Alternative Architectures and Protocols for Providing Strong Consistency in Dynamic Web Applications

M. Hossein Sheikh Attar, M. Tamer Özsu
School of Computer Science
University of Waterloo
{mhsheikh,tozsu}@uwaterloo.ca

Abstract

Dynamic Web applications have gained a great deal of popularity. Improving the performance of these applications has recently attracted the attention of many researchers. One of the most important techniques proposed for this purpose is caching, which can be done at different locations and within different stages of the process of generating a dynamic Web page. Most of the caching schemes proposed in literature are lenient about the issue of consistency; they assume that users can tolerate receiving stale data. However, an important class of dynamic Web applications are those in which users always expect to get the freshest data available. Any caching scheme has to incur a significant overhead to be able to provide this level of consistency (i.e., strong consistency); the overhead may be so much that it neutralizes the benefits of caching. In this paper, three alternative architectures are investigated for dynamic Web applications that require strong consistency. A proxy caching scheme is designed and implemented , which performs caching at the level of database queries. This caching system is used in one of the alternative architectures. The performance experiments show that, despite the high overhead of providing strong consistency in database caching, this technique can improve the performance of dynamic Web applications, especially when there is a long network latency between clients and the (origin) server.

Keywords: database caching, web caching, strong consistency, caching dynamic web, mid-tier database caching

1 Introduction

1.1 Background

The Internet, in general, and the Web, in particular, have changed their role from tools used by a few researchers to an important part of people's everyday lives. This rapid increase in usage has caused problems, such as network bandwidth contention and server overload. These problems are more critical for servers containing popular documents and services. Moreover, due to physical limits of data transfer speed, there is an inherent latency problem when a client accesses data from a remote Web server. Several solutions have been proposed to alleviate the above-mentioned problems. Caching, especially proxy caching, seems to be one of the most effective solutions, if not the most effective one. A proxy cache intercepts requests of a (selected) group of Web clients and tries to answer them using locally cached data. An important and challenging issue that should be addressed in any Web caching scheme is *cache consistency*. Cache consistency means ensuring that caches "use only fresh cached content" [32]. Some caching systems provide *strong consistency*, i.e., they guarantee that stale data are never served to users, while others provide *weak consistency*, i.e., they eventually propagate the most up-to-date copies of data to all caches, but allow stale data to be served for some time.

Research shows that the maximal hit ratio in proxy caches is about 50% [36]. It has been stated that "this limitation is mainly due to the dynamic nature of many HTML documents, which prevents them to be cached at the proxy level" [36]. Caching dynamic Web pages involves many more challenges, compared to caching static Web pages. Firstly, contrary to their static counterparts, dynamic Web documents do not exist as plain HTML files in the Web server. In fact, they are dynamically generated per each user request. Secondly, dynamic pages often change more frequently [25], compared to static pages. This is because Web applications are usually based on some data sources that may be constantly updated. Furthermore, if caches are allowed to update cached data locally, solving the cache consistency problem becomes even more difficult.

1.2 Problem Definition

In this paper, we study alternative architectures and protocols for a dynamic Web application that requires strong consistency. The focus is on finding the best architecture for situations in which the network latency between the client and the origin server is long. We consider three architectures:

- *No caching or replication*: In this architecture, there is one instance of each server (Web server, application server and database server), and no caching or replication is performed. Consequently, no measure needs to be taken to prevent serving stale or inconsistent data to users. Note that we assume, in this architecture, that the servers are co-located (or at least the network connection between them is such that the communication delay is negligible). This architecture is depicted in Figure 1.

- *Using proxy servers to replicate static data and application code*: This setting implies having full-fledged Web and application servers at each proxy. Since all the requests

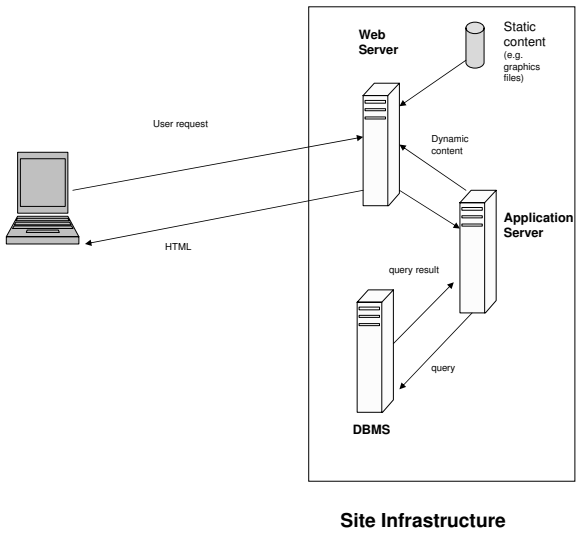

Figure 1: No Caching or Replication

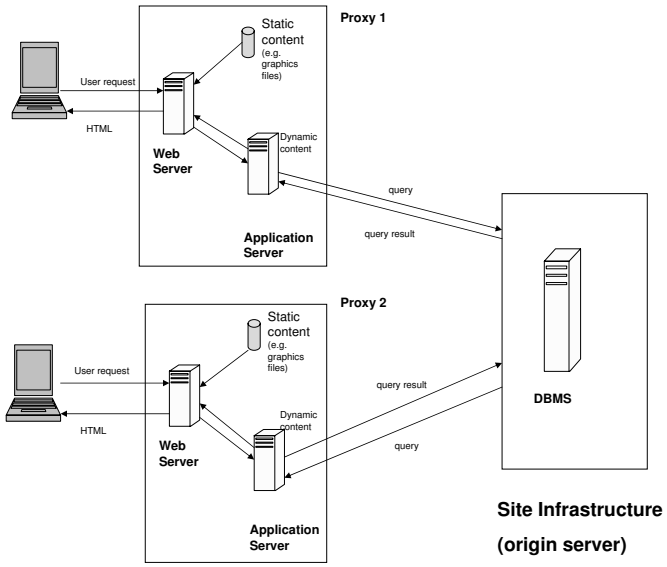

Figure 2: Using Proxy Servers to Replicate Static Data and Application Code

for the database objects (queries) eventually go to a single data source (origin DBMS), strong consistency is automatically provided. Figure 2 shows this architecture.

- *Using proxy servers to replicate static data and application code as well as to cache*

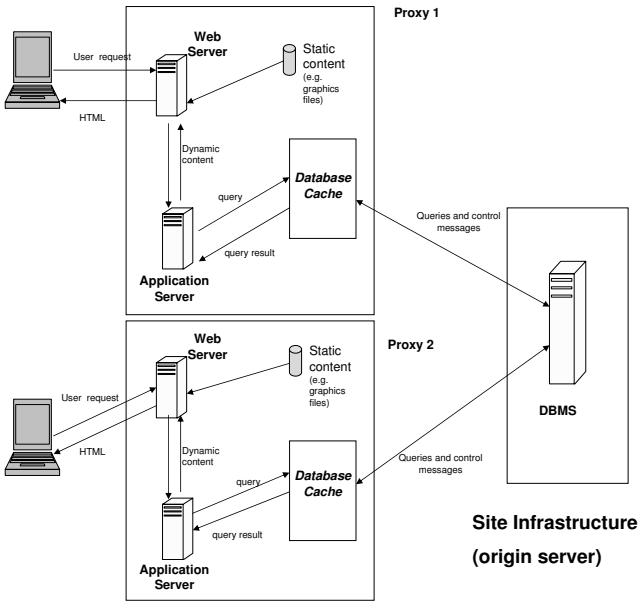

Figure 3: Using Proxy Servers to Replicate Static Data and Application Code as well as to Cache DBMS Objects

database objects: Here, similar to the previous case, each proxy server hosts full fledged Web and application servers. However, the application server sends its database queries to a local database cache, rather than the origin DBMS. In case of a cache hit, the query is answered without any need to contact the origin server. In case of a cache miss, the query is sent to the origin DBMS and the result is cached at the local database cache. This architecture is shown in Figure 3.

The first two alternatives are straightforward. The third one, however, involves using a protocol to keep database objects cached at proxies consistent with the data at the origin database server. In this paper, we examine each of these alternatives and their performance using TPC-W [35] as a benchmark that represents a large group of web-based applications (i.e., e-commerce applications). For the third architecture, we have designed a consistency protocol and implemented a database cache that uses it to answer the queries of the local application server.

1.3 Motivation

As mentioned in Section 1.1, providing (some level of) consistency is an essential need in caching systems. However, different Web applications require different levels of consistency. In some cases users can tolerate getting stale data, while in other applications (e.g., online shopping, stock quotes, and online auctions) users always expect to receive the most up-to-date data available.

It is important to notice that caching database objects while providing strong consistency is inherently expensive in terms of the number of necessary checks and control messages. It

may be argued that this overhead neutralizes or even exceeds the benefits of caching. This can explain why, as can be seen in Section 2, most of the works that propose dynamic content caching at the database level either provide only weak consistency, or are designed for applications with read-only queries. Although providing strong consistency in caching dynamic Web applications imposes a significant amount of overhead, it may still have benefits. In particular, the benefits of caching in terms of reducing the *network latency* may potentially outweigh its overhead. For example, consider a popular e-commerce web site that is located in North America, but serves customers from all over the world. For customers from other continents, the inter-continental network latency accounts for a considerable part of the response time of each Web Interaction. An important observation is that (unlike other factors that increase response time; e.g., overloaded servers or limited bandwidth) the problem of network latency cannot be alleviated by using more powerful hardware or software; it is caused by a physical limitation, namely, the maximum signaling speed. One way to solve this problem is to use proxy servers. In the above example, several proxy servers can be set up in different locations around the world, such that every customer connects to a near-by proxy instead of the origin server. Another scenario in which such proxies can be useful is e-commerce services for the cellular phones that can access the Internet. Using proxy servers results in much shorter network latencies, which may yield better total performance (e.g., shorter response times) provided that the performance benefits gained in terms of decreasing the network latency exceeds the overhead of providing strong consistency. Therefore, it is useful to examine the benefits and/or limitations of providing strong consistency in caching of dynamic Web applications.

1.4 General Approach

We have implemented a database cache that executes queries issued by the local application server providing strong consistency. We have then compared our implementation with two alternative architectures that do not cache dynamic contents at all. Our aim is to investigate which architecture yields better performance results in a benchmark that is representative of the behavior of e-commerce applications. Note that caching for dynamic Web applications can be done at different levels and granularities. Examples include caching HTML pages, page fragments, and database caching. In this paper, we consider only database caching.

1.5 Paper Organization

Section 2 is an overview of the related work. Section 3 presents a detailed explanation of the alternative architectures examined in this paper. We discuss the experimentation platform and the experimental results in Section 4. Finally, Section 5 presents our conclusions, as well as directions for future work.

2 Background and Related Work

Caching results in the creation of multiple copies of a data item; therefore, some strategy should be adopted to guarantee the consistency of data. This problem is well studied especially in the context of static Web pages; a number of consistency algorithms have been

proposed (e.g., [21, 38, 20, 31, 23, 37]), and a number of experimental results have been reported(e.g., [9, 28]).

In weak consistency mechanisms, caches check the validity (freshness) of cached items with the origin server only periodically (in order to reduce the network bandwidth consumption and the server load). Two main mechanisms for providing weak consistency are *Time-To-Live (TTL)* and *Client Polling*.

In strong consistency, it is guaranteed that no stale item is ever returned to users. Contrary to common belief, it is possible that providing strong consistency may not necessarily result in considerably lower performance compared to providing weak consistency [28]. The experimental results show that one of the strong consistency algorithms they examined, namely Invalidation, can show the same performance, in terms of network traffic, average client response time, and server CPU loads, as (a variation of) TTL [28]. In addition, experiments performed using the TPC-W benchmark show similar results [9].

Based on the classification used in [9], [20], and [28], there are three main categories for strong consistency mechanisms: *cache-driven*, *server-driven*, and *hybrid*. In this section, we present representative examples for each of the above-mentioned classes of algorithms.

In cache-driven mechanisms, consistency control is invoked from the cache, and servers are not responsible for assuring consistency of cached data. *Polling-Every-Time* [28] is the simplest strong consistency algorithm of this type.

In server-driven consistency algorithms, the origin server is responsible for providing consistency. This task usually involves maintaining a considerable amount of state at the server. *Invalidation* is a representative server-driven consistency algorithm. In invalidation, when an item is updated, the origin server sends invalidation messages to all the caches holding that item. Some variations of the basic Invalidation algorithm have been proposed. For example, in [21] authors suggest using *propagation* along with *invalidation*, where an object is propagated to all replicas when it is changed at the origin server.

