A Scalability Service for Dynamic Web Applications

Christopher Olston, Amit Manjhi, Charles Garrod,
Anastassia Ailamaki, Bruce M. Maggs, Todd C. Mowry

School of Computer Science
Carnegie Mellon University

Abstract

Providers of dynamic Web applications are currently unable to accommodate heavy usage without significant investment in infrastructure and in-house management capability. Our goal is to develop technology to enable a third party to offer scalability as a subscription service with "per-click" pricing to application providers. To this end we have developed a prototype proxy caching system able to scale delivery of dynamic Web content to a large number of users. In this paper we report initial positive results obtained from our prototype that point to the feasibility of our goal. We also report the shortcomings of our current prototype, the chief one being the lack of a scalable method of managing data consistency. We then present our initial work on a novel approach to scalable consistency management. Our approach is based on a fully distributed mechanism that does not require content providers to assist in managing the consistency of remotely cached data. Finally, we describe our ongoing efforts to characterize the inherent tradeoff between scalability and data secrecy, a crucial issue in environments shared by multiple organizations.

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 (DoS) attacks. Administrators currently face a dilemma: whether to (a) waste money by heavily overprovisioning systems, or (b) risk loss of availability during critical times. This problem is largely addressed for static content by Content Distribution Network (CDN) technology [9], which offers dynamic scalability as a plug-in service. CDN's make use of a large, shared infrastructure to absorb load spikes that may occur for any individual content provider. Hence, they can offer seemingly unlimited scalability and charge content providers on a per-usage basis.

CDN technology does not meet all of the needs of richly interactive applications such as online classrooms, bulletin boards, civic emergency management, and e-commerce, which are representative of the future landscape of the Internet and naturally implemented as "dynamic" Web applications. At Carnegie Mellon we are developing the foundations for offering scalability as a plug-in subscription service to dynamic Web applications. Such a service would enable cost-effective deployment of richly interactive online applications.

1.1 Example Scenarios

We illustrate the potential benefits of subscription-oriented scalability services for dynamic Web applications with two example scenarios.

1.1.1 E-Commerce

Consider a relatively small-scale Web-based e-commerce operation whose customer base is expanding. New customers bring more revenue, but may also lead to major management difficulties behind the scenes. Suppose the relatively low-cost equipment on which the e-commerce site was originally built (with the company's meager start-up funds) is becoming saturated with load, and will soon be unable to service all the customers. Standard solutions include upgrading to faster equipment on which to run the web, application, and/or database servers, or moving to a parallel cluster-based architecture as used by big e-commerce vendors. Unfortunately, these solutions require a large investment in equipment, and, perhaps more significantly, funding

Proceedings of the 2005 CIDR Conference

for staff with the expertise necessary to manage the more complex infrastructure. Moreover, transitioning to a new architecture will undoubtedly create new bugs and may lead to costly application errors or system downtime.

A much more palatable option would be to subscribe to a scalability service on a pay-by-usage basis. A cost curve proportional to usage could potentially save the company large sums of money, especially if demand flattens out or drops. In this scenario, the equipment and management costs are shifted to the scalability service provider, where they can be amortized across many subscribers. Of course, in a shared environment privacy and security are central requirements.

1.1.2 Civic Emergency Management

Suppose the local government of a large city, such as Chicago, is ordered to prepare a response plan in case of a natural disaster. The government would like to have the capability of providing each citizen, even after the event has occurred, with both general and individualized instructions on how to protect themselves. In particular, the city would like to be able to provide maps and directions for each citizen explaining where to find medical treatment, shelter, uncontaminated food and water, etc. In addition, the city would like to be able to collect requests for immediate medical treatment from citizens who are immobile, and to collect reports from citizens and professionals about the effects of the incident in various sections of the city.

This application lends itself naturally to a Web-based implementation, but there are several inherent difficulties. First, demand for the application is likely never to occur, but if it should occur, it will come very quickly and in a very large dose. It would be costly for the city to invest in enough permanent infrastructure to satisfy the demand, and not cost-effective to keep this infrastructure idle. Second, it is critical to give all end users prompt and reliable access to information (recall the frustration of end users who were unable to retrieve even general news from http://www.cnn.com on September 11, 2001, because CNN was not utilizing the services of a CDN that morning [26]). It is even more important that data collected from end users requiring immediate assistance be recorded reliably. Third, the delivery of information customized to each end user, such as the generation of maps and directions, requires significant computational resources. This information cannot easily be conveyed through a telephone conversation, and in any case, the scale of the demand would make a call-center solution impractical.

The ability to tap into a secure scalability service would alleviate these difficulties. The city would have to prepare software in advance and maintain a modest amount of permanent infrastructure, but could rely on the scalability service when demand suddenly arrived. The scalability service would then shoulder the network and computational load, even while potentially serving other, also critical, applications.

1.2 Creating a Scalability Service for Dynamic Web Applications

The above example scenarios illustrate the potential benefits of a secure plug-in scalability service for current and future dynamic Web applications on the Internet. Constructing such a service for dynamic applications is much more challenging than doing so for traditional static content.

1.2.1 Challenges

Dynamic Web applications are characterized by capabilities for personalization and distributed updating of data. These features, coupled with the often sensitive nature of the associated data, create new systems and security challenges in building an effective scalability service for dynamically-generated Web content. Web applications are typically deployed on a three-tiered server-side architecture consisting of one or more instances each of: a web server, an application server, and a database server. Most advanced Web applications rely on the database server(s) for the bulk of the data management tasks, and indeed the database servers often become the bottleneck in terms of maximum supportable load. Consequently, an effective scalability service would have to offload at least some of the database work from the home organization. However, that task is encumbered by two major difficulties inherent in dynamic Web applications:

1. Most advanced Web applications require strong consistency for their most important data. For example, in our civic emergency management scenario (Section 1.1.2), inventory data for emergency supplies must be managed precisely—inconsistencies could cost lives. It is well-known that maintaining strong consistency among replicas in a distributed setting presents significant scalability challenges [12].

2. Administrators are typically reluctant to cede ownership of data and permit data updating to take place outside the home organization. This reluctance arises with good reason, due to the security concerns, data corruption risks, and cross-organizational management difficulties entailed.

