Scalable Consistency Management for Web Database Caches

Charles Garrod⋆, Amit Manjhi⋆, Anastassia Ailamaki⋆, Phillip B. Gibbons‡, Bruce Maggs⋄, Todd C. Mowry‡, Christopher Olston⋆, Anthony Tomasic⋆

⋆School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
USA
{charlie, manjhi, natassa, bmm, tcm, olston, tomasic}@cs.cmu.edu

‡Intel Research Pittsburgh
4720 Forbes Ave., Suite 410
Pittsburgh, PA 15213
USA
phillip.b.gibbons@intel.com

⋄Akamai Technologies

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. Our architecture generates dynamic content at proxy servers which also 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. Our solution to this problem is a fully-distributed consistency management infrastructure that uses a scalable publish/subscribe substrate to propagate update notifications. We introduce several design alternatives mapping database requests to publish/subscribe groups, and evaluate these alternatives empirically to determine which design is best for typical dynamic web workloads.

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 [4, 14, 5, 18]. In each of these systems users interact with proxy servers that mimic a traditional three-tiered architecture. 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. However, they introduce the problem maintaining the consistency of database caches in scalable way. Most current systems do not adequately address this issue and rely upon simple consistency mechanisms that scale poorly. For example, at least one notable system requires the back-end database server to track the contents of the proxy caches and forward relevant updates to each proxy server.

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 tech-
nology 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. To implement our design we must address the fundamental problem of efficiently mapping database requests into a topic-based subscription language.

In this paper we introduce S3, a new scalability service for database-backed dynamic content, including our fully distributed consistency management system.

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 different mappings.
- The design and implementation of a complete prototype, and an initial evaluation of a scalability service for database-backed dynamic content, including our fully distributed consistency management mechanism.

In Section 2 we further 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 to 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 on a per-usage basis to scale their dynamic content. 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 is not present in the cache, it forwards the request to the centralized database server and caches the reply. Whenever a proxy server caches a query, the consistency module subscribes to some set of multicast groups related to that query.

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 changed.

The S3 system is designed to be implemented as transparently as possible to the application and central database server. All communication between the proxy server and database server occurs between a S3 proxy module and a S3 server module. The server module accesses the central database using whatever database driver the application would ordinarily use, and the proxy module may implement whatever standard database driver interface is used by the application. In general, the only necessary change to the database configuration or application code could be to ensure that the S3 proxy module is loaded instead of the standard database driver.

S3 uses the Scribe [8] publish / subscribe system as its standard group communication mechanism. Scribe is a completely decentralized multicast system implemented on top of the Pastry [21] peer-to-peer routing system. Scribe’s implementation is well-suited to the S3 design. In particular, it incurs no load for empty multicast groups, supports efficient subscription and unsubscription for even very large networks, and can sometimes reduce the load at hot spots in the network.

To focus on the issues fundamental to scalability we chose simple implementations for the caching and consistency management policies used in our simula-
or 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 the determining the invalidation relationship between a query and update would require significant implementation work, we simply 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.

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.

The database requests in modern web applications usually 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.

We leverage this key observation to bridge the gap between the semantically rich language of database requests and the rigid subscription language supported by topic-based publish / subscribe systems. To support a new web application we first inspect its database templates and apply offline query-update independence analysis [12] 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 that the set of dependent pairs to ensure the proper delivery of update notifications. For a given set of dependent templates there are a wide variety of potential mappings between queries, updates, and multicast groups. For a configuration to be correct it must ensure that for each dependent query-update pair there exists at least one multicast group to which the query and update are both mapped.

We say that a correct multicast configuration $C$ is minimal if, for every smaller subset $C'$ of its mappings, $C'$ 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 (1) minimal, (2) maps related queries into the same multicast groups, and (3) maps unrelated queries into different multicast groups. In doing this, it (1) avoids unnecessary update notifications for which no queries are affected, (2) avoids duplicate update notifications for related queries when their common data is updated, and (3) avoids unnecessary update notifications for unrelated queries that are not cached.

This intuition yields a natural association between multicast groups and data objects. In principle a multicast group might be associated with any arbitrary data object. Here we consider two fundamental paradigms: (1) multicast groups based upon the data on which a query depends, which we call Group-by-query, and (2) multicast groups based upon the data an update affects, or Group-by-update.

3.1 An Example Database Application

To illustrate these paradigms consider the following inventory application:

**Update Template 1:**
INSERT INTO inv VALUES (id = ?,
name = ?, qty = ?, entry_date = NOW())

**Update Template 2:**
UPDATE inv SET qty = ? WHERE id = ?

**Query Template 3:**
SELECT qty FROM inv WHERE name = ?

**Query Template 4:**
SELECT name FROM inv WHERE entry_date > ?

**Query Template 5:**
SELECT * FROM inv WHERE qty < ?

In this example, Update 1 affects instantiations of Query 3 when the same name parameter is used, affects instantiations of Query 4 for any past entry date, and affects instantiations of Query 5 whose quantity parameter was greater than the newly inserted item’s.

