Scalable Consistency Management for Web Database Caches

Charles Garrod[1], Amit Manjhi[1], Anastassia Ailamaki[1],
Phil Gibbons[2], Bruce Maggs[1,3], Todd Mowry[1,2],
Christopher Olston[1,4], Anthony Tomasic[1]

July 2006
CMU-CS-06-128

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[1]Carnegie Mellon University
[2]Intel Research Pittsburgh
[3]Akamai Technologies
[4]Yahoo Research

This research was supported by the National Science Foundation (NSF) under grant number CNS-0435382. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, Carnegie Mellon University, Intel Research, Akamai Technologies, or Yahoo Research. All unidentified trademarks mentioned in the paper are properties of their respective owners.

Abstract

We have built a prototype of a scalable dynamic web-content delivery system, which we call S3. Initial experiments with S3 led us to conclude that the key to achieving scalability lay in reducing the workload on back-end databases. S3 utilizes proxy servers to generate dynamic content and cache the results of queries forwarded to the back-end database. This approach introduces the challenge of maintaining cache consistency when the database is updated. In this paper we introduce a fully-distributed consistency management infrastructure that uses a scalable publish / subscribe substrate to propagate update notifications. We use static analysis of the database workload to introduce several design alternatives for how to map database requests to publish / subscribe groups. Finally, we develop a simulation framework and use both simulation and our S3 prototype implementation to evaluate these alternatives empirically to determine which design is best for typical dynamic web workloads.

Keywords: consistency management, database caching, database scalability services, publish / subscribe

1 Introduction

Applications deployed on the Internet are immediately accessible to a vast population of potential users. As a result, they tend to experience unpredictable and widely fluctuating degrees of load, especially due to events such as breaking news (e.g., 9/11), sudden popularity spikes (e.g., the "Slashdot effect"), or denial-of-service attacks. Content Distribution Network (CDN) technology largely addresses this issue for static content by using a large shared infrastructure to absorb load spikes. Recently, a number of systems have proposed a similar architecture for scaling the delivery of database-backed dynamic content [14, 16, 11, 8]. In each of these systems users interact with proxy servers that mimic a traditional three-tiered architecture (containing a web server to handle user requests, an application server to generate dynamic content, and a database server as a back-end data repository). These proxy servers typically cache static content and generate dynamic content locally, using a database cache and forwarding requests to a back-end database server as needed.

These design decisions successfully lift the application server load from the centralized infrastructure and reduce the number of requests sent to the back-end database server. They introduce the problem, however, of maintaining the consistency of database caches in a scalable way. Most current systems do not adequately address this issue and rely upon simple consistency mechanisms that scale poorly. For example, several notable systems require the back-end database server to track the contents of the proxy caches and forward relevant updates to each proxy server [16, 11, 18].

We propose a fully distributed consistency management infrastructure for this proxy architecture, using group multicast as a communication primitive. In our design the back-end servers track no information about the proxy caches and perform no additional work to maintain cache consistency. The main advantage of our design over previous efforts is that it lifts the burden of consistency management from the back-end database server.

There is, however, a fundamental trade-off between the scalability of group communication and the expressibility of its subscription language. The current state of fully-distributed group communication technology led us to choose a topic-based publish / subscribe system. This choice was attractive because there are mature implementations whose performance is fairly well understood. In topic-based publish / subscribe, a client receives all publications to each group to which it is subscribed. No filtering takes place based on the contents of the published objects. To limit the publications that a client receives, it must selectively subscribe to only those topics that are relevant to it. In the spectrum of publish / subscribe implementations, topic-based systems are the most primitive that support a non-trivial subscription language.

Given this group communication choice, the key design question is how to map database requests into topics in the communication infrastructure to ensure the consistency of the database caches. In this paper we use the static analysis of dynamic applications and their database workloads to develop two alternative policies. We then study the empirical behavior of these policies in depth using both simulation and a prototype implementation of the proxy architecture.

The contributions of this paper are:

- A new, fully distributed approach to consistency management that reduces load on back-end database servers.
- The use of static analysis of a dynamic application and its database requests to determine efficient mappings between database requests and the topic-based groups utilized by our consistency management system.
- A simulation framework for studying the performance of such mappings.
- The design and implementation of a complete prototype scalability service for database-backed dynamic content – S3 – including our fully distributed consistency management mechanism, and an initial evaluation of this prototype.

In Section 2 we describe the design and implementation of the S3 system. Section 3 describes the fundamental problem of efficiently mapping database requests into a topic-based subscription language and explores several alternative strategies for doing so. Section 4 describes the implementation of the SimS3 event-based simulator that models S3 for large network sizes, enabling us to evaluate our configuration choices for systems too large to otherwise test. Section 5 describes our fully functioning S3 prototype, which we use to validate the results of the SimS3 simulator and obtain an initial performance evaluation of the S3 design.

2 The S3 Scalability Service

The S3 design is similar to that of a static CDN. We envision a system in which businesses or other organizations would pay to scale their dynamic content on a per-usage basis. As with a CDN, S3 utilizes a large shared infrastructure to absorb spikes in a particular customer's demand.

Like other systems that seek to scale dynamic content, S3 users connect directly to proxy servers instead of the centralized home server. Each proxy server consists of a static web cache, an application server, a database cache which stores materialized views of query results, and a cache consistency module. When a dynamic web application issues a database query, the proxy server responds immediately using its database cache if possible. If the query result is not present in the cache, the proxy forwards the request to the centralized database server and caches the reply. Whenever a proxy server caches a query result, the consistency module subscribes to some set of multicast groups related to that query. Whenever a query result is removed from a proxy's cache, the proxy server unsubscribes from any multicast groups on which no remaining queries depend.

