Scaling and Availability for Dynamic Content Web Sites

Abstract

We investigate the techniques necessary for building highly-available, low-cost, scalable servers, suitable for supporting dynamic content web sites. We focus on replication techniques for scaling and availability of a dynamic content site using a cluster of commodity computers running Web servers and database engines. Our techniques allow scaling without undue development, maintenance, and installation costs, avoiding modifications to both the Web server and the database engine.

Our results on an eight node database cluster show good scaling for the e-commerce TPC-W benchmark provided that suitable load balancing and replication strategies are in place. Key among these strategies is replication with relaxed consistency, in which the server allows controlled internal data inconsistencies to improve performance while hiding these inconsistencies from the user. The actual choice of load balancing strategy is less important. Locality-based load balancing policies based on data caching, found very profitable in static content servers have almost no impact.

1 Introduction

As commercial use of the Internet continues to grow, the issues of scaling and availability of web sites become increasingly important. Most recent work on scaling web sites has focused on the delivery of static content [21, 26, 5]. For example, replication of static content on multiple servers and locality-aware load distribution [21] with simple metrics to estimate server load have been shown to produce good scaling behavior. In this paper, we show that web sites delivering dynamic content can also be made to scale by means of content replication, but that different techniques are necessary. Intuitively, the need for different techniques arises from the basic differences between static and dynamic content. The latter is typically more CPU intensive, and, above all, contains updates to the content. Thus, in contrast to static content, a fundamental challenge introduced by dynamic content is the need for a specific level of data consistency. Previous techniques for web site scaling such as Web-proxy caching, are based on the implicit assumption that loose data consistency is always acceptable. Thus, stale or inconsistent data is routinely served to users. On the other hand, in dynamic-content applications, loose consistency is acceptable for only some interactions. Strict consistency and up-to-date information is required for others. For example, a user would be upset if a display operation for his last order or account balance would not give accurate information.

Web sites serving dynamic content commonly consist of a front-end Web server, an application interpreter, and a back-end database. 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 application script's URL and some parameters. The web server invokes the application interpreter to execute the script, which issues a number of SQL queries 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.
In small sites the web server front-end, the application interpreter, and the database run on the same machine, which may become a bottleneck. In this paper we study how to scale up such sites by using clusters. The first step to scaling is to run the Web server, the application interpreter, and the database on separate machines. Beyond that, multiple machines may be used for each. We impose the constraint that the web server, the application interpreter, the database server, and the application scripts used for accessing the dynamic content must remain the same as for the single-machine case. We recognize that additional performance enhancements may be available if this constraint is lifted, but we argue that it is essential for transparent scaling of dynamic content sites without undue development or administration.

We show that a web site serving dynamic content can use a series of optimizations based on relaxed consistency internally, thereby improving scalability and data availability, while at the same time hiding any inconsistencies from the user. In our implementation, we exploit the usual level of indirection in accessing the database offered by any practical web site implementation. Specifically, we introduce a dispatcher between the application layer and the database server. This dispatcher is responsible for relaxing the replication consistency while still conforming to the user consistency requirements. Through replication of the dispatcher and the maintenance of reliable state at the dispatcher, we avoid the dispatcher itself becoming either a bottleneck or a single-point of failure.

We focus on read-one, write-all replication schemes. Read-only queries are distributed according to a load balancing scheme, while the execution of write queries is replicated on all database servers. We prefer this approach over data partitioning approaches [9, 4], because it is simpler, and, in particular, because it leaves the application layer scripts unchanged. More importantly, using content replication allows us to leverage the high availability of the data that comes with it.

Through a series of experiments and simulations, we evaluate the relative performance impact of different strategies for load balancing among the database servers, scheduling of queries to the database servers, and consistency maintenance. The load balancing policies we have implemented and evaluated are similar to the ones commonly used for static content. We further compare in-order versus out-of-order query scheduling and strict versus relaxed consistency maintenance.