In hybrid algorithms, the origin server and caches share the responsibility of providing consistency. An important algorithm in this category is *Leases*, first proposed by Gray and Cheriton [23], and later revised by others [38, 20, 31, 37].

While caching static Web pages is not a new technique, caching dynamically generated pages is a relatively new idea. In the past, most caching systems treated dynamic pages as uncacheable; but recently, many techniques for caching dynamic Web pages have been proposed. Caching systems can be classified in terms of different (orthogonal) dimensions, such as the location of the cache and type of cached data.

Caches can be placed in several places. Back-end caches cache content within the site structure. Examples of such systems include [36, 34, 11, 27, 14].

Reverse proxy caches are located between the origin server firewall and the Internet cloud [17]. Oracle Web Cache [5] and the system proposed in [8] are examples of reverse proxy caches. Content Delivery Networks (CDNs) can be thought of as reverse proxies shared by multiple Web sites [32]. Inktomi Content Delivery Suite [15] is an example of a CDN.

Forward proxy caches are located on the network path between (a set of) clients and the origin server. Client caches reside on the client machine. These two types of cache are generally controlled by end-users and the origin server may even be unaware of their existence. Systems proposed in [30, 17, 10] can be used as forward proxy caches. [7] discusses the design of a client-side Web cache.

As mentioned before, we can classify caches in terms of the type of cached objects. Some caching systems cache HTML pages [8, 11, 10, 25, 33]. Some systems cache page fragments rather than entire HTML pages in order to provide a finer granularity [7, 5, 12, 17, 19]. Another approach is caching database data and/or query results. Some caching systems cache query results inside the DBMS. This can be done by using materialized views or special relations called *cache functions* [36, 22]. Another technique is caching parts of the database data and letting the Web application use the locally cached data instead of querying the origin database server. Systems discussed in [30, 34, 14, 29, 3, 6, 1, 2] are all examples of such systems. One of the architectures studied and implemented in this paper (namely Architecture III) is also a database cache; however, contrary to the systems mentioned above, our database cache provides strong consistency. Finally, in some systems, different types of data items are cached at the same time. Providing such flexibility is motivated by the fact that there is no single performance bottleneck in a data-intensive Web site [36].

3 Architectural Alternatives

3.1 Assumptions and Scope

Although dynamic Web applications have many common properties, they have many differences as well.

In this paper, we focus on examining alternative architectures for dynamic Web applications that have the following characteristics:

1. They are data intensive, in the sense that they need to frequently read/update the contents of their data source to generate the dynamic pages.

2. They use a relational database as their data source. In these applications the front-end (Web/application server) queries a back-end DBMS to get the necessary data for generating the dynamic content and to update the application data.

3. Their users expect the system to provide them with the most recent information available. In other words, caching and/or replication should not cause any inaccuracy or staleness in the information given to the users.

There are several real-world applications with these characteristics. Examples include e-commerce Web sites, online auctions, bulletin board system, and Web sites that provide their users with stock quotes.

Our research is restricted in the following ways:

- We do not consider architectures in which the back-end database is replicated. Our focus is on caching. Other works (e.g., [4]) have studied database replication for Web-based applications extensively.

- We only consider caching database items, not HTML pages or fragments.

It is worth mentioning that, as argued in Section 1.3, our primary motive is to find the best architecture in terms of *user experience*, in particular the average response time. As a result, we are not concerned about issues such as comparing the bandwidth consumption and the load on the servers in each architecture.

3.2 Database-Centric Characterization of Dynamic Web Applications

The properties of the cached data items have a strong influence on the performance of any caching system. The system that we are considering deals with database objects that are used in a dynamic Web application. Therefore, we should first consider the characteristics of such an environment (e.g., the access pattern to data items). We rely on TPC-W [35] as a benchmark that specifies a representative workload of e-commerce Web applications. The following is a summary of our observations.

- In an e-commerce application most of the transactions are short; that is, they contain only a few SQL statements and each SQL statement is usually a simple SELECT, INSERT, or UPDATE. However, there are usually a few transactions that involve a long atomic sequence of operations [4].
- Data access pattern exhibits a great deal of locality.
- Many users merely browse the Web site and hence, generate only read requests. The users that actually buy something tend to generate several read requests while browsing the site, and finally issue a few updating transactions.[1] Therefore, the number of read requests is much higher than write requests.
- Database updates are usually confined to one record. This lowers the overhead of invalidations caused by the update operations. Moreover, there are tables that are read-only or are rarely updated.

3.3 Alternative Architectures

In this paper, we examine three alternative architectures and study the characteristics and performance of each.

3.3.1 Architecture I: No Caching or Replication

In this architecture, each client (browser) directly communicates with the origin Web server. The Web server invokes the application logic to generate dynamic content. In addition, it serves static content, such as graphic files. The application sends its queries to the DBMS. As depicted in Figure 1, the Web and application servers and the back-end database all reside at the same site so that they can communicate with minimum overhead. Because of its simplicity, this setting is used in many dynamic Web sites. We use this architecture as our *base case*.

3.3.2 Architecture II: Replicating Static Data and Application Logic

In this architecture, in addition to the origin server, there can be several proxies. Each proxy hosts full-fledged Web and application servers, as well as the whole static contents of the

[1] Note that adding items to shopping cart or removing them is not necessarily a database update; it can be done in temporary data structures.

site and the complete application code, but no database. Each application server sends its queries directly to the DBMS at the origin server. As shown in Figure 2, in this architecture proxies are responsible for running the application, as well as for storing and serving of static contents (such as graphic files). The origin server takes care of the database.

The rationale for using this setting is as follows. A client (browser) usually has to connect to the Web server to send a request and get an HTML page as the response. It, then, parses the HTML page to find references to static data items, such as graphic objects, and fetches them using additional requests. This means that the client may have to connect to the Web server several times. In situations where there is considerable network latency between the client and the server, the client incurs a long response time. The existence of a nearby proxy helps the client fetch everything, except for the HTML page, with short network latency. Replicating the application logic on the proxy brings the additional benefit of reducing the load on the origin server.

3.3.3 Architecture III: Database Caching in Addition to Replicating Static Data and Application Logic

Similar to the second architecture, every proxy has full fledged Web/application servers and the whole static contents and the complete application code. However, each proxy has also a database cache. The application sends its queries to the local database cache, rather than the origin server DBMS. In fact, to the application running in a proxy, the database cache appears as a database. If the cache can answer the query, it does so and the whole process of generating the response to the user is done without any need to go to the origin server. If the cache cannot answer the query, it communicates with the DBMS at the origin server to get the result of the query. Figure 3 shows how this setting works; we discuss it further in the next section.

3.4 A Closer Look at Architecture III

The first two architectures are straightforward and do not involve using any special caching algorithm or consistency protocol. The third architecture, however, is quite complex and needs to be described in more detail. Note that in this section, unless otherwise stated, the term *cache* is used to refer to a database cache at a proxy.

3.4.1 The Relationship between System Components

At each proxy, the Web and application servers act as if they have no knowledge of the origin server. That is because each proxy has the complete application logic, full-fledged Web/application servers, and the entire static contents.

The DBMS at the origin server and the database cache at each proxy, however, have full knowledge about the existence of each other. To guarantee strong consistency, on one hand, the origin DBMS should keep a considerable amount of state information about each database cache; on the other hand, each database cache has to communicate with the origin server to handle cache misses and to coordinate its actions with the origin DBMS.

3.4.2 What is Being Cached?

Caching only individual tuples is not a good idea because of the type of queries posed to a relational DBMS: *point queries* and *range queries* [26]. Range queries are quite common and account for a large percentage of the queries. A database cache that keeps only individual tuples cannot answer any range query, even if the result of the query turns out to be a single tuple. To solve this problem, the database cache should support *query-result caching* instead of, or in addition to, tuple caching. Although caching query results sounds like a trivial task, it involves many challenges, especially in the presence of strong consistency requirements. For example, consider a system in which several database caches store the results of their previous queries. The origin server and/or caches should be able to precisely determine which query results become invalid as the result of an insert or update. Keller and Basu [26] present an exhaustive study of the problems that arise in caching query results in a relational database. In summary, providing strong consistency automatically (without any hint from the database designer and with no change in the applications) is difficult. In our implementation, we assume that the database designer provides the system with a small number of hints that determine the query results that should be invalidated in response to each type of insert/update request. This can be done easily for any application in which the set of expected queries is finite and known. The hints (or rules) given to the system are of the following form:

if request to update tuple in table T with primary key k

then invalidate cached results of query type q whose parameters are $f_1(k)$, ..., $f_n(k)$

In other words, all the queries that the application can send are extracted and grouped into query types. Each query type is a parameterized SQL query. Then, for each possible insert/update statement in the application, a set of binary relations in the form of (query type, parameters) is determined. These binary relations correspond to queries that are affected by the insert/update.

An alternative approach is to use the *change notification*(also known as *query notification*) feature recently added to some commercial DBMSs[2]. Using this mechanism, an application (usually a mid-tier cache) can register queries with the database server and receive notifications whenever a future DML or DDL statement affects the result of any registered query. Note that the database server merely sends notifications to registered applications (caches) informing them that a certain query result has changed; it does not send the updated result to the caches nor does it block an update transaction until all caches have evicted the invalid query result. Therefore, the change notification mechanism, by itself, is not equivalent to Architecture III or its consistency protocol. In other words, one can use the change notification mechanism to implement a necessary part of Architecture III (namely determining cached query results affected by an update), but several other parts (especially the consistency protocol described below) need to be in place to provide strong consistency.

[2]To the best of our knowledge, this feature was not available in DB2, Microsoft SQL Server, and Oracle at the time we were building our prototype system.

3.4.3 How Is Cached Data Stored?

There are two options with regard to the storage of cached items: memory and disk (using files or a DBMS). We have chosen to store the cached items in a local database. This approach facilitates having a larger cache. Orthogonal to the choice of storage medium, there are two alternatives for caching the results of the queries that involve more than one table (join queries). First, they can be cached as one item (in one piece). Second, they can be split into tuples (or subtuples) of the tables involved in the query and stored in local (partial) copies of the original relations. We have chosen the second approach, because in that approach, each cache stores any tuple at most once, even if the tuple belongs to several cached query results (no duplication).

In summary, each database cache is implemented as a database in a local DBMS. For each relation in the origin server, there is a corresponding relation in the local database. These *cache relations* have all the columns of the corresponding origin server relation, plus a few columns for cache metadata (e.g., validity and reference count columns, which are discussed later). Caching individual tuples can easily be done by merely inserting the tuple into the corresponding table of the local database. Query result caching involves decomposing a query result into tuples of participating relations and inserting each tuple in the corresponding table. The details of both tuple and query result caching are presented in the following sections.

3.4.4 Tuple Caching

As mentioned above, caching individual tuples is useful for answering point queries. To provide strong consistency, the server should keep track of the caches holding copies of each tuple, so that it can send invalidation messages to those caches when the tuple is modified. When a cache receives a point query on a table, it checks the local (partial) copy of the table to see if

1. the tuple is already cached, and
2. the cached copy of the tuple is *valid* (according to the value of the "validity" column)

If both conditions hold, the cached copy of the tuple is returned to the user. Otherwise, the query is sent to the origin server and the result (which is a tuple) is cached. As mentioned above, the origin server keeps track of cached tuples. Therefore, if cache c_i sends a point query for tuple t_j, the server registers the fact that a copy of tuple t_j is cached in c_i. We call this *tuple caching registry* and implement it by adding an extra column to each table at the origin server, which stores a bitmap representing the proxies that have cached a copy of that tuple. The newly cached tuple remains *valid* until the server explicitly sends to the cache an invalidation message for the tuple. Note that if the query received by the database cache involves projection, the cache eliminates the projection before sending the query to the origin server so that a complete tuple is received.

3.4.5 Query Result Caching

To provide query result caching, every cache keeps an in-memory data structure called *Proxy-Side Cache Descriptor (PCD)*, which is a list of elements that stores information about

Query	Read Lock
Q1, (12, 7, "12/12/2004")	2
Q1, (15, 6, "04/07/2002")	0
Q2, (4, 17)	1

Query	Caching Proxies Bitmap
Q1, (12, 7, "12/12/2004")	01100001
Q1, (15, 6, "04/07/2002")	01000000
Q7, (23, 12)	10001000

Figure 4: sample cache descriptors: PCD (left) and SCD(right)

locally cached query results. Each entry in PCD is a *key-value* pair: a *key*, which is a binary relation in the form of *(query type*[3]*, parameter values)*, represents the cached query (see Section 3.4.2), and the corresponding *value* represents the read lock on that query (i.e., the number of active transactions that are using the cached query result). If the value is negative, it means that the corresponding query result in not valid. For efficiency, PCD is implemented as a hash table. Figure 4 depicts a PCD instance.