When a dynamic web application issues a database update the proxy server always forwards the update to the centralized database server, which contains the persistent state of the system. The proxy's consistency module then publishes an update notification to some set of multicast groups related to that update. When a proxy server receives an update notification the consistency module may then process that update in whatever fashion necessary to implement its consistency policy. In general, the S3 design specifies only a mechanism for communicating update notifications among proxy servers and does not require the choice of any particular consistency or caching policy. Different customers could choose different policies from each other, or conceivably even vary their policies as their applications' needs change.

The S3 system is designed to be as transparent as possible to the application and central database server. All communication between the proxy server and database server occurs between an S3 proxy module and an S3 module at the back-end database server. The server module accesses the central database using whatever database driver the application would ordinarily use, and the proxy module implements the standard database driver interface used by the application. In general, the only necessary change to the database configuration or application code is to ensure that the S3 proxy module is loaded instead of the standard database driver.

S3 uses the Scribe [15] publish / subscribe system as its standard group communication mechanism. Scribe is a completely decentralized multicast system implemented on top of the Pastry [23] peer-to-peer routing system, which uses a distributed hash table based on PRR trees to route network messages to their destination. Scribe's implementation is well-suited to S3's design. In particular, it incurs no storage load for empty multicast groups and supports efficient subscription and unsubscription for even very large networks and large multicast groups, without introducing hot spots in the network. Scribe obtains these benefits at the cost of introducing additional network load for each message, since each message in the Pastry overlay may require a logarithmic number (in the size of the proxy network) of physical messages.

To focus on the issues fundamental to scalability we chose simple implementations for the caching and consistency management policies used in our simulator and prototype. We implement an unbounded cache at each proxy server. Although this choice would be unrealistic in practice, it reasonably approximates the typical situation in which proxy servers use their large physical disk to store the cache. As a practical experimental benefit, it also eliminates the variable effect of a cache replacement algorithm on the performance of the consistency management system, allowing us to more clearly measure its characteristics.

For consistency management we implement a weak consistency model based on read-only proxy caches and best-effort invalidation. In some situations where determining the invalidation relationship between a query and update requires significant implementation work, we choose to always invalidate and accept that we are invalidating conservatively. We never choose to under-invalidate or intentionally cache inconsistent data, but our decision to avoid implementing a fully serializable transactional model inevitably results in occasional inconsistencies within our experiment executions.

The primary metrics of performance are database throughput and the number of simultaneous users that S3 can support without exceeding a threshold latency per request. For configurations that can support equal system loads, our secondary goal is to reduce the overall network traffic within the system. We accomplish this by attempting to reduce the number of subscriptions, unsubscriptions, publications, and notifications in the consistency management infrastructure.

3 Utilizing Topic-based Publish / Subscribe

The primary open question is how to associate database requests with multicast groups to ensure the efficient communication of update notifications among proxy servers. We call this the *database multicast configuration problem* and often refer to a particular association of database requests with multicast groups as a *multicast configuration*.

In this section we discuss the database multicast configuration problem in more detail, show how we utilize knowledge of a web application's embedded database requests to develop several competing multicast configurations for that application, and use static properties of those multicast configurations to predict their performance relative to each other.

3.1 The database multicast configuration problem

In our design a proxy cache subscribes to some set of multicast groups when it caches a query, and broadcasts to some set of multicast groups for each update it issues. Formally, a database multicast configuration is a function that maps database requests to sets of multicast groups. For a database multicast configuration to be *correct* it must ensure that for each update, any cached query result potentially affected by that update must cause a subscription to at least one multicast group to which that update is published.

We say that a correct multicast configuration $\mathcal{C}$ is *minimal* if, for every smaller subset $\mathcal{C}'$ of its mappings, $\mathcal{C}'$ is an incorrect configuration. We call two queries *related* if they are distinct but both depend on some of the same data. Intuitively, a good multicast configuration is minimal, maps related queries into the same multicast groups, and maps unrelated queries into different multicast groups. In doing this, it avoids unnecessary update notifications for which no queries are affected and avoids duplicate update notifications for related query results when their common data is updated.

3.2 Generating multicast configurations with static analysis

The database requests in many web applications consist of a small number of static templates within the application code. Typically, each template has a few parameters that are bound at run-time. Because of this, for a given application the proxy consistency module does not need to support consistency for general database requests. It needs only to support the range of queries and updates that could potentially be issued by that application code.

To bridge the gap between the semantically rich language of database requests and the rigid subscription language supported by topic-based publish / subscribe systems, we first inspect an application's database templates and apply offline query-update independence analysis [20] to each potential query-update pair. This analysis identifies pairs of query-update templates for which the update will not affect the query result. We then take the complement of the independent pairs – dependent templates for which the update may invalidate the query – and use the set of dependent pairs to ensure the proper delivery of update notifications.

Template	Group-by-query Associated Multicast Groups	Group-by-update Associated Multicast Groups
Update 1	{QUERY3:NAME=?, QUERY4, QUERY5}	{UPDATE1:NAME=?, UPDATE1}
Update 2	{QUERY3, QUERY5}	{UPDATE2}
Query 3	{QUERY3:NAME=?, QUERY3}	{UPDATE1:NAME=?, UPDATE2}
Query 4	{QUERY4}	{UPDATE1}
Query 5	{QUERY5}	{UPDATE1, UPDATE2}

Figure 1: Two correct, minimal multicast configurations for our sample inventory application.

Here we consider two natural paradigms in which templates are used to determine the multicast configuration: (1) multicast groups based upon the data on which each query depends, called *Group-by-query*, and (2) multicast groups based upon the data an update affects, or *Group-by-update*.