To evaluate our scaling techniques, we use the TPC-W benchmark [25]. This benchmark is designed to be representative of an e-commerce workload, specifically, an on-line bookstore. It specifies the site's data and the possible interactions with the data. It has three workload mixes. The shopping mix is meant to be the most representative. The browsing mix reflects a read-heavy workload and the ordering mix 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 [2], the PHP web-scripting/application development language [22], and the MySQL database server [18]. Since PHP is implemented as an Apache module, the web server and application interpreter must coexist on the same machine(s) (See Figure 1). This environment has become a de facto standard, at least on the Unix platform: A recent Netcraft survey [19] showed 63% of all Web sites running Apache. About 40% of these sites had the PHP module compiled in, making it the most widely used Apache module.

Our results are obtained in two ways: At a small scale, we run the implementation on a cluster of PCs. Our largest experimental setup includes 8 database server machines. At a larger scale,
we replace components of the system, such as the web server and the database with simulators that mimic their behavior at a lower cost. Thus, a single machine can simulate many data servers. We still, however, run the actual dispatcher. We use these simulations to model scaling for larger cluster sizes of up to 60 database server nodes.

Our main conclusions are:

1. With appropriate load balancing and scheduling, the TPC-W benchmark scales for all three mixes: The browsing mix has linear scaling up to 60 database machines, while the (common) shopping workload scales well up to 32 database nodes. The ordering workload scales up to 8 database engines but then flattens out. In general, the number of web servers required to achieve peak throughput grew at a slower rate than the number of database nodes.

2. Replication strategies that use relaxed consistency have the most beneficial impact.

3. Load balancing has a secondary impact and optimizing for locality has almost no impact.

4. The overhead of adding fault tolerance and data availability to the basic design is under 16% for all three mixes, with negligible overheads for the browsing and shopping mixes.

The remainder of this paper is structured as follows. Section 2 describes the scalable cluster architecture, and motivates our use of replication. Section 3 describes the scheduling techniques explored in the paper and Section 4 describes the load balancing techniques. We experimentally and by simulation investigate how the different scheduling and load balancing techniques affect scaling in Section 7. Section 8 discusses related work. Section 9 concludes the paper.

2 Cluster Architecture

We consider clusters of commodity hardware components, i.e., PCs, LANs, and an L4 switch.

2.1 Hardware

Our cluster-based server architecture consists primarily of a set of web server/application interpreter front-end nodes and a set of database back-end nodes (see figure 2). This architecture allows us to deal with bottlenecks at either the front-end nodes or the back-end nodes. Depending on the workload characteristics, a web site may require a different number of front-end nodes than back-end nodes. For example, in TPC-W, the application scripts are quite simple to interpret in comparison to many of the database queries that they generate, making it necessary to have more back-end nodes than front-end nodes.

When more than one front-end node is present, an L4 switch is also included. The use of an L4 switch makes the distributed nature of the server transparent to the clients. We assume that the L4 switch simply performs a round-robin distribution of incoming requests among web server front-ends (WS). In our experience, for delivery of dynamic content, this leads to adequate load distribution among the front-ends, and there appears to be little gain to be expected from any more sophisticated strategy.

A set of collaborating Dispatcher processes (D) act as load balancers and schedulers for the database queries. The web/application server sends the Dispatcher database queries and receives responses. A particular Dispatcher can service many web/application servers. The Dispatcher processes can be run on a separate set of machines or can be co-located with (some of) the web/application servers. The Dispatchers use a set of database proxies (DP), one at each database engine, to communicate with the databases. Each Dispatcher process sends the queries to a database
proxy who, in turn, delivers the queries to their corresponding database engine. Both the Dispatcher and database proxy layers are transparent. To the web/application servers, a dispatcher 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) and any off-the-shelf database (e.g., MySQL) without modification. Moreover, the system is easy to configure and reconfigures itself automatically in case of failures. Dispatchers and database proxies read a configuration file at startup, and set up connections accordingly.

2.2 Replication

We replicate the entire database on each of the database engines. This strategy allows a site to grow incrementally, by simply replicating the database on the new machine, and updating the Dispatchers with the identity of the new machine. There is no need for a redistribution of a partitioned database.

Traditionally, database load balancing has involved declustering (data partitioning across the cluster) [9, 4]. By using this shared-nothing model, clustered database systems have avoided consistency maintenance overheads. On the downside, declustering requires either expert administrators for database configuration and reconfiguration, or rather complex optimizers to minimize the data movement between machines [7].