Difficulty 1 precludes caching techniques based entirely on timed data expiration, i.e., *time-to-live* (TTL) protocols [8], which are the norm in current approaches to scaling the delivery of static Web content. The growing number of dynamic Web applications with sensitive and mission-critical data require a high degree of data fidelity and rely on systems that adhere to the transactional

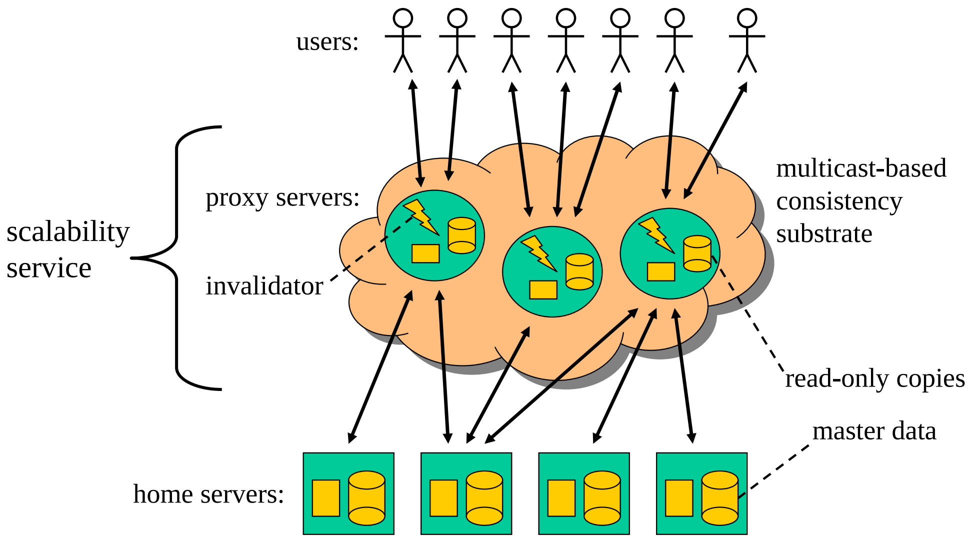


Figure 1: Envisioned high-level scalability service architecture.

model of consistency. Transactional consistency is supported by a plethora of distributed data replication algorithms (e.g., [14]), but these are not a good fit because of Difficulty 2. In many cases, capabilities for data updating simply must remain within the boundaries of the home organization, and decentralized data updating is simply unacceptable. Our project explores a middle ground between TTL-based caching and fully distributed replica management, to achieve good scalability while retaining strong consistency and centralized ownership of data.

1.2.2 Our Approach

Our approach exploits two properties shared by many Web applications: the underlying data workloads tend to (1) be dominated by reads, and (2) consist of a small, fixed set of query and update templates (typically 10-100). The first property makes it feasible to handle all data updates at servers within each application's home organization. That way, no data updating is performed outside of the home organization, and tight control over authentication of updates and overall data integrity is retained. By exploiting the second property, i.e., predefined query and update templates, we believe we can create a fully distributed mechanism for enforcing strong cache consistency that does not burden home organizations with this responsibility.

The architecture we envision for our scalability service is illustrated in Figure 1. At the bottom we see *home servers*, which host code and data for individual applications within the boundaries of their home organizations. Scalability is provided by a collection of cooperating *proxy servers* that cache data on behalf of home servers. Users access applications indirectly, by connecting to proxies within the scalability service network. At the proxy servers, application data is cached on a strictly read-only basis, ensuring that server failures do not affect data integrity or require expensive quorum protocols. All data update requests are forwarded directly to application home servers for local processing. Cache consistency is managed entirely by the proxy servers themselves, which notify each other of updates via a fully distributed multicast network.

Distributed Consistency Management
Our distributed consistency mechanism combines two technologies: query/update independence analysis [17] and distributed multicast [7, 28, 34]. Query/update independence analysis is concerned with deciding whether a given update affects the result of a particular query. Distributed multicast environments offer efficient distributed routing of messages to multiple recipients based on application-level concepts rather than network addresses.

In our approach, application code is first analyzed statically to identify pairs of embedded query and update templates that are in potential conflict, meaning that an instantiation of the update template might affect the result of an instantiation of the query template. At runtime, when an update template is invoked at a proxy server, the precomputed set of potentially conflicting queries is narrowed based on the parameter bindings supplied with the update. At that point it must be ensured that any cached results of conflicting queries are either updated or invalidated. Hence, all proxy servers caching conflicting data must be notified of the update.

To limit dissemination of updates to just the servers that cache potentially conflicting data, our approach leverages distributed multicast technology. Proxy servers organize themselves into an overlay network, and transmission of updates is handled via a group multicast environment built into the overlay. Multicast groups are established in correspondence with query templates embedded in the application, and servers caching data for a particular query template subscribe to the corresponding multicast group(s). When an update occurs, notification of that update is routed by the multicast protocol to any and all servers caching data that may be affected. Our project will assess the feasibility of using multicast as a substrate upon which to construct a large-scale distributed data consistency mechanism, which to our knowledge has not been attempted before.

Cache Invalidation
When a proxy server receives notification of an update that may conflict with locally-cached data, some action must be taken to ensure consistency with the master copy maintained by the home server(s). Since one of our design principles is to avoid distributed updating of data, we plan to rely on *invalidation*, in which potentially inconsistent data is evicted from caches. In our design (Figure 1), each proxy server is equipped with a local *invalidator* module, which decides what data to evict in response to notification of an update. Cached data is never updated locally, so home organizations can more readily monitor and control the security and integrity of their data. Authentication of update requests need take place only at home servers. (While unauthenticated updates may induce spurious invalidation of cached data, the impact is only on performance, not correctness.)

There is an important tradeoff to be considered in the design of invalidator modules. Clearly, to achieve the best scalability, a minimal number of data invalidations should be performed by proxy servers. The most selective invalidation strategies require inspection of the content of cached data. However, the presence of data originating from databases managed by different DBMS products severely complicates this process, and it is well-known that increased implementation complexity tends to degrade reliability. In addition, allowing proxy servers to inspect the content of cached data precludes the use of certain cryptographic approaches for ensuring privacy and security. We are working toward formally characterizing the tradeoff between scalability, on the one hand, and the combination of low implementation complexity and secrecy of data on the other hand.

1.3 Outline

The remainder of this paper is structured as follows. First, we discuss related work in Section 2. Then, in Section 3, we describe our working prototype and report some preliminary results obtained via experiments on benchmark dynamic Web applications. Our current prototype employs a TTL-based approach to consistency management, and strong consistency can only be achieved by setting TTL's to zero, which negates the benefits of caching. In Section 4 we describe some of our ongoing work on developing a fully distributed mechanism to achieve strong (transactional) consistency in a scalable manner. Then, in Section 5 we present our initial work on local invalidation, in which we seek to formally characterize the tradeoff between scalability and the combination of data secrecy and low implementation complexity. Our plans for future work are outlined in Section 6, and we summarize the paper in Section 7.