3.2.1 An example database application

To illustrate these paradigms consider this inventory application:

Update 1:
```
INSERT INTO inv VALUES
  (id = ?, name = ?, qty = ?,
  entry_date = NOW())
```
Update 2:
```
UPDATE inv SET qty = ?
  WHERE id = ?
```
Query 3:
```
SELECT qty FROM inv
  WHERE name = ?
```
Query 4:
```
SELECT name FROM inv
  WHERE entry_date > ?
```
Query 5:
```
SELECT * FROM inv
  WHERE qty < ?
```

In this example, Update 1 affects instantiations of Query 3 when the same name parameter is used, affects any instantiation of Query 4 for which the entry date was in the past, and affects instantiations of Query 5 whose quantity parameter was greater than the newly inserted item's. Update 2 affects instantiations of Query 3 whose name matches the id parameter that was used, is independent of Query 4, and affects instantiations of Query 5 if the change in the item's quantity crosses the quantity parameter used to instantiate the query.

Figure 1 gives minimal correct multicast configurations for this example based on the Group-by-query and Group-by-update paradigms. A question mark in the group name indicates that the appropriate parameter should be bound at run-time when the template is instantiated. We say that a group is *parameter-independent* if the group name contains no parameters that are bound at run-time, and otherwise that it is *parameter-dependent*.

First suppose that the consistency mechanism uses the Group-by-query based mapping and proxy server A starts with a cold cache. If server A caches Query Template 3 with the parameter "fork" it would then subscribe to the multicast groups corresponding to QUERY3 and QUERY3:NAME=FORK. If proxy server B then used Update Template 1 to insert a new item with the name "spoon" then server B would issue update notifications to the groups QUERY3:NAME=SPOON, QUERY4, and QUERY5.

Now suppose this workload was executed on a cold system using Group-by-update. When server A caches Query Template 3 it instead subscribes to the multicast groups for UPDATE1:NAME=FORK and UPDATE2 since those are the updates that could affect it. When server B inserts the new item it now needs to publish notifications to just two groups, one for the group UPDATE1:NAME=SPOON and one for UPDATE1.

3.2.2 Comparing group-by-query and group-by-update

The key static property of a multicast configuration is the number of multicast groups on which each database request depends: for queries, the number of groups to which each cached query result requires a subscription to, and for updates the number of groups to which an update notification must be broadcast. On average, we expect (1) queries to depend on fewer multicast groups with Group-by-query than with Group-by-update since multiple update templates may affect a query template. Similarly, we expect (2) updates to typically publish to more multicast groups with Group-by-query than with Group-by-update. Finally, we expect (3) that Group-by-update is more likely to aggregate related queries into the same multicast group since queries dependent on the same underlying data would tend to be affected by the same update templates.

Because of (2), we expect Group-by-update to result in a lower publication rate than Group-by-query for most workloads. However, most workloads are dominated by queries rather than updates, and the overall subscription rate is dependent on the competing factors (1) and (3). For workloads where Group-by-update's aggregation of related queries is low, we expect Group-by-query to result in a substantially better subscription rate. The relative performance of the two configurations is unclear when Group-by-update succeeds in aggregating related queries into the same group.

3.3 Multicast configuration and caching strategies

So far we have discussed the performance of the consistency management system as a function of the database workload but the true input to the system is actually the *cache* workload, which itself is dependent on the database workload, the consistency management policy, and the cache replacement algorithm. For simplicity we study only the case where the consistency management policy is invalidation and the cache size is unbounded. This choice, however, does not preclude us from studying configurations in which we intentionally avoid caching some objects for which the cost of caching is higher than the cost of regenerating the result. In this section we introduce a caching policy designed to complement our choice of topic-based publish / subscribe as our multicast substrate.

Specifically, we examine a *Selective Caching* strategy that avoids caching objects for which our multicast design performs poorly. With selective caching the proxy servers cache only queries that exclusively subscribe to parameter-dependent groups. For example, in our simple inventory application of Section 3.2.1, the selective caching strategy would not cache any instance of Query Template 4 since this query depends on a parameter-independent group (regardless of whether Group-by-query or Group-by-update is used). Intuitively, query results that cause subscriptions to parameter-independent groups often rely on data that cannot be determined from the query or query result (such as a join condition) and are frequently invalidated.

We call the strategy where proxy servers cache all queries *Indiscriminate Caching*. Selective caching will clearly result in worse cache performance, but should result in significantly better consistency management performance since it avoids caching the most volatile queries.

The open question is whether this will compensate for the additional load on the central database system caused by queries we chose not to cache. To answer this question we test four configurations: the combination of the two multicast configurations with the two caching strategies. We subsequently refer to these configurations as Indiscriminate Group-by-query, Indiscriminate Group-by-update, Selective Group-by-query, and Selective Group-by-update. We study this question via simulation in the following section and validate the results experimentally in Section 5.

4 S3 Simulation-based Results

It is impractical to test a full-scale working prototype for the full range of system sizes for which S3 could be used. To test configurations for very large networks we have built SimS3, an S3 cache and network simulator. In this section we first describe SimS3 and the workloads we use as input to in, and then describe and discuss the simulation results.

4.1 SimS3: the S3 simulator

SimS3 is a Java-based simulator that takes as input a collection of traces of database requests and models the cache and network state of the S3 system as if it executed those requests. SimS3 simulates an unbounded cache and invalidation policy as described in Section 2.

Each input trace is a log of database requests and the time at which each request was issued. For each trace, SimS3 models S3's behavior as if that trace were executed at a single proxy node. The cache state at each proxy and the membership of every multicast group are exactly traced through a SimS3 execution. SimS3 processes each database request atomically and interleaves the traces based on the requests' time stamps. No actual database activity occurs and SimS3 does not impose any additional constraints on the ordering of simulated database requests.