Replication brings with it high data availability, but also the cost of replicating the execution of update queries for maintaining the table replicas consistent. Fortunately, queries that update the database are usually lightweight compared to read-only requests (See section 6). For instance, in e-commerce, typically only the record pertaining to a particular customer or product is updated, while any given customer may browse through the whole product database. Replication also brings with it the need for synchronization for bringing all replicas up-to-date. We address these issues through our scheduling algorithms in Section 3.

2.3 Overall Approach

The key observation for achieving good scaling and availability in a dynamic content site is that we can decouple the server’s internal view of the data from the user’s view of the data. The data is never accessed directly by the user but through the execution of scripts found at the site’s web server. This level of indirection allows the server to use a relaxed data consistency model internally, while externally it conforms to user consistency requirements.
We assume that, for each web application, the dynamic content script running at the server provides a specification for the required data consistency level of the application. The server guarantees to present to the user only data meeting the consistency specification. On the other hand, internally, the server allows controlled data inconsistencies between the database replicas to improve scaling. In this model, any database, at a given point in time can be either up-to-date or can contain stale information. The server maintains internal state to reflect the current status of each transaction and database. It then uses this state to control the level of data consistency, and to service database queries according to the application requirements specified by script annotations. Thus, any temporary data inconsistencies can be hidden from the user.

In particular, we implement the level of indirection in accessing the data and all relaxed consistency policies in the Dispatchers. The Dispatchers control the level of consistency and the load on all databases. The Dispatchers also provide high data availability. If a Dispatcher crashes, another Dispatcher takes charge of rolling forward the databases that are behind, such that users perceive only data at the required consistency level.

### 2.4 Consistency Specification

Throughout this paper, we will assume the consistency specification given by the dynamic content script is as follows.

1. All queries in a read-write script must execute as a transaction, and must operate only on up-to-date data.

2. Queries in read-only scripts are allowed to read out-of-date information with only a best effort data freshness guarantee.

This follows the consistency specification provided by TPC for the TPC-W e-commerce benchmark. Thus, the only script annotations we use are those strictly required for transaction delimitation. For more complex consistency specifications, other types of script annotations may be needed.

For all read-write PHP scripts, we insert database lock operations that obtain all locks necessary for the queries in the script (for both read and write operations) before the first query. Scripts that contain only read-only queries do not obtain explicit locks. This pre-declaration of locks in a transaction allows for deadlock-free execution. On the other hand, the locking needs to be done conservatively because we may not know beforehand the exact locks that are going to be used at run-time.

### 2.5 Operation

The Web server receives incoming client requests and executes the corresponding scripts, as before. The queries, however, are sent to a Dispatcher instead of to the database. The Dispatcher parses each incoming query to determine its type (read-only or write) and the tables accessed. Subsequently, read-only (SELECT) queries are sent to only one database machine according to the load balancing scheme in use, while the execution of write queries (INSERT, UPDATE, DELETE) is replicated on all machines. The transaction delimiters are treated like write-type queries (i.e., their execution is replicated on all machines).

Since multiple Dispatchers can send write type queries to the database engines, the Dispatchers need to enforce a consistent order for the execution of writes across all engines. To this end, the Dispatchers coordinate to assign globally unique version numbers to write-type queries. Furthermore, each database proxy delivers write-type queries to the database engine on its machine, in
the order of their version number. After the database executes the queries, it returns the results to its database proxy which forwards them to the Dispatcher. The Dispatcher updates its state and forwards the results to the Web server. At any given time, the Dispatcher state contains load information, the current availability and consistency status of each database back-end. From this information, the Dispatcher can infer which databases are accessible and up-to-date at a given time, and thus can make load balancing and scheduling decisions for queries. For the above scheme to work even in the presence of machine failures, the Dispatcher state needs to be persistent and available.

To this end, the Dispatchers communicate among themselves to replicate the Dispatcher state, and also log state changes to disk.