Update 2 affects instantiations of Query 3 when the same name parameter is used, is completely 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 group corresponding to \( \text{QUERY3:name=fork} \). 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 for \( \text{QUERY3:name=spoon} \) as well as \( \text{QUERY4} \) and \( \text{QUERY5} \).

Now suppose this same workload is executed on a cold system using Group-by-update. When server \( A \) caches Query Template 3 it instead subscribes to the multicast group for \( \text{UPDATE1:name=fork} \) as well as the parameter-independent group for \( \text{UPDATE2} \) since those are the update objects that could affect it. When server \( B \) inserts the new item it now needs to issue update notifications to just two groups, one for \( \text{UPDATE1:name=spoon} \) and one for \( \text{UPDATE1} \).

### 3.2 Static Comparison of Multicast Configurations

Ideally we would like to compare two multicast configurations by studying their static properties. In particular, it seems like properties such as the number of multicast groups on which a query depends could be used to predict performance metrics like the relative subscription rate. In practice, though, the performance of a particular configuration is highly dependent on dynamic workload characteristics. This section discusses how various properties of a multicast configuration may influence its overall performance and then applies that analysis to compare the Group-by-query and Group-by-update paradigms.

Intuitively, two multicast configurations should result in the same cache performance if the same consistency policy and caching policies are used; the multicast configuration only determines where update notifications are sent and not how they are utilized. However, two highly dissimilar configurations could result in secondary effects on the cache performance if an excess of multicast activity were to significantly affect the overall performance of the system.

Because subscriptions take place only when objects are added to a proxy server’s cache, we expect the overall subscription rate to be closely related to the cache miss rate. Naively, one might expect the subscription rate to the product of the cache miss rate and the (weighted) average number of groups on which each query depends. However, this fails to account for natural aggregation that may occur when related queries both depend on the same multicast group. A trivial example of this phenomenon can be observed in a broadcast configuration that aggregates all queries into a single group. For most common workloads, such a strategy reduces the subscription cost to zero but causes horrid performance when publications occur.

If the system is near executing near steady state, the long-term unsubscription rate should approximately equal the subscription rate. In steady state the total number of groups to which the system is subscribed should remain statistically unchanged, and this can only occur if the number of subscriptions and unsubscriptions are nearly equal.

In our design a notification is published to each group on which the update depends and no aggregation of these publications takes place. This results in an overall publication rate that is simply the number of updates times the weighted mean number of groups on which those updates depend. Thus, any aggregation of publications must occur as a consequence of our statically-determined configuration.

The number of notifications delivered is similarly just the publication rate times the (weighted) average number of proxy servers subscribed to the groups on which notifications take place. However, this average is complexly dependent on factors including the cache miss rate and the aggregation of subscriptions, mak-
ing an overall prediction of the number of notifications difficult to determine.

3.2.1 Group-by-query vs. Group-by-update

Queries typically depend on fewer multicast groups with Group-by-query as compared to Group-by-update, but updates typically depend on more. However, we expect that Group-by-update is more likely to result in related queries being aggregated into the same multicast group. Overall we expect this to be the determining factor for which configuration is better for a given workload.

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. We generally expect Group-by-update to outperform Group-by-query for both the publication and notification metrics, but since most workloads are dominated by queries rather than updates, we expect Group-by-query to be a better choice in these cases.

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 it is important to realize that the true input to the system is actually the cache workload (which itself is highly dependent on the database workload). We choose to study only configurations using a unbounded cache, but this choice does not preclude us from studying configurations in which we intentionally avoid caching some objects.

Earlier distributed systems have employed similar partial-caching algorithms for a particular purpose, such as not caching critical data that must be kept strongly consistent [18]. We feel that such a strategy only approximates the true criteria by which selective caching should be employed: avoid caching objects for which the cost of caching is relatively high compared to the cost of an additional cache miss. If the consistency management system is sufficiently scalable and can support strong consistency management, this precludes the need to avoid caching objects that require strong consistency.

In particular, we choose to avoid caching objects for which the multicast system performs poorly. We propose a Selective Caching strategy in which the proxy servers cache only queries that exclusively subscribe to parameter-dependent groups. 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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>DB size</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUBBoS (bboard)</td>
<td>1.4 GB</td>
<td>213,292 comments</td>
</tr>
<tr>
<td></td>
<td></td>
<td>500,000 users</td>
</tr>
<tr>
<td>RUBIS (auction)</td>
<td>990 MB</td>
<td>33,667 items</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10,000 items</td>
</tr>
<tr>
<td>TPC-W (bookstore)</td>
<td>217 MB</td>
<td>100,000 registered users</td>
</tr>
<tr>
<td></td>
<td></td>
<td>86,400 registered users</td>
</tr>
</tbody>
</table>

Figure 2: Configuration parameters for each benchmark.

question is whether this will compensate for the additional load on the central database system caused by queries we chose not to cache.