To provide strong consistency, the server must keep track of query results cached at each proxy. This is done through another in-memory data structure called *Server-Side Cache Descriptor (SCD)*. Each entry in SCD corresponds to a single query result cached at one or more proxies. Similar to PCD, a SCD entry is a *key-value* pair: *key*s are binary relations in the form of *(query type, parameter values)*, each representing a cached query, and the *value* corresponding to each key is a bitmap representing the proxies currently caching that query result. A sample SCS is shown in Figure 4.

When a cache receives a range query, it checks the PCD see if

1. the query result is already cached, and

2. the cached query result is *valid* (The value of "Read Lock" column is non-negative).

If both conditions hold, the query is executed in the local DBMS and the result is returned to the user.[4] Otherwise, the query is sent to the origin server, where the SCD is updated and the result is returned to the cache. At the proxy, the result is split up into base table tuples and put into the corresponding tables. In addition, a new entry is added to the PCD.

As in tuple caching, the cache may augment a query (i.e., eliminates projections) before sending it to the server. This is necessary because the cache needs to store complete tuples. If a database cache receives a query that involves more than one table (a join query), it eliminate projections and sends the query to the origin server. After receiving the query result from the origin server, the cache looks at the schemas of the tables participating in the join and decomposes the result into tuples of different tables. Note that, as mentioned before, the schema of each cache table is identical to the schema of its corresponding table at the origin server; therefore, all necessary metadata for result decomposition is readily available at any database cache. After decomposing, a simple duplicate elimination operation is applied to each group of decomposed tuples, and the result is inserted into the corresponding cache table. The SCD, PCD, and the consistency protocol guarantee that executing the same

[3] Two queries are of the same type, iff everything in query texts except for parameter values are similar.

[4] As we elaborate in the following sections, while the query is being executed, the corresponding read lock in PCD is incremented.

query on the cache database produces the same join query result (as long as the cached query result is deemed valid by PCD).

Note that in this paper, we do not use partial query results; that is, our database cache can answer a query using the results of a previous query only if the two queries match exactly. Using query containment techniques to exploit partial results can be the subject of future work. Our experimental results show that, although we consider only exact matches, the cache hit ratio is high. This high cache hit ratio can be attributed to the high locality of reference in the queries.

3.4.6 The Consistency Protocol

In this section, we give a detailed explanation of the consistency protocol used in our implementation of Architecture III. First, we give a brief overview of the system components involved in consistency control. Then, the protocol is illustrated by considering all possible scenarios and specifying the server-side and proxy-side actions that are performed in each scenario.

System Components

To guarantee strong consistency, the server and proxies exploit several components of the system; namely the SCD, PCDs, tuple caching registry, and locking systems at the server and proxy DBMSs. We have already introduced SCD, PCD, and tuple caching registry in the previous sections. The locking systems at the server and at database caches, however, deserve a more detailed explanation.

In our prototype implementation, we have implemented a *tuple-level* locking system at the server, that is we lock at the granularity of tuples. As elaborated in the following sections, our consistency protocol requires locking and unlocking individual tuples at the server, but we cannot obtain such a fine grained control on the locking subsystem of the origin server DBMS (which is a non-open source, commercial DBMS). Therefore, our locking system works as an in-memory structure on top of the locking system of the DBMS. However, in a real-world implementation of the consistency protocol, it would be desirable to merge the two locking systems to increase efficiency.

In contrast to server-side locking, the proxy-side part of our protocol does not need to maintain fine-grained control over locking; therefore, we use the locking scheme of the local DBMS. In fact, the transaction isolation guarantees given by the local database are sufficient for our purpose. For example, when a transaction is being executed at a database cache, none of the tuples it reads can be updated until the transaction is committed or aborted. The local DBMS concurrency control takes care of this, so there is no need to have fine-grained control over the details of locking or have an additional locking scheme on top of the existing one.

In addition to the above-mentioned components, the server keeps some state information for each active transaction. The state information includes the list of locked tuples, the list of pending updates, and the list of invalidation messages sent as the result of update statements in the transaction. The following sections illustrate how this information is used.

The prototype database cache that we have implemented is DBMS- independent; the server-side and proxy-side components use JDBC to interact with the origin and proxy-side DBMSs respectively. As a result, the DBMSs at the origin server and proxies do not need

to be from the same vendor. Nevertheless, we believe that implementing the server-side component inside the DBMS engine of the origin DBMS and integrating the locks needed by the caching system with the traditional DBMS locks can boost the system performance. The down side of the latter approach is that it may make the caching module dependent on a specific DBMS. However, if different DBMS engines implement the protocol in the same way and use some standard for server-proxy communication, both optimum performance and interoperability across different systems can be achieved.

SELECT Queries

When the application server at a proxy sends a SELECT query to its local database cache, depending on the type of query, different actions are taken.

- *Point Query*

  1. The cache executes the query. If the result is a tuple marked as valid, the cached tuple is sent back to the application; no other action is required. Otherwise, the cache eliminates any projections and sends the modified query to the origin DBMS server and the following steps are performed.
  2. The origin server executes the query and locks the resulting tuple in read-mode. If the tuple cannot be locked (due to a lock-request time-out), the server sends a negative response to the proxy and the transaction is aborted.
  3. The tuple is registered in tuple caching registry at the origin server.
  4. The tuple is sent back to the requesting proxy.
  5. The proxy caches the tuple and marks it as valid.

- *Range Query*

  1. The cache checks its PCD to see if the results of the given query are already cached.
  2. If so, it locks the corresponding entry in the PCD (increments the read lock), executes the query on the local database data, and returns the result to the application. The cache needs to execute the query because it does not cache query results in one piece, but decomposes them into individual tuples (see Section 3.4.3); therefore, has to run the query later to compose the result. Also note that queries always lock PCD entries only for reading, so several transactions can obtain locks on the same PCD entry.
  3. Otherwise, the cache eliminates any projections and sends the modified query to the origin DBMS server.
  4. The origin server executes the query and locks all of the tuples that are part of the result in read-mode. If any of the necessary locks cannot be obtained (due to a lock-request time-out) the server sends a negative response to the proxy and the transaction is aborted
  5. SCD is updated to reflect the fact that the result of the given query is cached at the requesting proxy.

6. The query result is sent back to the requesting proxy.
7. The proxy splits the result into tuples of the tables participating in the query and inserts them in the corresponding tables of the local database. If a tuple already exists in the local database, its *reference count* is incremented. Otherwise, the reference count is set to 1. As we discuss later, tuple reference counts are needed for cache replacement.
8. The proxy updates its PCD; i.e., it adds the newly cached query and also locks it in read-mode.

UPDATE/INSERT Queries

When the application server at a proxy sends an UPDATE or INSERT query to its local database cache, the following events are triggered.

1. The cache invalidates (removes) all the PCD entries corresponding to the query results that are (potentially) affected by the update.
2. The proxy sends the update to the origin server asynchronously; that is, it does not wait for the response from the server to go to the next step.
3. The update is applied to the local database. Note that the local update is not committed.
4. At the server, the tuples to be updated are locked in write-mode and a positive response is sent back to the proxy. If the request for locking is timed out (e.g., due to a deadlock), the corresponding transaction should be aborted, so a negative response is sent back to the proxy.
5. The server adds the update statement to the list of *pending updates*.[5]
6. The proxy continues processing the current transaction if it receives a positive response. Otherwise, it aborts the current transaction.
7. The server looks up the tuple-caching registry and SCD to find all the tuples and query results affected by the update.
8. The server sends *invalidation messages* to all the proxies found in the previous step asynchronously.

As can be seen from the above explanation, the server optimistically sends a positive response to an update query as soon as it can lock the tuples affected by the update. It does not defer sending the positive response until it gets acknowledgement of invalidation messages. This check is postponed until the commit time.

[5]The server keeps a list of pending updates for each transaction. This list holds the update statements issued by the transaction until the commit point, at which time those updates are executed.

Committing a Transaction

At transaction commit point, the proxy executing the transaction sends a *transaction commit request* to the origin server.

The following actions are performed at the server.

1. If any invalidation message was sent to other proxies as the result of an update statement in the committing transaction, the responses are checked. If any of the responses has not yet been received, the server waits until it receives the missing response.
   - If there is a negative response, the transaction is aborted and all its state information is deleted. Then, a negative response is sent back to the proxy in response to its commit request.
   - Otherwise, the tuple caching registry and SCD are updated to reflect the changes. For example, if a proxy was requested to invalidate a certain query result and it has done so and sent a positive response, the server updates SCD by deleting the proxy from the list of proxies which cache that query result.
2. If the transaction is not aborted in the previous step, all (pending) updates of the transaction are performed and a positive response is sent back to the proxy's commit request.
3. All the server-side locks obtained by the transaction are released.

After the database cache receives the response to its commit request, it either aborts or commits the transaction based on the response. Note that in both cases, the cache releases the locks on the PCD entries locked by the transaction. In addition, it either commits or aborts all uncommitted queries that the transaction has issued to the local database. This implicitly releases all the locks obtained by the transaction on local tuples.

Aborting a Transaction

A transaction is aborted in several cases:

- When the application explicitly aborts the transaction;
- When the server cannot grant a lock request and the request times out (e.g., due to a deadlock);
- When the server sends a negative answer to a commit request. This happens when the server receives at least one negative response for the invalidation messages it has sent to other proxies.

In any of the above scenarios, the transaction should be aborted both at the server and at the cache in which the transaction is being executed.

At the proxy side, the PCD entries locked by the transaction are released. In addition, all uncommitted queries of the transaction in the local database are aborted.

At the server side, the state information of the transaction is deleted and all the locks it has obtained are released.

Note that, except for the case where the application explicitly aborts the transaction, the process of aborting is initiated by the server; therefore, the proxy does not need to send an abort message to the server.

Special Case: Single-Statement Transactions

Single-statement transactions (i.e., a transaction containing only one SQL query) are treated as a special case. First, there is no need to issue an explicit commit request for a single-statement transaction. Second, the server does not need to store explicit state information for them. If a cache needs to send such a query to the origin server (e.g., because of a cache miss), the origin server executes the query and immediately tries to commit it.

Invalidation Message Processing

A cache receives *invalidation messages* from the origin server. Each invalidation message contains one or more *invalidation requests*. An invalidation request is a request to invalidate either a specific tuple or cached query result. Upon receiving an invalidation message, the cache extracts all invalidation requests and applies them. To invalidate a tuple, it is enough to mark the tuple as invalid. Invalidating a query result is done through deleting its corresponding entry from PCD and decreasing the reference count of all the tuples in the query result.

A tuple invalidation may fail if the tuple is currently locked by an active transaction. If the conflicting transaction is not committed or aborted within a certain period of time, the invalidation times out and a negative invalidation response is sent back to the server. A query result invalidation may fail as well. This happens when the PCD entry is locked for reading. If the locks on the entry are not released within a time-out period, a negative response is sent back to the server.

SCD-PCD Consistency

It should be noted that both the SCD and PCDs keep track of the query results cached in proxies. However, the SCD at the server and the PCD of a proxy P_i do not need to be strictly consistent in their description of the query results cached in P_i.

Neither SCD nor PCD needs to keep *exact* descriptions of the cached query results [26]. A PCD should be either exact or *conservative*. By being conservative we mean that PCD is allowed to falsely assume that it does not have the results of a query, but can never falsely assume that it has cached a query result. Being conservative may result in inefficiency (by causing false cache misses), but does not harm correctness. On the other hand, the SCD can be *liberal* in its description of what is cached in proxies. In other words, it may assume that a query result is cached in a proxy, whereas in fact it is not. However, it is not allowed to miss any cached query in any proxy. A liberal description of the contents of caches may cause unnecessary invalidation messages to be sent. Similar to the previous case, this can harm efficiency, but not correctness [26]. This explains why the temporary inconsistencies between the SCD and PCDs do not cause a problem.

Cache Replacement

Since we store the cache contents in a database, rather than memory, cache replacement is not a critical issue. However, invalidations cause a great deal of unusable data (invalid tuples). To reclaim the unused space, we can evict every tuple whose *reference count* reaches zero.