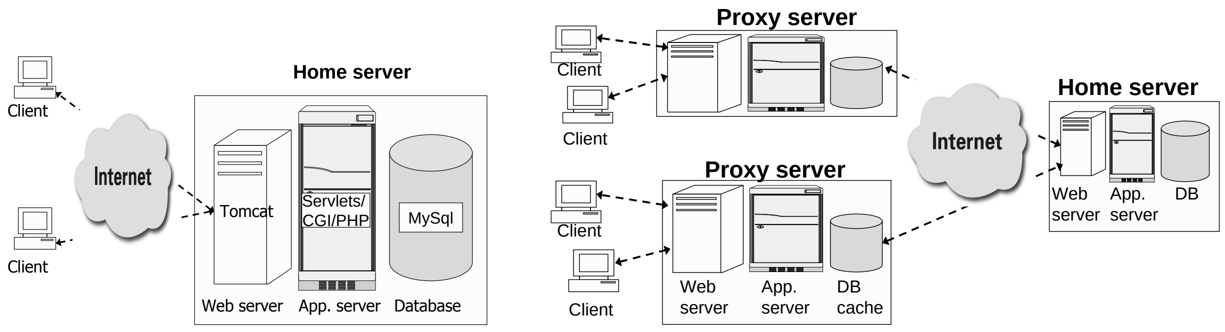
2 Related Work

Remote caching of database objects first received significant attention during a flurry of research activity on client-server object-oriented databases that began in the late 1980's. A prominent research question in that context was whether the central server should maintain consistency by invalidating data cached at remote clients, or by propagating changed data to clients (e.g., [11]). To our knowledge the work in that area did not consider fully distributed data invalidation or update propagation methods.

More recently, database caching has been investigated as a means to scale the delivery of dynamically-generated Web content. Ongoing efforts in this area include the DBCache [1, 19] and DBProxy [3] projects at IBM Research, and the CachePortal project [18] at NEC Laboratories. To ensure consistency, these approaches rely on propagation of data updates from the home organization's local database to exterior cache nodes. While there has been some recent work devoted to developing more efficient means of handling consistency maintenance at a centralized server [4], to our knowledge researchers in this area have not considered distributed consistency management mechanisms.

Distributed consistency management has been studied extensively in the context of distributed and federated databases, which have received a great deal of attention from the database research community over the past two decades. The work in this area most closely related to our own is on consistency management for widely replicated data. Most approaches permit distributed updating of data replicas, and rely on propagation (rather than invalidation) of updated data to all nodes to ensure consistency; see, e.g., [14]. Furthermore, most work in the data replication area aims to support general-purpose, ad-hoc querying and updating, in which arbitrary SQL statements can be executed over the data. In contrast, our approach specifically leverages the fact that in our context, querying and updating is mainly restricted to templates specified in advance.

There has been some prior work pertaining to invalidation of cached materialized views. Candan et al. [6] introduced techniques for deciding whether to invalidate cached views in response to database updates. These techniques leverage "polling queries" to inspect portions of the database not available in the materialized view. The need to invalidate a view in response to a particular update can in some cases be ruled out by analysis of the view definition and update statements alone, without inspecting any data. Levy and Sagiv [17] provide methods



(a) Traditional centralized architecture. (b) Distributed architecture.

Figure 2: Traditional versus distributed architecture.

of ruling query statements (and hence view definitions) independent of updates in many practical cases, although the general query/update independence problem is undecidable. With our work, the focus is not on developing new strategies for deciding whether to invalidate cached views. Rather, we develop a formal characterization of view invalidation strategies in terms of what data they access, as a basis for studying the tradeoff between scalability and data secrecy.

2.1 Akamai EdgeJava Product

Akamai Technologies, a leading CDN, has recently released its “EdgeJava” product, which allows Web content providers to execute Java servlets on Akamai’s proxy servers. The largest significant use of EdgeJava to date was a widely advertised promotion staged by Logitech Corporation, in which peak demand exceeded 60,000 user requests per second. For each request, a Java servlet dynamically generated an HTML document, indicating whether the end user was a winner, which was then served to the end user’s browser. Other Akamai customers have used EdgeJava to perform server-side transformations of XML to HTML. Akamai provides weak consistency for cached data via TTL-based protocols, with the option of associating “do-not-cache” directives with objects that require strong consistency. Akamai has not, as yet, explored the possibility of applying database query/update independence analysis or distributed invalidation mechanisms to enforce strong data consistency.

3 Initial Prototype

As a proof-of-concept demonstration and to expose fruitful avenues for research, we have built a prototype scalability service for dynamic Web applications. Our prototype enables application providers to offload the dynamic generation of Web content to a distributed network infrastructure. From our preliminary experimental results, we have identified key shortcomings in the current implementation that motivate our ongoing work described in Section 4. Before presenting these results, we first describe our prototype implementation and benchmark applications.

In contrast with previous work on distributed generation of dynamic Web content, we explore an approach that replicates all three tiers (web server, application server, back-end database) of the traditional centralized Web architecture. The traditional architecture is illustrated in Figure 2a (here all three tiers are depicted as executing on a single home server; in general they may be spread across multiple physical servers). As in the centralized architecture, our distributed architecture (Figure 2b) includes a home server for each application provider, located within the provider’s home organization. In the distributed architecture, the home server primarily houses code and data, while a set of proxy servers processes client requests and executes programs. A client request is sent to a proxy server which then generates the appropriate response by a running a program, accessing locally cached code and data when possible, and obtaining additional code and data from the home server as necessary.

In the following subsections we describe the design choices we made while building our prototype. We first present the design of the three distinct types of nodes in our system: home servers, proxy servers, and clients.

3.1 Home Servers

Each home server embodies the traditional three-tiered architecture, which enables it to generate Web content dynamically. While we aim to offload the generation of dynamic content to the proxy servers, our prototype is compatible with this well-established home server architecture and hence also allows the application provider to serve directly as much dynamic content as it chooses. The top tier is a standard *web server*, which manages HTTP interactions with clients. The web server is augmented with a second tier, the *application server*, which can execute a program to generate a response to a client’s request based on the client profile, the request header, and the information contained in the request body. Finally, the third tier consists of a *database server* that the application provider uses to manage all of its data.

In our prototype each home server is implemented as follows. We use Tomcat [16] in its stand-alone mode as both a web server and a servlet container, enabling it to process client requests and invoke and run Java Servlets. We use MySql4 [22] as our back-end database management system and mm.mysql [20], a type IV JDBC driver, as our database driver.

3.2 Proxy Servers

The novelty of our prototype lies in the architecture and implementation of the proxy servers. A traditional proxy server such as Squid [31] is sufficient to serve static content on behalf of application providers. To generate dynamic content we have added a servlet container to Gemini [21], a Squid proxy cache [31] augmented with IBM's Java virtual machine [15]. The proxy server also contains a simple database cache designed to reduce CPU utilization and bandwidth consumption of the home server, as well as the impact of database queries on the execution time of a servlet.

3.2.1 Caching Granularity

Our proxy server caches the results of database queries (i.e., materialized views) rather than the tables of the database itself, or arbitrary subtables. The primary rationale for this design choice is to make the proxy independent of the back-end database implementation. This flexibility is required since different application providers may choose different back-end databases. This approach also has the advantage that complex queries need not be re-executed, and the proxy server does not have to implement full database functionality (e.g., it does not need a query optimizer or query plan evaluator). In our prototype, database query results are cached at the JDBC level. All updates are forwarded directly to the back-end database at the home server.