The key to good scaling for the entire system lies in the Dispatcher design. Inter-Dispatcher communication is necessary for the purpose of assigning version numbers to write-type queries and for maintaining the Dispatcher state with high availability. We are currently using a designated sequencer Dispatcher for assigning version numbers. Furthermore, only one message to the Dispatcher is necessary at the beginning of each read-write transaction since all accesses are pre-declared (see section 2.4). On the other hand, while not strictly necessary, we choose to replicate the Dispatcher state to all Dispatchers. Thus, for good scaling, the Dispatcher code should be lightweight enough such that each Dispatcher can support a large number of Web server front-ends and database back-ends.

3 Scheduling Queries in the Presence of Writes

3.1 Synchronous Replication with Strict Consistency

This is a basic replication scheme with in-order execution of all queries and synchronous execution of writes on all replicas. The dispatcher waits for completion of every write-type query on all database back-ends, before returning the answer to the Web server. In this basic version, each database engine is responsible for read-write and write-write conflict resolution. The Dispatchers and database proxies only pass through queries and ensure that the writes occur in the order of their assigned global version numbers.

3.2 Synchronous Replication with Conflict Avoidance

This scheduling algorithm enhances the basic synchronous scheme with out-of-order issue of queries to avoid conflicts at the database. A query is sent to a database engine only if there is no locking conflict with an outstanding operation. Queries that cannot be sent are held back. Any held back query can be sent out of order when there are no more outstanding conflicting operations. Queries from the same script are issued in-order.

This enhancement can be implemented by holding back queries either at the Dispatcher or at each database proxy. For reasons explained in section 2.5, we choose to implement conflict avoidance at each database proxy, while the Dispatcher code is kept as lightweight as possible.

3.3 Asynchronous Replication with Relaxed Consistency

This version maximizes the available parallelism by removing the restriction for synchronous execution of all write-type operations. For each write-type operation, as soon as the query completes on one database engine, the answer is returned to the Web server. This means that, at any given time, the same script can generate several outstanding queries. The dispatcher sends a new read or
write operation, only to the set of machines that have the required consistency level. In particular, reads belonging to read-only scripts are load-balanced among all databases, while reads and writes belonging to read-only scripts are sent only to up-to-date databases. To be able to do so, the dispatcher keeps a record of outstanding replicated operations.

As in the synchronous scheme, we enhance the asynchronous replication scheme with conflict avoidance at each database proxy.

4 Load Balancing Strategies for Read-Only Queries

4.1 Round Robin Schemes

The simplest load balancing policy assigns the requests in Round Robin (RR) order to back-ends. Slightly more complex, weighted round-robin is a common load balancing scheme in static-content cluster servers [14, 8]. The incoming requests are distributed in round-robin fashion, weighted by an estimate of the load on the different back-ends.

We compare two weighted round robin schemes with different load measures:

1. **Shortest Queue First (SQF)** uses the number of outstanding queries to a particular back-end as an estimate of the load on that back-end.

2. With **Shortest Execution Length First (SELF)**, we measure off-line the execution time of each query on an unloaded (idle) machine. We then estimate the load on a particular back-end as the sum of the (measured) execution times of all queries outstanding to that back-end.

SQF treats each query as equal, while SELF tries to take into account the widely varying execution times for different queries.

4.2 Locality-Aware Request Distribution Scheme (LARD)

LARD was developed and shown to be successful for load balancing static content requests in a cluster [21]. The goal of LARD is to combine good load balance and high locality. In our implementation of LARD, the dispatcher 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 dispatcher 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 dispatcher sends the query to the least loaded machine.

4.3 Load Limiting

Load limiting is a potential addition to any load balancing algorithm, in which there is a limit set on the load of outstanding queries to 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). As with the Conflict avoidance technique in section 3.2, we choose to implement this technique at each database proxy rather than at the Dispatchers. 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.

Limiting the load has the effect of smoothing out bursts of request traffic which could cause overload conditions on the database back-ends.
5 Fault Tolerance and Data Availability

Our distributed Dispatcher solution provides data availability through replication of the Dispatcher state. Thus, if one Dispatcher fails, a fail-over Dispatcher can continue the task of database coordinator for the transactions the failed Dispatcher was responsible for. Full replication of the entire Dispatcher state might however, imply extensive communication between the Dispatchers. Hence, our solution minimizes the overheads during normal operation, at the expense of increased complexity on recovery. We replicate (and log on disk) only a portion of the Dispatcher state. In particular, we maintain as replicated and hard state all the write queries that have not been applied to all database engines. On the other hand, the set of database back-ends where each write has been applied, and information about database availability are not replicated.

