# Scaling e-Commerce Sites

Cristiana Amza, Alan L. Cox, Willy Zwaenepoel
Department of Computer Science, Rice University
{amza, alc, willy}@cs.rice.edu

**Abstract**

We investigate how an e-commerce site can be scaled up from a single machine running a Web server and a database to a cluster of Web server machines and database engine machines. In order to reduce development, maintenance, and installation costs, we avoid modifications to both the Web server and the database engine, and we replicate the database on all database machines. All load balancing and scheduling decisions are implemented in a separate dispatcher.

We find that such an architecture scales well for the common e-commerce workload of the TPC-W benchmark, provided that suitable load balancing and scheduling strategies are in place. Key among these strategies is asynchronous scheduling, in which writes complete and are returned to the user 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. In particular locality-based load balancing policies, found very profitable for static Web workloads, offer little advantage.

## 1 Introduction

E-commerce sites commonly consist of a front-end Web server and a back-end database (See Figure 1). The (dynamic) content of the site is stored in the database. A number of scripts provide access to that content. The client sends an HTTP request to the web server containing the URL of the script and some parameters. The Web server executes 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 to the client as an HTTP response.

In small sites the Web server front-end and the database run on the same machine, which may become a bottleneck. In this paper we study how to scale up such e-commerce sites by using clusters. The simplest attempt to scaling is to run the Web server and the database on a separate machine. Beyond that, multiple machines may be used for each. We impose the constraint that the Web server, the database engine, and the 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 e-commerce sites without undue development or administration.

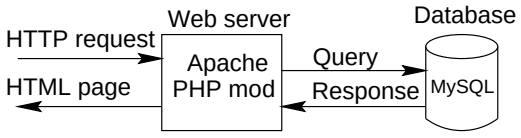


Figure 1: Common E-commerce Site Architecture

Subject to this constraint, we investigate issues such as: Should additional machines be devoted to Web servers or databases? What load balancing and scheduling policies should be used for scaling ?

Much recent work in scaling web servers has focused on load balancing for static content [15, 19, 4]. Content replication and locality-aware load distribution [15] with simple load measures such as the number of connections were shown to produce good scaling behavior. We show that e-commerce sites can also be made to scale by means of content replication, but that different load balancing and scheduling techniques are necessary. Intuitively, the need for different load balancing and scheduling techniques arises from the basic differences between static and dynamic content. The latter are typically more CPU-intensive, access a small number of tables through a small number of scripts, and, above all, contain writes.

We focus on read-one, write-all replication schemes. Read-only queries are sent according to a load balancing scheme, while the execution of write queries is replicated on all machines. We prefer this approach over data partitioning approaches [9, 3], because it is simpler, and, in particular, because it leaves the scripts unchanged. We have implemented and evaluated a number of load balancing and scheduling policies specifically designed for the characteristics of e-commerce sites.

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. We run three popular open source software packages - the Apache web server [1], the MySQL database server [13], and the PHP web-scripting/application development language [16] (See Figure 1). This environment has become a de facto standard, at least in the Unix world. The most recent Netcraft survey [14] showed 63% of all Web sites running Apache. About 40% of these sites had the PHP module compiled in. We use the TPC-W benchmark [18] to evaluate various load balancing and scheduling policies. The TPC-W benchmark models an e-commerce site. It specifies the data of the site 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.

Our main conclusions are:

1. With appropriate load balancing and scheduling techniques, the TPC-W benchmark scales well at least up to 8 machines for the browsing and the (common) shopping workloads. The ordering workload scales well up to 4 machines but then flattens out.

2. Scheduling strategies that optimize synchronization latency, such as asynchronous replication have the most beneficial impact.

3. Optimizing for locality has almost no impact.

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 load balancing techniques, and Section 4 describes the scheduling techniques explored in the paper. We experimentally investigate how the different load balancing and scheduling techniques affect scaling in Section 8. Section 9 discusses related work. Section 10 concludes the paper.