3.3 Clients and Benchmarks

To drive load against our prototype we use Java implementations of three publicly available benchmarks: TPC-W [30], a transactional e-Commerce benchmark that captures the behavior of clients accessing an online book store, RUBiS [23], an auction system modeled after e-Bay [10], and RUBBoS [24], a simple bulletin-board-like system inspired by Slashdot [25]. We will henceforth refer to these benchmarks as BOOKSTORE, AUCTION and BBOARD, respectively. Each benchmark contains between 11 and 16 distinct update templates, and between 28 and 37 distinct query templates.

We made one significant modification to the BOOKSTORE benchmark. In the original benchmark all books are uniformly popular. In our version we use a more realistic distribution of book popularity based on the work by Brynjolfsson et al. [5], who empirically verified that for the well-known online bookstore Amazon [2], the popularity of books follows a Zipf distribution. In particular, $\log Q = 10.526 - 0.871 \log R$, where R is the sales rank of a book and Q is the number of copies of the book sold within a short period of time.

Each of the three applications conform to the TPC-W client specification, called *Emulated Browsers* (EB's). The run-time behavior of an EB models a single user session. Starting from the home page of the site, each EB uses a Customer Behavior Graph Model (a Markov chain with various servlets in the Web site as nodes and transition probabilities as edges) to navigate among Web pages, performing a sequence of Web interactions. The behavior model also incorporates a *think time* parameter that controls the amount of time an EB waits between receiving a response and making the next request. This parameter emulates the behavior of human users who typically spend some time looking at a page after it has been received before clicking on the next link.

3.4 Data Consistency

In our preliminary prototype we adopted a simple and conservative consistency model. In particular, all query templates are classified as either uncacheable, or cacheable with a fixed time-to-live (TTL) threshold. Marking a query template uncacheable ensures strong consistency for all instantiations of that template. Examples of uncacheable data include the latest bid on an item in the auction benchmark and the number of copies of an item in stock in the bookstore. Examples of cacheable data include the ten best sellers in the bookstore and the latest posting in the bulletin board. This model has proven successful for managing collections of documents and images in Web caches and CDN's, and fits the three benchmarks naturally. However, marking data as uncacheable increases the load on the back-end database at the home server and hence limits the scalability of the prototype architecture.

Our ultimate goal is to mark all data as cacheable, and rely on a fully distributed consistency enforcement scheme to ensure strong consistency of the previously uncacheable data. Before describing our ongoing work on this aspect in Section 4, we first present some initial performance results obtained using our current prototype.

3.5 Preliminary Results

In our implementation, we carefully optimized the performance of the centralized solution by enabling the query caching feature of MySQL4, the back-end database, and by adding the indices necessary to make queries execute as quickly as possible. We also eliminated most static content from our workload by ensuring that the emulated browsers did not request any images. This modification makes our results conservative

Benchmark	DB size	Details
AUCTION	990 MB	33,667 items 100,000 registered users
BBOARD	1.4 GB	213,292 comments 500,000 registered users
BOOKSTORE	217 MB	10,000 items 86,400 registered users

Figure 3: Configuration parameters for each benchmark.

since the static content of a normal workload can easily be cached by our system.

We performed our experiments with a simple two-node configuration (one home server and one proxy server), running on Emulab [32]. All machines had an Intel P-III 850 MHz processor with 512 MB of memory. In all experiments the home server and proxy server were connected by a high latency, low bandwidth duplex link (100 ms latency, 2 Mbps). Each client was connected to the proxy server by a low latency, high bandwidth duplex link (5 ms latency, 20 Mbps). These network settings model a deployment in which the proxy servers are located on the same local area networks as the clients, which may be far from the home servers. Since the overhead for emulating the clients is very low, we used a single client machine to emulate multiple clients. Figure 3 gives the benchmark configuration parameters we used in our experiments. We ran each experiment for 10-15 minutes, starting the proxy server each time with a cold cache.

3.5.1 Cache Hit Rates

Figure 4 gives cache hit rates measured for each of the three benchmarks, under three levels of client load (load levels are characterized by the number of simulated clients, or EB's). The BBOARD benchmark achieves high hit rates, implying that proxy servers should be able to offload much of the database work from home servers hosting similar applications. The hit rates for the AUCTION and BOOKSTORE benchmarks are less impressive. With the BOOKSTORE benchmark, hit rates improve with increased client load but still remain fairly small due to the relatively low degree of commonality among client access patterns in that benchmark. With this application, the scalability of our approach has fundamental limits unless a large portion of the database can be cached. We plan to study collaborative caching techniques [33] as a means to circumvent this limitation (see Section 6 for brief discussion). As for the AUCTION benchmark, only 52% of retrieved query results are marked as cacheable, which explains the modest hit rates. We expect that the distributed consistency management scheme we are developing (Section 4) will improve this situation considerably by enabling all query results to be cached until a conflicting update occurs.

Benchmark	Low Load	Medium Load	High Load
AUCTION	39.4	41.2	41.4
BBOARD	86.8	87.6	89.8
BOOKSTORE	18.4	24.7	36.6

Figure 4: Cache hit rates under different client load conditions, in %.

3.5.2 Server Load

Figure 5 shows the average latencies, per HTTP request, observed from various vantage points in the system at three different load levels. The bar **C** represents the traditional centralized architecture, whereas the bar **D** shows results for our distributed prototype. The bar **C** consists of three components: the network latency (i.e., the time spent by an average request in going from the client to the home server), the processing time spent in the web and application servers, and the time spent in servicing database requests. As the number of clients (EB's) increases, time spent in the back-end database increases, reflecting growing load on the database server. For the AUCTION and BOOKSTORE benchmarks, the database is clearly the bottleneck. The BBOARD benchmark, however, is presentation heavy and stresses the web and application server more than the database.

The **D** bar consists of five components for the AUCTION and BBOARD benchmarks and one additional component for BOOKSTORE. The BOOKSTORE benchmark implementation includes an independent shopping cart database on the proxy, which maintains, for each user, a list of items that the user intends to buy. The shopping cart sees frequent additions, deletions and modifications. As an example of a high-level optimization that the application developer can suggest, we moved the task of maintaining the shopping cart from the home server to the proxy server. While this optimization allows many queries to be answered at the proxy server, some queries that perform a join of one shopping cart table with one of the other tables in the database must obtain relevant data from the home server. The additional component measures the time spent in the shopping cart code at the proxy server. The other five components are the times spent in going from the client to the proxy server, processing at the proxy server, processing in the database caching module at the proxy server, going from the database module at the proxy server to the back-end database at the home server, and processing at the back-end database. An artifact of our current implementation is that there is limited multi-threading available at the proxy server (i.e., it is unable to handle a large number of threads efficiently). Hence, at higher loads many requests become enqueued at the proxy server, and the time spent there increases substantially. We are working to address this problem.

