Thesis Proposal

Increasing the Scalability of Data-intensive Web Applications

Amit Manjhi
Carnegie Mellon University
manjhi@cs.cmu.edu

Abstract

Data-intensive Web applications are becoming increasingly popular and represent the future landscape of Web applications. Currently there is no effective way to increase the scalability of such applications without significant investment in infrastructure and in-house management capability. In this thesis we present two ways to scale such applications that only rely on the knowledge of the applications' code: First we develop the foundations of a Database Scalability Service (DBSS) that is able to offer scalability as a plug-in subscription service to data-intensive Web applications. As key parts of this work, we design the overall architecture of a DBSS, and present two techniques to help applications manage the important security/privacy-scalability tradeoff in the DBSS setting: a) a scalability-conscious security design methodology that achieves good scalability while adhering to a security policy, and b) invalidation clues, a general technique that enables applications to balance their privacy and scalability needs at a fine granularity. Second we optimize the collection and the schedule of database accesses an application issues. While these optimizations are useful in any setting the application operates, they are particularly important for DBSS settings, where the “network distance” between the DBSS node and the application’s database is likely to be large. Our experiments with all three realistic benchmark applications we study confirm the effectiveness of our methods.

1 Introduction

Applications deployed on the Internet are immediately accessible to a vast population of potential users. As a result, they tend to experience fluctuating and unpredictable load, especially due to events such as breaking news (e.g., 9/11) and sudden popularity spikes (e.g., the “Slashdot Effect”). It is crucial that the scalability of Web applications—the number of clients that the applications can support concurrently—be high. Among Web applications, data-intensive Web applications like bulletin-boards and e-commerce applications are becoming increasingly popular and represent the future landscape of Web applications. However, there is currently no effective way to increase the scalability of such applications without significant investment in infrastructure and in-house management capability.

In this thesis we present two important ways in which the knowledge of a data-intensive Web application’s code can be used to increase its scalability. First we develop the foundations of a Database Scalability Service that is able to offer scalability as a plug-in subscription service to data-intensive Web applications. Second we optimize an application’s interaction with its database. We provide an overview of our methods in the next two subsections.

1.1 Database Scalability Service (DBSS)

We start in Section 1.1.1 by providing the overall architecture of a DBSS. Section 1.1.2 and Section 1.1.3 then provide techniques for applications to manage the privacy/security-scalability tradeoff that exists in the DBSS setting.
1.1.1 Overall Architecture

We have designed a **Database Scalability Service** (DBSS) [32] to support applications where the database is the bottleneck. As in CDNs, a third-party (Database Scalability Service Provider, or DSSP) provides such service by maintaining a large, shared infrastructure to offload work from and to absorb load spikes for any individual database. Figure 1 depicts the resulting architecture, in which (1) user requests are handled by a CDN, (2) the CDN in turn fires off database queries and updates that are handled by a DBSS, and (3) queries that cannot be answered by the DBSS and updates are sent to backend databases within the application’s “home” organization. We have built a prototype DBSS with this general architecture, and used it to scale three data-intensive Web application benchmarks.

A key challenge in the design of a DBSS is providing this shared scalability infrastructure while protecting each organization’s sensitive data. The goals are (1) to limit the DBSS administrator’s ability to observe or infer an application’s sensitive data, and (2) to limit an application’s ability to use the DBSS to observe or infer another application’s sensitive data. Such concerns have been increasing in the past few years, as borne by well-publicized instances of database theft [36]. From the viewpoint of the home organization, these are security concerns; from the viewpoint of an individual user whose personal data may be revealed, these are privacy concerns.

Security/privacy concerns dictate that a DBSS should be provided encrypted updates, queries and query results. The home servers of applications maintain master copies of their data and handle updates directly\(^1\), and the DBSS caches read-only (encrypted) copies of query results that are kept consistent via invalidation.