## 2 Cluster Architecture

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

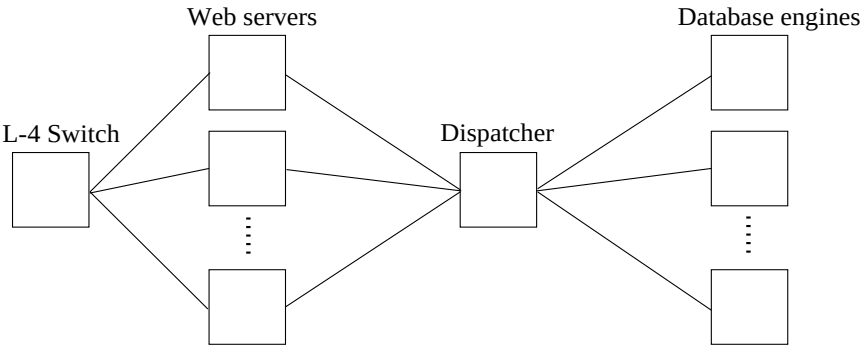


Figure 2: Cluster Architecture Design

## 2.1 Hardware

Our cluster-based server architecture primarily consists of a set of Web-server front-end nodes and a set of database engine back-end nodes (see figure 2). This architecture allows us to deal with bottlenecks both in the Web server front-ends and in the database back-ends. In general, we have found that for e-commerce workloads the database back-ends become the bottleneck before the Web server front-ends (See Section 8.1). With enough back-ends, however, there is inevitably a point where a single front-end becomes the limiting factor, so our architecture allows for multiple front-ends.

When more than one front-end is present, an L4 switch and a dispatcher node are 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. In our experience, 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.

The dispatcher implements the load balancing and scheduling of queries coming from the Web server front-ends and going to the database back-ends. If there is only a single front-end, then this functionality can be implemented on the front-end node. To the Web servers, the dispatcher looks like a database engine. The web server sends it queries and receives responses as before. Likewise, the database engines interact with the dispatcher as if it were a regular Web 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. The single dispatcher potentially introduces a scaling bottleneck. We have not found that to be the case for the number of machines we considered. We have also experimented with a primary-backup Dispatcher implementation (see section 6).

## 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 dispatcher with the identity of the new machine. There is no need for a re-organization of the database.

Traditionally, database load balancing has involved declustering (data partitioning across the cluster) [9, 3]. By using this shared-nothing model, clustered database systems have avoided consistency maintenance overheads. On the downside, it requires either expert administrators for database configuration and re-configuration, or rather complex optimizers to minimize the data movement between machines [7].

Replication brings with it the cost of replicating the execution of update queries for maintaining

the table replicas consistent. Fortunately, e-commerce queries that update the database are usually lightweight compared to read-only requests (See section 7). For instance, 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 4.

### 2.3 Operation

The Web server receives incoming client requests and executes the corresponding scripts, as before. The queries, however, are sent to the 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. We also support transaction isolation. The transaction delimiters are treated like write-type queries (i.e., their execution is replicated on all machines). The database executes the queries, as before, and sends the results to the dispatcher. The dispatcher updates its state and forwards the results to the Web server.

## 3 Load Balancing Strategies for Read-Only Queries

### 3.1 Weighted Round Robin Schemes

*Weighted round-robin* is a common load balancing scheme in static-content cluster servers [11, 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 to calculate. 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.

### 3.2 Load Limiting

Load limiting is an addition to SQF and SELF, 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 number of queries for SQF and in terms of execution time for SELF. If the load for all back-ends is over the limit, the dispatcher holds on to the queries and does not assign them to any back-end, until the load on a back-end drops below the limit. Limiting the load has two beneficial effects. First, when there is sufficient load on each back-end, the load balancer delays its decision until later and can make an assignment based on more current information. Second, it avoids overload conditions on the database back-ends.

### 3.3 Locality-Aware Request Distribution Scheme (LARD)

LARD was developed and shown to be successful for load balancing static content requests in a cluster [15]. 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 Scheduling Queries in the Presence of Writes

### 4.1 Synchronous Replication without Conflict Avoidance

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, while the Dispatcher only ensures a consistent order for the execution of writes on all engines.