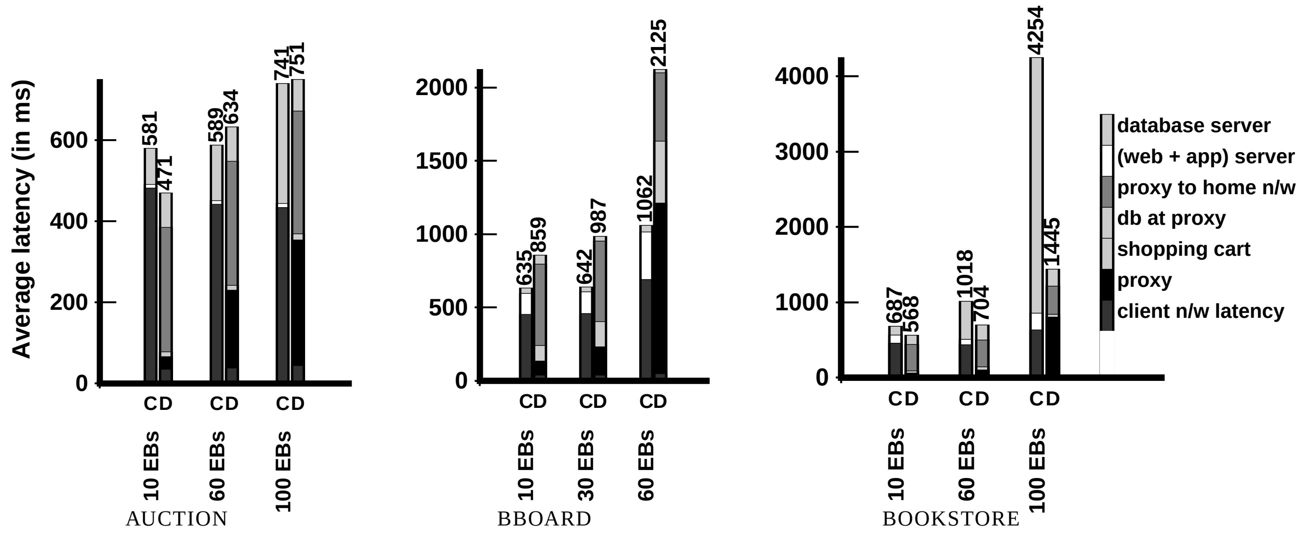


Figure 5: Latency breakdown, and how it varies with increasing load. The **C** bar shows the observed latency as a request passes through the different components of a traditional centralized architecture. The **D** bar shows the corresponding numbers for our distributed architecture.

Comparing the **C** and **D** bars, we see that our distributed architecture successfully eliminates all web and application server load from the home server. Plus, the database load on the home server is substantially reduced in the presence of a large number of clients (EB's), for the AUCTION and BOOKSTORE benchmarks (reflected in the lower average latency to service back-end database requests). In our distributed architecture the back-end database load does not rise dramatically with increasing client population, which is the case in the centralized architecture. This observation provides some evidence that in our distributed architecture the proxy server is able to shield home servers from increasing load. On the other hand, our hit rate measurements (Figure 4) indicate that substantial numbers of queries are still being sent to the back-end database, and hence the degree of scalability achievable by our current prototype is limited. Ultimately, the scalability of our approach rests on whether it can be improved to become more effective at shielding back-end databases from queries.

4 Distributed Management of Cache Consistency

To shield back-end databases from as many queries as possible, all data should be marked as cacheable, and only evicted from caches when a conflicting update occurs at some node in the system. When an update occurs, that update must be conveyed to all proxy servers caching views (query results) that might be affected by the update, so that they can either update or invalidate those views accordingly. Our approach is to employ a fully distributed multicast infrastructure to disseminate updates among proxies. When a proxy server caches a view, it subscribes to one or more multicast channels related to the cached view. Any time an update occurs, a message is broadcast to all channels related to views that may be affected by the update, and proxy servers that have subscribed to those channels can update or invalidate the affected views as necessary.

A good multicast environment to use for this purpose should support a large number of channels with highly dynamic groups of subscribers for each channel. In our scheme the rate of subscribing to and unsubscribing from channels is dictated by the rate at which views are cached and evicted. If invalidation is used as a consistency enforcement mechanism as we intend (see Section 1.2.2), then the rate of subscription churn can conceivably exceed the rate of multicast messages used to disseminate updates. Therefore it is crucial that subscribing and unsubscribing from multicast channels be highly efficient operations. At some point we may consider designing a custom multicast environment that can be optimized specifically for our context. In the meantime, we plan to use an out-of-the-box solution. The multicast environment Scribe [7], which uses a Pastry DHT overlay [29] as a basis for organizing multicast trees, seems to meet most of our needs.

4.1 Transactional Semantics

Recall from Section 1.2.2 that in our approach, whenever a data update is generated by application code running at a proxy server, the update is forwarded directly to the home server(s) corresponding to that application. Meanwhile, notification of that update must be sent to all proxy servers caching potentially conflicting views (including, of course, the proxy server at which the update originated). In this setting, to ensure proper transactional consistency semantics, i.e., one-copy serializability, a distributed consistency enforcement mechanism such as distributed locking must be used. The asynchronous, unidirectional messaging capability offered by Scribe and other multicast environments is not a sufficient basis upon which to build any distributed consistency mechanism. Bidirectional messaging is required to synchronize actions across multiple nodes. We are in the process of considering how best to graft bidirectional messaging onto a multicast environment for this purpose, and weighing the merits and drawbacks of different transactional consistency and availability mechanisms in our context.

At present the primary focus of our distributed consistency management work is on studying alternative ways to configure multicast channels. We discuss this issue next.

4.2 Configuration of Multicast Channels

There are many ways to configure multicast channels for dissemination of updates. To illustrate some of the alternatives and showcase the issues involved, we introduce a simple example inventory application in which the following two parameterized update templates and three query templates are embedded (question marks indicate parameters bound at execution time):

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

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

Query Template 1:
```
SELECT qty FROM inv WHERE name = ?
```

Query Template 2:
```
SELECT name FROM inv
WHERE entry_date > ?
```

Query Template 3:
```
SELECT * FROM inv WHERE qty < ?
```

4.2.1 Channel-By-Query Versus Channel-By-Update

For now let us ignore optimizations that can be made by considering parameter bindings supplied at execution time (we return to this issue in Section 4.2.2). Consider a bipartite graph between update templates and query templates, in which an edge from update template U to query template T indicates that whenever an instantiation of U is issued, all proxy servers currently caching one or more instantiations of Q are to be notified. In our example application there is an edge between every U/T pair with the exception of Update Template 2 and Query Template 2, which are independent. We refer to this graph as the *nonindependence mapping*. For a given set of query/update templates this mapping can be obtained using offline query/update independence analysis [17]. [1]

Given a precomputed nonindependence mapping, to ensure proper delivery of update notifications there are two basic alternatives: *channel-by-query* and *channel-by-update*:

- **Channel-By-Query:** There is a multicast channel corresponding to each query template. Numeric channel identifiers can be assigned using hashing. (Collisions in the channel identifier space may result in some proxy servers receiving spurious update notifications, but do not affect correctness.) When an update occurs at some proxy server, the server applies the precomputed nonindependence mapping to determine which query templates might be affected, and forwards the update along the corresponding multicast channels.