Communication occurs only at the end of the transaction. Before a commit is issued to any database, the Dispatcher sends a commit transaction message to the other Dispatchers. This message contains a log of write type queries that have occurred during the transaction, with the tables accessed and their update version numbers. The originator of the commit transaction message also logs the message contents to disk. The Dispatcher delays committing the transaction until the intention to commit is stable (both the disk write and remote in-memory replication are complete). In the rare case where all Dispatchers fail, the state is reconstructed from the disk logs of all Dispatchers. The more common case of isolated failures is handled through Dispatcher fail-over.

5.1 Dispatcher fail-over

When a Dispatcher fails, all live database proxies sense the failure due to broken connections and roll-back all active transactions on their corresponding database. At the same time, the fail-over Dispatcher contacts all available database proxies and asks for their current state in terms of the version numbers for all database tables. The fail-over Dispatcher then rolls-forward all databases by sending to each the writes that they have missed. Furthermore, for each Web server that was pointing to the failed Dispatcher, the fail-over Dispatcher also modifies that server's configuration to direct further requests to itself.

5.2 Database recovery from failure

When a database recovers from a failure, its database proxy has to know the state that the database is in, and needs to be able to coordinate the recovery process. This is in general not possible to achieve transparently, because database engines do not offer interfaces for this purpose.

We opt for a solution where the database proxy does periodic checkpoints of the database together with the current state (in terms of the last version numbers of the writes it has applied). To make a checkpoint, the database proxy stops all write operations going out to the database engine, and when all pending write operations have finished it takes a snapshot of the database and writes the new state. If any tables have not changed since the last checkpoint, they do not need to be included in the new checkpoint.

On recovery, the database proxy re-installs the database from the last checkpoint containing the database snapshot of the last known clean state. After this point, the database proxy contacts an available Dispatcher and provides its current database state. The Dispatcher rolls-forward the database providing the remaining portion of the write backlog, corresponding to the database state. In case of a long failure or adding an entirely new database back-end, the Dispatcher instructs the database proxy to download a snapshot of the database and the corresponding state from a recent checkpoint on another database back-end.
6 TPC-W Benchmark

The TPC-W benchmark from the Transaction Processing Council [25] 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. The read-only interactions include access to the home page, listing of new products and best-sellers, requests for product detail, and two interactions involving searches. Read-write interactions include user registration, updates of the shopping cart, two interactions involving purchases, two involving order inquiry and display, and two involving administrative tasks. We use the same distribution of script execution as specified in TPC-W. An interaction may also involve requests for multiple embedded images, each image corresponding to an item in the inventory. With one exception, all interactions query the database server. The complexity of the interactions varies widely, with interactions taking between 20 ms and 700 ms on an unloaded machine, and read-only interactions up to 30 times more heavyweight than read-write interactions. The weight of a query (and interaction) is the same for a given query type largely independent of the 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%.

6.1 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.

6.2 Hardware Platform

We use the same hardware for all machines running the emulated-client, web-servers, Dispatchers and 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.

6.3 Software

We use Apache v.1.3.22 [2] for our web-server, configured with the PHP v.4.0.1 module [22] providing server-side scripting for generating dynamic content. We use MySQL v.3.23.43-max [18] as our database server.
7 Experimental Results

7.1 Baseline Experiment

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. A dispatcher is not necessary in this configuration. There is no measurable difference in terms of throughput, however, when we interpose a dispatcher between the Web server and the database machine.

More importantly, figure 3 presents the CPU utilization on the Web server machine and the database server machine. For all the three interaction mixes, the database server is the bottleneck. For the ordering mix, the CPU utilization on the database server machine does not reach 100%. We attribute this to lock waiting times in this write-heavy workload.

All further results belong to two categories: experimental and simulated.

The experimental numbers are obtained running a prototype 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 was 7. We present these experiments in section 7.2, and section 7.3.

Secondly, we simulate further scaling for up to a cluster of 60 database nodes and present these results in section 7.4.