### 4.2 Synchronous Replication with Conflict Avoidance

The dispatcher sends a query to a particular database engine only if there is no read-write or write-write conflict with an outstanding operation. Queries that cannot be sent are held back at the dispatcher. Held back queries can be sent out of order when there are no outstanding conflicting operations on any of the machines. As before, replication is synchronous (i.e., the dispatcher waits for completion on all machines before returning the answer to the Web server). As with the load limitation technique in Section 3.2, withholding conflicting queries has two beneficial effects: It allows the dispatcher to make decisions on more current load information and it reduces database overload.

### 4.3 Asynchronous Replication

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. All the queries from the same script are issued in-order. The dispatcher sends a new read or write operation, only to the set of machines that are up-to-date. To be able to do so, the dispatcher keeps a record of outstanding replicated operations. As in scheduling for conflict avoidance, a query is not sent to a particular machine, if there is a conflicting outstanding operation on that machine.

## 5 Dispatcher Implementation

The dispatcher is a multithreaded process using POSIX threads, that handles all communication between the Web servers and the database engines. It has one thread, created dynamically, for each Web server process processing a script.

In the basic synchronous implementation, all threads simply pass all queries to and from the database engines, without waiting.

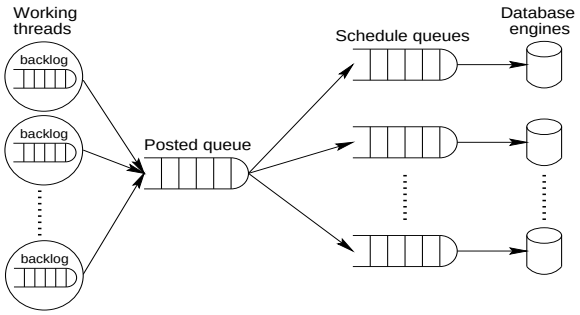

Figure 3: Dispatcher Implementation for the Asynchronous Version

Under load limitation and scheduling for avoiding conflicts, when a new query is received, the corresponding worker thread parses it, and posts it in a shared `Posted Queue` (See Figure 3). All sends to the databases engines are handled by a separate manager thread. The manager examines the queries from the queue, and decides when and to which engine(s) to send each query. When the manager cannot send any further queries, it blocks on a semaphore, waiting for posting events.

For asynchronous replication, we decouple the individual execution of the database engines through a set of `Schedule Queues`, one per engine. For each write-type query in the `Posted Queue` the manager replicates the needed information to all the `Schedule Queues`. Once put in a schedule queue, queries can proceed if conflicts have cleared on their corresponding engine. Furthermore, each worker thread keeps a `Backlog Queue` with detailed information about the status of each pending query. The status is updated whenever a query is posted, moved to schedule queues, sent to database engines or when one of the engines sends back a reply.

For each ongoing transaction, the Dispatcher keeps a `Dispatcher transaction state` which contains information about each active transaction (such as the associated script and which databases have committed the transaction). In addition, the `Dispatcher transaction state` records the current availability of all database back-ends. From this information, the Dispatcher can infer which databases are up-to-date at a given time.

## 6 Fault Tolerance and Data Availability

This section discusses enhancements to the basic e-commerce cluster design such as robustness and data availability.

In the basic design, despite having high database replication, data availability is limited. If the Dispatcher is down, we cannot access the data. We can provide both fault tolerance and data availability through *replication of the Dispatcher state* onto a backup Dispatcher. If the primary Dispatcher fails, the backup Dispatcher can continue the task of database coordinator for the transactions the primary was responsible for. We do not perform any disk logging at the Dispatcher(s). If both Dispatchers fail, the Dispatcher state can be reconstructed from the logs kept at the Web servers and databases.

Full transaction state replication onto the backup would imply extensive communication between the two Dispatchers. We choose a solution which minimizes the overheads during normal operation, at the expense of increased complexity on recovery. In our solution, communication occurs only at the end of the transaction. Before any commit is issued to a database, the primary Dispatcher sends a commit_transaction message to the backup. This message contains a transaction identifier and a log of write_type queries that have occured during the transaction.