Benchmark	DB size	Details
TPC-W (bookstore)	217 MB	10,000 items 86,400 registered users
RUBiS (auction)	990 MB	33,667 items 100,000 registered users
RUBBoS (bboard)	1.4 GB	213,292 comments 500,000 users

Figure 2: Configuration parameters for each benchmark.

4.2 Experimental workloads

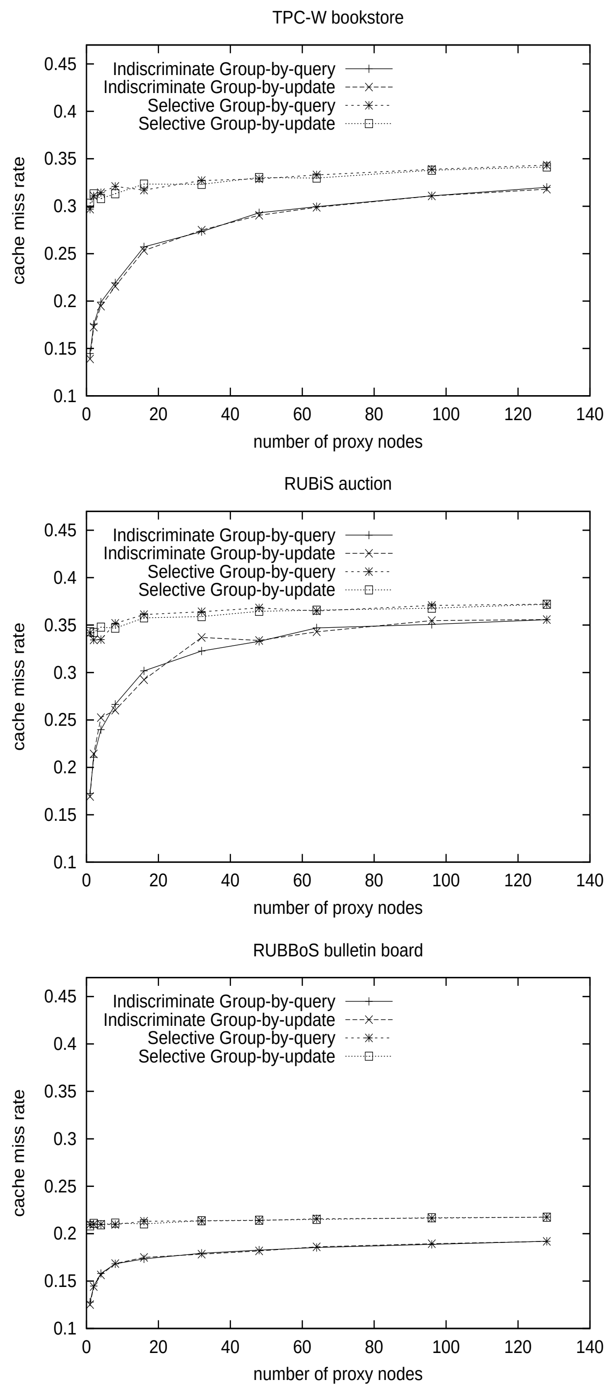
As input we use database traces from short executions (5-10 minutes) of the TPC-W bookstore [24], RUBiS auction [21], and RUBBoS bulletin board [22] benchmarks. Figure 2 shows the database configuration used for each benchmark. All three benchmarks conform to the TPC-W client specification for emulated browsers, which we configure to use a think time and session time exponentially distributed around means of 7 seconds and 15 minutes, respectively.

Our implementation of the TPC-W benchmark contains one significant modification. In the original benchmark all books are uniformly popular. Our version uses a more realistic Zipf distribution based on Brynjolfsson et al.'s work [2]. In particular, we model book popularity as $\log Q = 10.526 - 0.871 \log R$ where R is the sales rank of a book and Q is the number of copies sold in a short period of time.

Each experiment assigns a fixed set of 1000 users to each proxy server, and each proxy server supports the activity of 160 simultaneous emulated browsers. Thus, in every experiment the overall system load is proportional to the number of proxy servers. As our primary interest is the steady-state behavior of S3, each proxy server starts with a warm cache derived from the execution of other input traces of the appropriate benchmark. For reasons of practicality, each proxy cache was warmed using multiple database traces of short length rather than a single trace of long length. Although the total length of these traces approximates the length of a single trace of several-hour duration, the short traces may bias the cache states toward queries that are executed early in the benchmark workloads, slightly biasing the cache performance in the simulation results. We later discuss how this effect introduces differences between the simulation-based results described here and the results from the S3 prototype in Section 5.2.

4.3 SimS3 results

Figure 3 shows SimS3's measured cache performance for all three benchmarks and systems from 1 to 128 proxy servers. As expected, Group-by-query and Group-by-update result in the same cache miss rate when the same caching and invalidation policy is used. For all three benchmarks selective caching results in a significantly worse cache miss rate. This fact is particularly true for small proxy networks when the overall workload is lower and the