**Privacy-Scalability Tradeoff.** There is an important security/privacy-scalability tradeoff in the DBSS setting. When a data update occurs, to maintain consistency, the DBSS must invalidate (at least) all the cached query results that changed. Because all data that the DBSS sees is encrypted, the DBSS needs help from the application in order to know which results to invalidate; such help, however, inevitably reveals some properties about the data. (The application could provide the help, either by not encrypting the data passing through the DBSS, an approach we use in Section 1.1.2, or by sending invalidation clues, a more general technique we present in Section 1.1.3 that allows applications to manage their privacy and scalability needs at a fine granularity.) Thus, in providing help to the DBSS, the application faces an important dilemma. On the one hand, revealing less about the data means that the DBSS will invalidate far more than needed, resulting in more queries passed through to the home server, degrading scalability. On the other hand, revealing more about the data to the DBSS raises security concerns. In the next section, we illustrate

---

\(^1\)In many Web applications, updates are infrequent; so the load on home servers due to updates is low.
Table 1. An example toystore application, denoted SIMPLE-TOystore, with three query templates \(Q_1^T, Q_2^T, Q_3^T\), one update template \(U_1^T\), and two base relations: \(\text{toys} \) with attributes \(\text{toy}\_id, \text{toy}\_name, \text{qty}\), and \(\text{customers} \) with attributes \(\text{cust}\_id, \text{cust}\_name\). The question marks indicate parameters bound at execution time.

<table>
<thead>
<tr>
<th>Information available?</th>
<th>Templates</th>
<th>Parameters</th>
<th>Query Results</th>
<th>Invalidated Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>All of (Q_1^T, Q_2^T, Q_3^T)</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>All (Q_1^T), all (Q_2^T)</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>(Q_2^T) if (\text{toy}_id=5), all (Q_1^T)</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>(Q_1^T) if (\text{toy}_id=5), (Q_2^T) if (\text{toy}_id=5)</td>
</tr>
</tbody>
</table>

Table 2. Invalidations differ depending on the amount of information the DBSS can access. The table is for update \(U_1^T\) with parameter 5.

the security-scalability tradeoff through an example, and present our scalability-conscious security design methodology.

1.1.2 Scalability-Conscious Security Design Methodology

To illustrate this important tradeoff between security and scalability, we introduce a simple example application called SIMPLE-TOystore, specified in Table 1. We focus on the application’s database access templates—queries or updates missing zero or more parameter values. Table 2 lists the invalidations the DBSS needs to make on seeing a specific update in four different scenarios; each scenario is represented by a row of the table. The scenarios differ in what help the application provides to the DBSS. This help determines what data is available to the DBSS for making invalidation decisions. For example, if application provides no information, i.e., all data for queries and updates is encrypted, as in the first row, then the DBSS invalidates all cached query results on seeing an instance of update \(U_1^T\). However, if the application does not encrypt the template information, as in the second row, then in response to an instance of \(U_1^T\), the DBSS invalidates cached query results of all instances of only \(Q_1^T\) and \(Q_2^T\). As the application provides more information to the DBSS (moving down the rows), the number of invalidations the DBSS needs to make decreases, thereby increasing scalability.

We study this security-scalability tradeoff, both analytically and empirically, as a function of the four scenarios represented by the four rows of Table 2. To help manage this tradeoff, we present a static analysis method [26] for identifying segments of an application’s database that are never useful for invalidation decisions. The application can stop worrying about making such data available to the DBSS. Moreover, for all three benchmark applications we study (details in Section 3.5), most of the data that can be encrypted without impacting scalability is of the type that application designers will want to encrypt, all other things being equal.\(^3\)

Based on our static analysis method, we propose a new scalability-conscious security design methodology that features: (a) compulsory encryption of highly sensitive data like credit card information, and (b) encryption of data for which encryption does not impair scalability. As a result, the security-scalability tradeoff needs to be considered only over data for which encryption impacts scalability, thus greatly simplifying the task of managing the tradeoff.

\(^2\)With security, the goal is to limit the DBSS administrator’s ability to read the application’s data whereas with privacy/security, the goal is to limit the DBSS administrator’s ability to either read or infer the application’s data.

\(^3\)See Section 4.4.3 for details on what data is kept private using our static analysis method.
1.1.3 Invalidation Clues

We present invalidation clues, a general framework (the solution of Section 1.1.2 is a special case) for applications to reveal little data to the DBSS, yet prevent wholesale invalidations. Invalidation clues (or clues for short) are attached by the home server to query results returned to the DBSS. The DBSS stores these query clues with the encrypted query result. On an update, the home server can send an update clue to the DBSS, which uses both query and update clues to decide what to invalidate. We show how specially designed clues can achieve three desirable goals:

- **Limit unnecessary invalidations**: Our clues provide relevant information to the DBSS that enable it to rule out most unnecessary invalidations.
- **Limit revealed information**: Our clues enable the application to achieve a target security/privacy by hiding information from the DBSS.
- **Limit database overhead**: Our clues do not enumerate which cached entries to invalidate. Instead, they provide a “hint” that enables the DBSS to rule out unnecessary invalidations. Thus, the home server database is freed from the excessive overhead of having to track the exact contents of each DBSS cache in order to enumerate invalidations.