On fail-over, the backup Dispatcher rolls-back all active transactions on the database back-

ends. If a database has already committed a transaction, the attempted roll-back would return an error message. After the roll-back, the log of writes is replayed on all the databases that have not already committed. On each commit_transaction message, the primary piggybacks information about transactions that have been committed on all databases. This allows the backup to update its list of active transactions.

For the above scheme to work, we need a mechanism which enables the backup to roll-back transactions on behalf of the primary who initiated them. This is not possible by default with our database engine. We add a communication daemon at each database engine. This daemon serves as an intermediary between the Dispatchers and database engines. It receives, and delivers the queries to its database engine. Furthermore, upon receiving a a special command from the backup Dispatcher, this daemon becomes in charge of rolling-back all active transactions on its machine and reporting back on the outcomes. This scheme also takes care of the case where the primary Dispatcher fails before reaching the commit point of the transaction, and thus the backup doesn't know what transaction needs to be rolled-back.

# 7 TPC-W Benchmark

The TPC-W benchmark from the Transaction Processing Council [18] is a transactional Web benchmark specifically designed for evaluating e-commerce systems. The performance metric reported by TPC-W is the number of *Web interactions per second*. 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. TPC-W simulates three different interaction mixes by varying the ratio of *browse* to *buy*: **browsing**, **shopping**, and **ordering**.

Table 1 lists all the 14 different interactions and their proportions in the different profiles. The column *Time*, in the table refers to the average time (in milliseconds) measured for each interaction on an unloaded machine. This gives an idea of the relative complexity of the interactions. Each interaction also involves requests for multiple embedded images, each image corresponding to an item in the inventory. Except for *Order Inquiry*, all interactions query the database server.

Table 2 lists the database tables and their sizes in our test-bed. The sizes include that of the necessary indexes on each of the tables to make the queries in the interactions efficient. The inventory images, totaling 18.3 GB of data, are resident on each of the Apache servers.

We implement each of the interactions as a separate PHP script. We also 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 session time and think time are generated from a random distribution with the specified mean. We use 100 clients, a mean session length time of 2 minutes, and a mean think time of 1 second for all our experiments.

## 7.1 Hardware Platform

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

| Web Interaction | Browsing | Shopping | Ordering | Time (ms) |
|---|---|---|---|---|
| **Browse** | **95%** | **80%** | **50%** | |
| Home | 29.00% | 16.00% | 9.12% | 26 |
| New Products | 11.00% | 5.00% | 0.46% | 241 |
| Best Sellers | 11.00% | 5.00% | 0.46% | 701 |
| Product Detail | 21.00% | 17.00% | 12.35% | 25 |
| Search Request | 12.00% | 20.00% | 14.53% | 22 |
| Search Results | 11.00% | 17.00% | 13.08% | 288 |
| **Order** | **5%** | **20%** | **50%** | |
| Shopping Cart | 2.00% | 11.60% | 13.53% | 37 |
| Customer Registration | 0.82% | 3.00% | 12.86% | 19 |
| Buy Request | 0.75% | 2.60% | 12.73% | 47 |
| Buy Confirm | 0.69% | 10.20% | 10.18% | 40 |
| Order Inquiry | 0.30% | 0.75% | 0.25% | 3 |
| Order Display | 0.25% | 0.66% | 0.22% | 77 |
| Admin Request | 0.10% | 0.10% | 0.12% | 21 |
| Admin Confirm | 0.09% | 0.09% | 0.11% | 4869 |

Table 1: Web Interaction Mix and Characteristics

| Table Name | Number of Rows | Table Size (KB) |
|---|---|---|
| Customer | 2,880,000 | 1341139 |
| Address | 5,760,000 | 861567 |
| Orders | 2,592,000 | 269090 |
| Order_Line | 7,782,313 | 988555 |
| CC_XACTS | 2,592,000 | 252800 |
| Item | 100,000 | 63680 |
| Author | 25,000 | 9540 |
| Country | 92 | 8 |

Table 2: Database Table Characteristics

### 7.2 Software

We use Apache v.1.3.22 [1] for our web-server, configured with the PHP v.4.0.1 module [16] providing server-side scripting for generating dynamic content. We limit the maximum number of Apache processes to 100, the same value as the number of clients. We use MySQL v.3.23.43-max [13] as our database server.