Deadlocks

Both getting locks at the server and invalidating at the proxies may result in deadlocks. To deal with deadlocks, every *wait* operation has a time-out value. If a request times out, its corresponding transaction is aborted. We chose this method of deadlock handling to avoid the overhead of deadlock avoidance/prevention algorithms.

4 Evaluation

In this section, we evaluate the performance of each of the three architectures using the TPC-W benchmark. TPC Benchmark W (TPC-W) [35] is a transactional Web benchmark that simulates the activities of the Web site of an online store. According to the benchmark specification, any implementation of TPC-W must include features such as browsing, various types of search, shopping carts, and a payment gateway emulator (PGE), which represents an external system that authorizes the payment of funds. The main purpose of this benchmark is to have a standard workload that is representative of the workload of real-world e-commerce Web sites and can be used to measure and compare the performance of different components of an online business system.

The TPC-W specification defines fourteen *Web interactions* that represent the typical services provided by e-commerce sites. Examples of these interactions include *Customer Registration, Search Results, Product Detail, Best Sellers, Buy Request,* and *Buy Confirm.* More than 90% of these Web interactions involve generating dynamic pages using the data stored in a back-end database. The benchmark has determined the schema of the database. The database consists of eight database tables: *Customer, Country, Address, Orders, Order_Line, Author, CC_Xacts (which stores transactions),* and *Item.* According to the benchmark, the size of the bookstore (the size of the *Item* table) is chosen from a given set of *scale factors*; that is, it can range from 1000 to 10,000,000 book items in tenfold increments.

The fourteen Web interactions defined in the benchmark impose different loads on the Web and database servers. Some interactions involve only reading the database data, whereas others both read and update the data. The benchmark defines two types of interactions: *browsing* and *ordering.* Browsing interactions include read-only interactions , whereas ordering interactions include interactions that update the underlying data. Since different businesses have different types of customers, the benchmark defines three *mixes*, based on the ratio of browsing and ordering interactions: *browsing mix, shopping mix,* and *ordering mix.* The browsing mix, which has a 95%-5% ratio between the browsing and ordering interactions, models users that usually browse the items and seldom buy anything. In the shopping mix the ratio between the browsing and ordering interactions is 80%-20%, which represents a typical users shopping activity. Finally, the ordering mix emulates a B2B type of workload and has a 50%-50% ratio between the two types of interactions.

To drive the TPC-W workload, a component called *Remote Browser Emulator (RBE)* simulates the users of the online store. The RBE manages a collection of site users, each of which is called an *emulated browser (EB).* Each EB sends an HTTP request and waits for the response from the server. Based on the response received, it sends another request and this process continues until the *session* ends, at which time the EB starts a new session. Each EB session represents the activities of a single user in a single shopping session. TPC-W specifies a minimum duration for each session, which is a randomly selected number from a

negative exponential distribution.

There are several implementations of the TPC-W benchmark. We use the University of Wisconsin-Madison implementation [18]. We modified this implementation slightly to simulate network latency. For Architecture III, we made additional modifications so that our database cache is used. For simplicity, we assume that the network latency between clients/proxies and the origin server is a constant value. However, in reality, the network latency between two hosts is not a constant value.

4.1 Software Components

We use Jakarta Tomcat version 5 [13] as our Web/application server and IBM DB2 Universal Database version 8.1 (Enterprise Edition) [16] as the database server. In architecture III, we have two more software components, namely database cache and server-side consistency controller. Each proxy hosts an instance of the database cache. In our implementation, the database cache consists of two software components. The first is a Java program that intercepts the database queries of the Web/application server and implements the proxy-side part of the consistency control. The second component is a full-fledged DBMS (IBM DB2 Universal Database version 8.1) used by the former component to store cached tuples and also run the queries that can be answered locally.

The server-side part of the consistency protocol is implemented by another Java application, which resides at the origin server and is called *server-side consistency controller*. In a real-world implementation, the server-side consistency controller should not be a stand-alone component, but part of the consistency control mechanism of the origin DBMS. However, since we are using a non-open source, commercial DBMS, we had to implement a separate Java program that resides in the origin server and is responsible for consistency control. This limitation made us implement a locking system on top of the DB2 locking, leading to inefficiency and duplication. Thus, the results we report for Architecture III are more conservative than what can be achieved in reality.

4.2 Hardware Configuration

We use the following hardware configuration in our experiments.

- *Origin Database Server:* We run the origin database server (IBM DB2) on a PC with Pentium IV 1.5GHz CPU and 1GB RAM.

- *Origin Web Server (Architecture I only):* We run the origin Web server (Tomcat) on a PC with Pentium III 997MHz CPU and 512MB RAM.

- *Proxy Servers (Architectures II and III only):* We have two proxy servers, each on a PC with Pentium IV CPU (one 1.5GHz and the other 1.6GHz) and 512MB RAM. A Web server (Tomcat), application servlets, and all the static contents (image files) reside on each proxy. In addition, in architecture III, a database cache (including an IBM DB2 DBMS and the Java program mentioned in Section 4.1)runs at each proxy.

- *Client (RBE):* We use two Sun Solaris machines, each running an instance of RBE. Each RBE emulates half of the EBs required for the experiment. For example, to run

parameter	values
scale factor	10,000; 100,000
number of EBs	20, 50, 100, 150
Web interaction mix	browsing, shopping, ordering
network latency	0ms , 350ms, 700ms, 1,000ms

Table 1: The Parameters of the Experiments and Their Values

an experiment with 100 EBs, we run 50 EBs on each client to make sure that the client machine resources are not system bottlenecks.

- *Network:* A 100 Mbps LAN is used to connect all the hosts. We used network monitoring tools to make sure that network resources (especially network bandwidth) do not create bottlenecks.

All of the machines, except the machines that execute RBEs, run on Microsoft Windows XP operating system.

4.3 System Parameters

Table 1 presents a summary of the system parameters used in our experiments. In this table, the default value for each parameter is underlined.

There are a few points regarding the choice of parameters that need to be explained.

- According to TPC-W, the size of the bookstore (i.e., the number of books) must be chosen from the following set of values: 1,000; 10,000; 100,000; 1,000,000; 10,000,000. For our experiments, we have chosen two scale factors: 10,000 and 100,000. Due to hardware and operating system limitations, we cannot use the large scale factors, namely 1,000,000 and 10,000,000.

- The number of EBs determines the amount of load on the origin server and proxies. There is no theoretical bound on the number of EBs; however, the servers are saturated after the load exceeds a certain threshold, and adding further load beyond saturation makes the behavior of the system unpredictable. Our results show that in all of the architectures, running 150 EBs saturates the Web server; therefore, there is no point in running experiments with higher values.

- To evaluate the effect of network latency on the performance of each architecture, we use the following values as the round-trip network latency between the clients (and/or proxies) and the origin server: 0ms, 350ms, 700ms, 1,000ms. The first value (0ms) represents the situations in which the network latency is negligible (e.g., a LAN). The next two values (350ms and 700ms) represent the network latency for a WAN; especially 700ms is close to the network latency of intercontinental communications or satellite links. The last value (1,000ms) is used for investigating the performance of each architecture in the presence of extremely long network delays to stress-test the alternatives. We use 700ms as the default value for network latency, because as

mentioned in Chapter 1, our primary motive for using a database cache (which has a high consistency overhead) is to improve the performance in situations where there is a high network latency between the origin server and clients.

Note that we assume that the network latency between the clients and the proxies is negligible. This is a realistic assumption, because in our research, we are considering forward proxies, which are most often placed close to the end-users.

4.4 Performance Evaluation Measures

To compare the three architectures considered in this research and investigate the effect of changing different system parameters, we use the following measures: Web Interaction Response Time (WIRT), Web-Interaction-per-Second (WIPS), Cache Hit Rate, and Caching-Related Transaction Abortion Rate (The two latter measures are applicable only to Architecture III).

4.4.1 Web Interaction Response Time (WIRT)

According to the TPC-W specification, WIRT measures the interval between the time that the first HTTP request of a Web interaction is sent from the EB to the Web server and the time that the last response completing the interaction is received by the EB. In our research, the average WIRT is the most important measure, because our main objective is to find the best architecture in terms of the user experience.

Response time is mainly determined by the network latency and the server latency. Network latency is affected by the physical distance between the two communicating nodes and also by the network traffic. Server latency (i.e., the time needed for the server to process a user request) is determined by the hardware/software configuration of various servers involved (Web, application, and database servers), as well as the load on the server (i.e., number of clients). Increasing the number of clients increases the server latency, because user requests have to spend more time in various operating system and application queues.

4.4.2 Web-Interaction-per-Second (WIPS)

Web Interaction per Second (WIPS) measures the system throughput. Ideally, throughput increases with the number of clients until one of the servers (Web, database, or application server) becomes saturated, at which point the throughput remains fixed at its maximum. In practice, however, once the server is saturated, increasing the number of clients degrades throughput, because the saturated server spends a considerable amount of time dealing with the excess load.

4.4.3 Cache Hit Rate

This measure is applicable only to architecture III, and we use it to compare the effect of parameters such as the number of EBs or the type of workload (TPC-W mix), on the database cache. As mentioned in Section 3.4.2, we differentiate between two types of queries (namely, point queries and range queries). We measure the cache hit rate for these two query types separately, so that we can analyze the effectiveness of caching for each type of query.

4.4.4 Caching-Related Transaction Abortion Rate

In every database application, some transactions may be forced to abort because of problems such as deadlocks or time-outs. However, in Architecture III, our consistency protocol causes additional types of transaction aborts. For example, when a database cache wants to execute an update, it sends a request to the origin server, and the origin server sends invalidation messages to all the caches that may be affected by the update. If any of these caches fail to acknowledge the invalidation within a timeout period, the server aborts the updating transaction. We measure the rate of this type of abort for two purposes. First, it can give us an estimation of the cost of database caching in terms of additional aborts. Second, it helps us understand the effect of different parameters (such as the type of workload) on the behavior of our caching system.

4.5 Experiments and Results

In this section, we present the results of our experiments. Note that since we are interested in comparing the relative performance of three architectures, the real numbers obtained for average response time and system throughput are not important. Therefore, we normalize the results and report them relative to the performance of the baseline case, namely Architecture I. However, since the performance of each architecture with respect to the particular performance parameter is of interest, for each experiment, we show the relative values with respect to the lowest value obtained for Architecture I or with respect to the value corresponding to the performance of Architecture I with the default value of the parameter. For example, in Figure 5, all the values are normalized with respect to the performance of Architecture I for 20 EBs. Therefore, in addition to showing the relative performance of different architectures, this figures shows the effect of increasing the number of EBs on the performance of each architecture.

4.5.1 Web Interaction Response Time (WIRT)

We have conducted a set of experiments using different values for system parameters to measure the average response time, over all interactions, for each architecture. Figure 5 shows the effect of client population (number of EBs) on response time. In general, as the number of clients increase, the average response time becomes longer. Our results show that Architecture III is more robust as the load increases, because in this architectures the load is distributed among the server and a number of proxies (in our experiments, two proxies), whereas in Architecture I all of the load is on the origin server. For example, with 20–50 EBs, the response times of Architectures I and III are almost equal; but with 150 EBs, the response time of Architecture III is considerably better than that of Architecture I.

As depicted in Figure 5, the average response time in Architecture II is worse than that in other architectures. In our experiments, Architecture II became unstable (dropping most of the requests) under heavy load (e.g., 150 EBs). Because of this, the WIRT of Architecture II for 150 EBs is not shown in Figure 5. Later we explain why, despite the positive effect of load balancing, Architecture II has a long response time.

