Publish / Subscribe for the Consistency of Web Database Caches

Charles Garrod

Abstract

Group communication primitives have long been among the tools used to facilitate database replication, and publish / subscribe systems are a natural mechanism to denominate update notifications for the consistency management of database replicas. However, the subscription and publication languages of current publish / subscribe systems do not efficiently capture the complex relationship between a database update and the data that update affects.

In this thesis we specialize the publish / subscribe mechanism to meet the requirements of web database caches. To do so, we exploit foreknowledge of a web application’s code and embedded database requests to efficiently support consistency management for typical database workloads. We compare the performance of our system to that obtainable by similar publish / subscribe systems that do not assume such foreknowledge, and examine how our system performs in an environment where the database schema and web application may evolve over time.

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 [1, 2, 3, 4]. 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 a 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.

Group communication primitives have long been among the tools used to facilitate database replication [1] and publish / subscribe systems are a natural mechanism which may enable scalable consistency management among the proxy caches. Using a publish / subscribe system, each proxy server issues some set of multicast subscriptions that corresponds to the data stored in its cache. When an update to the database occurs the system publishes an update notification, which the publish / subscribe system then delivers to all proxy caches which may be affected by that update. Ideally, the publish / subscribe system should be scalable so as to avoid introducing a new performance bottleneck, avoiding relying on centralized infrastructure if possible.

Current publish / subscribe systems do not implement subscription and publication languages that efficiently capture the complex relationship between a database update and the data that update affects. There is, however, a fundamental trade-off between the scalability of a publish / subscribe system and the expressibility of its subscription language [21]. Thus, the goal of extending publish / subscribe systems to better support database requests is antagonistic to the goal of maintaining a scalable publish / subscribe system.
In this thesis we will show that scalable publish / subscribe systems can be used to propagate update notifications efficiently to web database caches. We have designed and built a prototype Database Scalability Service (DBSS) to scale the generation of dynamic content, using database caches at proxy servers and utilizing publish / subscribe as the communication mechanism for propagating update notifications. We conquer the trade-off between the expressibility and scalability of the publish / subscribe system by exploiting the foreknowledge of a web application’s code and embedded database requests. This foreknowledge allows us to constrain the range of requests issued by a web application, enabling us to more efficiently support those requests at the expense of poorer performance for requests the application does not issue.

This thesis will make the following contributions:

1. We specialize publish / subscribe systems for the propagation of database update notifications, showing how a static analysis of the database workload can be used to obtain scalability while supporting a subscription and publication language sufficiently expressive for efficient update propagation.

2. We implement and quantitatively evaluate such a publish / subscribe system as part of the S3 DBSS, using multiple configurations of S3 and typical web database workloads as a benchmark application for the publish / subscribe infrastructure.

3. We quantitatively evaluate the costs and benefits of using a static analysis of a web application to improve publish / subscribe performance, comparing the performance obtained by our design to that obtainable by more general systems which do not exploit foreknowledge of the database workload.

The remainder of this proposal is organized as follows. Section 2 discusses the S3 Database Scalability Service, presenting our prototype implementation and alternatives in its design. Section 3 briefly introduces publish / subscribe in more detail, motivates the need for publish / subscribe for a DBSS, and describes the current state of research in scalable publish / subscribe systems. Section 4 describes our current work implementing publish / subscribe for database update notification, and section 5 outlines our future research goals and the expected time line for achieving these goals.

2 The S3 Database Scalability Service

This section describes the S3 DBSS, a prototype dynamic-web-content delivery system that utilizes publish / subscribe for propagating update notifications. We describe our prototype implementation, its initial performance results, and discuss how S3 is related to others’ work.

2.1 The S3 Prototype Implementation

A high-level diagram of S3’s architecture is shown in Figure 1. 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. Unlike previous systems, however, S3 implements a fully-distributed consistency management infrastructure, using publish / subscribe as a communication primitive.

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.

Our initial prototype uses the Scribe [?] publish / subscribe system as its group communication mechanism for update notifications. Scribe is a completely decentralized topic-based multicast system implemented on top of the Pastry [18] peer-to-peer routing system, which uses a distributed hash table based on PRR trees to route network messages to their destination.