## 8 Experimental Results

### 8.1 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 the dispatcher between the Web

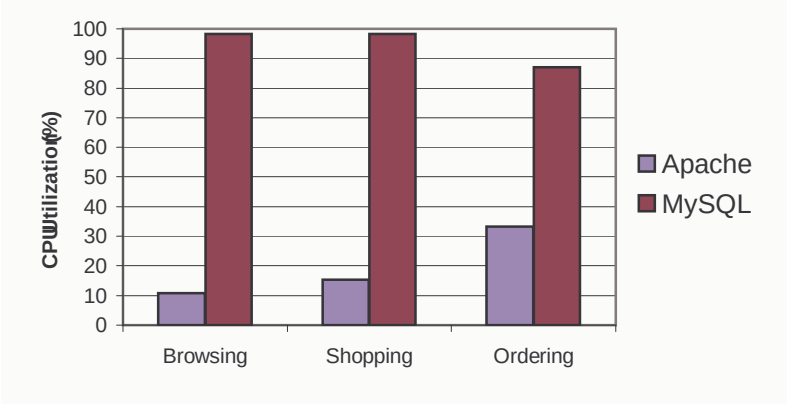


Figure 4: CPU Utilization for the Web server and Database server in the 1-1 configuration

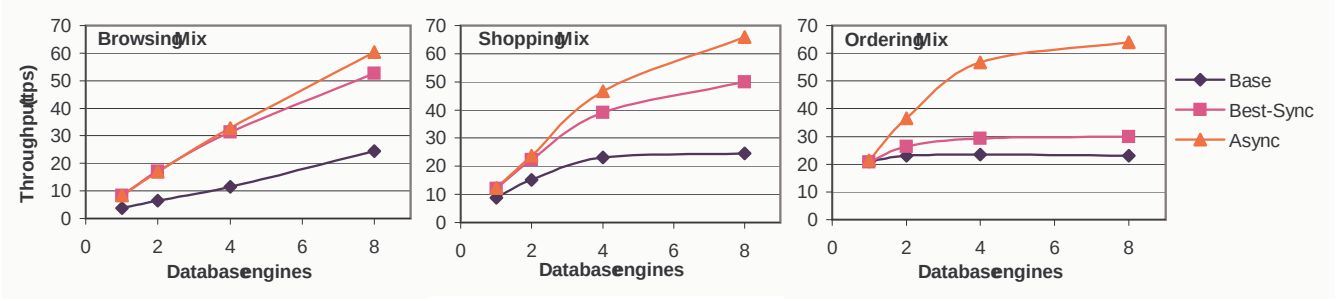


Figure 5: Throughput scaling for the browsing, shopping and ordering mixes

server and the database machine.

More importantly, figure 4 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 are obtained with 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 3. The dispatcher is never a bottleneck for these cluster sizes.

## 8.2 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 strategies in more detail in Section 8.3. The top graph in Figure 5 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.

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 7 improvement for the browsing mix and a factor of 6 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