Figure 6 depicts how network latency affects the average response time in each of the architectures. As we expected, for long network delays, Architecture III performs better

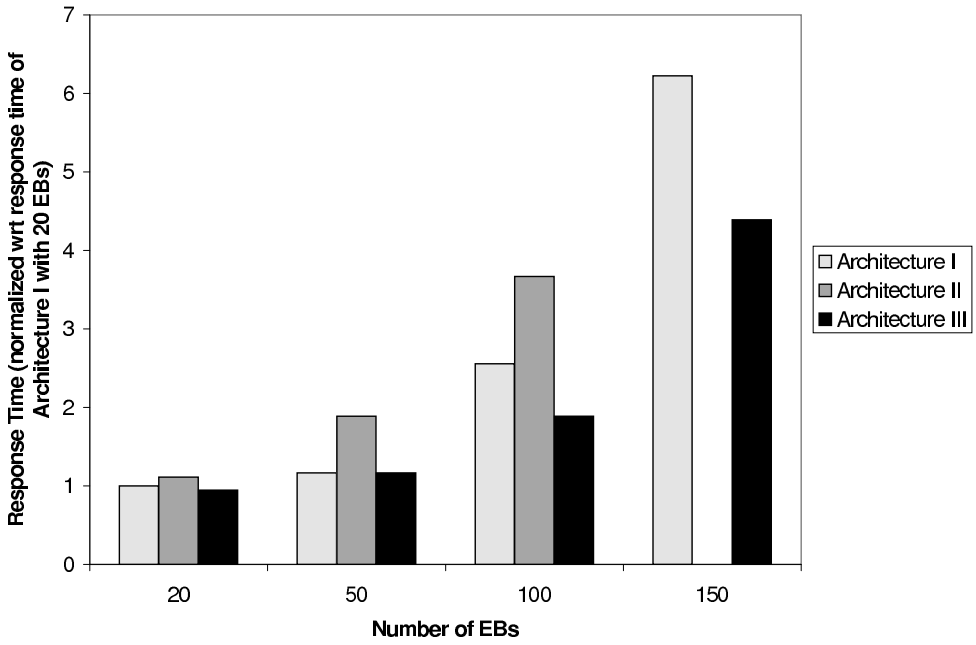

Figure 5: Average Response Time vs. Number of Clients

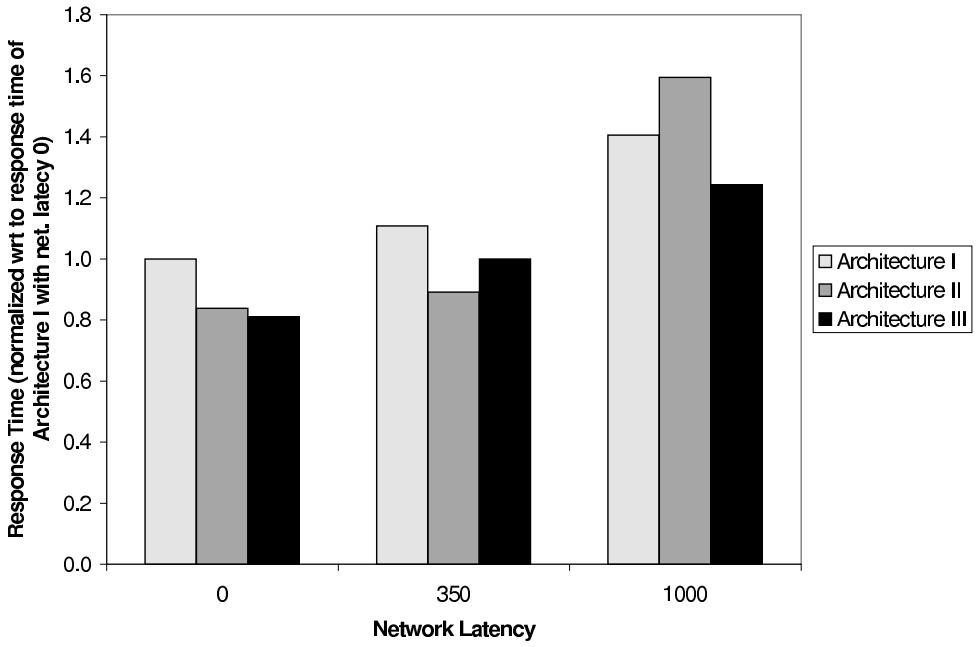

Figure 6: Average Response Time vs. Network Latency

than architecture I. However, interestingly, this holds even when the network latency is 0 ms. We believe this is due to the positive effect of load distribution in Architecture III.

The above results show that, in Architecture II, despite the positive effects of serving static items locally and distributing the load on proxies, the average response time is considerably longer than the response time in the other two architectures. We believe that the

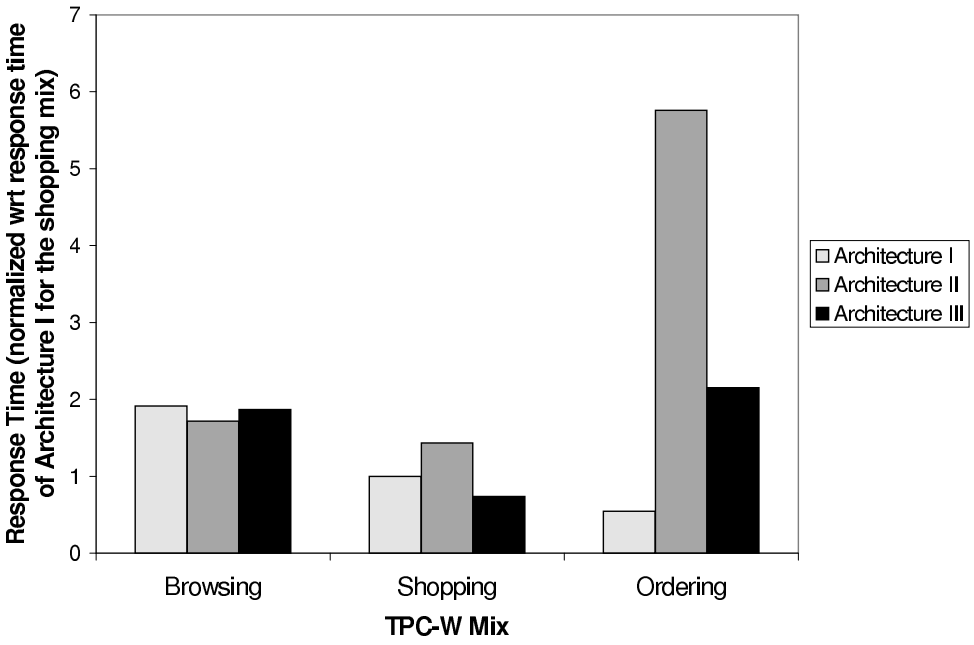

Figure 7: Average Response Time vs. TPC-W Mix

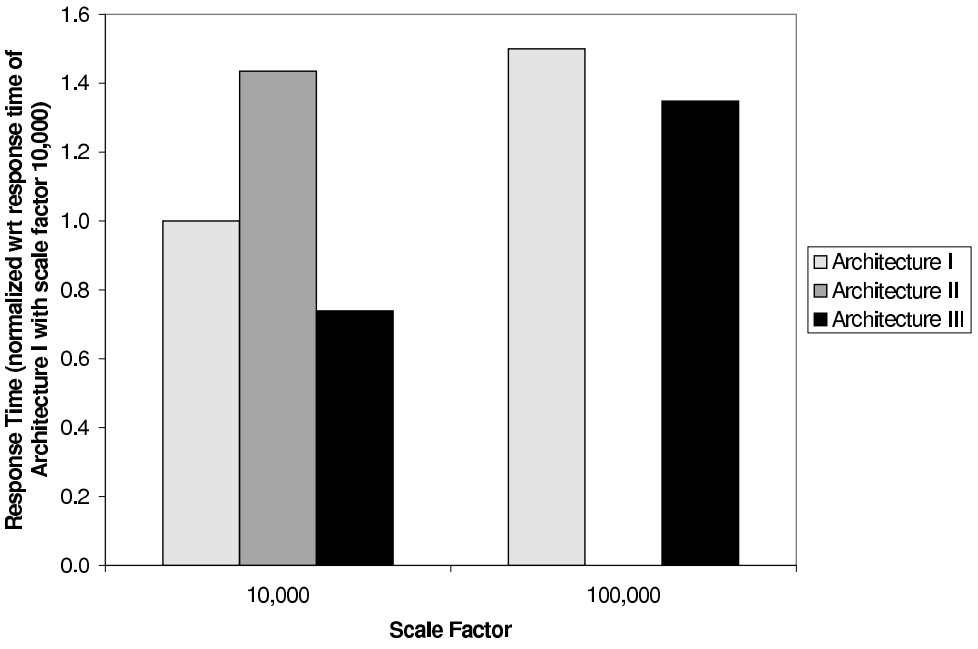

Figure 8: Average Response Time vs. TPC-W Scale Factor

reason for the poor average response time of Architecture II is the following. In database-backed dynamic Web applications, creating one dynamic Web page may involve sending multiple queries to the database; therefore, in Architecture II, the application server at a proxy has to send a query to the origin server and wait for its response, then do some computation and send another query to the origin server, and repeat this cycle a few times before a complete HTML page can be constructed. Considering the network latency between the

proxy and the origin DBMS, the process of creating a Web page takes a long time in this architecture. Note that the application server cannot send all the queries to the origin DBMS at once (as a batch), because the parameters of each query are computed using the results of the previous queries. This problem can be solved by moving some parts of the application logic into the database; however, this solution prohibits transparent caching (caching without or with minimal changes to the application code). It is worth mentioning that the *query-per-page ratio* (i.e., the number of queries necessary for generating a page) is different for different pages. A higher average query-per-page ratio, as well as longer network latency between the origin server and proxies, exacerbates the latency problem of Architecture II.

Note that Architecture III does not suffer a similar response latency problem, because many of the queries can be answered using the local cache (without going to the origin server).

The effect of the type of workload (TPC-W mix) on the response time is shown in Figure 7. Browsing the bookstore site (browsing mix) mostly results in issuing one or two queries per generated page. These queries are generally read-only range queries. The low query-per-page ratio hides the latency problem of Architecture II, therefore its response time is even slightly better than that of the other two architectures. On the other hand, Architecture I shows a very long response time (compared to its response time for other mixes), because of the higher cost of evaluating range queries. On the whole, all of the three Architectures show similar response times in the browsing mix, with Architecture II being slightly better than the other two.

As mentioned before, in the ordering mix, 50% of the interactions are ordering interactions. These interactions usually involve issuing several database queries, many of which are update queries. The high query-per-page ratio makes the response time of Architecture II extremely long. On the other hand, the high rate of update queries degrades the performance of Architecture III (because of the increased cost of providing consistency). Therefore, in the ordering mix, Architecture I shows the best response time by far.

In the shopping mix, Architecture III performs the best. The existence of proxies decreases the network latency (in case of cache hits) and also decreases the load on the origin server. Since the update queries are not dominant, the overhead of strong consistency protocol is compensated for by the above-mentioned savings.

To investigate the effect of scale factor (database size) on the response time of each architecture, we have conducted a set of experiments whose results are depicted in Figure 8. These results show that Architecture III outperforms Architecture I at both scale factors, which verifies that Architecture III is scalable in terms of database size. Note that in our experiments, Architecture II becomes unstable with 100,000 items. As shown later, the throughput of Architecture II in that case is nearly 0, which means that most of the user requests are dropped.

In summary, the results show that architecture II does not yield good response times, unless the majority of pages have a low query-per-page ratio. Architecture III provides the best response time, except for update-intensive applications, in which case Architecture I has the shortest response time. Note that in our prototype implementation, we did not have access to the code of the commercial DBMS that we used as our origin DBMS, so we had to implement our server-side part of the consistency protocol as a stand-alone application. We believe that a more efficient implementation (in which the consistency protocol is incorpo-

rated into the DBMS consistency control subsystem) may considerably improve the response time of Architecture III in update-intensive applications.