(See Section 5.1 for an illustrative example of how clues enable applications to balance their security/privacy and scalability requirements.)

We show how invalidation clues offer applications a low overhead control to balance their privacy and scalability needs at a fine granularity [25]. Furthermore, for many query/update template pairs, extra data, beyond data that is a function of query and update statements and query results, is necessary for precise invalidation. We identify such pairs, and show how precise invalidation can be achieved in such cases by generating “database-derived” clues. We also empirically measure the scalability benefits of using precise invalidations for the three application benchmarks we study (described in Section 3.5).

As our proposed work, we intend to measure the scalability advantages of using different invalidation strategies in the more realistic setting when (a) the applications use multiple DBSS nodes (so far, we have been using a single DBSS node), and (b) each DBSS node starts with a warm cache (so far, each DBSS node starts with a cold cache).

1.2 Optimizing an Application’s Interaction with its Database

A Web application (Table 5 shows a fragment) is a collection of programs commonly written in a procedural language like CGI or Java Servlets. These programs interact with the application’s database, which houses and manages all of its data, by issuing queries or updates in a declarative language, commonly SQL. The queries and updates are executed by the database, while the remainder of the Web application is executed by the application server. Thus the collection of queries and updates determines how the execution work is divided between the database and the application server.

There is often a lot of flexibility in choosing how this execution work is divided (between the database and the application server). Flexibility arises because most of the computations that a Web application is required to do can be performed either in the procedural code or the declarative code. The different divisions of execution work not only affect the load on the database, but also influence the latency seen by the clients. And any reduction in client latency increases the scalability\(^4\) of the application.

Furthermore, since the execution of a single program can result in multiple database requests, there is also opportunity for optimizing the schedule in which the multiple database requests are issued. For example, after issuing a database access, a program either waits for the query result or waits for the confirmation that the update has been applied. In many cases, this waiting behavior is unnecessary since the the next database access does not depend on the answer to the current database access. Although this waiting behavior is fine in a centralized setting, where the application code is executed “close” to the database, in a DBSS setting or a shared Web-hosting setting\(^5\), where the “network distance” between the application server and

\(^4\) We measure scalability from a client’s perspective, as the number of requests the system can serve such that the majority of requests have a response time lower than a certain threshold.

\(^5\) In a shared hosting scenario with a Web-hosting service, the application and the database of a website typically runs on separate cluster of machines. For www.dreamhost.com, a leading Web-hosting company, we measured the latency between the clusters to be between 16ms and 20ms.
### Table 3. A code fragment from the AUCTION application. We focus on two base relations: users with attributes user_id and user_name, and comments with attributes from_user_id and to_user_id. The left hand side shows the original code, while the right hand side shows the optimized code.

<table>
<thead>
<tr>
<th>Original code</th>
<th>Functionally-equivalent code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$query := SELECT from_user_id FROM comments WHERE to_user_id = ? $result := execute ($query)</td>
<td>$query := SELECT from_user_id, user_name FROM comments, users WHERE from_user_id = user_id AND to_user_id = ? $result := execute ($query)</td>
</tr>
<tr>
<td>foreach ($row in $result) $from_user_id := get-from_user_id-from-row ($row) $query2 := SELECT user_name FROM users WHERE user_id = $from_user_id $result2 := execute ($query2)</td>
<td></td>
</tr>
</tbody>
</table>

the database can be large, waiting for each database access individually can significantly increase the client latency. And an increase in the client latency decreases the scalability.

For data-intensive Web applications, there is even greater opportunity to apply these two optimizations because such applications make heavy use of the database. Furthermore, these two optimizations are beneficial even in a non-DBSS setting, but are more important in a DBSS setting, where the code is executed far from the database, in terms of the “network distance.” We next provide more details about these two optimizations.