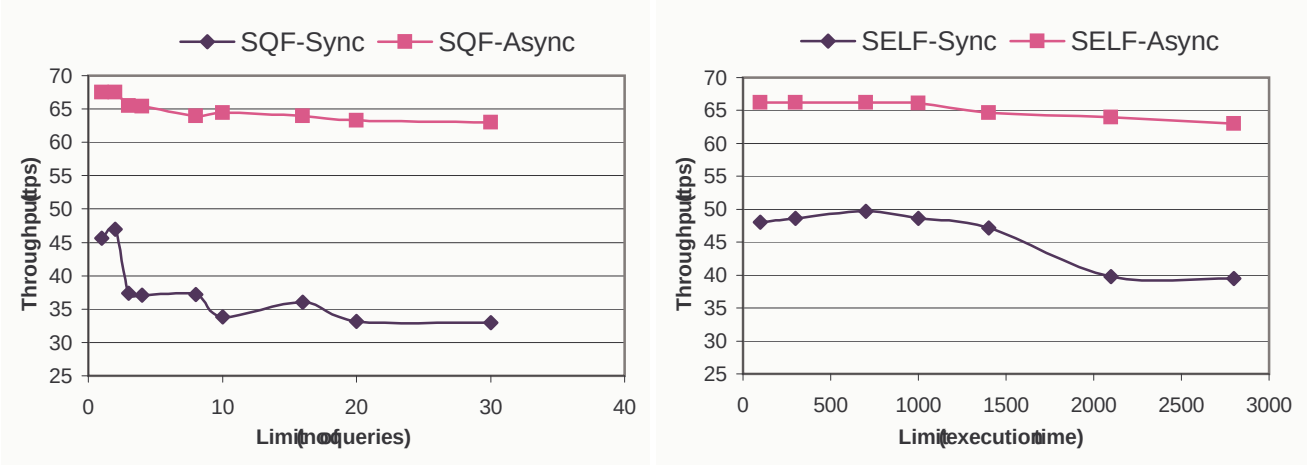


Figure 6: Throughput for asynchronous versus synchronous replication as a function of threshold value for SFQ and SELF for shopping mix

less well. It scales linearly until 4 databases machines, with an improvement of almost a factor of 3 at 4 machines, but there is only a 10% further improvement for eight machines. We attribute this to a 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. Fortunately, this mix is considered less representative of the normal operation of e-commerce sites than the shopping mix.

### 8.3 Comparison of Load Balancing and Scheduling Methods

Asynchronous replication is the method of choice. Figure 5 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 graphs in Figure 5 presents the scaling results for a shortest queue first (SQF) distribution strategy with synchronous scheduling of queries, no load limiting, and no attention paid to conflicts. The results are poor for all workloads.

For asynchronous replication, the choice between SQF, SELF, or LARD, and the exact value of the threshold matter much less in terms of overall throughput when compared to the difference between synchronous and asynchronous replication. In Figure 6, for instance, we present the 8-machine throughput for the shopping mix for asynchronous versus synchronous replication for SQF and SELF as a function of the threshold used. Unlike synchronous, the resulting throughput for asynchronous is almost independent of the choice between SQF and SELF for a large range of thresholds.

### 8.4 Detailed Comparison

As a final comparison, we present the results for various combinations of load balancing and scheduling strategies for 8 processors for each workload mix in Figures 7 to 9.

The graphs for all the three mixes show: all load balancing versions with Basic Synchronization (the first three bars), Conflict Avoidance (the bars in the middle) and Asynchronous Scheduling (the last three bars). All the Basic Synchronization versions assign queries immediately, while all the other versions can accumulate queries. We want to separate the gains through Conflict Avoidance and Load Limiting since they have the same net effect: reducing database congestion and increasing the decision window for the load balancer. To show the gains obtained by each