To focus on the issues fundamental to scalability the S3 prototype uses simple implementations for its caching and consistency management policies. 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.

2.1.1 Performance Metrics

The primary performance metrics for S3 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 network traffic incurred by the consistency management infrastructure. We accomplish this by attempting to reduce the number of subscription, unsubscription, publication, and notification messages in the publish / subscribe system.

2.2 Initial Results from S3

Figure 2 compares the performance of the S3 system to two default configurations: (1) a CDN-like solution in which proxy nodes run web and application servers to generate dynamic content but no database caching is done, and (2) a system where proxy nodes cache data as with S3 but use centralized broadcast as the communications mechanism to maintain cache consistency. With the CDN-like system, throughput slightly
Figure 2: Overall database throughput achieved by the S3 scalability service.

Figure 3: Simulated S3 cache miss rates for some typical dynamic web workloads.

benefits from the addition of proxy servers since generation of content by the web and application servers are at first the bottleneck in system performance; however, as the number of proxy servers increases the centralized database quickly becomes the bottleneck and additional performance gains are not realized. When data caches are used but update notifications are broadcast to the proxy servers, performance significantly benefits from the addition of proxy nodes, but the high cost of maintaining cache consistency soon becomes a performance bottleneck and peak performance is achieved with only about four proxy nodes. S3 with a publish / subscribe cache consistency mechanism far out-scales either competing system, and for the system sizes shown here the cost of maintaining consistency never becomes a bottleneck in system performance.

Figure 3 shows the expected cache miss rates for the TPC-W bookstore, RUBis auction, and RUBBoS bulletin board benchmarks, for S3 proxy networks of 1 to 128 nodes. These results were obtained using an S3 event-based network simulator which exactly tracks the cache and consistency-management state for each proxy, using database traces from actual executions of the various benchmark applications. As the number of proxy nodes increases the system supports a greater overall load, which increases the frequency of invalidations experienced by any single proxy cache. However, the frequency of a query at any single proxy cache is independent of the network size, and thus the cache miss rate increases as more nodes are added. For each workload, the miss rate seems to increase asymptotically as frequently updated items become increasingly rare in each proxy’s cache, but infrequently updated items remain present and result in cache
hits. For large networks the TPC-W bookstore results in a miss rate of about 20 percent while the auction and bulletin board benchmarks each reach a miss rate of about 35 percent.

These miss rates strongly suggest that even when caching is used, the ability of the proxy servers to shield load from the database is limited. When publish / subscribe is used as the cache consistency communications mechanism, throughput at the central database is likely to be the overall performance bottleneck.

Together, these results indicate that the key to improving performance of the S3 system is to improve the database cache hit rate at the proxy servers while avoiding broadcast communication to maintain cache consistency.

2.3 Others’ Work Related to S3

The database, filesystems, and middleware communities have all addressed the general problem of maintaining the consistency of replicated data in a multi-reader multi-writer environment, and overall the body of this work is too great to directly address. In particular, Gustavo Alonso [7, 11, 17, 1, 12, 15], Cristiana Anza [2], Arun Iyengar [8, 6, 10], Jim Challenger [5, 4, 6, 10], Mike Dahlin [8, 6, 12, 15], Alexandros Labrinidis [2], and Marta Patiño-Martínez [12, 15] have all made significant contributions toward supporting the scalable, consistent generation of dynamic web content.

Alonso, Kemme, Martínez, Dahlin, and Anza have each advanced the state-of-the-art in protocols to maintain consistency in a distributed environment. The first three have implemented GlobeDB [7], which explicitly enables weak consistency management for replicated web databases. In contrast, Dahlin’s work primarily seeks to divorce consistency management policy from mechanism. Anza has implemented 1-copy serializability for replicas of web databases using table- and column-level granularity for conflict-checking.

Challenger and Iyengar have primarily focused on the caching of the end results of a dynamically generated page, not the data used to generate it. In their work, they exploit data dependency analyses to determine when a page – or parts of a page – need to be regenerated when the underlying data is updated, and have even utilized a publish / subscribe system to propagate update notifications. They have not, however, explored the more general problem of using publish / subscribe to propagate database update notifications to database caches. Iyengar’s work does exploit foreknowledge of a web application’s code, using a static analysis to derive data dependencies much as we do. Labrinidis has studied the performance tradeoff of alternative strategies of caching and regenerating dynamic data: from no caching, caching at the database (view materialization) level, or caching of the final generated dynamic page.