#### 1.2.1 Determining the Procedural-Declarative Boundary

To illustrate the flexibility an application has in choosing the collection of queries and updates to issue, consider the functionally-equivalent code fragments in the two columns of Table 3. On the left-hand side, the program issues several small inter-related queries and the procedural code combines them to get a result, whereas on the right-hand side, the program issues a single query that gets all the relevant data from the database. The two code fragments are functionally equivalent, but have unique advantages. For example, if there is no caching and a to_user_id value with which eight from_user_id values are associated is used, then the CPU-load on the database due to executing the program on the right-hand side is about $\frac{1}{8}$th the CPU-load for the left-hand side. (We use the database of the AUCTION benchmark, and run it on the set-up described in Section 3.) Moreover, if there is no caching, the program on the right reduces the latency of the clients, particularly in DBSS settings where the “network-distance” between where the program is executed and the application’s database is large. This reduction in client latency increases the perceived scalability—the number of Web responses the system can answer within a certain threshold—of the system. However, the program on the right-hand side is less useful if the query results are being cached, and there is an update of the form UPDATE user_name FROM users WHERE user_id = ?. Such an update invalidates the whole query result on the right-hand side, whereas it only invalidates a part of the query result on the left-hand side. Thus, the program on the right may require the database to re-do a lot of the work.

Our preliminary study of the three application benchmarks confirms that many opportunities exist\(^6\) for optimizing the execution work between the application server and the database. For our proposed work, we first intend to evaluate the benefit of this optimization in a centralized setting. Next, we plan to explore how we can combine the unique advantages of the two approaches in the DBSS setting. Finally, we intend to evaluate this optimization in a DBSS setting.

#### 1.2.2 Optimizing the Schedule of Database Requests

To illustrate how the schedule of database requests issued by a program can be optimized, consider Table 4, which shows two functionally-equivalent code fragments from the BOOKSTORE application (full details in Section 3.5). The program on the right is obtained by optimizing the schedule of database accesses of the program on the left. (The method execute_non_blocking does not block and only serves to populate

---

\(^6\)See Section 6.2.1 for what opportunities exist and why we believe they exist.
Table 4. A simplified code fragment from the BOOKSTORE application. We focus on two base relations: users with attributes user_id and user_name, and items with attributes item_id, item_name, and related. The left hand side shows the original code, while the right hand side shows the optimized code.

<table>
<thead>
<tr>
<th>Original code</th>
<th>Code after optimizing the schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{query1} := \text{SELECT item_name} ) FROM items i1, items i2  (\text{WHERE} \ i1.\text{id} = i2.\text{related} ) AND i2.\text{id} = ? (\text{result1} := \text{execute} (\text{query1}))</td>
<td>(\text{query2} := \text{SELECT user_name FROM users} ) WHERE user_id = ? (\text{execute_non_blocking} (\text{query2})) (\text{query1} := \text{SELECT item_name} ) FROM items i1, items i2 (\text{WHERE} \ i1.\text{id} = i2.\text{related} ) AND i2.\text{id} = ? (\text{result1} := \text{execute} (\text{query1}))</td>
</tr>
</tbody>
</table>

the cache with the query result.) If the latency of the first database request is \(t_a\) and the latency of the second request is \(t_b\), the optimization reduces the overall latency from \(t_a + t_b\) to \(\max\{t_a, t_b\}\).

In our preliminary work, we have evaluated a simple version of this idea. We used a Java bytecode analysis tool called soot [12] to automatically insert non-blocking calls in the BOOKSTORE application. Our conservative compiler analysis was effective in reducing the client latency, but was applicable only in a limited number of cases. For our proposed work, we intend to evaluate this technique on all three benchmark applications we study (details in Section 3.5), using a less conservative strategy.

1.3 Outline

The rest of the thesis proposal is organized as follows. Section 2 describes related work. Section 3, 4, and 5 respectively summarize our completed work on designing a prototype, devising a scalability-conscious security design methodology, and devising invalidation clues. Section 6 discusses our proposed future work: exploring the privacy-scalability tradeoff in presence of more than one DBSS node, and optimizing the interaction of a Web application with its database. Finally, Section 7 shows our proposed schedule for completing the thesis.

2 Related Work

Prior work related to ours can be partitioned into five categories: database services, view invalidation, privacy, interaction with database, and commercial efforts. We discuss each in turn.

2.1 Database Services

Existing work on providing database services can be classified into two categories: the Database Outsourcing (DO) model and the Database Scalability Service Provider (DSSP) model.

In the DO model, an application outsources all aspects of management of its database to a third party [16]. A key concern is to safeguard the application’s sensitive data. Since the DO provider houses the application’s entire database, one way to ensure security of an application’s data is to store an encrypted database at the DO provider, and use encryption schemes that permit query processing on encrypted data [2, 15, 17]. Aggarwal et al. [1] suggest an alternative—distribute data across multiple independent providers that do not communicate with one another.

