write interactions containing transactions. The weight of a particular query (and interaction) is largely independent of its arguments.

TPC-W uses three different workload mixes, differing in the ratio of read-only to read-write scripts. The browsing mix contains 95% read-only scripts, the shopping mix 80%, and the ordering mix 50%.

All read-only interactions exhibit locality in their access patterns, which ranges from hot-spot rows satisfying a condition (such as top-k published items and top-k recent orders) to larger sets of frequently accessed rows in the item table for bestseller, new product and promotional items and for return customers in the customer table.

The item table is the most frequently read. It is also updated in every order-placement transaction. Other tables that both appear in compute-intensive queries and are updated frequently are orders and order_line.

7.1 Client Emulation Implementation

We implemented a client-browser emulator. A session is a sequence of interactions for the same customer. For each customer session, the client emulator opens a persistent HTTP connection to the web server and closes it at the end of the session. Each emulated client waits for a certain think time before initiating the next interaction. The next interaction is determined by a given state transition matrix that specifies the probability to go from one interaction to another. The session time and think time are generated from a random distribution with a specified mean.

7.2 Hardware Platform

We use the same hardware for all machines running the client emulator, the web servers, the schedulers and the database engines. Each one of them has an AMD Athlon 800Mhz processor running FreeBSD 4.0, 256MB SDRAM, and a 30G ATA-66 disk drive. They are all connected through 100MBps Ethernet LAN.

7.3 Software

We use Apache v.1.3.22 [1] for our web-server, configured with the PHP v.4.0.1 module [30] providing server-side scripting for generating dynamic content. We use MySQL v.4.0.1 [26] with InnoDB transactional extensions as our database server.

8 Experimental Results

8.1 Baseline Experiments

We run the TPC-W benchmark with one web server machine and one database engine machine. We obtain 5.1, 8.5, and 20.4 interactions per second for the browsing, shopping and ordering workload mix, respectively. The database is already a bottleneck in this configuration, as we can see from the CPU utilization on the web server and the database server machines in Figure 4. For the browsing and shopping mixes, the CPU usage on the database is almost 100%, while for the ordering mix it does not reach 100% due to lock waiting times. The database CPU is less of a bottleneck in the ordering mix, due to a much smaller fraction

of complex single read queries, compared to the browsing and shopping mixes. A scheduler is not necessary in this configuration. There is no measurable difference in terms of throughput, however, when we interpose a scheduler between the web server and the database machine.

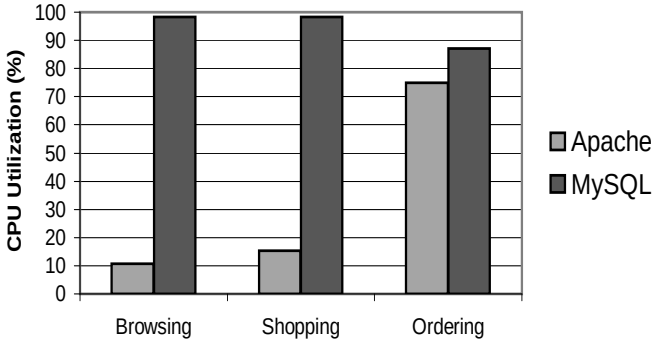

Figure 4. CPU Utilization for the web server and the database server in a single-replica configuration

All further experimental numbers are obtained running an implementation of our dynamic content server on a cluster of 1 to 8 database server machines. We use a number of web server machines sufficient for the web server stage not to be the bottleneck. The largest number of web server machines used for any experiment is 8. We use two schedulers to ensure data availability. The invalidation mechanism we use for caching is the finer-grain column-based invalidation.

In all further experiments, the performance metric used is the standard one in TPC-W, throughput in terms of interactions per second. We present performance measurements in Section 8.2, and Section 8.3.

8.2 Overall Scaling Results

In this section we discuss the overall results for the best combination of load balancing and scheduling. We discuss the relative merits of various load balancing and scheduling strategies in Section 8.3. Figure 5 shows in the x-axis the number of database machines and in the y-axis the number of interactions per second. The three graphs correspond to the three different workload mixes. For each point in the graphs, we drive the server with increasing number of clients, until performance peaks, and we report the peak-point throughput.