4.5.2 System Throughput (WIPS)

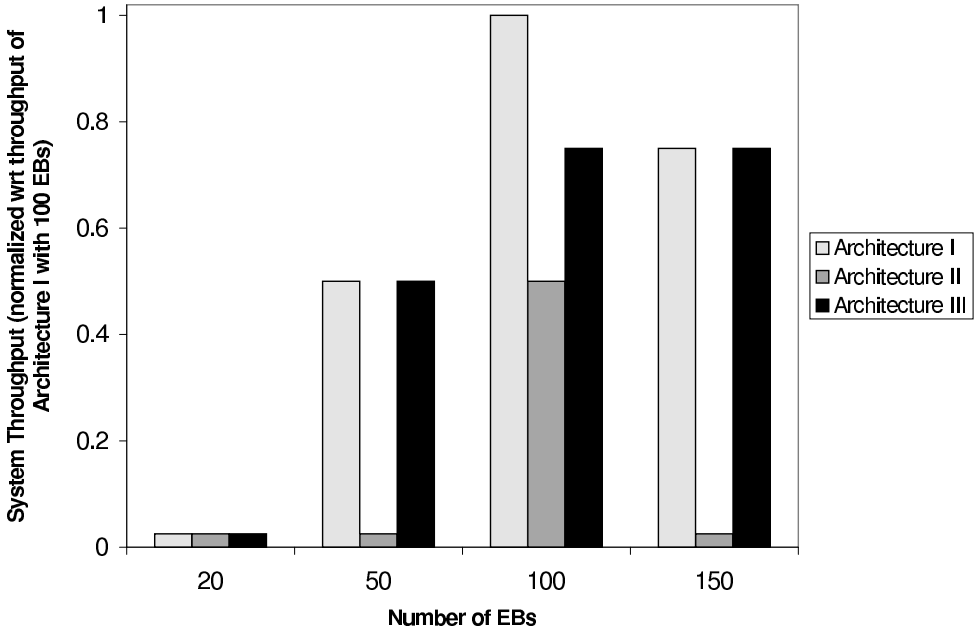

Figure 9: System Throughput vs. Number of Clients

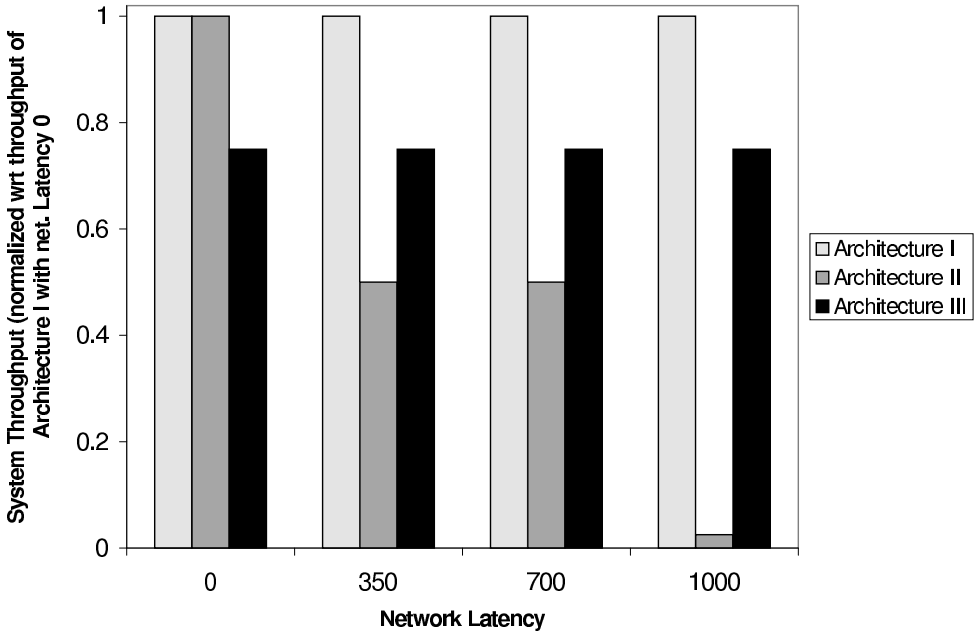

Figure 10: System Throughput vs. Network Latency

Figure 9 depicts the effect of the number of clients on the system throughput. As mentioned before, throughput increases with increasing number of clients until the system is saturated, at which point it begins to degrade. Our results show that Architecture I has the highest average throughput.Note that in this set of experiments the network latency is fixed at its default value (700ms), which is a rather high latency. Later, we explain how this long latency has a negative effect on system throughput in Architecture II. Architecture III performs as well as Architecture I with small number of clients. However, increasing the number of clients adds to the (computation) overhead of the consistency protocol and this results in faster server saturation. Part of this overhead might have been avoided if we could have implemented our protocol inside the DBMS, rather than on top of it.

Figure 10 shows the results of the experiments in which we change the network latency between the client (or proxies) and the server, while fixing all other system parameters at their default values. The results show that in Architectures I and III, network latency does not have a visible effect on throughput. This observation verifies the fact that, in general, throughput depends on the amount of necessary computation and I/O (as well as the processing power of the machines), rather than the latency between the producer and the consumer. However, in Architecture II, the network latency has a strong effect on the throughput. Note that as shown in Section 4.5.1, high network latencies result in extremely long response times in Architecture II. In this situation, some HTTP requests may time out before the response becomes ready, resulting in poor system throughput.

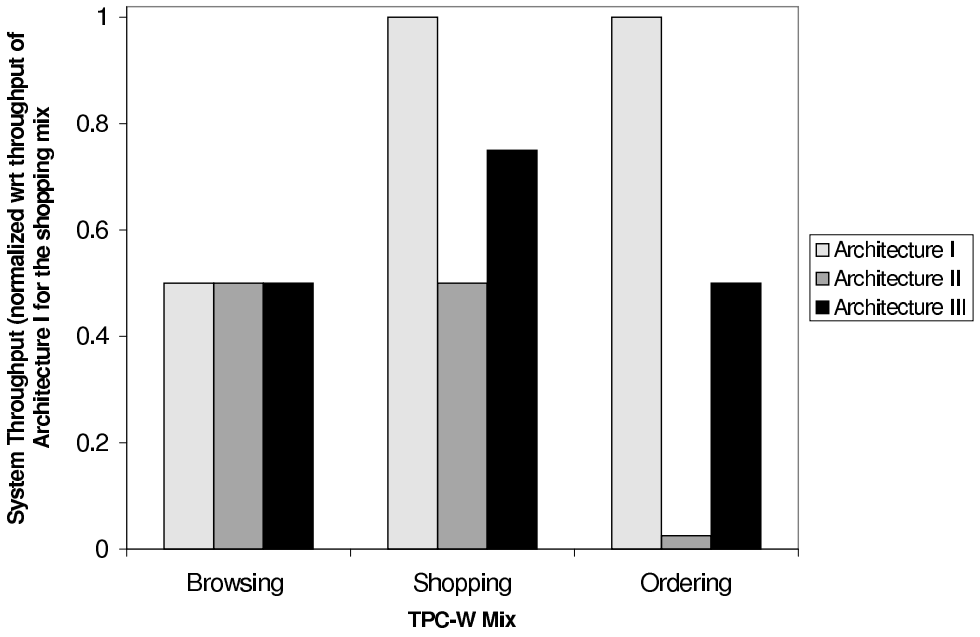

Figure 11: System Throughput vs. TPC-W Mix

System throughput for different TPC-W mixes is depicted in Figure 11. Generally, the throughput in browsing mix is lower than that in other mixes, because the evaluation of range queries, which are frequent in this mix, is generally more complex compared to that of point queries.

The extremely low throughput of Architecture II in the ordering mix is because of the high query-per-page ratio of ordering interactions. In a typical ordering interaction, several

queries update the system data to reflect user's purchase. As for Architecture III, the low throughput in the ordering mix can be justified by the higher rate of aborted transactions and HTTP timeouts (due to long response times).

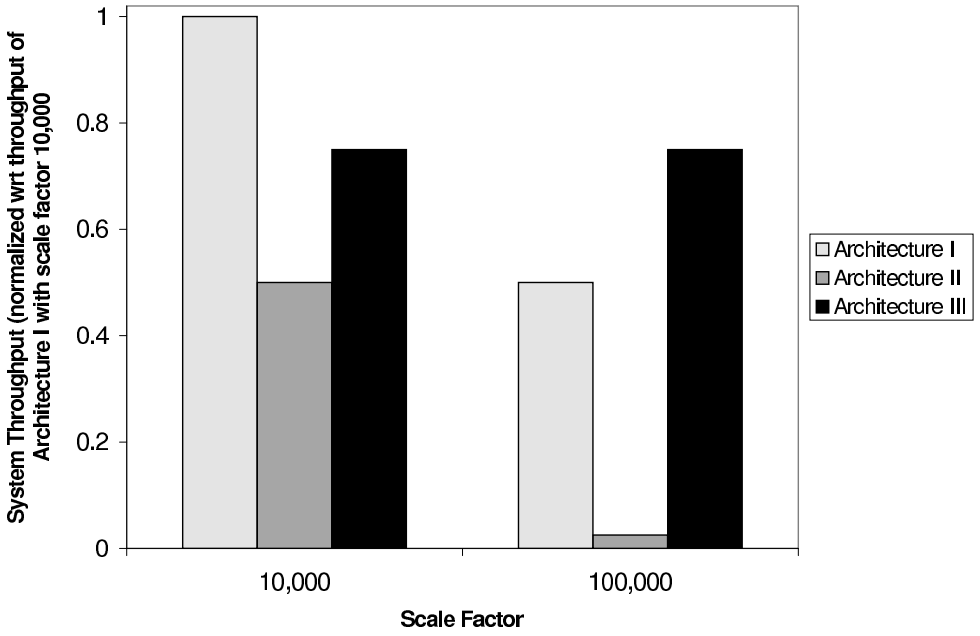

Figure 12: System Throughput vs. TPC-W Scale Factor

Figure 12 shows the effect of database size on system throughput in each architecture. As mentioned earlier, increasing the database size renders Architecture II unstable; the system cannot complete most of the interactions and the throughput is almost zero. Architecture III proves to be more scalable than Architecture I in terms of database size. Having a larger database adds some overhead in both Architectures I and III. However, we believe that since, with a fixed number of users, increasing the number of book items decreases the probability of contention for getting exclusive access to each item, the overhead of consistency protocol of Architecture III is reduced; hence, its overall throughput is not degraded.

4.5.3 Cache Hit Rate

Figure 13 depicts the cache hit rate[6] for experiments with different number of clients. The cache hit rate for point queries does not show a significant dependence on the number of clients. This is because many of the point queries in the bookstore application are used for retrieving user information (e.g., first/last name, and address); when a user first logs into the system, the first few point queries force the database cache to fetch this information from the origin server and store it; therefore, the subsequent point queries issued for the same user can be answered locally. Since we use a local database as our cache and, hence, are not limited by memory, having more clients does not cause cache evictions and therefore, does not affect cache hit rate.

[6]Note that this performance measure applies only to Architecture III.

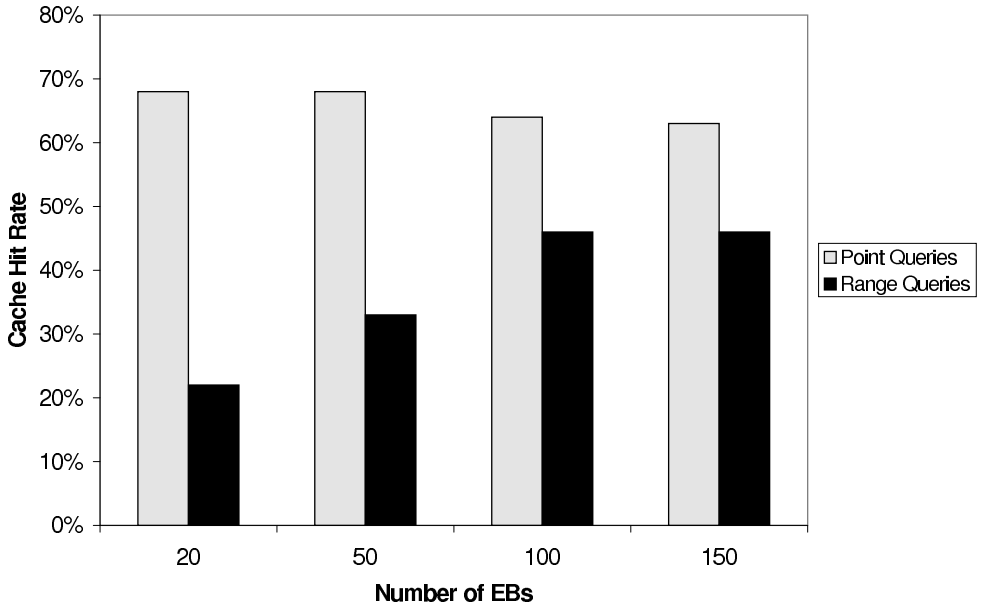

Figure 13: Cache Hit Rate vs. Number of Clients

On the other hand, the cache hit rate for range queries has a direct correlation with the number of clients. This happens because the bookstore application has a large number of different range queries, each of which has several parameters. Therefore, there are a large number of possible query results and the more clients use the database cache, the more query results become cached and hence, the number of cache misses is decreased.