In contrast to work in the context of the DO model, we consider the DSSP model, in which only database scalability, and not full-fledged database management, is outsourced to a third party [32]. Under the DSSP model, application providers retain master copies of their data on their own systems, with the DSSP only caching and serving read-only copies on their behalf. In our DSSP approach, query execution on third party servers is not needed, so arbitrarily strong encryption of the remotely-cached data is possible. We contend that from a security and data integrity standpoint, the scalability provider model is more attractive than the DO model in the case of Web applications with read/write workloads (e.g., e-commerce applications).

Other ongoing efforts to create DSSP technology include the DBCache [4, 23] and DBProxy [5] projects at IBM Research, the MTCache project at Microsoft Research [20], and the CachePortal project [22] at NEC.
Laboratories. To the best of our knowledge, no prior work has studied security issues in the DSSP model, which is the focus of our work.

2.2 View Invalidation

There has been prior work pertaining to the invalidation of cached materialized views. Choi and Luo [11] proposed a technique that leverages statically available query/update templates to speed up runtime invalidation decisions. Levy and Sagiv [21] provide heuristic methods for determining when query statements (and hence view definitions) are independent of updates in many practical cases, although the general query/update independence problem is undecidable. In the data-warehousing context, a plethora of work has been done on view maintenance, in which, on any update, the view is updated to reflect the update; view maintenance strategies can be used to implement view invalidation strategies. Gupta and Blakeley [14] provide techniques to update views using a subset of the query statement, the update statement, and the updated base relation. Quass et al. [33] study view self-maintenance—for a given view, find a set of extra views, called auxiliary views, so that on any update, the view and the set of auxiliary views can be updated without inspecting the base relations. However, these techniques are not well-suited to our setting, as argued in Section 5.3.2.

The works cited above are only special cases of clues, which are often ill-suited to our setting. For example, we demonstrate the necessity and advantages of specially designed “database-derived” clues, in order to achieve precise invalidations. The work closest to this in technique is by Candan et al. [10]. They suggested using “polling queries” to inspect portions of the database in order to decide whether to invalidate cached query results in response to database updates. However, they used polling queries as a heuristic to get better invalidations, and did not use them to implement precise invalidations.

In the context of Coda, a filesystem for mobile computing environments, Mummert et al. [28] maintained cache entries at multiple levels of granularity which allowed trading precision of invalidation for speed of invalidation. Satya [34] then showed by examples how this concept of cheap but conservative invalidation can be useful in a variety of settings. In our work, we also see the effects of this tradeoff, as explained in Section 6.1.2. However, in addition to the speed of invalidation, we also focus on privacy and security—on how using conservative invalidation exposes less data to the DBSS.

2.3 Privacy

There has been a lot of recent interest in keeping data private, yet allowing the computation of several functions on the data. For example, Agrawal et al. [3] showed how to transform a database $D$ to $D'$ so that $D'$ is privacy-preserving, but still allows a user to compute a function $f$ on the database such that $f(D) = f(D')$. Agrawal et al. [2] present order-preserving encryption schemes; these could be used to enable order-comparisons over clues. However, under our attack model where the adversary can have access to some plain-text to encrypted-value mappings, this scheme does not work.

Hore et al. [18] study the privacy-utility trade-off in the choice of the “coarseness” of the index on encrypted data. Our bucketization technique in Section 5.4.4 is similar. However, the different domain we consider requires different optimization objectives.

2.4 Optimizing an Application’s Interaction with its Database

The paper [38] on Hilda, a high-level language for specifying data-intensive Web applications, mentions of optimizing an application’s interaction with its database, but defers their discussion to future work.

Most database vendors support stored procedures [35] that allow applications to either invoke a block of procedural and declarative code at the database or send a block of procedural and declarative code to be invoked at the database. However, with stored procedures, all work for the stored procedure is still done centrally at the database—it does not help scale the database. With our work, we want to combine the unique advantages of stored procedures with the advantages of caching query results at a small granularity in a DBSS setting.

2.5 Commercial Efforts

Akamai Technologies, a leading CDN, has released its “EdgeJava” product, which allows Web content providers to execute Java servlets on Akamai’s proxy servers. 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. Avokia (www.avokia.com) is another notable commercial effort which aims to scale the database, although in a centralized setting. Neither Akamai nor Avokia has, as yet, explored either the security concerns, or optimizing an application’s interaction with its database for the DSSP setting.