Overall 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.

4 SimS3: The S3 Simulator

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, a S3 cache and network simulator.

As input SimS3 takes a collection of database traces. Each trace is a log of database requests and the time at which each request was issued. For each trace input to SimS3, 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 tracked through a SimS3 execution. SimS3 processes each database request atomically and correctly interleaves each trace based on the requests’ time stamps. No actual database activity occurs and SimS3 does not impose any additionally constraints on the ordering of simulated database requests. SimS3 models the unbounded cache and same invalidation policy described in Section 2.

As input we use database traces from short executions (5-10 minutes) of the TPC-W bookstore [24], RUBiS auction [16], and RUBBoS bulletin board [17] 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 standard 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 of book popularity based on the work by Brynjolfsson et al. [7]. In particular, we model 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 in 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 each experiment we test all four strategies described in Section 3.3.

### 4.1 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 inferior cache miss rate. This fact is particularly true for small proxy networks when the update rate for even relatively “frequently” invalidated objects may be low. As the number of proxy servers increases, 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 very slightly since it always avoids caching many of the objects that are frequently invalidated.

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. Although not pictured, the number of unsubscriptions for each experiment is similar to the number of subscriptions as we expect. 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 merely 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 be-

![Figure 3: Simulated cache miss rates and how they vary with the size of the proxy network, using 1000 users and 160 simultaneous emulated browsers per proxy node.](image-url)
cause 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, 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. Note that the vertical scale for the y-axis varies for these figures. The TPC-W bookstore requires far fewer notifications per update than does the RUBiS auction or RUBBoS bulletin board, where updates are more likely to affect global data. Once again, selective caching significantly outperforms indiscriminate caching in both scalability and overall network load. 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 our worst case, only about 15% of the proxy servers receive update notifications.

4.2 SimS3 conclusions

Overall, the 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 clearly better since it significantly outperforms Group-by-update for database queries but is only slightly outperformed during database updates and the overall workload is query-dominated. For the RUBiS auction and RUBBoS bulletin board, the analysis is not so clear. Even for a fixed benchmark, Group-by-update will perform better for small network sizes but worse as the network size increases.

As expected, selective caching results in significantly less load on the consistency management infrastructure but greatly increases the cache miss rate. Additionally, for these workloads, selective caching scales almost perfectly as the system load increases.

5 The S3 Prototype Implementation

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 report initial performance measurements and show that the prototype validates the conclusions of Section 4.

The S3 server and proxy modules are both implemented in Java 1.4 and are executed using Sun Microsystems’s standard JVM for Linux [23]. The central database server utilizes MySQL4 [15] as the back-end database management system, which the S3 server module accesses using the MySQL Connector/J JDBC driver [1]. The central server also runs the Apache Tomcat [11] server in its stand-alone mode as both a web server and servlet container. However, this Tom-
Figure 5: Simulated notifications per update and how it varies with the size of the proxy network, using 1000 users and 160 simultaneous emulated browsers per proxy node.

cat server is not utilized in our experiments since each proxy server begins with a warm static cache.

Each proxy server also runs the Apache Tomcat server as a static cache and as a servlet container to generate dynamic content on behalf of the central server. The S3 proxy module implements the standard Java 2 JDBC API. We use the Scribe [8] multicast system that is distributed with FreePastry 1.4.3.02 [21]. The S3 proxy module also implements simple invalidation and a unbounded cache as described in Section 2.

All experiments run on the Emulab testbed [25]. The database server runs on a 3 GHz Intel Pentium Xeon processor with 2 GB of memory and a large 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 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 bandwidth). These network settings attempt to 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. We intend to fix this flaw in subsequent analyses.

Each client is configured to run a modified copy of the TPC-W bookstore benchmark as described in Section 4. 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 initialized to a warm state obtained from a single execution of long duration. All experiments that start from the warm cache state are of medium duration: typically 15-20 minutes.

5.1 S3 Prototype Results

Figure 6 shows the S3 prototype performance for the metrics evaluated by the SimS3 simulator, for systems from one to eight proxy servers. Characteristically, each result is very similar to that obtained using SimS3 for the equivalent experiment. As before, the cache performance of Group-by-query and Group-by-update are equivalent within experimental uncertainty when used in conjunction with an equivalent
Figure 6: Cache and multicast network performance of the TPC-W benchmark on the S3 scalability service.

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

caching policy, although the experimental uncertainty is higher when using the real implementation. 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. SimS3 also correctly predicts the degree to which each configuration scales for each performance metric.