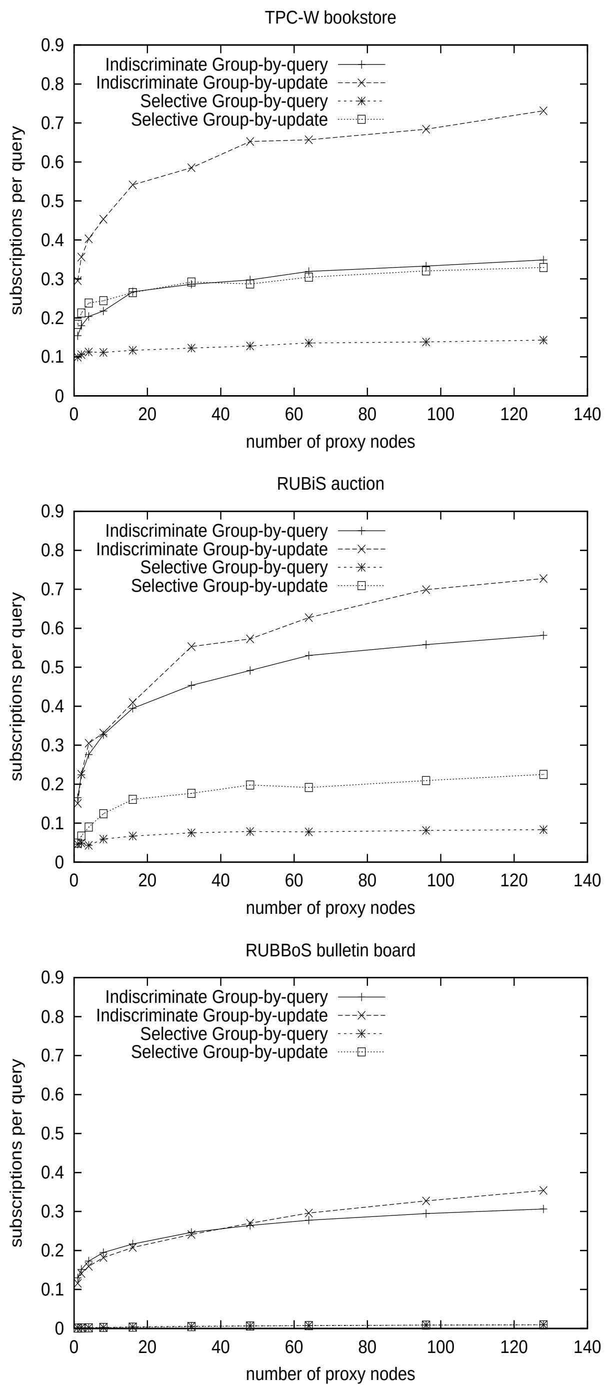


Figure 3: Simulated cache miss rates

Figure 4: Simulated subscriptions per query

update rate may be low even for relatively frequently invalidated objects. As the number of proxy servers and overall workload size increase, the performance of indiscriminate caching degrades quickly at first but less as the introduction of additional nodes only slightly affects the relative update rate. In contrast, the performance of selective caching degrades only slightly since it tends to avoid caching the objects that are frequently invalidated. One caveat to recall here is that overall cache performance may be biased by our use of short traces to warm the proxy caches.

Figure 4 shows the number of multicast subscriptions sent per query for each benchmark, again for 1 to 128 proxy nodes. Notice that we show subscriptions per query and not subscriptions per cache miss since the former metric is proportional to the total number of subscriptions in the network. Notably, selective caching always requires significantly fewer subscriptions since it avoids the thrashing caused by attempts to cache volatile queries. In all cases selective caching scales much better than indiscriminate caching.

For TPC-W, Group-by-update requires about twice as many subscriptions as Group-by-query, regardless of the network size. This fact is not too surprising since each query in TPC-W depends on nearly twice as many groups when using Group-by-update. This indicates that Group-by-update does not successfully aggregate related queries for TPC-W. For RUBiS and RUBBoS, however, Group-by-update performs similarly to Group-by-query for small networks and even outperforms it in some circumstances. This shows that Group-by-update does successfully aggregate related queries for these workloads. Notably, this effect is reduced for large networks, presumably because the high relative update rate reduces the probability of any related queries already being cached. Thus, for benchmarks where Group-by-update aggregates related queries successfully into the same multicast group we may expect Group-by-update to scale more poorly than Group-by-query, an unanticipated result.

Finally, Figure 5 shows the number of multicast notifications delivered per update for each benchmark. The TPC-W bookstore requires far fewer notifications per update than the RUBiS auction or RUBBoS bulletin board, with which updates are more likely to affect global data. Once again selective caching incurs significantly less network load than indiscriminate caching, and this network load scales better than with indiscriminate caching as the number of users and proxy servers increase. In all cases Group-by-update results in fewer notifications than Group-by-query. Notably, all four configurations scale well compared to traditional broadcast mechanisms since even in the worst case only about 15% of the proxy servers receive update notifications.

4.4 SimS3 conclusions

Overall, SimS3 demonstrates that neither Group-by-query nor Group-by-update dominates the other for all workloads. For the TPC-W bookstore, Group-by-query is better since it significantly outperforms Group-by-update for database queries but is only slightly worse for database updates, and the overall workload is query-dominated. For the RUBiS auction and RUBBoS bulletin board the analysis is not so clear. For these benchmarks Group-by-update may perform better for some network sizes and numbers of users, but worse as the overall system load increases.