- **Channel-By-Update:** The determination of which update templates potentially affect a cached query result is performed eagerly, when the result is cached at some proxy server, as opposed to lazily, i.e., whenever an update occurs. In this scheme whenever a proxy server begins caching a query result, it applies the nonindependence mapping to determine which update templates might affect the query result, and subscribes to the corresponding multicast channels. When an update occurs, notification is sent along a single channel corresponding to the template for that update.

In both schemes, the set of proxy servers notified of a particular update is the same. The difference is that with channel-by-query, less work is performed each time a query result is cached or evicted, while more work is performed each time an update occurs, compared with channel-by-update. The cost of each strategy is highly dependent on the workload, of course, as well as the overhead of applying the nonindependence mapping, the overhead of subscribing to and unsubscribing from multicast channels, and the overhead of sending multicast messages. Overall, since most Web applications have read-dominated workloads, including our testbed applications described in Section 3.3, we expect channel-by-query to perform better.

4.2.2 Parameter-Specific Channels

The number of proxy servers notified of an update can, in some cases, be reduced substantially by considering runtime parameter bindings. It is not clear what can be done in the case of inequality predicates, so we focus

[1]The general query/update independence problem is undecidable [17]. As a result, in the presence of certain classes of query/update templates the nonindependence mapping can turn out to be conservative, i.e., contain unnecessary edges. This aspect can impact performance but not correctness, in our setting.

on equality predicates. Also, to simplify exposition in this section we assume the channel-by-query scheme is in place, although our discussion can be translated to fit the channel-by-update scheme.

Consider Query Template 1, which contains an equality predicate on the name attribute. Suppose that in addition to the template-level multicast channel corresponding to Query Template 1 we have an additional channel corresponding to each of the possible runtime bindings of the name parameter (there may be infinitely many). Numeric channel identifiers can be assigned using a hash function operating over the combination of template identifier and parameter binding. In most multicast environments, including Scribe, zero bookkeeping overhead is incurred for channels with no subscribers (i.e., those channels are not materialized).

Suppose proxy server A starts with a cold cache, and then begins caching the result of issuing Query Template 1 with parameter binding name='fork.' Server A must subscribe to the template-level channel for Query Template 1 as well as the parameter-specific channel for name='fork.' Then, if Update Template 1 is instantiated at proxy server B to insert a new record with name='spoon,' B transmits the update on a channel corresponding to Query Template 1 with name='spoon,' which does not reach A. (B must also transmit the update on the template-level channel for each of Query Templates 2 and 3.)

While the simple example given above illustrates the main idea, the presence of more sophisticated predicates (e.g., those containing conjunctive and disjunctive clauses) complicates the situation. We are currently studying the general problem of how best to configure parameter-specific channels so as to minimize the number of recipients of update notifications. We are also considering alternatives to channel-by-query and channel-by-update in which multicast channels can correspond to arbitrary units of data, and need not be tied to particular query or update templates embedded in the application. If it turns out that there is no universal best scheme that dominates the others in all cases, we will investigate adaptive policies that adjust the channel configuration dynamically in response to current conditions.

5 Tradeoffs in Invalidation Strategies for Cached Views

When a proxy server receives notification of an update (either a local update or an update forwarded by another node) that potentially conflicts with one or more locally-cached views, the next step is either to update or invalidate all affected views to ensure consistency. Recall from Section 1.2.2 that we choose to focus on invalidation rather than maintenance of cached views to avoid distributed updating of data, which causes numerous practical difficulties in our context. Hence, the issue is to decide which locally-cached views, if any, to invalidate upon receiving notification of an update.

One rather conservative strategy is to invalidate any view that cannot be ruled independent of the update by inspection of the view definition (the methods of [17] can be used, for example). Under this strategy invalidation of a view in response to an update is only avoided if it can be determined that any instance of the view would remain unchanged after application of the update (different view instances result from different database content). Often, the particular instance of a view being cached is not affected by an update, although some conceivable instances of the same view would be affected. Hence, any approach that relies solely on inspection of the view definition to determine independence tends to be quite conservative. A less conservative determination of independence can be made by inspecting the content of the currently cached view instance, using the techniques of [13] or others. Doing so can reduce the number of view invalidations performed, thereby improving scalability.

Customers, however, may be reluctant to permit access to sensitive data by cache management code that is written by a third party and lies outside their trusted code base. With statement-level view invalidation strategies that do not inspect data, applications are free to encode and encrypt data arbitrarily, a significant advantage in terms of security and privacy. These are crucial concerns when data flows through an infrastructure shared by many organizations. Additionally, when cache management entails inspection of data, interoperability with a wide range of DBMS products also becomes significantly more complex to achieve.

We illustrate the inherent tradeoff between scalability, on one hand, and a combination of low implementation complexity and data secrecy on the other hand, using a simple example. Consider an online bookstore application with data stored in two relations: `Books(Title, Author, Subject)` and `Authors(Author, Awards, Country)`. Suppose that a proxy server has cached an instance of the following view, giving the awards of American authors who have written history books:

```
CREATE VIEW MyView(Author, Awards) AS
SELECT  A.Author, A.Awards
FROM    Authors A, Books B
WHERE   B.Author = A.Author
        AND A.Country = "USA"
        AND B.Subject = "history"
```

Say the following update occurs:

```
UPDATE Authors SET Country="France"
  WHERE Author="Tocqueville"
```

An instance of MyView will be affected by the update if and only if it contains one or more records for author Tocqueville. Hence, an implementation that conservatively determines independence without inspecting view contents will invalidate, whereas one that inspects the contents of cached views can avoid invalidation in some cases. The former strategy has a simpler implementation that preserves data secrecy, whereas the latter strategy accommodates greater scalability.

Consider now the following update:

```
UPDATE Books SET Subject="fiction"
  WHERE Title="Napoleon's Television"
```

Here, inspecting the view does not help rule out conflict, but inspecting the base relations might help, say if no book with title "Napoleon's Television" occurs in the Books relation. Therefore, inspection of data that lies outside the view in question, using, say, the techniques of [6], can reduce the number of invalidations further, potentially leading to even better scalability. However, caution is warranted since caching additional data for the purpose of making more informed invalidation decisions is not necessarily a win in terms of overall scalability because the additional data must also be kept consistent. Plus, implementation complexity increases under such a strategy.

Which of these approaches, then, is the right one to take? This question has no one-size-fits-all answer. A system designer might opt for a simple implementation that treats cached views as black boxes and thus permits arbitrary data encryption by applications, as long as doing so does not unduly compromise scalability. Alternatively, the designer might implement a sophisticated invalidation strategy that inspects data, but switch to a statement-level strategy for customers that demand high security, even if it means decreased scalability for those customers. To determine the best approach, the system designer will need quantitative estimates of the impact these design decisions are likely to have on scalability. To this end we are developing an analytical framework for comparing various view invalidation implementation strategies quantitatively, in terms of the number of invalidations performed under a given workload. Our work is grounded in a formal characterization of view invalidation strategies, provided next.