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 performance metric. We believe that the discrepancies are systemic and caused by the different methodology used to warm the proxy caches in the different experiments. For reasons of practicality, the SimS3 caches were warmed from traces of many short executions, while the S3 caches are warmed from a single long execution. Although the aggregate length of the short traces was chosen to approximate the length of the S3 execution, we hypothesize that the database requests of the short traces were more closely correlated to database trace simulated by SimS3, resulting in an over-optimistic cache performance that subsequently resulted in differences with the other metrics as well.

Finally, Figure 7 shows the overall rate at which the S3 scalability service can execute database requests on one to eight nodes. In addition to our four standard configurations, we test two new configurations: one in which no caching is performed and one where the central database server broadcasts invalidations to each proxy server. As expected, all configurations that involve caching far 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. For this network and proxy de-
sign, however, even the configuration without caching benefits somewhat from the addition of proxy servers. This is because that for small networks, the relatively slow 850 MHz proxy server is the performance bottleneck instead of the 3 GHz database server. But without caching the database server quickly becomes overloaded and the overall throughput plateaus and even declines since we do not employ adequate admission control at the database.

Also notable is that all four of our standard configurations obtain the same overall throughput, within experimental uncertainty. We expected this for the Group-by-query and Group-by-update policies since that parameter primarily affects proxy server load and the overall system bottleneck is the database server. What is surprising is that selective caching achieves a throughput comparable to that of indiscriminate caching. By relative cache performance, selective caching performs as much as 40% worse than indiscriminate caching. 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 expect selective caching to outperform indiscriminate caching: particularly in configurations where the bandwidth of the proxy nodes becomes a performance bottleneck.

6 Related Work

Since various aspects of our work touch on a wide variety of areas, we share some characteristics with a large number of other projects.

There has been a recent explosion of work on edge computing platforms. IBM DBCache [4, 14], IBM DBProxy [5] and the NEC CachePortal [13] projects all provide some form of database caching for dynamic web content. Each of these systems use some sort of centralized invalidation mechanism to propagate updates to the proxy servers. More recently, GlobeDB [22] provides some very weak consistency guarantees and focuses on automatic data placement to improve scalability.

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 [3].

Recently, several groups have utilized static analyses to improve the performance of replication. Amiri et al. [6] exploited templates to scale consistency maintenance, while Gao et al. [9] used application-specific data to improve replication performance for TPC-W.

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.

Finally, Hacigumus et al. [10] and Aggarwal et al. [2] 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.

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 a collection of web applications, distributed databases will solve the over-provisioning problem faced by database administrators. However, the main problem in building a scalability service is the maintenance of the consistency of database caches as the system scales in size. Protocols for consistency in distributed databases have been studied since the late 1970s [19]. However, existing approaches do not scale well to the Internet.

Our design consists of a combination of technologies to efficiently maintain cache consistency: the caching (but not maintenance) of materialized views, static and dynamic analysis of queries and updates to determine cache invalidation, and the mapping of cache invalidation notification to multi-cast publish/subscribe channels. We investigated (a) mapping queries to channels and (b) mapping updates to channels. We also investigated caching (i) all database requests and (ii) caching less dynamic database requests. We combined these two design issues into four mapping designs.

To evaluate our design, we simulated, using three standard benchmarks, four mapping designs and measured the impact of these mappings on cache hit rates and network traffic. We then confirmed our analysis of our simulation results by implementing a prototype system. We compared our four policies running on our prototype system to a broadcast policy and to a no-caching policy as a base-line. In short, we show that mapping of query templates and update templates to publish/subscribe channels performs well as the number of caches grows from 1 to 8, corresponding to a system load that grows from 160 to 1260 emulated browsers. Each emulated browser represents many actual users of a system.

Overall, this work demonstrates the powerful technique of applying a static analysis database requests to simplify an otherwise intractable general problem of maintaining consistency. Finally, in examining selective caching in conjunction with a fully distributed multi-cast infrastructure, we elucidated a situation in which a counter-intuitive design choice may potentially result in better overall performance due to
its interaction with another complex system.

7.1 Future work

In building S3 upon a fully distributed multi-cast infrastructure we have explored only one point in a very large design space. In general we are interested in the relative advantages of a number of configurations in this space. In particular, we would like to explore how using a slightly less centralized but semantically richer publish / subscribe system like Hermes [20] would affect the overall performance of our update notification mechanism. Ideally we would like to obtain much better efficient coverage for interesting portions of the database request language.

Our use of static analysis can be significantly extended to similar problems. Some of our earlier work applied well-understood compiler techniques to enable the safe and efficient pre-fetching of database queries in some circumstances. We are currently exploring how static analysis might allow us to automatically augment database results with auxiliary data that vastly improve invalidation accuracy. In general, our use of foreknowledge of the application code is a powerful tool that has so far been largely unexplored.

References