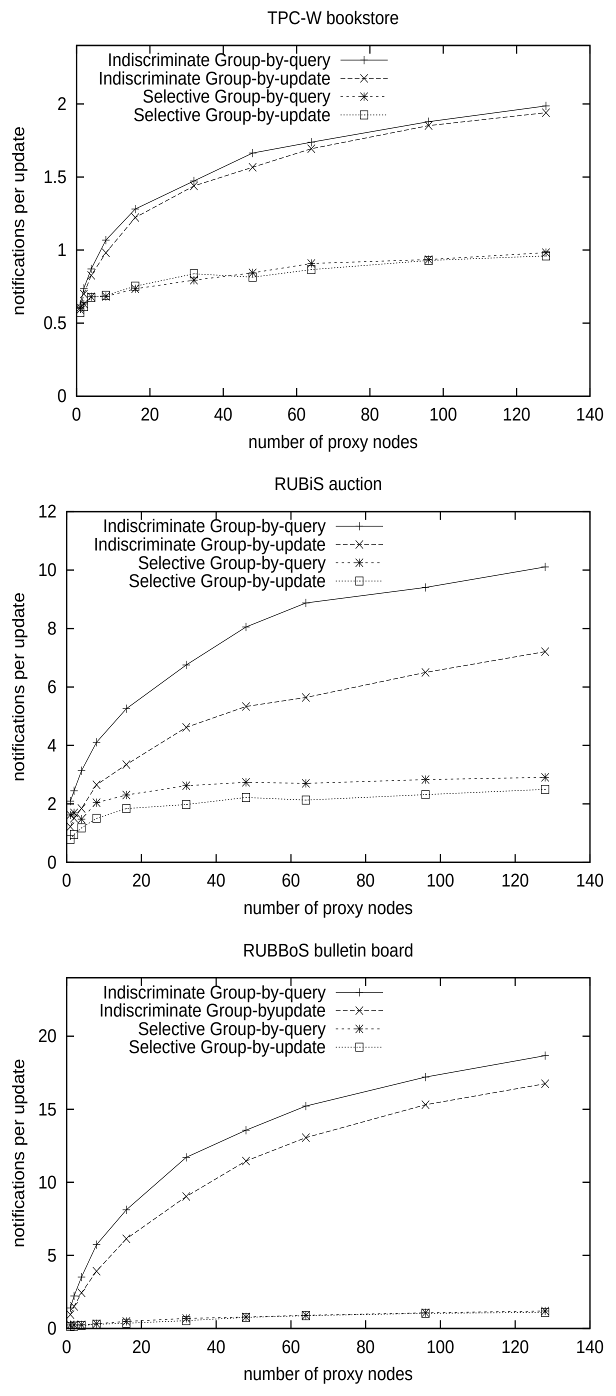


Figure 5: Simulated notifications per update

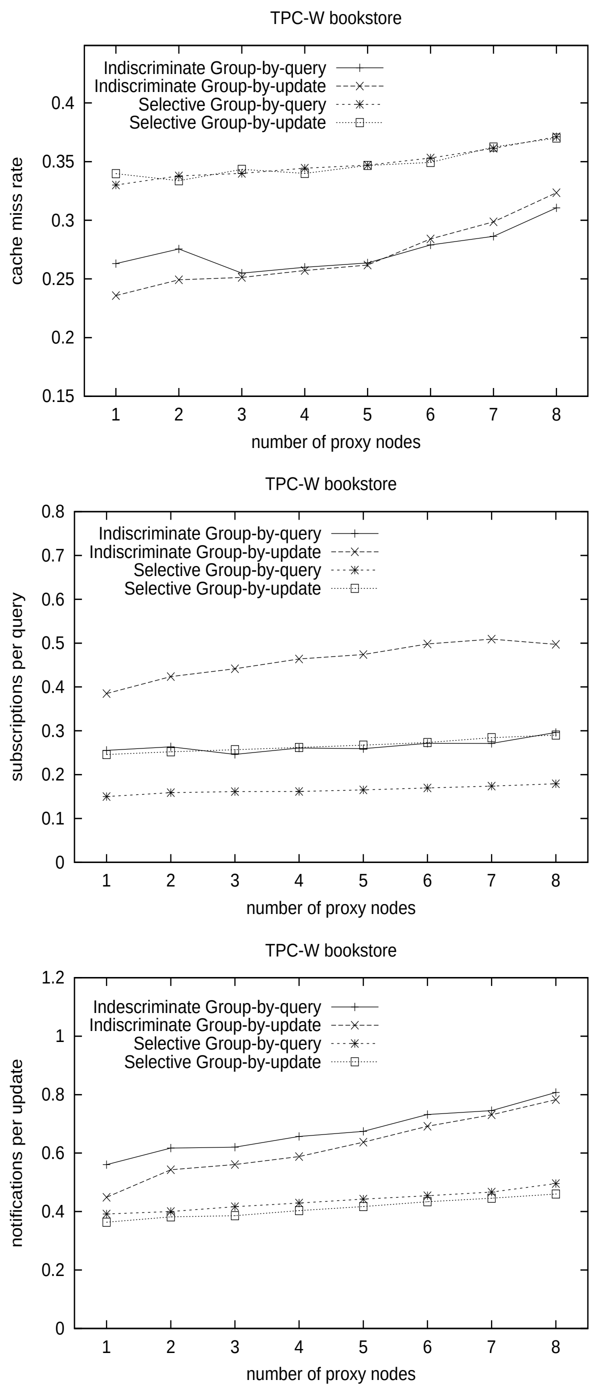


Figure 6: Cache and multicast network performance of the TPC-W benchmark on the S3 scalability service.

As expected, selective caching results in significantly less load on the consistency management infrastructure but increases the cache miss rate. For these workloads selective caching scales almost perfectly as the number of users and proxy servers increase.

5 S3 Prototype Results

To confirm the experimental results obtained with SimS3 and provide a platform for more extensive performance testing we have implemented a fully-functioning prototype of the S3 system. In this section we describe the S3 prototype, report initial performance measurements, and show that the prototype validates the conclusions of Section 4.

5.1 The S3 prototype

The S3 server and proxy modules are both implemented in Java 1.4 and executed using Sun Microsystem's standard JVM for Linux. The central database server utilizes MySQL4 as the back-end database management system, which the S3 server module accesses using the MySQL Connector/J JDBC driver. The central server also runs Apache Tomcat as both a web server and servlet container, but this Tomcat server is not utilized in our experiments since each proxy server begins with a warm static cache.

Each proxy server runs the Apache Tomcat server as a static cache and as a servlet container to generate dynamic content. The S3 proxy module implements the standard Java 2 JDBC API. We use the Scribe multicast system that is distributed with FreePastry 1.4.3_02. The S3 proxy module implements the simple invalidation policy and unbounded cache described in Section 2.