7.2 Overall Scaling Results

In this section we discuss overall results for the best combination of load balancing and scheduling. We discuss the relative merits of various load balancing and scheduling strategies in more detail in Section 7.3. The top curve in Figure 4 shows the overall scaling results for the best strategy (asynchronous replication) for each of the three workload mixes. In the x-axis we have the number of database machines and in the y-axis the number of interactions per second. For all the points on each curve, we are driving the server with increasing number of clients, until performance peaks, and we report the peak-point throughput.

The browsing and the shopping workload 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 good results of the shopping mix are especially encouraging as this mix is considered to be the most representative of e-commerce
site operation. The performance of the browsing mix reflects a read-heavy workload with little synchronization. Its good performance is therefore not surprising. The ordering workload mix performs less well. It scales almost linearly until 4 databases machines, with an improvement of a factor of 3.2 at 4 machines, but there is only a 24% further improvement for eight machines. This is due to the lack of parallelism in the workload. About 50% of this mix consists of update queries, which are executed on all replicas and therefore offer no room for improvement. Moreover, the higher frequency of longer transactions in this mix increases even more the probability for conflicts. Fortunately, TPC considers this mix less representative of the normal operation of ecommerce sites than the shopping mix.

Asynchronous replication is the method of choice. Figure 4 also contains the best result for any strategy that does not include asynchronous replication. Clearly, the results are inferior for all mixes. Furthermore, the bottom curves in Figure 4 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.

7.3 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.

7.3.1 Comparison of Load Balancing Methods

Figure 5 shows the relative performance of the four load balancing policies when used in conjunction with the best scheduling policy (Async). We use no limit for the database load. Setting a limit could smooth out imperfections in the load balancing due to the queueing capacity of the database proxies.

We see that the maximum difference between the performance of any two policies is around 20%. Secondly, we see that 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 access a few hot tables (e.g. item, author, order_line) parts of which become replicated in most caches independent of policy.

7.3.2 Relative Impact of All Strategies

The graphs for all the three mixes show the contribution of each strategy. Starting from the simplest strategy of RR with synchronous scheduling (Base), we add all the other enhancements, one by one.
First we add the best load balancing strategy as shown in the previous section (SELF), then we add conflict avoidance (AConfl), the limit on database load (Limit) and lastly asynchronous replication. 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 this limit. This is in contrast to choosing a limit in terms of number of outstanding queries. We use a 1 second load limit for all mixes.

Overall the most important factor is asynchronous replication and the least important is the choice of load balancing. Secondly, we see that in the browsing mix, Load Limiting is an important factor. In this workload, synchronization is rare, while heavy-weight reads can cause the database to be congested during bursts of request traffic. In the shopping mix, where read-write conflicts are relatively frequent, avoiding conflicts helps. In the ordering mix, Conflict Avoidance and Load Limiting help somewhat but the improvement is dwarfed by the factor of 3 impact on performance of asynchronous replication.

### 7.4 Simulation of Larger Clusters

To study the scaling of our techniques on a larger number of nodes and to determine the number of Dispatchers needed for larger clusters, we developed two configurable cluster simulators: one for the web/application server front-ends and the other for the database back-ends. We use these front-end and back-end simulators to drive actual Dispatchers and database proxies. Each simulator models a powerful server of the given type equivalent to running a much larger number of real-life servers. We used our experiments on the 8 database cluster to validate our simulators. At most, the results obtained through simulation differed by 12% from our experimentally obtained results.

The web/application-server simulator takes each http request generated by the client emulator and sends the queries embedded in the corresponding scripts with dummy arguments to one of
the Dispatchers, one query at a time. As before, the Dispatcher parses each query, selects a back-end node and sends the query to the corresponding database proxy, which performs any necessary reordering and passes the query to the database simulator.

Figures 7 and 8 show the aggregate cluster throughput for the three mixes with the best (Async) protocol, on a range of database cluster sizes (up to 60 nodes) with increasing number of Dispatchers (1, 2 and 3 Dispatchers). As with the experimental results, for a given number of databases we are increasing the number of clients (and web servers) until the system achieves peak throughput. For all mixes, we found that, as we scaled the system, the number of simulated web servers needed for the peak throughput grows more slowly than the number of database engines. For the browsing mix (figure 7), we see that the throughput increases almost linearly up to 60 database nodes, and one Dispatcher is enough to sustain scaling up to around 48 database nodes, when two Dispatchers are needed to continue scaling. Two Dispatchers are sufficient up to the simulated largest cluster of 60 nodes, and the addition of another Dispatcher does not improve performance.