3 Prototype DBSS

We have built a prototype with the architecture of Figure 1, which enables home servers to offload the tasks of generating dynamic content and answering query results. In Section 3.1–3.3, we first present the design of the four distinct nodes in our system: Home servers, CDNs, DBSSs, and the clients. Next, in Section 3.4, we discuss how invalidations are carried out in our prototype DBSS. Finally we describe our three benchmark applications in Section 3.5 and our experimental methodology in Section 3.6.

3.1 Home Server

Each home server embodies the traditional three-tiered architecture, which enables it to generate Web content dynamically and server 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 content provider uses to manage all of its data.

In our prototype each home server is implemented as follows. We use Tomcat [19] 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 [29] as our back-end database management system and mm.mysql [27], a type IV JDBC driver, as our database driver. Additionally, a Java module, through which all database requests pass, computes “database-derived” clues for queries and updates.

3.2 CDN and DBSS Node

The DBSS node provides answers to database queries using its store of cached query results. To keep the prototype simple, the DBSS node also provides the functionality of a CDN node, i.e., the ability to run Web applications and to interact with a user running a Web browser. We used Tomcat [19] to provide both functionalities. Cached query results are kept consistent with the home server’s database using non-transactional invalidation of cached query results. The cache store and the invalidation logic are implemented in Java.

3.3 Clients

Since the overhead for emulating clients is low, one machine was used to emulate multiple clients. As in the TPC-W [37] specification, clients simulate human usage patterns by issuing an HTTP request, waiting for the response, and pausing for a think time of X seconds before requesting another Web page—X is drawn from a negative exponential distribution with a mean of seven seconds.

3.4 Flow of Invalidations

The overall system architecture is as depicted in Figure 1. The flow of queries, updates, and invalidations in the system is shown in greater detail in Figure 2. In the figure, diagonal shading denotes information that is subject to encryption. The DBSS maintains a cache of encrypted queries and encrypted query results. Along with each cache entry, it stores query clues sent by the home server’s database when returning the encrypted query result. On receiving an encrypted query Q, the DBSS determines if an entry for Q is in its cache and, if so, it returns the cached encrypted query result. Otherwise, the encrypted query is forwarded to the home database server, which returns an encrypted query result and any associated query clues. All updates are encrypted by the CDN and routed to the home organization via the DBSS. The home organization applies the updates, and returns the encrypted updates with associated update clues. The DBSS monitors completed updates, and uses the query clues and update clues to invalidate cached query results as needed to ensure consistency.
3.5 Benchmark Applications

We sought Web benchmarks that make extensive use of a database and are representative of real-world applications. We found three publicly available benchmark applications that met these criteria: RUBiS [30], an auction system modeled after eBay.com, RUBBoS [31], a simple bulletin-board-like system inspired by slashdot.org, and TPC-W [37], a transactional e-Commerce application that captures the behavior of clients accessing an online bookstore.7 We used Java implementation of these applications. We will henceforth refer to these applications as AUACTION, BBBOARD, and BOOKSTORE, respectively.

Figure 3 provides the database configuration parameters we used in our experiments. Please refer our publication [24] for sample update numbers for the three applications.

3.6 Experimental Methodology

We performed our experiments with a simple two-node configuration—a home server machine, which had an Intel P-III 850 MHz processor with 512 MB of memory, and a DBSS node machine, which had an Intel 64-bit Xeon processor with 2048 MB of memory. In all experiments, the home server and DBSS node were connected by a high latency, low bandwidth duplex link (100 ms latency, 2 Mbps). We used just one additional node to emulate all clients—the client was connected to the DBSS node by a low latency, high bandwidth duplex link (5 ms latency, 20 Mbps). These network settings model a deployment in which a DBSS node (because there are many of them) is “close” to the clients, most of which are “far” from any single home server.

Each experiment ran for ten minutes, and the DBSS node started with a cold cache each time. Scalability was measured as the maximum number of users that could be supported while keeping the response time below two seconds for 90% of the HTTP requests.