The top two curves in each graph show the overall scaling results for the best combination of load balancing and scheduling strategies (including conflict-awareness and asynchrony, labeled "ConflA" in the figure), with and without caching. Comparing the two, we see that roughly the same factor of improvement was obtained through caching, independent of the cluster size. The factor of improvement is 2 for the browsing mix, 1.2 for the shopping mix, and 1.1 for the ordering mix. The decreasing cache effectiveness is due to a decrease in the fraction of reads and an increase in the number invalidations as a result of writes. In the largest configuration we get a 40%, 31% and 17% hit rate for the reads in the browsing, shopping and ordering mixes, respectively. During a one-hour run, the caches for each scheduler did not exceed the maximum set size of 50 MB, and thus there were no cache replacements. This is explained in part by the high locality in the application and by the relatively coarse-grained invalidation scheme (column-based), which invalidates many cache entries when hot tables are written.

Looking at the effectiveness of clustering alone, we see that with increasing cluster size, the browsing and shopping mixes scale very well. We get almost linear improvement with each added database machine up to 8 machines, where we get a factor of 8 improvement for the browsing mix and a factor of 7.4 for the shopping mix. The lower overall scaling (a factor of 5.0 at 8 machines) in the ordering mix is explained by a lower degree of parallelism. This mix contains a significant fraction of order-placement transactions accessing several hot tables (such as item, orders and order_line). While out-of-order query issue at the database proxy and conflict-aware scheduling can overlap query waits if a variety of conflict classes exist, these techniques have limitations when a single conflict type is dominant, as in this mix.

Conflict-aware scheduling with asynchronous replication is the method of choice. Figure 5 also contains the best result for any strategy that does not include these two ingredients (Best-Sync). Clearly, the results are inferior for all mixes. Furthermore, the bottom curves in Figure 5 present the scaling results for a simple round robin (RR) distribution strategy with synchronous scheduling of queries, no load limiting, and no attention paid to conflicts. The results are poor for all workloads.

8.3 Detailed Comparison

In this section we compare the impact of the different strategies on performance. All numbers represent peak-performance and were derived through measurements using the experimental platform with 8 databases.

8.3.1 Relative Impact of All Scheduling Strategies

The graphs in Figure 6 for all the three mixes show the contribution of each strategy. Starting from the simplest strategy of round-robin with synchronous scheduling (Base), we add all the other enhancements, one by one. First, we add the best load balancing strategy (SELF), then we add the

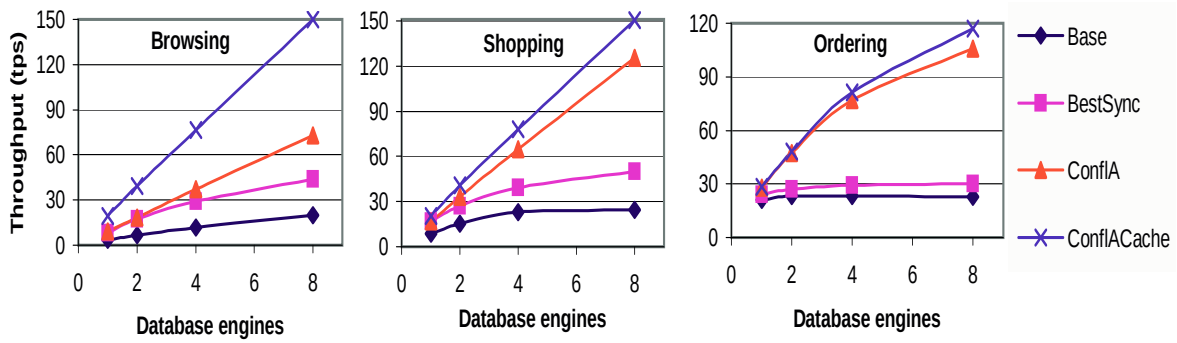

Figure 5. Throughput scaling for the browsing, shopping and ordering mixes

limit on database load (Limit), then asynchronous replication (Async) and scheduling reads with conflict-awareness (ConflA). We found that using the load measure in terms of execution length that comes with SELF allows for relative independence on the exact value of the load limit. This is in contrast to choosing a limit in terms of number of outstanding queries in SQF, as shown in Figure 7. We use a 1 second load limit with SELF for all mixes.

First, the most important factors are related to consistency maintainance (asynchronous replication and conflict awareness) and the least important is the choice of load balancing. Second, in the browsing mix, admission control is a relatively important secondary factor. In this workload, writes are rare, while heavy-weight reads can cause congestion at the database. For all mixes, given that traffic is non-uniform due to client think times, limiting database congestion during bursts of traffic becomes more important than the particular choice of load balancing. On the other hand, in the shopping and ordering mixes, where transactions are more frequent than in the browsing mix, any other improvement is dwarfed by the combined impact on performance of asynchronous scheduling and conflict-awareness (ConflA).