5.1 Formal Characterization of View Invalidation Strategies

A view invalidation strategy $\mathcal{S}$ is a function whose arguments include an update statement, a view definition, and possibly other information, and that evaluates to one of I (for "invalidate") or DNI (for "do not invalidate"). A view invalidation strategy is *correct* if and only if whenever a view changes in response to an update, all corresponding cached instances of that view are invalidated. A formal definition of correctness is as follows:

Correctness: A view invalidation strategy $\mathcal{S}$ is correct iff for any view definition Q, database D, and update μ, $(Q(D) \neq Q(D^\mu)) \Rightarrow (\mathcal{S}(\mu, Q, \ldots) = I)$.

Here, D^μ denotes the state of database D after applying update μ, and $Q(D)$ denotes the result of evaluating view definition Q over database D. We assume that updates are applied sequentially, and that all invalidations necessitated by one update are carried out before the next update is applied.

We now define three classes of view invalidation strategies, which differ by which portions of the database they may access:

- **Black-Box Strategy** $\mathcal{S}(\mu, Q)$**:** Only the update μ and the view definition Q may be used to make the invalidation decision. Correctness requires that $(\exists D\ (Q(D) \neq Q(D^\mu))) \Rightarrow (\mathcal{S}(\mu, Q) = I)$.

- **View-Inspection Strategy** $\mathcal{S}(\mu, Q, V_p)$**:** The update μ, the view definition Q, and the content of the view $V_p = Q(D_p)$, where D_p denotes the state of the database at the time the view was evaluated (i.e., prior to application of the update), may be used to decide whether to invalidate V_p. Correctness requires that $(\exists D\ ((Q(D) = V_p) \wedge (Q(D) \neq Q(D^\mu)))) \Rightarrow (\mathcal{S}(\mu, Q, V_p) = I)$.

- **Full-Access Strategy** $\mathcal{S}(\mu, Q, V_p, D_p)$**:** The update μ, the view definition Q, and any portion of the database as it stood prior to application of μ, D_p (including the view $V_p = Q(D_p)$ itself), may be used to decide whether to invalidate V_p. Correctness requires that $(Q(D_p) \neq Q(D_p^\mu)) \Rightarrow (\mathcal{S}(\mu, Q, V_p, D_p) = I)$.

Every correct black-box strategy is a correct view-inspection strategy, and every correct view-inspection strategy is a correct full-access strategy, as shown in Figure 6.

We are now ready to define minimality:

Minimality: A view invalidation strategy $\mathcal{S}$ belonging to class $\mathcal{C}$ is *minimal* if and only if it is correct and there exists no view definition Q, update μ, and database D such that $\mathcal{S}$ invalidates the view $Q(D)$ in response to μ, while another correct view invalidation strategy in class $\mathcal{C}$ does not.

Corresponding to each class of invalidation strategy we give a simple criterion that is a sufficient and necessary condition for minimality:

- **Minimal Black-Box Strategy:** A black-box strategy $\mathcal{S}(\mu, Q)$ is minimal if and only if it is correct and whenever it invalidates a view defined by Q in response to update μ, there exists some database D such that the outcome of $Q(D)$ changes as a result of applying μ to D. Combining this condition with the correctness criterion for black-box strategies, we determine that a black-box strategy $\mathcal{S}$ is minimal iff $(\mathcal{S}(\mu, Q) = I) \Leftrightarrow \exists D\ (Q(D) \neq Q(D^\mu))$.

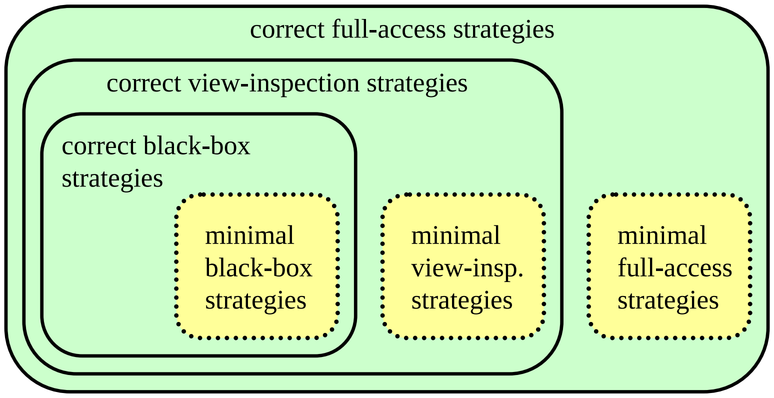

correct full-access strategies
correct view-inspection strategies
correct black-box strategies
minimal black-box strategies
minimal view-insp. strategies
minimal full-access strategies

Figure 6: Venn diagram depicting relationships among classes of view invalidation strategies.

- **Minimal View-Inspection Strategy:** A view-inspection strategy $\mathcal{S}(\mu, Q, V_p)$ is minimal if and only if it is correct and whenever it invalidates view V_p in response to update μ, there exists some database D from which V_p could have been derived such that the outcome of $Q(D)$ changes as a result of applying μ to D. Formally, a view-inspection strategy $\mathcal{S}$ is minimal iff $(\mathcal{S}(\mu, Q, V_p) = I) \Leftrightarrow \exists D\ ((Q(D) = V_p) \wedge (Q(D) \neq Q(D^\mu)))$.

- **Minimal Full-Access Strategy:** A full-access strategy $\mathcal{S}(\mu, Q, V_p, D_p)$ is minimal if and only if it is correct and whenever it invalidates view V_p in response to update μ, applying μ to database D_p changes the outcome of $Q(D_p)$. Formally, a full-access strategy $\mathcal{S}$ is minimal iff $(\mathcal{S}(\mu, Q, V_p, D_p) = I) \Leftrightarrow (Q(D_p) \neq Q(D_p^\mu))$.

It is not difficult to construct a formal proof to show that no correct black-box strategy is a minimal view-inspection strategy. Similarly, one can prove that no correct view-inspection strategy is a minimal full-access strategy. Figure 6 depicts the relationships among classes of view invalidation strategies as a Venn diagram.

5.2 Ongoing Tradeoff Analysis Work

Our eventual goal is to be able to answer the following question (without having to perform arduous manual analysis or build a sophisticated simulator): *Given an expected query/update workload, how many invalidations would be incurred under each class of minimal invalidation strategy?* As a starting point, we are working to identify restricted classes of workloads for which there is provably no advantage to inspecting cached views, i.e., a minimal view-inspection strategy would incur precisely the same set of invalidations as a minimal black-box strategy. The next step is of course to examine cases in which view inspection does offer an advantage in terms of fewer invalidations, and characterize the size of the advantage as a function of the workload.