4 Security-Scalability Tradeoff

To underpin our study of the security-scalability tradeoff, we begin in Section 4.1 by presenting our formal characterization of cache invalidation strategies, each of which represents a natural choice in the space of

---

7To make the TPC-W application more representative of a real-world bookseller, we changed the distribution of book popularity in TPC-W from a uniform distribution to a Zipf distribution based on the work by Brynjolfsson et al. [8]. Brynjolfsson et al. verified empirically that for the well-known online bookstore amazon.com, the popularity of books varies 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 of the book sold within a short period of time.
security-scalability options. Section 4.2 provides an overview of our main contribution: a static analysis method for determining which data can be encrypted without impacting scalability. Section 4.3 presents our scalability-conscious security design methodology for managing the security-scalability tradeoff. In Section 4.4 we highlight our empirical findings, which point to the effectiveness of our technique. For full details on any aspect of this work, please refer to the full version of our paper [24].

4.1 Framework for Studying the Security-Scalability Tradeoff

In this section we characterize when an update necessarily causes invalidation of the cached result of a query, as a function of the information that is accessible. This formal characterization underpins our study of the security-scalability tradeoff. We begin in Section 4.1.1 by providing the details of our basic query and update model, and introducing the terminology and notation we use in the rest of the paper. Then, in Section 4.1.2 we characterize four distinct classes of invalidation strategies, i.e., strategies for deciding when to invalidate a cached query result in response to an update, that differ in the amount of information available to them.

4.1.1 Query and Update Model

Our query and update model is based on our study of three benchmark Web applications (an auction, a bulletin-board and a bookstore—details in Section 3.5); hence we believe it to be reasonably realistic.

In our model, there are a fixed set of query templates and a fixed set of update templates. Example templates are given in Table 1 and Table 5. A query $Q$ is composed of a query template $Q_T$ to which parameters are attached at execution time. Likewise, an update $U$ is composed of an update template $U_T$ to which parameters are attached at execution time. A sequence of queries and updates issued at runtime constitutes a workload.

The query language is restricted to select-project-join (SPJ) queries having only conjunctive selection predicates, augmented with optional order-by and top-k constructs. SPJ queries are relational expressions constructed from any combination of project, select and join operations. As in previous related work [6, 33], the selection operations in the SPJ queries can only be arithmetic predicates having one of the five comparison operators $\{<, \leq, >, \geq, =\}$. The order-by construct affects tuple ordering in the result; and the top-k construct is equivalent to returning the first $k$ tuples from the result of the query executed without the top-k construct. We assume multi-set operation; the projection operation does not eliminate duplicates.

The update language permits three kinds of updates: insertions, deletions and modifications. Each insertion statement fully specifies a row of values to be added to some relation. Each deletion statement specifies an arithmetic predicate over columns of a relation. Rows satisfying the predicate are to be deleted. Each modification statement selects a row of a relation according to an equality predicate on the relation’s primary key and modifies non-key attributes of the selected row.

To simplify the presentation of our static analysis of which information can be encrypted without impacting scalability, we make some assumptions about the update and query templates. Please refer our full paper [26] for details.

4.1.2 Formal Characterization of View Invalidation Strategies

Recall that in our current design, the DBSS caches views, which are results of queries. A view invalidation strategy $S$ is a function whose arguments possibly include an update statement, a query statement, and other information such as a cached query result. It 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 $S$ is correct iff for any query $Q$, database $D$, and update $U$, $\left(Q[D] \neq Q[D + U]\right) \Rightarrow S(U, Q, . . .) = I$.

(Assume that updates are applied sequentially, and that all invalidations necessitated by one update are carried out before the next update is applied.)

A view invalidation strategy is invoked whenever an update occurs. Based on what information they access in making invalidation decisions, four classes of view invalidation strategies, one for each row of
Table 2, may be defined as follows: (a) Blind Strategy corresponding to the first row, (b) Template Inspection Strategy (TIS) corresponding to the second row, (c) Statement-Inspection Strategy (SIS) corresponding to the third row, and (d) View-Inspection Strategy (VIS) corresponding to the last row. In each case, the strategy can only use the accessible information for making invalidation decisions. We provide a formal definition for each strategy in an extended technical report [24] .

These four view invalidation strategies, natural points in the invalidation strategy design space, are largely based on previous work in the area of view invalidations. For example, the methods of [14] can be used to implement a view-inspection strategy. Similarly, the methods of [21] can be used to implement a template- or a statement-inspection strategy. Finally, implementing a blind strategy is simple: invalidate all cached query results on any update.

Also, every correct blind strategy is a correct template-inspection strategy, every correct template-inspection strategy is a correct statement-inspection strategy, and every correct statement-inspection strategy is a correct view-inspection strategy. The relationships are depicted in Figure 4.