8.3.2 Comparison of Load Balancing Methods

Figure 8 shows the relative performance of the four load balancing policies when used in conjunction with the best scheduling policy (conflict-awareness with asynchronous replication). We use no limit for the database load. Setting a limit could smooth out imperfections in the load balancing due to the queuing capacity of the database proxies.

First, the content-aware schemes (SELF and LARD), perform better than the generic ones. The maximum difference, however, between the performance of any two policies is around 20%. Second, locality (LARD) does not bring any benefits compared to SELF. This is mainly due to the compute-intensive nature of the read queries. Furthermore, all complex read queries such as the ones necessary for computing best sellers and new products access

Strategy	CPU (%)	Memory (MB)	Network (Mb/sec)	Disk (MB/sec)
No-cache	4%	0.8	3.8	< 1
Cache	13%	33.1	6.0	< 1

Table 1. Average resource usage at each of the two schedulers for the TPC-W shopping mix, at the largest configuration with and without caching

hot rows in only a few tables (e.g., item, author, order_line), which become replicated in most memories independent of policy. These complex queries have a large performance impact, and each replica's memory can easily accommodate the data necessary for computing all of their results. Locality-aware request distribution can make a difference only for sets of requests with combined working sets that do not fit into the memory of any single node. This is a common scenario [28, 7] in web sites serving *static content* where this technique was shown to work well.

8.3.3 Costs of Content-Awareness at the Scheduler

In table 1 we show the memory, disk, and network usage at the scheduler in the largest configuration with 8 databases, with and without caching (disk accesses result from fault tolerance actions in both configurations). All resource usage is very low. Throughout our experiments, the bottleneck is either the front-end tier or the database tier (with oscillations between the two especially due to cache invalidations), while the scheduler never becomes the bottleneck. From additional profiling experiments, we have determined that for a scheduler without caching, query parsing accounts for around 2% of the total CPU processing time, while the rest is spent in handling connections in the event-driven loop.

The CPU usage for each database proxy is negligible for all mixes in all experiments.

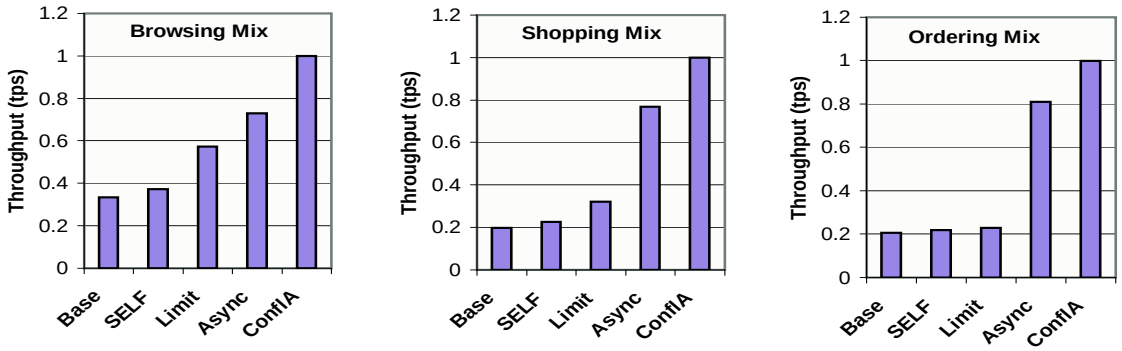

Figure 6. Contributing factors in content-aware strategies (all protocols normalized to ConflA)

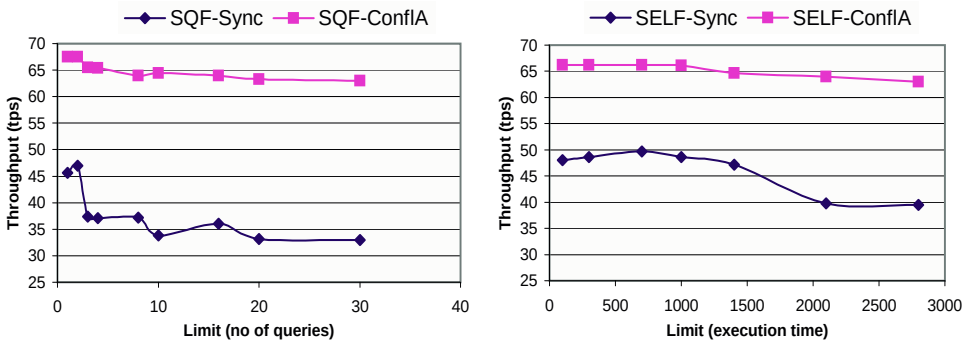

Figure 7. Throughput for conflict-aware asynchronous versus synchronous replication as a function of threshold value for SFQ and SELF for shopping mix on 4 databases