In the shopping mix, one Dispatcher is sufficient for supporting scaling up to 24 nodes, after which more Dispatchers are needed. This number is lower than in the browsing mix, because transactions are more fine grained and the system can sustain a higher throughput at a given database cluster size.

In the ordering mix, a single Dispatcher can support scaling up to the saturation point.

Lastly, in Figure 9 we show by varying the database speed, that the scaling limitation is a function of the replication of writes and level of conflicts and not due to a Dispatcher bottleneck. Specifically, we assume a database engine twice as fast as the one we have used to this point (e.g., a multiprocessor database node) and compare the new scaling curves to our earlier results for the shopping mix. In both experiments we use 3 Dispatchers, the minimum number necessary for good scaling. Figure 9 shows that the faster database has a higher saturation point (by a factor of 1.6) in terms of throughput, and reaches this point at roughly the same number of nodes.
7.5 Fault Tolerance and Data Availability

In Figure 10 we compare the performance of the Async protocol without fault tolerance, the same protocol with the overhead of logging to disk (Reliable), and Async with logging to disk plus replicating the state using two and three Dispatchers (Reliable-2 and Reliable-3 respectively). All the measurements were done using the experimental platform with 8 databases at the peak throughput.

We can see that the overhead for fault tolerance and data availability is negligible for the browsing and shopping mixes and around 16% for the ordering mix. Furthermore, the overhead of replication in addition to logging is negligible. We do not present any numbers that reflect the overhead of checkpointing, because well known techniques for minimizing the time for taking file snapshots exist [13].

8 Related Work

Current high-volume Web servers such as the official Web server used for the Olympic games [6] rely on expensive supercomputers to satisfy the volume of requests. Nevertheless, performance of such servers is still a problem during periods of peak load. Hence, this brute-force approach of ever increasing the capacity of a stand-alone data server seems an unlikely solution for the scalability problem in the foreseeable future.

Luo et al. [17], and Oracle's 9i Database Cache product [20], use a middle-tier database cache. They rely on replication tools to periodically propagate updates from the back-end database to
the cache tier. This approach is orthogonal to ours, although it achieves the same goal, that of scaling the database tier. For applications that allow out-of-date data, we could also benefit from a similar update propagator. On the other hand, our work attempts to parallelize even e-commerce interactions that need strict consistency. Thus, we need control and accurate information on exactly when updates occur at each database.

Our LARD scheme is similar to the locality-aware request distribution proposed by Pai et al. [21], 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. [26] 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 [23] 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.

In practice, replication has previously been used mainly for fault tolerance and data availability [3, 10]. Gray et al. [11] shows that classic solutions based on eager (synchronous) replication do not scale well. Lazy replication algorithms [15, 24, 16, 1, 12] used in mobile or wide area applications scale well but do expose inconsistencies and stale data to the user. More recent work [1] has explored the possibility of using lazy replication while still providing serializability. 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. In our work, the database replicas, the consistency level between them, and the data that the user sees are all controlled by dispatchers, based on the consistency specification provided by the dynamic content script.

9 Conclusions

In this paper, we investigate how a dynamic content site can be scaled up from a single machine running a Web/application server and a database to a cluster of Web/application server machines and database engine machines. We avoid modifications to the Web/application server, the database engine, or the scripts for accessing dynamic content. 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 scaling methods are applicable without burdensome development or reconfiguration of a web site. We use the various workload mixes of the TPC-W benchmark to evaluate overall scaling behavior and the contribution of various load balancing and scheduling algorithms to good scaling behavior.

We find that a cluster architecture scales well for the most representative of the TPC-W workload mixes, the shopping mix, and also for the browsing mix. The write-heavy ordering mix scales less well. The key ingredient of a scalable load balancing and scheduling policy is asynchronous replication, in which writes complete and are returned to the Web/application server as soon as a single instance of the write completes at one of the database engines. 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.
References