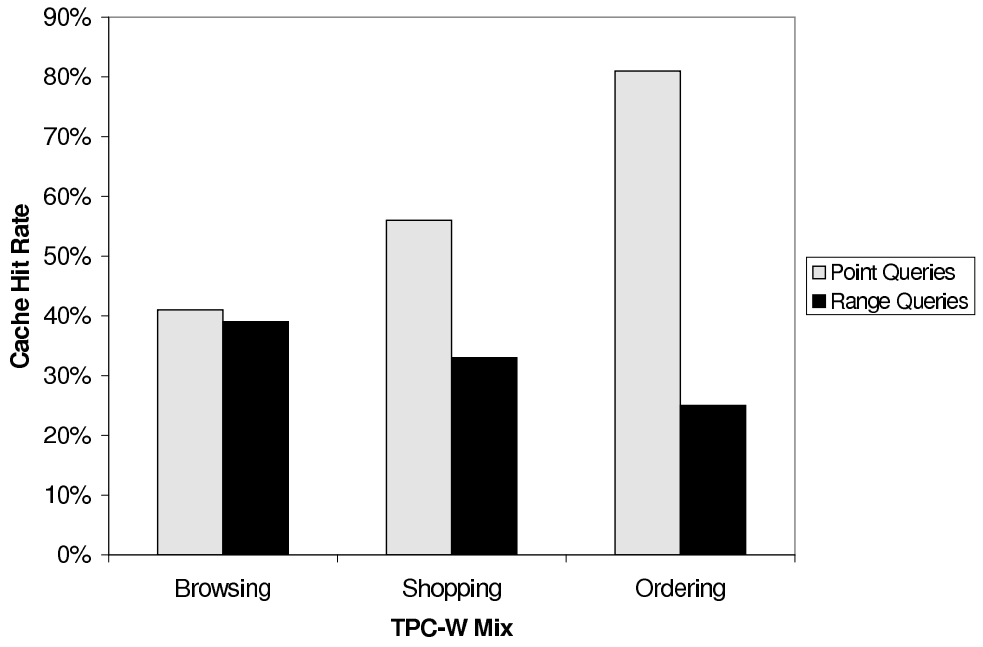

Figure 14: Cache Hit Rate vs. TPC-W Mix

Generally speaking, in TPC-W, browsing activities (e.g., various searches) result in the

issuing of range queries, whereas shopping activities (e.g., adding an item to shopping cart, or checking out items) create more point queries. The results shown in Figure 14 verify this argument. The highest cache hit rate of range queries belongs to the browsing mix, which has the highest ratio of browsing interactions. Similarly, the cache hit rate of point queries is maximum in the ordering mix, in which 50% of the interactions are ordering.

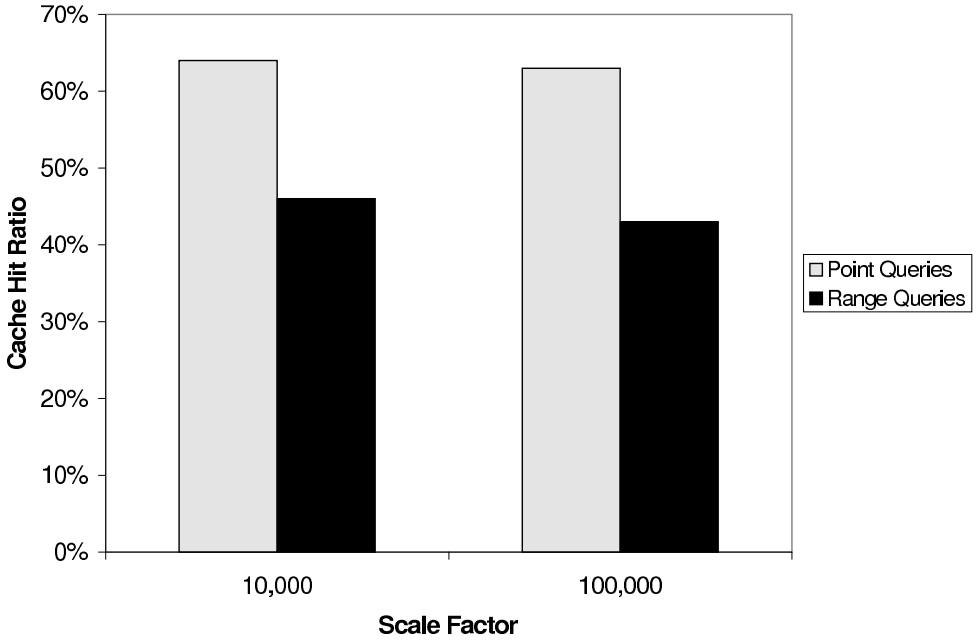

Figure 15: Cache Hit Rate vs. TPC-W Scale Factor

Figure 15 shows that the database size does not have a significant effect on the cache hit ratio. This is due to the fact that we use a database with virtually unlimited space for storing cached items. Had we used main memory, the hit ratio would have decreased in larger scale factors.

4.5.4 Caching-Related Transaction Abortion Rate

Figures 16 and 17 show how the rate of aborted transactions is affected by changes in the number of clients and the type of workload, respectively[7]. Increasing the number of clients increases the contention for getting exclusive access to database tuples (for the purpose of updating) and therefore, adds to the number of transactions that are aborted, because they could not obtain the necessary locks.

The extremely high rate of transaction aborts in the ordering mix is because the ordering interactions issue longer database transactions (i.e., transactions with several SQL statements) that update data. These transactions are more likely to be aborted compared to single-statement read-only transactions of browsing interactions. A surprising observation is that, although the browsing mix has a lower ratio of ordering interactions compared to that of the shopping mix, the abort rate of the shopping mix is higher. We believe that this anomaly is due to an inefficiency in our implementation. As mentioned earlier, we use

[7]Note that this performance measure applies only to Architecture III.

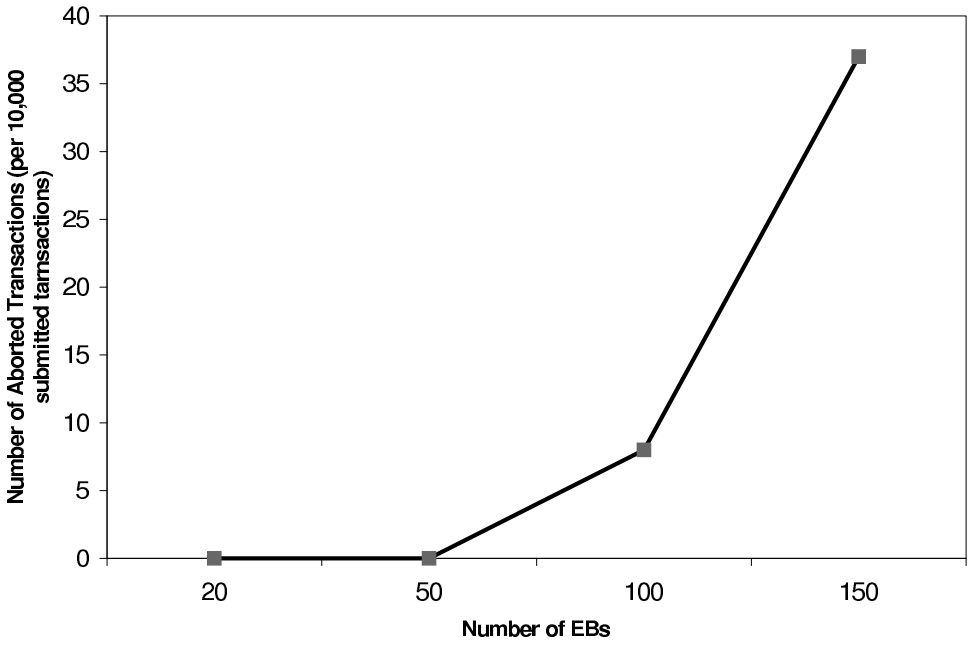

Figure 16: Transaction Abort Rate vs. Number of Clients

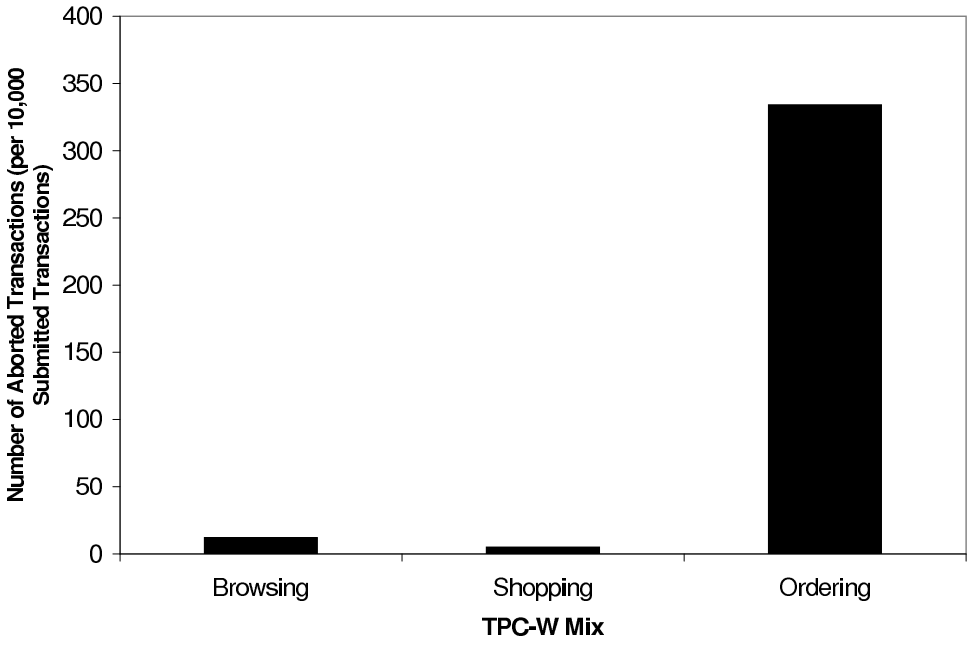

Figure 17: Transaction Abort Rate vs. Transaction Mix

a DBMS at each proxy to store cached items. Since, in our prototype implementation, we did not have access to the code of the commercial DBMS that we use, we had to implement our database cache as a stand-alone application that uses the database just like any other external application. This leads to some inefficiencies, one of which is in the way we insert new query results in the cache, which causes additional aborts. We believe that a more efficient implementation can solve this problem.

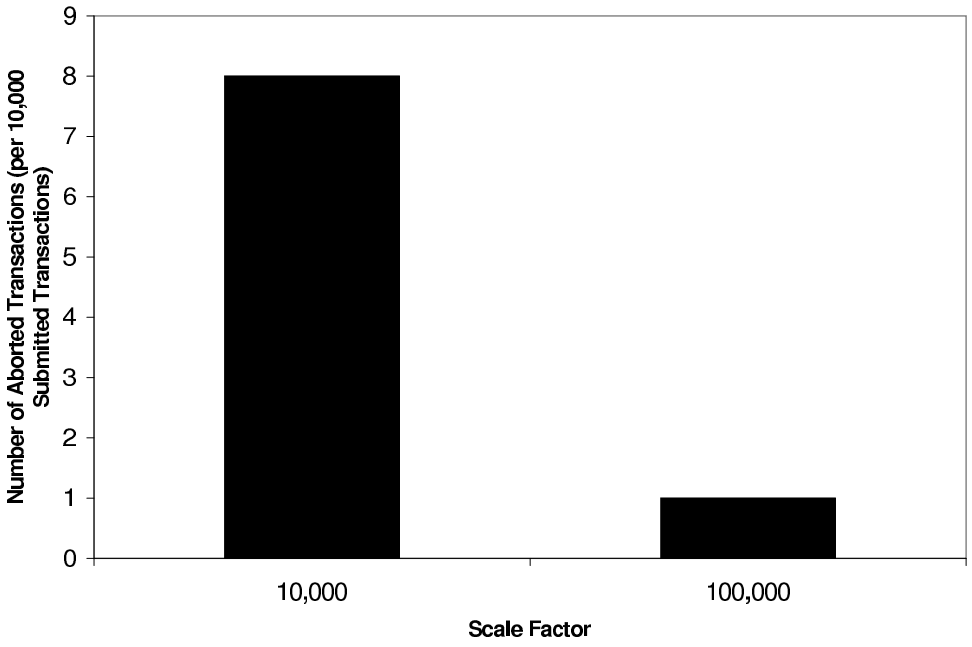

Figure 18: Transaction Abort Rate vs. TPC-W Scale Factor

The results depicted in Figure 18 show that the transaction abort rate dramatically decreases by using a larger database. We attribute this improvement to the fact that having more items in the database reduces the chances of contention for getting exclusive access to each item.

4.5.5 Discussion

The results of experiments presented in Section 4.5 show that no architecture is a definite winner. In this section, we summarize the results of the experiments and discuss the criteria for choosing the best architecture based on the system requirements and conditions.

Architecture I performs better than Architecture III if there are many update interactions. In such cases, query-per-page ratio is typically large, so Architecture I outperforms Architecture II as well (unless the network latency is very short as we see later). Another scenario in which Architecture I can be favored is when the network latency is short and there is not much load on the server, in which case using a more complex architecture can be overkill; for example, a small application with a limited number of users running on a small server.