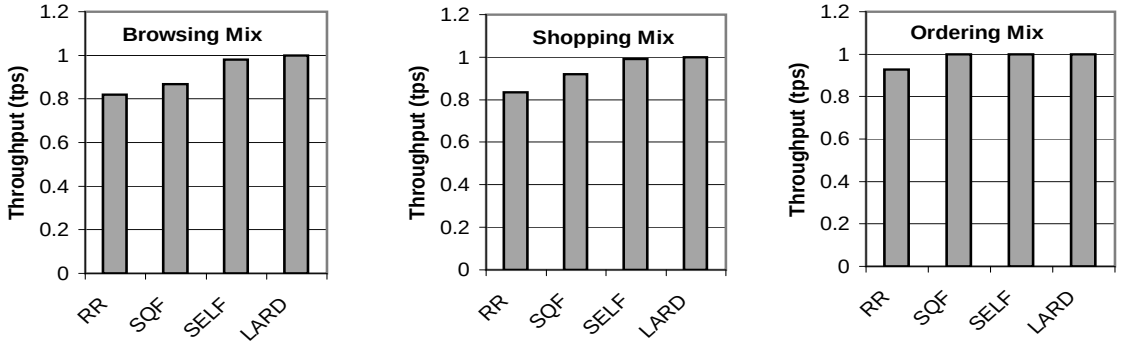

Figure 8. Comparison of load balancing policies (normalized to LARD)

8.4 Discussion

Overall our results indicate that: i) Scheduling queries with content awareness is a promising technique for scaling dynamic content sites. The computational costs involved are very low and employing such techniques can dramatically reduce consistency-induced overheads. ii) Traditional load balancing strategies even when enhanced with content-awareness become less important than congestion control measures due to the non-uniform traffic on such sites caused by client think times. iii) Simple query-caching techniques can help scale the site for typical browse-oriented client sessions.

Finally, more research is needed to investigate the database features, the interfaces, and the protocols necessary for a closer integration between the scheduler and the internal database consistency maintenance policies, in replicated database clusters. We believe that further improvements are possible if the database is no longer treated as a black box without sacrificing transparency. For instance, the scheduler could benefit from further augmenting its awareness with knowledge about the presence or absence of advanced concurrency control techniques used by the databases (e.g., multiversion concurrency control [6], database statistics on fine-grain lock conflicts, database logging of row-level write-sets in a standard format, etc).

9 Related Work

Current high-volume web servers such as the official web server used for the Olympic games [10, 9] and actual e-commerce site configurations reported by industry [19, 36], rely on expensive supercomputers to satisfy the volume of requests. Nevertheless, performance of such servers may become a problem during periods of peak load. Our solution provides scalability and availability by using commodity hardware and software with no modifications.

This paper is related to all previous work studying transparent performance optimizations in static or dynamic content servers, such as web caching [32, 2], scaling through replication [5, 35, 21, 22, 38, 39, 31], load balancing [28, 7, 25] or admission control [37, 12]. In contrast, we present our experience with scaling a dynamic content server system, as a whole. We strive to provide insights into what techniques are important in practice when engineering such sites and pinpoint areas that are worth further investigation.

Our dynamic content caching approach differs from previous transparent approaches to dynamic content caching [32, 2] because we design and evaluate a cluster dynamic content cache, with automatic invalidations and strong consistency guarantees.

In practice, database replication has previously been used mainly for fault tolerance and data availability [14]. Recent efforts in the database research community focus on providing *both* scaling and serializability in replicated databases [5, 35, 21, 22, 38]. These approaches differ from ours in that databases are considered independent (e.g., distributed on a wide area network), where clients have little choice other than executing transactions locally, and the replication is implemented by modifying the database layer.