Several projects have previously applied the edge-computing paradigm to the generation of dynamic web content, including IBM’s DBCache [7], IBM’s DBProxy [7], NEC’s CachePortal [7], and GlobeDB [7]. The first three projects do not support any distributed consistency management, and instead rely on centralized infrastructure to track the contents at each proxy server and forward update notifications to the relevant proxies. In contrast, GlobeDB does enable scalable, distributed replication, but only implements a weak form of consistency, essentially disallowing multi-statement transactions for strongly consistent data.

3 State-based Publish / Subscribe for Web Database Caches

This section discusses the requirements on the publish / subscribe system for the consistency of web databases. We start by introducing publish / subscribe and motivating that it is the right architecture for what we want. We then describe the current state of publish / subscribe technology and describe our work in building a publish / subscribe system for propagating updates to web database caches.

3.1 A Brief Introduction to Publish / Subscribe

Publish / subscribe is a relatively new and extremely popular paradigm in modern distributed systems. In general, publish / subscribe systems support two basic operations: subscribe($S$), where $S$ is a subscription in some form, and publish($O$), where $O$ is an object of some type. A subscriber receives notification of a published object $O$ if the $O$ ”matches” their subscription, for some notion of matching. The semantic complexity of allowed subscriptions, the complexity of published objects, and semantics of publication-subscription matching may all vary between different publish / subscribe systems.
3.1.1 Why Publish / Subscribe?

There are a few basic strategies for propagating updates in a partitioned, replicated environment: broadcast, a central repository tracking replica contents, selective partitioning of replica data, or publish / subscribe.

Broadcast is the most basic propagation strategy, in which each cache is sent a copy of each update, regardless of whether that update affects data in the cache. Broadcast may work well when updates are very infrequent, affect a large fraction of caches, or there are a small number of caches, but in practice broadcast is a poor choice for most data intensive applications.

Compared to broadcast, network demand may be significantly reduced by tracking the contents of each cache by some central directory. When an update occurs, the system then may consult the directory and forward updates to only those caches affected by the particular result. Using a central directory effectively reduces network load, but at the expense of performing additional computation at some central location. When processing at the central server is the performance bottleneck in the system – as is the case for many data-intensive applications – using a central directory only exacerbates the problem.

One very appealing alternative is to simplify the tracking of replica contents by carefully partitioning the data and deterministically assigning different portions of the cache to different proxy servers. Using this selective partitioning approach, determining which proxies contain which data can be a trivial process, and selective partitioning is the fundamental principle behind the construction of some highly scalable databases, like federated systems.

However, partitioning replicated data to deterministic locations is an administrator-intensive process and can be difficult or impossible with some database and application designs. Some recent work explores automatic tools for partitioning data [?], but even this is contrary to the general principle of placing data near the end-user rather than at arbitrary locations in the network. In practice, the need to lower the latency of database interactions by checking only contents in the local cache can be critical to system performance.

In contrast, the goal of publish / subscribe is to perform the function of a central directory and the function of message delivery. Some publish / subscribe systems rely on a central directory to match publications to subscriptions, but in practice most implementations avoid a central implementation to prevent a centralized performance bottleneck. Some of the simplest publish / subscribe systems essentially consist of a distributed repository tracking cache contents, and a basic communication infrastructure for delivering messages.

3.2 Publish / Subscribe for Web Database Caches

Each database cache consists of the union of some set of SQL query results. When a proxy inserts a new query $Q$ into its cache, the proxy should execute a subscribe($Q$) call to publish / subscribe system, signaling that the proxy should now receive all updates that affect $Q$. When a proxy issues an SQL update $U$ it should execute publish($U$). The publish / subscribe system is then responsible for ensuring that any subscriptions $Q_i$ should be notified of publication $U$ if the update $U$ could potentially affect query $Q_i$.