All experiments were run on the Emulab testbed [7]. The database server runs on a 3 GHz Intel Pentium Xeon processor with 2 GB of memory and a 10,000 RPM SCSI disk. Each proxy server executes on an Intel P-III 850 MHz processor with 512 MB of memory and a large 7200 RPM IDE disk. Each proxy is connected to the database server with a high latency, medium bandwidth Emulab "net" link (100 ms latency, 4 Mb/s total bandwidth). Client servers also execute on 850 MHz nodes, and each client server is connected to a single proxy server by a low latency, high bandwidth duplex connection (5 ms latency, 20 Mb/s bandwidth). These network settings approximate a CDN-like deployment in which the proxy servers are located on the same local area network as the clients, which may be far from the central database server. Unfortunately, we were unable to control the ethernet interface on which Scribe opened its server port. In these experiments, invalidation traffic is routed over the Emulab control network rather than the lower performance experimental network. Although this potentially improves the performance of the S3 system on some metrics, we do not expect a significant effect for any metrics on which we evaluate it here.

Each client runs the modified copy of the TPC-W bookstore benchmark described in Section 4.2. We configured the benchmark to resemble the SimS3 experiments as closely as possible. The database configuration is also as described in Figure 2 and we again use 1000 users and 160 simultaneous emulated browsers per proxy server. Each proxy cache is

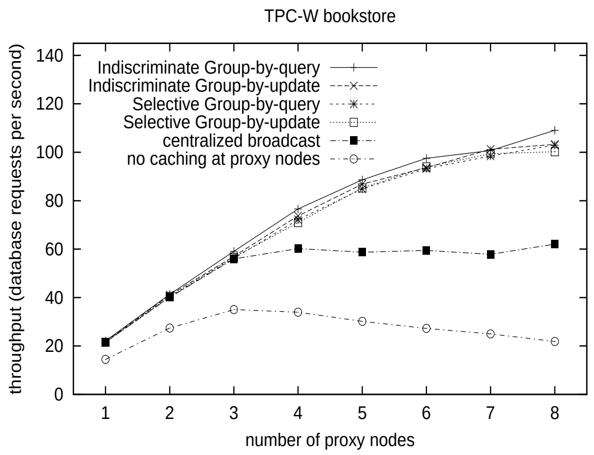


Figure 7: Overall database throughput achieved by the S3 scalability service for various multicast and caching configurations.

initialized to a warm state obtained from a single execution of about 3 hours. All experiments start from the warm cache state and are of medium duration, 15-20 minutes.

5.2 Results from the S3 prototype

Figure 6 shows the S3 prototype performance for the TPC-W bookstore benchmark for systems from one to eight proxy servers. Qualitatively, each result is similar to that obtained from SimS3 for the equivalent experiment. As before, the cache performance of Group-by-query and Group-by-update are equivalent within experimental uncertainty when an equivalent caching policy is used. Likewise, indiscriminate caching significantly outperforms selective caching but requires far more communication between proxy servers. And as SimS3 predicts, Group-by-query requires significantly less invalidation traffic than Group-by-update on the TPC-W benchmark.

Although the prototype results are consistent with SimS3 when predicting the relative performances of the various strategies, their results are inconsistent in both the constant factor and overall scalability for each metric. We believe that the discrepancies are systemic and caused by the different methodology used to warm the proxy caches in the different experiments. Our hypothesis is that the database requests of the short traces used to warm the SimS3 proxy caches were more closely correlated to the experimental load, resulting in an over-optimistic cache performance in our simulation-based results compared to those from the S3 prototype.

Finally, Figure 7 shows the overall rate at which S3 can execute database requests on one to eight nodes. Here we additionally two new configurations: one in which no caching is done and one where the central database server broadcasts invalidations to each proxy. As expected, all configurations with caching outperform the configuration without caching. More significant is that all four multicast-based configurations far out-scale the broadcast experiment, which reaches its peak performance at around four proxy nodes. This demonstrates the value of using publish / subscribe to propagate update notifications rather than using traditional broadcast methods.

A surprising result is that the throughput achieved by selective caching is comparable to that achieved by indiscriminate caching, even though by relative cache hit performance selective caching performs as much as 40% worse. This result elucidates a fundamental fact of caching: cache hit or miss rate may be a misleading metric for evaluating overall cache performance. A simple cache hit or miss rate fails to account for the *cost* of the cache miss. Here we hypothesize that the mean cost of a cache miss for a frequently invalidated object is less than for other objects. This is probably because most data for such queries remains in the active buffer pool and is relatively cheap to access. For some network configurations we even could expect selective caching to outperform indiscriminate caching: if the bandwidth of the proxy nodes is a performance bottleneck, the high invalidation traffic at the proxy nodes caused by indiscriminate caching may be more costly than regenerating the query results at the database server.

6 Related Work

Aspects of our work touch on a wide variety of areas, and we share some characteristics with many other projects.

There has been a recent explosion of work on edge computing platforms. IBM DB-Cache [14, 16], IBM DBProxy [11] and NEC CachePortal [18] all implement database caching for dynamic web content, but each of these systems use a centralized invalidation mechanism to propagate updates to the proxy servers. More recently, GlobeDB [17] focuses on automatic data placement to improve scalability and provides weak consistency guarantees.

Group communication has long been among the tools used to facilitate database replication. The earliest example of this we find is Alonso's 1997 work [1]. Recently, Chandramouli et al. [4] use a content-based publish / subscribe system to propagate update notifications for distributed continuous queries, much as we do using topic-based publish / subscribe for data caches here.

Recently, several groups have utilized static analyses to improve the performance of replication. Amiri et al. [12] exploited templates to scale consistency maintenance, while Gao et al. [13] used application-specific data to improve replication performance for TPC-W. Challenger et al. [3] analyzed the data dependencies between dynamic web pages and the back-end database to maintain the coherence of the dynamic content in proxy caches. However, this seminal work relies on centrally tracking of the contents of each cache.