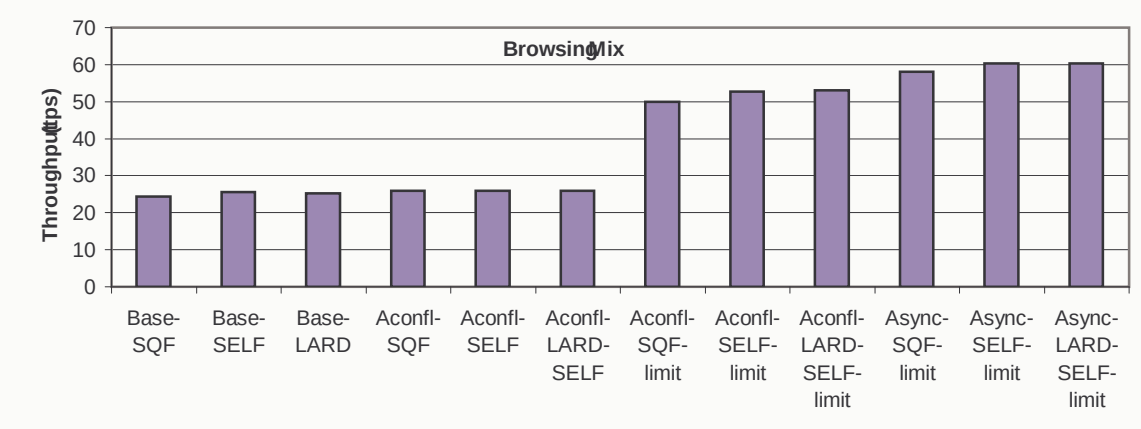


Figure 7: Throughput comparisons for the TPC-W browsing mix


Figure 8: Throughput comparisons for the TPC-W shopping mix

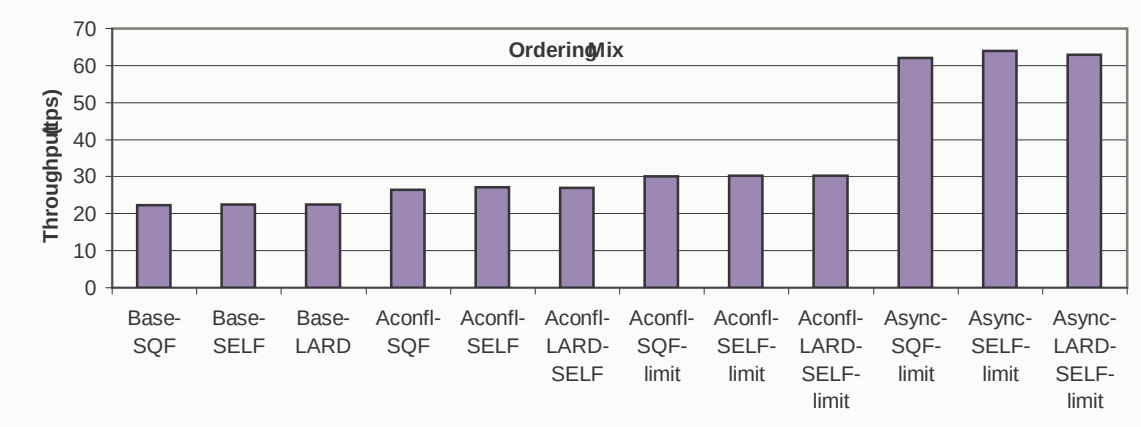


Figure 9: Throughput comparisons for the TPC-W ordering mix

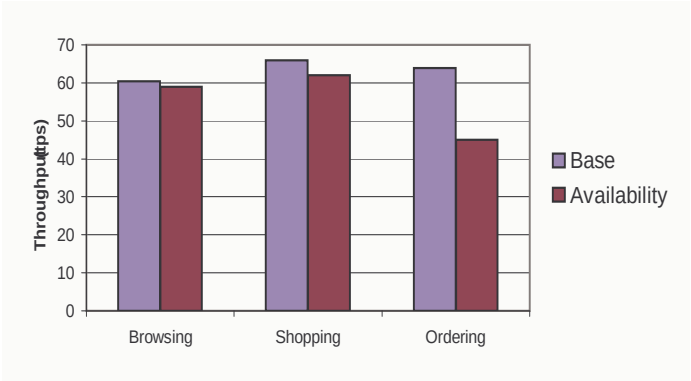


Figure 10: The overhead of providing fault tolerance and data availability

individually, we present the Conflict Avoidance results with and without Load Limiting. For all the bars where a limit is shown, we use the best experimentally determined limit for SQF and SELF. In particular, the best limit value for SQF was measured through sensitivity experiments as: 1 for the browsing mix, 2 for the shopping mix and 10 for the ordering mix. A 1 second load limit for SELF gives good results for all protocols and mixes. In all LARD experiments, we use a load measure in terms of execution time, with the same overload threshold of 1 second as in SELF.

For the browsing mix, we see that Load Limiting is an important factor, with scheduling for synchronization a second order improvement. In this workload, synchronization is rare, while frequent heavy-weight reads cause the database to be congested.

For the other two mixes, scheduling for synchronization plays the most important role. In the shopping mix, where there is still database congestion due to heavyweight reads, and conflicts are relatively frequent, avoiding conflicts helps. All the asynchronous versions further improve performance by a significant fraction compared to the best synchronous version.

In the ordering mix, the database is not congested due to the lightweight workload with a high fraction of writes, thus both techniques that reduce congestion (Conflict Avoidance and Load Limiting) are of little help. On the other hand, the high frequency of synchronizations explains the factor of 2 impact on performance of all Asynchronous schedulers.