Our architecture requires that the publish / subscribe system be decentralized so as to avoid becoming a performance bottleneck. However, one major difference between a traditional publish / subscribe system and our needs is that some unnecessary notifications – notifications that occur for a publication that doesn’t actually match a particular subscription – are okay as long as the over-notifications don’t themselves become a performance bottleneck.

Thus, the key goal of our publish / subscribe work is to build a scalable publish / subscribe system with SQL as both the subscription and publication language, that matches a publication $U$ to a subscription $Q$ if, for a given database state, $U$ affects $Q$.

3.3 The Current State of Publish / Subscribe

There are two major paradigms in modern publish / subscribe systems: topic-based systems and content-based systems. We also discuss state-based publish / subscribe, a relatively new paradigm in publish / subscribe that closely matches our requirements for propagating database update notifications.
3.3.1 Topic-based Publish / Subscribe

In a topic-based system, clients may only subscribe to simple topics, or groups, and all clients in a group receive all publications to that group. Some systems implement publication-batching and other such modifications to improve system performance, but in general no additional filtering of publications is done and any processing of the published objects must be done at the client after a subscriber is notified of the publication.

An example application of topic-based publish subscribe is a simple stock ticker. In this application, clients may subscribe to a simple topic for each stock, for example "GOOG" or "MSFT". Any updates to the stock price could then be published to the appropriate stock topic, and all interested subscribers would then receive such updates.

In the spectrum of publish / subscribe systems, topic-based systems are the least complex and potentially the most scalable. One highly scalable topic-based implementation is Scribe [?], a fully decentralized system.

3.3.2 Content-based Publish / Subscribe

Content-based publish / subscribe systems implement more complex subscription and publication languages, and the system may match a publication to a subscription based on the contents of the published object. In a content-based system, the above stock-ticker example may allow filtering of messages based on the current stock price, eg. a subscription such as "GOOG, price > 500" would allow a client to receive only publications of updates that set GOOG’s price above $500.

Content-based publish / subscribe systems are inherently more complex, and potentially less scalable, than similar topic-based systems. However, the realm of content-based systems is not as uniform as topic-based systems, and content-based systems may implement subscription and publication languages of varying complexity. The most basic content-based systems allow only subscriptions based on simple equality and inequality predicates on publications, such as the price example in the stock-ticker application above; these systems may be implemented with highly scalable hierarchical or hybrid approaches like Hermes [16]. Other systems may match publications and subscriptions based on more complex properties of the subscription and publication objects, for which only centralized matching algorithms are known. One such system would be Tomasic et al.’s SPS system. [19]

3.3.3 State-based Publish / Subscribe

In topic-based and content-based publish / subscribe systems, only the content of the subscription and publication at hand is considered in determining whether a publication matches a subscription. With state-based publish / subscribe the match algorithm may additionally take into account some external database state. For example, one possible subscription with state-based publish / subscribe might be "GOOG, MAX(price)”, in which case the subscriber would only be notified of publications for which GOOG’s stock price exceeded the previous maximum GOOG price. Such a subscription would not be directly possible using either a topic-based or content-based system.

The potential complexity of state-based publish / subscribe is inherently higher than for either topic-based or content-based systems. Like content-based systems, the potential scalability of a state-base system may vary widely based on the complexity of the particular subscription- and publication-matching algorithm it allows.

State-based publish / subscribe systems were recently independently introduced by Chandramouli et al. [?], and it is their work that most closely resembles ours. They examine state-based publish / subscribe systems from the standpoint of subscription and message reformulation, in which stateful subscriptions and publications are transformed into a stateless form, which are then implemented using an existing topic-based or content-based publish / subscribe system. To perform the reformulation, the state-based system maintains and updates whatever external state necessary to convert subscriptions and publications to an underlying stateless form. When notification occurs in the underlying stateless implementation, their state-based system references the external state to determine whether a match, and hence a notification, actually occurs in the state-based system.

As a trivial example, one such reformulation state would be to include any reference to the external state as part of each subscription, and augment all publications with a copy of the entire database state. These
subscriptions and publications could then be directly matched with a content-based publish / subscribe system. This procedure, however, would of course be infeasible in practice, as including the entire database state in every message would overwhelm the system for most applications.