We hope to do the same for minimal full-access strategies, although the situation is more complex because full-access strategies are “double-edged swords,” so to speak. In certain cases they necessitate bringing additional portions of the database into the cache solely for the purpose of making invalidation decisions. An interesting open problem is to find a portion of the database to cache for this purpose that can be kept consistent with little overhead (where consistency is ensured through invalidation or other means). This problem is related to the problem of finding the minimal set of auxiliary views to enable self-maintenance of a particular view [27]. It remains to be seen to what extent the problem differs in our context.

6 Future Work

Regardless of the exact invalidation strategy employed, the practice of always invalidating cached data in response to updates affecting that data places bounds on scalability. Our ultimate goal is to design a service that provides virtually unlimited scalability to applications, so that users are never denied access due to overload situations. As future work we plan to explore cache management techniques that always avoid overloading home servers by continuously monitoring and reacting to changing conditions. For one, we intend to study load-aware collaborative caching schemes in which proxy servers attempt to serve cache misses from other proxy servers before resorting to retrieving the data from a heavily-loaded home server. The more heavily-loaded the home server, the more additional proxy servers are contacted in an effort to locate a copy of the data. Such an “adaptive reach” policy would trade request response time against home server load as needed. Additionally, we plan to consider cache eviction policies that avoid evicting data housed at home servers currently experiencing heavy load. Finally, we may investigate policies that, when all else fails, relax data consistency as needed to avoid overload situations. One possibility is to defer invalidation of cached views when the impact on consistency is not severe, as judged in the context of the application.

7 Summary

In this paper we presented our ongoing work on creating a scalability service for dynamic Web applications. The purpose of a scalability service is to allow application providers to accommodate fluctuating demand without investing in costly equipment and in-house management expertise. We began by describing our preliminary working prototype and reporting some early performance results. The primary remaining challenge in realizing our vision of scalability offered as a plug-in service lies in the development of a scalable consistency enforcement mechanism for data requiring strong consistency. We presented our initial work on this topic, aimed at developing a fully distributed update propagation scheme. Our scheme exploits the fact that Web applications typically contain a small set of predefined query and update templates. In our scheme independence relationships between query and update templates are determined offline and then used at runtime to limit the number of proxy servers receiving notification of each update. Lastly, we turned to a discussion of alternative strategies for invalidating cached data in response to update notifications. In particular, we described our ongoing effort to characterize the inherent tradeoff between scalability and data secrecy, a crucial issue for an environment managing data from multiple organizations.

References

[1] M. Altinel, C. Bornhvd, S. Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald. Cache tables: Paving the way for an adaptive database cache. In *Proc. Twenty-Ninth International Conference on Very Large Data Bases*, Berlin, Germany, Sept. 2003.

[2] Amazon.com, Inc. Amazon.com. `http://www.amazon.com`.

[3] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A dynamic data cache for Web applications. In *Proc. IEEE International Conference on Data Engineering*, Bangalore, India, Mar. 2003.

[4] K. Amiri, S. Sprenkle, R. Tewari, and S. Padmanabhan. Exploiting templates to scale consistency maintenance in edge database caches. In *Proc. Eighth International Workshop on Web Content Caching and Distribution*, Hawthorne, New York, Sept. 2003.

[5] E. Brynojolfsson, M. Smith, and Y. Hu. Consumer surplus in the digital economy: Estimating the value of increased product variety. 2002. `http://www.heinz.cmu.edu/~mds/cs.pdf`.

[6] K. Candan, D. Agrawal, W. Li, O. Po, and W. Hsiung. View invalidation for dynamic content caching in multitiered architectures. In *Proc. Twenty-Eighth International Conference on Very Large Data Bases*, Aug. 2002.

[7] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralised application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication*, Oct. 2002.

[8] E. Cohen, E. Halperin, and H. Kaplan. Performance aspects of distributed caches using TTL-based consistency. In *Proc. Twenty-Eighth International Colloquium on Automata, Languages and Programming*, Crete, Greece, July 2001.

[9] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

[10] eBay, Inc. eBay. `http://www.ebay.com`.

[11] M. Franklin and M. Carey. Client-server caching revisited. In *Proc. International Workshop on Distributed Object Management*, Edmonton, Canada, Aug. 1992.

[12] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, June 1996.

[13] A. Gupta and J. A. Blakeley. Using partial information to update materialized views. *Information Systems*, 20(9):641–662, 1995.

[14] J. Holliday, R. Steinke, D. Agrawal, and A. El-Abbadi. Epidemic algorithms for replicated databases. *IEEE Transactions on Knowledge and Data Engineering*, 15(3), May/June 2003.

[15] IBM Corporation. Java2 development kit for Linux.

[16] Jakarta Project. Apache Tomcat.

[17] A. Y. Levy and Y. Sagiv. Queries independent of updates. In *Proc. Nineteenth International Conference on Very Large Data Bases*, Dublin, Ireland, Aug. 1993.

[18] W. Li, O. Po, W. Hsiung, K. S. Candan, D. Agrawal, Y. Akca, and K. Taniguchi. CachePortal II: Acceleration of very large scale data center-hosted database-driven web applications. In *Proc. Twenty-Ninth International Conference on Very Large Data Bases*, Berlin, Germany, Sept. 2003.

[19] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton. Middle-tier database caching for e-business. In *Proc. ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, June 2002.

[20] M. Matthews. Type IV JDBC driver for MySQL.

[21] A. Myers, J. Chuang, U. Hengartner, Y. Xie, W. Zhuang, and H. Zhang. A secure, publisher-centric web caching infrastructure. In *Proc. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, Apr. 2001.

[22] MySQL AB. Mysql database server.

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

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

[25] Open Source Development Network, Inc. Slashdot. `http://slashdot.org`.

[26] C. Partridge, P. Barford, D. Clark, S. Donelan, V. Paxson, J. Rexford, and M. K. Vernon. *The Internet Under Crisis Conditions: Learning from September 11*. National Academy Press, Washington, DC, Jan. 2003.

[27] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proc. Fourth International Conference on Parallel and Distributed Information Systems*, Miami Beach, Florida, Dec. 1996.

[28] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *Proc. Third International Workshop on Networked Group Communication*, London, U.K., Nov. 2001.

[29] 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, Nov. 2001.

[30] Transaction Processing Council. TPC-W specification, version 1.7.

[31] D. Wessels et al. Squid web proxy cache. `http://www.squid-cache.org`.

[32] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. Fifth Symposium on Operating Systems Design and Implementation*, Boston, Massachusetts, Dec. 2002.

[33] J. Yang, W. Wang, and R. R. Muntz. Collaborative web caching based on proxy affinities. In *Proc. SIGMETRICS 2000 International Conference on Measurements and Modeling of Computer Systems*, Santa Clara, California, June 2000.

[34] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proc. Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video*, Port Jefferson, New York, June 2001.