Our goal of aggregating related queries into the same multicast group is effectively a special case of the general aggregation problem in the publish / subscribe field. The goal there is to develop general mechanisms to aggregate more general publications into single notification messages to reduce the overall load on the system. Challenger et. al's [10] fragment-based approach for the dependence between dynamic web pages and back-end data is similar to our approach here.

Hacigumus et al. [19] and Aggarwal et al. [9] have previously considered systems in which a database server is employed as a shared service, but both of these works have focused on the security implications of such a system rather than its scalability.

We previously introduced the S3 architecture and showed that the central database server was the performance bottleneck in the system design [8]. Since then Manjhi et al. have studied the performance cost of adding security to the S3 architecture. In [6] Manjhi showed how a static analysis of an application's database templates can be used to determine which data in the database workload can be encrypted without causing unnecessary invalidations of query results in the proxy caches. In [5] we used static analysis to determine how additional data from the central database may be used to further improve invalidation performance without compromising security. Our work here, however, is the first to introduce the fully distributed S3 design including the publish / subscribe mechanism, the first complete S3 prototype, and the first study of how to utilize publish / subscribe to efficiently propagate update notifications in this setting.

7 Conclusions

Scalability services promise to be the next step in distributed database research. By efficiently sharing the investment of a large server farm among many web applications, distributed databases can help solve the over-provisioning problem faced by database administrators. One key problem in building a database scalability service is to maintain the consistency of database caches as the system scales in size.

Our design combines several technologies to efficiently maintain cache consistency: the caching of materialized views, static and dynamic analysis of queries and updates to determine their dependencies on each other, and the mapping of update notifications to multicast publish / subscribe channels. In this paper we investigate two strategies of how to associate database requests with topic-based multicast groups: (a) associating multicast groups with the data on which queries depend, and (b) associating multicast groups with the data which updates affect. We also explored two fundamental caching strategies: (a) caching all database requests, and (b) caching only database requests that are not extremely volatile. We combined these two design issues into four competing configurations.

To evaluate our design we simulated these configurations using three standard benchmarks and determined the impact of each configuration on cache hit rate and the network traffic needed to correctly propagate update notifications. We then confirmed our simulation results by implementing a prototype system and compared our configurations to less sophisticated systems that use either a no-caching policy or the broadcast of update notifications to all proxy servers.

Overall, all configurations of S3 significantly outperform configurations where either no caching or a broadcast policy is used. We also show that neither of our proposed multicast configurations is universally preferable to the other; on some workloads Group-by-query is superior while on other workloads Group-by-update is preferable. Finally, we show that our strategy of selective caching may greatly improve the performance of distributing update notifications in the multicast infrastructure without degrading overall system performance by substantially increasing the workload at the central database server.

References

[1] G. Alonso. Partial database replication and group communication primitives. In *Proc. ERSADS*, 1997.

[2] E. Brynojolfsson, M. Smith, and Y. Hu. Consumer surplus in the digital economy: Estimating the value of increased product variety at online booksellers. MIT Sloan Working paper No. 4305-03, 2003.

[3] J. Challenger, P. Dantzig, and A. Iyengar. A scalable system for consistently caching dynamic web data. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*, New York, New York, 1999.

[4] B. Chandramouli, J. Xie, and J. Yang. On the database/network interface in large-scale publish/subscribe systems. In *Proc. SIGMOD*, 2006.

[5] A. Manjhi et al. Invalidation clues for database scalability services. Technical Report CMU-CS-06-139, Carnegie Mellon University, July 2006.

[6] A. Manjhi et al. Simultaneous scalability and security for data-intensive web applications. In *Proc. SIGMOD*, 2006.

[7] B. White et al. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, 2002.

[8] C. Olston et al. A scalability service for dynamic web applications. In *Proc. CIDR*, 2005.

[9] G. Aggarwal et al. Two can keep a secret: A distributed architecture for secure database services. In *Proc. CIDR*, 2005.

[10] J. Challenger et al. A fragment-based approach for efficiently creating dynamic web content. *ACM Trans. Inter. Tech.*, 5(2):359–389, 2005.

[11] K. Amiri et al. DBProxy: A dynamic data cache for Web applications. In *Proc. ICDE*, 2003.

[12] K. Amiri et al. Exploiting templates to scale consistency maintenance in edge database caches. In *Proc. IWCW*, 2003.

[13] L. Gao et al. Improving availability and performance with application-specific data replication. *IEEE Transactions on Knowlege and Data Engineering*, 17(1):106–120, Jan 2005.

[14] M. Altinel et al. Cache tables: Paving the way for an adaptive database cache. In *Proc. VLDB*, 2003.

[15] M. Castro et al. SCRIBE: A large-scale and decentralised application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication*, October 2002.

[16] Q. Luo et al. Middle-tier database caching for e-business. In *Proc. SIGMOD*, 2002.

[17] S. Sivasubramanian et al. GlobeDB: Autonomic data replication for web applications.

[18] W. Li et al. CachePortal II: Acceleration of very large scale data center-hosted database-driven web applications. In *Proc. VLDB*, 2003.

[19] H. Hacigumus, B. Iyer, and S. Mehrotra. Providing database as a service. In *Proc. ICDE*, 2002.

[20] A. Y. Levy and Y. Sagiv. Queries independent of updates. In *Proc. VLDB*, 1993.

[21] ObjectWeb Consortium. Rice University bidding system. `http://rubis.objectweb.org/`.

[22] ObjectWeb Consortium. Rice University bulletin board system. `http://jmob.objectweb.org/rubbos.html`.

[23] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, November 2001.

[24] Transaction Processing Council. TPC-W specification ver. 1.7.