Chandramouli’s work is motivated by continuous-query processing within stream database systems, and their overall system requirements are similar to our own. In particular, Chandramouli describes a scalable implementation for the reformulation of some stateful SQL aggregates that could be directly applied to our own state-based publish / subscribe system.

4 The S3 Publish / Subscribe System

In our initial S3 prototype we chose the highly scalable Scribe system as an underlying publish / subscribe implementation, and addressed the question of how to reformulate the complex state-based publication and subscription languages required for update notification into the simple topic-based paradigm supported by Scribe. This section describes our reformulation approach. In particular, we show how we use a static analysis of a web application’s database workload to develop a reformulation strategy, introduce two such reformulation approaches based on static analyses, and evaluates the merits of each approach using a simulation of the S3 system.

4.1 Static Analysis of Dynamic Web Applications

The key observation is that 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 reformulation 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 support a new web application we first inspect its database templates and apply offline query-update independence analysis [?] 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.

Intuitively, a good reformulation maps queries into the same topic groups if they are affected by the same updates, and avoids mapping unrelated queries into the same topic groups. This yields a natural association between topic groups and data objects. Although in principle a topic group might be associated with any arbitrary data object, here we consider two fundamental strategies: (1) topic groups based upon the data on which a query depends, which we call Group-by-query, and (2) topic groups based upon the data an update affects, or Group-by-update.

4.1.1 An Example Database Application

To illustrate these strategies 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 4 gives correct reformulations for the example inventory application based on the Group-by-query and Group-by-update strategies. A question mark in the topic 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 topic name contains no parameters that are bound at run-time, and otherwise that it is parameter-dependent.

First suppose that the Group-by-query reformulation is used 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 topics 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 topics update1: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 topics 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 topics, one for update1:name=spoon and one for update1.

### 4.2 Comparing Group-by-query and Group-by-update

The key static property of a reformulation strategy is the number of topic groups to which each database request belongs. On average, we expect (1) queries to belong to 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 topics 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 topic groups 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.

In this section we use an S3 simulator to compare Group-by-query and Group-by-update on typical web database workloads. We first describe the S3 simulator and the workloads we use to compare the two strategies, and then present our simulation results.

#### 4.2.1 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 2.1.

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.

### 4.2.2 Experimental Workloads

As input we use database traces from short executions (5-10 minutes) of the TPC-W bookstore [20] and RUBBoS bulletin board [13] benchmarks. Figure 5 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 [3]. 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.

### 4.2.3 Results from the S3 Simulator

Figure 6 shows the number of topic-based subscriptions sent per query for each benchmark, 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.

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 belongs to 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 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 topic group we may expect Group-by-update to scale more poorly than Group-by-query, an unanticipated result.

Figure 7 shows the number of topic-based notifications delivered per update for each benchmark. The TPC-W bookstore requires far fewer notifications per update than the RUBBoS bulletin board, for which

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>DB size</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPC-W</td>
<td>217 MB</td>
<td>10,000 items</td>
</tr>
<tr>
<td></td>
<td></td>
<td>86,400 registered users</td>
</tr>
<tr>
<td>RUBBoS</td>
<td>1.4 GB</td>
<td>213,292 comments</td>
</tr>
<tr>
<td></td>
<td></td>
<td>500,000 users</td>
</tr>
</tbody>
</table>

Figure 5: Configuration parameters for each benchmark.
Figure 6: Simulated subscriptions per query

Figure 7: Simulated notifications per query
updates are more likely to affect global data. In both cases here Group-by-update results in fewer notifications than Group-by-query. Also notable is that both configurations scale well compared to traditional broadcast mechanisms, since even in the worst case only about 15% of the proxy servers receive update notifications.

4.3 Conclusions from the S3 Publish / Subscribe System

Overall, our simulations demonstrate that neither the Group-by-query nor Group-by-update reformulation strategy dominates the other for all workloads. For the TPC-W bookstore, Group-by-query is better since it significantly outperforms Group-by-update for subscriptions but is only slightly worse for publications, and the overall workload is query-dominated. For the 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. Thus, our main conclusion is just that there is no "silver bullet" reformulation for all web database workloads.