In all the LARD combinations, locality does not bring any benefits compared to SELF. This is mainly due to the compute-intensive nature of the read queries. Furthermore, most read queries access only a few tables (e.g. item, author, order_line).

In Figure 10 we show that the overhead for fault tolerance and data availability is negligible for the browsing and shopping mixes and around 25% for the ordering mix. The explanation is that read-only scripts are not transactional in TPC-W, and thus do not incur overhead, while read-write scripts are lightweight and transactional (incur overhead).

### 8.5 Summary

For the interaction mixes that contain a non-trivial fraction of write-type queries such as shopping and ordering, asynchronous scheduling gives the relatively largest gains. Limiting machine load is still important for read-heavy workloads such as the browsing mix. However, using a load measure in terms of execution length allows for relative independence on the exact value of this limit. In contrast, we cannot choose any one limit in terms of number of outstanding queries that will give good performance for all three mixes.

## 9 Related Work

Current high-volume Web servers such as the official Web server used for the Olympic games [5, 6] and IBM's WebSphere, Commerce Edition [12] rely on expensive supercomputers for scaling to high request volumes. Our solution provides scalability using commodity hardware and software with no modifications.

Our LARD scheme is similar to the locality-aware request distribution proposed by Pai et al. [15]. 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, scheduling requests for reducing synchronization latency is more important than distributing requests for locality.

Zhang et al. [19] 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 web server, replication implies the need for consistency maintenance. In this paper, we develop and evaluate scheduling techniques that take into account the interplay between load balancing and consistency maintenance.

Our load balancer is also related to load balancing schemes used in cluster database systems [9], although the approaches are orthogonal. The traditional approach to load balancing in cluster database systems has been that data placement drives the load balancing. We use replication instead of declustering for data placement. Previously, replication has been mainly used for fault tolerance and data availability [2, 10].

Neptune [17] adopts a primary-copy approach to providind 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 Conclusion

In this paper, we investigate how an e-commerce site can be scaled up from a single machine running a Web server and a database to a cluster of Web server machines and database engine machines. We avoid modifications to the Web server, the database engine, or the scripts for accessing dynamic content. We also assume software platforms in common use: Apache web servers, mySQL database engine, and the PHP scripting language. As a result, our scaling methods are applicable without burdensome development or reconfiguration of the 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 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, in terms of response times and insensitivity to threshold values, 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


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

[2] J. F. Bartlett. A Non Stop kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 22–29, December 1981.

[3] Haran Boral, William Alexander, Larry Clay, George Copeland, Scott Danforth, Michael Franklin, Brian Hart, Marc Smith, and Patrick Valduriez. Prototyping Bubba, A Highly Parallel Database System. In *IEEE Transactions on Knowledge and Data Engineering*, volume 2, pages 4–24, March 1990.

[4] Enrique V. Carrera and Ricardo Bianchini. Efficiency vs. portability in cluster-based network servers. In *Proceedings of the 8th Symposium on the Principles and Practice of Parallel Programming*, pages 113–123, June 2001.

[5] 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*, 1998.

[6] Jim Challenger, Arun Iyengar, Karen Witting, Cameron Ferstat, and Paul Reed. A Publishing System for Efficiently Creating Dynamic Web Data. In *Proceedings of IEEE INFOCOM 2000*, March 2000.

[7] Surajit Chaudhuri and Gerhard Weikum. Rethinking database system architecture: Towards a self-tuning RISC-style database system. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases*, pages 1–10, September 2000.

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

[9] George Copeland, William Alexander, Ellen Boughter, and Tom Keller. Data placement in Bubba. In *Proceedings of Association for Computing Machinery Special Interest Group on Management of Data*, pages 99–108, June 1988.

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

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

[12] Anant Jhingran. Anatomy of a real e-commerce system. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data*, May 2000.

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

[14] The Netcraft Webserver Survey. http://www.netcraft.com/survey/.

[15] 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 3–7, 1998.

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

[17] 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.

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

[19] 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*, June 2000.