Luo et al. [24] and Oracle's 9i Database Cache product [27] use a middle-tier database cache. They rely on replication tools to periodically propagate updates from the back-end database to the cache tier. More recent efforts towards integration of database fine-grained concurrency control and replication techniques use snapshot isolation [13, 39, 31] to minimize consistency maintainance overheads.

These techniques are not completely transparent, since they either assume the presence of some particular features (e.g., multiversioning and write-set extraction) at the database or allow stale data [24]. In contrast, by treating databases as black-boxes, our techniques are applicable even to heterogeneous environments clustering databases of different types.

Our LARD scheme is similar to the locality-aware request distribution proposed by Pai et al. [28] for static content. They show that for a web engine serving static content, LARD outperforms both pure locality-based and weighted round-robin schemes. In contrast, we show that, when the web server is targeted at serving dynamic content, consistency maintenance techniques have more impact than distributing requests for locality. Zhang et al. [41] have previously extended LARD to dynamic content in their HACC project. Their study, however, is limited to read-only content workloads. In a more general dynamic content server, replication implies the need for consistency maintenance.

Neptune [34] adopts a primary-copy approach to providing consistency in a partitioned service cluster. However, their scalability study is limited to web applications with loose consistency such as bulletin boards and auction sites, where scaling is easier to achieve. They do not address e-commerce workloads or other web applications with relatively strong consistency requirements.

10 Conclusions

In this paper, we investigate the impact of several content replication and caching techniques on scaling in a dynamic content site using a cluster of web server and database engine machines. We avoid modifications to the web server, the application scripts and the database engine. We also assume software platforms in common use: the Apache web server, the MySQL database engine, and the PHP scripting language. As a result, our techniques are applicable without burdensome development or reconfiguration of the web

site. We use the various workload mixes of the TPC-W benchmark to evaluate the contribution of load balancing, scheduling and caching to good scaling behavior.

Our cluster architecture scales well for all the TPC-W workload mixes. The key ingredient of a scalable policy is content-aware asynchronous replication. The scheduler combines information about the individual queries and the state of the database replicas to, at the same time, improve performance and provide strong consistency. The actual choice of load balancing strategy is less important. Somewhat better results are obtained if query execution time is taken into account for load balancing. Locality-based load balancing policies, found very profitable for static web workloads, offer little advantage.

Finally, we have shown that clustering and caching, two traditional scaling methods can be combined successfully in a fully transparent integrated solution. Dynamic content caching improves performance by the same factor independent of cluster size. The impact is significant (a factor of 2) only for workloads with low write frequency, while for write-intensive workloads, conflict-aware scheduling with asynchronous replication is the main scaling option.

Acknowledgments

We would like to thank the anonymous reviewers for their comments. The work also benefited from informal discussions with Emmanuel Cecchet and Anupam Chanda and from the contributions of the DynaServer team towards a better understanding of dynamic content benchmarks. Finally, we thankfully acknowledge the support of Natural Sciences and Engineering Research Council of Canada and IBM.

References

[1] The Apache Software Foundation. http://www.apache.org/.

[2] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A dynamic data cache for Web applications. In *Proceedings of the 19th International Conference on Data Engineering*, March 2003.

[3] C. Amza, A. Cox, and W. Zwaenepoel. Conflict-aware scheduling for dynamic content applications. In *Proceedings of the Fifth USENIX Symposium on Internet Technologies and Systems*, pages 71–84, March 2003.

[4] C. Amza, A. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *4th ACM/IFIP/Usenix International Middleware Conference*, June 2003.

[5] T. Anderson, Y. Breitbart, H. F. Korth, and A. Wool. Replication, consistency, and practicality: are these mutually exclusive? *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, 27(2):484–495, June 1998.

[6] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[7] E. V. Carrera, S. Rao, L. Iftode, and R. Bianchini. User-level communication in cluster-based servers. In *Proceedings of the 8th IEEE International Symposium on High-Performance Computer Architecture (HPCA 8)*, February 2002.

[8] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. C-jdbc: Flexible database clustering middleware. In *Proceedings of the USENIX 2004 Annual Technical Conference*, Jun 2004.

[9] J. Challenger, A. Iyengar, K. Witting, C. Ferstat, and P. Reed. A publishing system for efficiently creating dynamic web data. In *IEEE INFOCOM*, March 2000.

[10] Jim Challenger, Paul Dantzig, and Arun Iyengar. A scalable and highly available system for serving dynamic data at frequently accessed web sites. In *Proceedings of Supercomputing'98*, November 1998.

[11] Cisco Systems Inc. LocalDirector. http://www.cisco.com.