4.4 Others’ Work Related to our Reformulation Techniques

No one has previously explored the use of a workload analysis in the reformulation of a state-based publish / subscribe system into a stateless implementation. However, several groups have independently considered a static analysis of a dynamic web application as a means to improve performance in a distributed or otherwise replicated setting.

In particular, Gao et al. rebuild the TPC-W benchmark to use carefully constructed distributed objects rather than its usual centralized model, allowing them to maintain strong consistency of a TPC-W execution in a distributed environment and thus improving its scalability [8, 9]. While their work demonstrates the feasibility of developing scalable web applications with strong consistency, however, they fail to demonstrate how their analysis and techniques might be automatically applied to general web applications, and in fact it seems that their optimizations are very TPC-W specific rather than general techniques.

Choi and Luo use a static data analysis very similar to ours to determine a mapping between SQL requests and URLs for dynamic web pages, and apply that mapping to the problem of maintaining coherence of a dynamic web cache [7]. However, they only consider the problem of central consistency maintenance and do not consider using template-based data analysis to optimize update notification in a distributed publish / subscribe-based setting. As such, their technique is analogous to using template-based data analysis to improve the performance of a central directory of cache contents, and their results admit that the central processing load remains the bottleneck of their system.

We have ourselves previously used static analysis to improve the scalability of a DBSS in the face of security constraints, using a static data analysis to evaluate the cost maintaining the consistency of encrypting replicated data [10].

5 Future Work

In my thesis I intend to continue work in three major directions: (1) extending the class of database requests efficiently supported by a publish / subscribe infrastructure, (2) formally evaluating the costs and benefits of using static analysis in the deployment of the communications infrastructure used to propagate update notifications, and (3) improving the cache performance of the S3 Database Scalability System.

5.1 Extending State-based Publish / Subscribe

While existing publish / subscribe systems can be used to propagate database update notifications, they only do so efficiently for a fraction of possible database requests. For instance, the reformulation into topic-based publish / subscribe used by S3 could efficiently propagate update notifications just for queries and updates that utilized only equality-based constraints in their select clause. More sophisticated content-based publish / subscribe systems allow select clauses containing simple inequalities, at the expense of requiring slightly more centralized infrastructure. Chandramouli et al. have extended the class of queries efficiently
supported by publish / subscribe systems to include some queries involving aggregates, notably range-min and range-max aggregates \cite{7}.

In this thesis I propose to extend the class of database requests efficiently supported by state-based publish / subscribe. My work will focus on enabling efficient update notification for database requests containing joins and aggregations not currently supported by Chandramouli et al., although I will also consider ways in which a static analysis of the application code and embedded database requests may offer an improvement over their existing techniques.

Specifically, my work will start with an analysis of public data-intensive web applications and benchmarks to identify candidate classes of database requests (1) which are commonly used in existing applications, (2) are not currently well-supported by publish / subscribe systems, and (3) for which a through examination of the data relationships between queries and updates indicates that effectively extending publish / subscribe to support the class of requests may be feasible. The S3 simulator will be a key tool in the determination of the potential value of accommodating a particular query class; it will reveal both the prevalence of the the query class and the amount of update traffic that could potentially be eliminated for a sample workload.

After selecting the top one or two candidate query classes I will design and implement the extension to the publish / subscribe mechanism and integrate the augmented system into S3. The ultimate measure of the success of my implementation will be the database throughput achieved by the S3 system and network traffic required for update propagation, as per the metrics discussed in Section 2.1.1.

5.2 Evaluating the costs and benefits of static analysis

Our initial work made extensive use of a static analysis of application code and associated database requests. However, we have not quantitatively evaluated the benefits obtained from that static analysis as compared to the performance that could be obtained from a reasonable run-time analysis of incoming database requests. We similarly have not examined the costs of such an analysis, such as the additional costs imposed by the evolution of the database schema or application code, or the possible presence of ad-hoc updates (updates that are not part of the code statically analyzed by the system).