In general, using Architecture II results in long latencies. Only when network latency is low and/or query-per-page ratio (defined in Section 4.5.1) is small, does Architecture II outperform the other alternatives. In these cases, Architecture II performs well because handling requests for static objects decreases the load on the server and forwarding the queries to the origin DBMS is not costly (as elaborated before). However, when network latency is low, the same benefits can be achieved by clustering the Web servers at the origin site rather than using proxy caches.

In most cases, Architecture III provides the best response time, especially when the load is high and/or the network latency between the origin server and the clients is long. In

general, Architecture III performs very well when the majority of the queries are read-only queries. However, if a workload has a high rate of update queries (in our experiments 50%), the overhead of the consistency protocol offsets the benefits of caching, making Architecture III inefficient. Consequently, using Architecture III is not recommended when the workload is update intensive. The experiments also show that a system using Architecture III may become saturated easier than a system using Architecture I, resulting in poor system throughput. We believe that a part, if not all, of this problem can be solved by implementing the consistency protocol inside the DBMS engine (rather than on top of it, as we did in our prototype implementation). If the problem is not completely solved by a more efficient implementation, more resources (hardware) should be given to a system using Architecture III in order to benefit from its good response time without losing throughput (caused by early saturation).

Type Of Queries	**Network Latency**	**Expected Load**	**Query-Per-Page Ratio**	**Recommended Architecture**
update–intensive	–	moderate	–	Architecture I
–	very short	very low	–	Architecture I
–	≈ 0	high	–	Architecture II or clustering Web servers
–	–	high	≈ 1	Architecture II or clustering Web servers
read–dominant	moderate–long	–	–	Architecture III
read–dominant	short–long	high	> 1	Architecture III

Table 2: Choosing the Best Architecture

Table 2 summarizes the above discussion by showing the *recommended architecture* with respect to the properties of the system and workload (specifically four properties). Each row represents a recommendation and each column represents a system or workload property. Note that when a cell is marked with "–", it means that the corresponding recommendation holds regardless of the value of the corresponding property (column).

In summary, except for some special cases, either Architecture I or Architecture III should be chosen for Web applications that require strong consistency and have a large number of users from different geographical locations. For such applications, Architecture III provides better response times when the workload is read-dominant; otherwise, Architecture I is the winner.

5 Conclusions and Future Work

5.1 Conclusions

In this paper, we examine three alternative architectures for database-backed dynamic Web applications whose clients need strict freshness guarantees. The first architecture is the most basic dynamic Web application architecture; it does not use any kind of caching whatsoever. The second and third architectures are motivated by the fact that using proxy servers brings the Web contents closer to the clients and reduces the network latency; therefore, it can improve the overall performance. The second architecture exploits caching of static contents at proxy servers, whereas the third architecture caches both static content and database tuples/query results at proxy servers.

The results of our experiments show that, compared to the other two architectures, the performance of the second architecture is by far the poorest. Although, in this architecture, proxy servers bring the static contents closer to the clients, proxies have to send every database query to the origin DBMS and incur the corresponding network latency. Considering the fact that generating each dynamic Web page may involve issuing several database queries, the experimental results suggest that this architecture suffers long response time and low throughput.

Our experiments show that Architecture III provides a better response time than Architecture I, except when there is a high ratio of updates, in which case Architecture I provides the best performance. We do not yet know whether an optimized implementation of server-side consistency controller can alleviate the poor performance of Architecture III when the update ratio is high.

The high cache hit rate in both tuple caching and query result caching shows that queries issued by a typical e-commerce Web application have a high degree of locality of access. If the cache hit ratio were not so high, the overhead of providing strong consistency could neutralize the benefits of caching.

In conclusion, our research shows that database caching at proxies is an effective technique for reducing the response time of a Web application and hence, gaining user satisfaction (even though providing strong consistency imposes a considerable overhead).

5.2 Contributions

Our first contribution is the design of a database cache and a strong consistency protocol that is optimized based on the typical workload of dynamic Web applications. Our second contribution is creating a prototype implementation of the database cache. The advantage of providing a complete implementation is that it can be used for accurate performance evaluations (e.g., performance comparison with other systems) using real-world system components (Web server, DBMS, etc). Another contribution is carrying out a thorough evaluation of the performance of three alternative architectures for dynamic Web applications in which strong consistency is mandatory (including an architecture that uses our database cache). We perform our experiments using real-world software components (Tomcat Web server, IBM DB2 Universal Database, etc.), which makes our results much more reliable and accurate compared to results obtained through simulations.

5.3 Future Work

As mentioned in Section 3.4.6, we have implemented the consistency protocol of Architecture III on top of the DBMS consistency control. A possible future extension to our work is to implement the consistency protocol as part of the database consistency control mechanism, which can result in considerable optimizations and increase the performance of Architecture III. In particular, we believe that it may increase the system throughput up to the level of Architecture I, or even higher.

Examining architectures not considered in our work, (e.g., architectures in which caches communicate with each other) can be the subject of future work.

This paper considers only database caching. An important direction for future research is investigation of techniques for providing strong consistency for other types of caching (e.g., Web page or page fragment caching). Most of the research bodies carried out in the area of dynamic Web caching either consider weak consistency (e.g., [24]), or propose systems that cannot work transparently (i.e., they need a great deal of addition to the application code, e.g., [8]).

Another direction for future research is relaxing the strong consistency condition and examining the performance as well as error ratio of the resulting caching system. The results *may* show that being (somewhat) lenient about consistency yields considerable performance gains, while imposing a low error ratio, which makes such caching systems a good choice for Web applications that can tolerate stale data. It is also useful to investigate different ways of controlling (limiting) the amount of staleness of the data served to the users.

Another possible extension is providing different levels of consistency for different Web interactions (of the same application). For example, in an e-commerce site, for interactions that deal with ordering, strong consistency is required, whereas users may put up with occasionally getting stale data in the interactions that deal with browsing.

Our research focuses on finding the best architecture in terms of system response time. Some of our findings suggest that distributing the server load by using proxies can improve scalability of a dynamic Web application considerably. It is beneficial to perform more elaborate analysis on the effect of database caching on scalability. In particular, database caching at proxies should be compared against database replication (at origin servers) to find out the merits of each.

References

[1] M. Altinel, C. Bornhovd, S. Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald. Cache tables: Paving the way for an adaptive database cache. In *Proc. 29th Int. Conf. on Very Large Data Bases*, pages 718–729, 2003.

[2] M. Altinel, Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, B. G. Lindsay, H. Woo, and L. Brown. Dbcache: database caching for web application servers. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 612–612, 2002.

[3] C. Amza. Transparent caching and consistency in dynamic content web sites (draft paper). available at http://www.eecg.toronto.edu/ amza/papers/osdi.ps.

[4] C. Amza. *Conflict-Aware Scheduling for Dynamic Content Applications.* PhD thesis, Department of Computer Science, Rice University, Houston, Texas, 2003.

[5] J. Anton, L. Jacobs, X. Liu, J. Parker, Z. Zeng, and T. Zhong. Web caching for database applications with oracle web cache. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 594–599, 2002.

[6] C. Bornhvd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptive database caching with DBCache. *Data Engineering*, 27(2):11–18, 2004.

[7] C. Brabrand, A. Møller, S. Olesen, and M. Schwartzbach. Language-based caching of dynamically generated HTML. *World Wide Web Journal*, 5(4):305–323, 2002. Kluwer.

[8] K. Candan, W. Li, Q. Luo, W. Hsiung, and D. Agrawal. Enabling dynamic content caching for database-driven web sites. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 532–543, 2001.

[9] L. Cao and M. T. Özsu. Evaluation of strong web caching techniques. *World Wide Web Journal*, 5(2):95–123, 2002.

[10] P. Cao, J. Zhang, and K. Beach. Active cache: Caching dynamic contents on the web. In *Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, pages 373–388, 1998.

[11] J. Challenger, P. Dantzig, and A. Iyengar. A scalable system for consistently caching dynamic web data. In *Proc. 18th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '99)*, pages 294–303, New York, 1999.

[12] J. Challenger, A. Iyengar, K. Witting, C. Ferstat, and P. Reed. A publishing system for efficiently creating dynamic web content. In *Proc. the INFOCOM Conference*, pages 844–853, 2000.

[13] Tomcat Servlet/JSP Container. http://jakarta.apache.org/tomcat/.

[14] Oracle Corp. Oracle 9i application server: Database cache, 2001.

[15] Inktomi Corporation. http://www.inktomi.com/products/network/traffic.

[16] IBM DB2 Universal Database. http://www-3.ibm.com/software/data/db2/.

[17] A. Datta, K. Dutta, H. Thomas, D. VanderMeer, Suresha, and K. Ramamritham. Proxy-based acceleration of dynamically generated content on the world wide web: An approach and implementation. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 97–108, 2002.

[18] Java TPC-W Implementation Distribution. http://www.ece.wisc.edu/p̃harm/tpcw.shtml.

[19] F. Douglis, A. Haro, and M. Rabinovich. Hpp: Html macro-preprocessing to support dynamic document caching. In *USENIX Symposium on Internet Technologies and Systems*, pages 83–94, 1997.

[20] V. Duvvuri, P. Shenoy, and R. Tewari. Adaptive leases: A strong consistency mechanism for the world wide web. In *Proc. INFOCOM Conference*, pages 834–843, Tel Aviv, Israel, March 2000.

[21] Z. Fei. A novel approach to managing consistency in content distribution networks. In *Proc. 6th Workshop on Web Caching and Content Distribution*, pages 71–86, Boston, MA, June 2001.

[22] D. Florescu, A. Levy, D. Suciu, and K. Yagoub. Optimization of run-time management of data intensive web sites. In *Proc. 25th Int. Conf. on Very Large Data Bases*, pages 627–638, Edinburgh, Scotland, September 1999.

[23] C. Gray and D. Cheriton. Leases: An efficient fault tolerant mechanism for distributed file cache consistency. In *Proc. 12th ACM Symposium on Operating Systems Principles*, pages 202–210, 1989.

[24] V. Holmedahl, B. Smith, and T. Yang. Cooperative caching of dynamic content on a distributed web server. Technical Report TRCS98-12, Computer Science Department, University of California Santa Barbara, 1998.

[25] A. Iyengar and J. Challenger. Improving web server performance by caching dynamic data. In *USENIX Symposium on Internet Technologies and Systems*, pages 49–60, 1997.

[26] A. M. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. *The VLDB Journal*, 5(1):35–47, 1996.

[27] A. Labrinidis and N. Roussopoulos. Webview materialization. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 367–378, 2000.

[28] C. Liu and P. Cao. Maintaining strong cache consistency in the world-wide web. In *Proc. 17th Int. Conf. on Distributed Computing Systems*, pages 12–21, 1997.

[29] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. Lindsay, , and J. Naughton. Middle-tier database caching for e-business. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 600–611, 2002.

[30] Q. Luo and J. Naughton. Form-based proxy caching for database-backed web sites. In *Proc. 27th Int. Conf. on Very Large Data Bases*, pages 191–200, 2001.

[31] A. Ninan, P. Kulkarni, P. Shenoy, K. Ramamritham, and R. Tewari. Cooperative leases: Scalable consistency maintenance in content distribution networks. In *Proc. 11th Int. World Wide Web Conference*, pages 1–12, 2002.

[32] M. Rabinovich and O. Spatscheck. *Web Caching and Replication.* Addison-Wesley, 2002.

[33] K. Rajamani and A. Cox. A simple and effective caching scheme for dynamic content. Technical Report, CS Dept., Rice University, September 2000.

[34] The Times Ten Team. High performance and scalability through application-tier in-memory data management. In *Proc. 26th Int. Conf. on Very Large Data Bases*, pages 677–680, 2000.

[35] Transaction Processing Performance Council (TPC). TPC Benchmark W (Web Commerce) specification, http://www.tpc.org/tpcw/default.asp, 2002.

[36] K. Yagoub, D. Florescu, V. Issarny, and P. Valduriez. Caching strategies for data-intensive web sites. In *Proc. 26th Int. Conf. on Very Large Data Bases*, pages 188–199, 2000.

[37] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Volume leases for consistency in large-scale systems. *Knowledge and Data Engineering*, 11(4):563–576, 1999.

[38] H. Yu, L. Breslau, and S. Shenker. A scalable web cache consistency architecture. In *Proc. ACM SIGCOMM*, pages 163–174, 1999.