[12] Sameh Elnikety, Erich Nahum, John Tracey, and Willy Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *Proceedings of the Thirteenth International World Wide Web Conference*, May 2004.

[13] Sameh Elnikety, Fernando Pedone, and Willy Zwaenepoel. Generalized snapshot isolation and a prefix-consistent implementation. Technical Report IC/2004/21, EPFL, 2004.

[14] Ron Flannery. *The Informix Handbook*. Prentice Hall PTR, 2000.

[15] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar. Application specific data replication for edge services. In *Proceedings of the 12th International World Wide Web Conference*, May 2003.

[16] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, 1996.

[17] IBM. Autonomic Computing Manifesto. http://www.research.ibm.com/autonomic/manifesto, 2003.

[18] IBM Corporation. IBM interactive network dispatcher. http://www.ics.raleigh.ibm.com.

[19] A. Jhingran. Anatomy of a real e-commerce system. In *Proceedings of the ACM SIGMOD*, May 2000.

[20] Minwen Ji, Edward W. Felten, Jaswinder Pal Singh, and Mao Chen. Query affinity in internet applications. Technical report, Computer Science, Princeton University, 2001.

[21] P. Keleher. Decentralized replicated-object protocols. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 143–151, May 1999.

[22] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. In *ACM Transactions on Data Base Systems*, volume 25, September 2000.

[23] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *Proceedings of the 26th International Conference on Very Large Databases*, pages 134–143, September 2000.

[24] Q. Luo, S. Krishnamurty, C. Mohan, H. Pirahesh, HW̃oo, B. Lindsay, and J. Naughton. Middle-tier database caching for e-business. In *Proceedings of the 2002 ACM International Conference on Management of Data*, pages 600–611, June 2002.

[25] J. M. Milan-Franco, Ricardo Jimenez-Peris, Marta Patio-Martnez, and Bettina Kemme. Adaptive middleware for data replication. In *Proceedings of the 5th ACM/IFIP/USENIX International Middleware Conference*, October 2004.

[26] MySQL. http://www.mysql.com.

[27] Oracle. Oracle9*i* Application Server Web Caching, October 2000.

[28] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, October 1998.

[29] M. Patiqo-Martinez, R. Jiminez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *International Symposium on Distributed Computing*, pages 315–329, October 2000.

[30] PHP Hypertext Preprocessor. http://www.php.net.

[31] Christian Plattner and Gustavo Alonso. Ganymed: Scalable Replication for Transactional Web Applications. In *Proceedings of the 5th ACM/IFIP/Usenix International Middleware Conference*, October 2004.

[32] Karthick Rajamani and Alan Cox. A simple and effective caching scheme for dynamic content. Technical Report 00-371, Computer Science, Rice University, September 2000.

[33] U. Rhom, K. Bhom, H.-J. Schek, and H. Schuldt. Fas - a freshness-sensitive coordination middleware for a cluster of olap components. In *Proceedings of the 28th International Conference on Very Large Databases*, pages 134–143, August 2002.

[34] Kai Shen, Tao Yang, Lingkun Chu, JoAnne L. Holliday, Doug Kuschner, and Huican Zhu. Neptune: Scalable replica management and programming support for cluster-based network services. In *Proceedings of the Third USENIX Symposium on Internet Technologies and Systems*, pages 207–216, March 2001.

[35] I. Stanoi, D. Agrawal, and A. El Abbadi. Using broadcast primitives in replicated databases. In *Proceedings of the* 18^{th} *IEEE International Conference on Distributed Computing Systems ICDCS'98*, pages 148–155, May 1998.

[36] Transaction Processing Council. http://www.tpc.org/.

[37] Matt Welsh and David Culler. Adaptive overload control for busy internet servers. In *Proceedings of the Fifth USENIX Symposium on Internet Technologies and Systems*, March 2003.

[38] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database replication techniques: a three parameter classification. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, pages 206–215, October 2000.

[39] Shuqing Wu and Bettina Kemme. Postgres-r(si): Combining replica control with concurrency control based on snapshot isolation. In *Proceedings of the 21st International Conference on Data Engineering*, Apr 2005.

[40] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 305–318, October 2000.

[41] Xiaolan Zhang, Michael Barrientos, J. Bradley Chen, and Margo Seltzer. HACC: An architecture for cluster-based web servers. In *Proceedings of the 2000 Annual Usenix Technical Conference*, pages 155–164, June 2000.