I propose to formally evaluate the costs and benefits of the static analysis. This section of the thesis consists of several sub-components: (1) comparing the performance obtainable using static analysis to that obtainable using a reasonable dynamic analysis, such as that of Chandramouli, (2) development and evaluation of reasonable methodologies for handling the evolution of the database schema and application code, and (3) the development and evaluation of reasonable methodologies for handling ad-hoc updates.

The first component – quantitatively comparing the performance of publish / subscribe systems that exploit static analysis to those that do not – will primarily utilize the S3 simulator and sample workloads from existing web applications and benchmarks. This stage will require some slight modification to the existing simulator implementation.

The handling of database schema evolution and ad-hoc updates primarily will be an analytical exercise, with some empirical results obtained from micro-benchmarks of our creation. These micro-benchmarks will consist of examples of schema changes or non-standard updates that an administrator may occasionally make to an existing database, to be determined by informally examining common scenarios with real-life databases. In practice, schema and application changes to real-life databases and web applications consist of the addition of tables or columns, the addition of new queries or updates, or the modification of existing queries to augment the attributes contained in the result set.

Overall, the utilization of static analysis in the development and execution of the publish / subscribe system will be considered a success if (1) the performance gains resulting from static analysis is at least a non-trivial constant factor, and (2) the performance cost of executing a schema change or ad-hoc update every 2-4 weeks does not negate the expected performance gains.

5.3 Improving the Cache Performance of the S3 DBSS

As discussed in sections 2.2, one of the limiting factors in S3 performance is the cache hit rate that S3 obtains. With miss rates of 10-20%, only a scalability of 5-10 times can conceivably be obtained.

There are several possible modifications to the S3 design that may result in significantly better caching performance. Two such optimizations are update propagation and cooperative caching.
5.3.1 Update Propagation

In the current implementation, the S3 DBSS invalidates cache entries upon receipt of an update notification. An alternative is update propagation, in which the cache would instead modify the cached entry to reflect the update rather than invalidate it.

The relative advantages of update propagation and invalidation have been studied previously by Labrini-dis [?] and others. Although we have not yet explored the addition of update propagation to the S3 system, we expect it to result in a much higher cache hit rate than S3 with view invalidation, at the expense of slightly more notification traffic for the updates of views that would have been previously invalidated.

5.3.2 Cooperative Caching

S3 currently implements a caching system where each proxy only checks its own cache for a query result, and otherwise forwards the query to the central database system. An alternative design might allow a proxy to check the cache at one or more other proxies for the result before forwarding the request to the back-end database.

Scalable cooperative caching schemes have been well-studied, eg. in [9], and the development of new cooperative caching designs is beyond the scope of this thesis. However, any interaction between the caching system and invalidation system is relevant to this thesis.

We have considered the incorporation of a distributed cooperative cache into S3’s architecture in the following manner. When a query is executed at a proxy cache, the proxy first checks its own cache and if the query result is present returns the result. If the result is not present, rather than forward the query to the back-end database the query is instead forwarded to a cooperative distributed cache, eg. by lookup in a distributed hash table. If the query result is present in the cooperative cache that result is returned, and otherwise the query is finally forwarded to the back-end database server.

This hierarchical cache architecture is compatible with our use of publish / subscribe as the mechanism for update notification without significant modification.

5.4 Time Line

<table>
<thead>
<tr>
<th>Task</th>
<th>Total time</th>
<th>Goal date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Utilize other DB state</td>
<td>30 days</td>
<td>15 Jan 2007</td>
</tr>
<tr>
<td>Richer pub/sub</td>
<td>25 days</td>
<td>28 Feb 2007</td>
</tr>
<tr>
<td>Cost model for ad-hoc updates</td>
<td>5 days</td>
<td>15 map 2007</td>
</tr>
<tr>
<td>Evaluate static analysis</td>
<td>40 days</td>
<td>30 Apr 2007</td>
</tr>
<tr>
<td>Cooperative caching</td>
<td>40 days</td>
<td>30 Jun 2007</td>
</tr>
<tr>
<td>Implementation correction</td>
<td>60 days</td>
<td>30 Sep 2007</td>
</tr>
<tr>
<td>Thesis</td>
<td>60 days</td>
<td>31 Dec 2007</td>
</tr>
</tbody>
</table>

References