We now define minimality:

Minimality: A view invalidation strategy $S$ belonging to class $C$ is minimal if and only if it is correct and there exists no query statement $Q$, update statement $U$, and database $D$ such that $S$ invalidates the view $Q[D]$ in response to $U$, while another correct view invalidation strategy in class $C$ does not. Corresponding to each class of invalidation strategy, the criterion for a minimal blind strategy (MBS), a minimal template-inspection strategy (MTIS), a minimal statement-inspection strategy (MSIS), and a minimal view-inspection strategy (MVIS), can be arrived at, by applying the definition of minimality to the respective class.

For arbitrary databases and workloads, no correct blind strategy is a minimal template-inspection strategy. Similarly, no correct template-inspection strategy is a minimal statement-inspection strategy and no correct statement-inspection strategy is a minimal view-inspection strategy. (We omit formal proofs for brevity.) Figure 4 depicts the relationships among classes of view invalidation strategies as a Venn diagram.

The choice of invalidation strategy determines what information can be encrypted. On the one extreme, if a view-inspection strategy is used, neither queries, nor updates, nor cached query results can be encrypted. On the other extreme, if a blind strategy is used, all queries, updates, and cached query results can be encrypted. To check whether a given query can be answered from the cache, a lookup operation is required to check whether the DBSS has a cached copy of the query result. For a VIS or SIS, the query statement serves as the lookup key. For a TIS, the query template along with encrypted parameters are used. For a BS, the encrypted query statement is used as the lookup key.

4.2 Overview of Our Static Analysis Method

Instead of using the same invalidation strategy for all update/query template pairs, a DBSS may use different invalidation strategies for different update/query template pairs. This allows the application more...
Query templates:

Update templates:

Figure 5. Starting with the California data privacy law, additional exposure reduction for query and update templates.

flexibility in choosing which data to encrypt and which data to not encrypt. Further, for a given update/query template pair $U_T/Q_T$, if it turns out that certain data is not useful for invalidation, then not making the data available to the DBSS has no scalability overhead. We identify such data by analyzing the invalidation behavior of the four invalidation strategies: MBS, MTIS, MSIS, and MVIS, for a update/query template pair. For example, if the invalidation behavior of MVIS and MSIS for a update/query template pair $U_T/Q_T$ is the same, then not making the query result of any instance of $Q_T$ available to the DBSS has no scalability impact. In our paper [26], we provide complete details for identifying data that can be encrypted (i.e., does not need to be made available to the DBSS) without any scalability overhead. Such an analysis can be done for any application that follows the query and update model of Section 4.1.1, and even if a part of the database is necessarily encrypted.

4.3 Overview of Our Scalability-Conscious Security Design Methodology

Based on our static analysis method, we propose a new scalability-conscious security design methodology that features: (a) compulsory encryption of highly sensitive data like credit card information, and (b) encryption of data for which encryption does not impair scalability. As a result, the security-scalability tradeoff needs to be considered only over data for which encryption impacts scalability, thus greatly simplifying the task of managing the tradeoff. For full details of the scalability-conscious security design, please see our paper [26].

4.4 Evaluation

Here, we summarize the results from our paper [26]. We executed our benchmark applications (Section 3.5) on our prototype DBSS, described in Section 3 as per our experimental methodology, described in Section 3.6. In Section 4.4.1 we confirm that blanket encryption of all data passing through the DBSS greatly hurts scalability. Then, in Section 4.4.2, we show how our scalability-conscious security design methodology achieves good scalability with adequate security. Finally, in Section 4.4.3 we find that our scalability-conscious security design methodology enables significantly greater security without impacting scalability.

4.4.1 Magnitude of Security-Scalability Tradeoff

Figure 6 plots the scalability of an application as a function of the invalidation strategy used by the DBSS, for all three applications. The y-axis plots scalability, measured as specified in Section 3.6. On the x-axis, we consider an instance of each of the four classes of invalidation strategies introduced in Section 4.1.2. (The
same invalidation strategy is used for all update/query template pairs.) For the BBOARD application, in which each HTTP request results in about ten database requests, with the poor cache behavior of a blind or a template inspection strategy, not even a small number of clients can be supported within the response time threshold specified in Section 3.6.

For each application, the leftmost strategy, a minimal view inspection strategy (MVIS), offers the best scalability, but the worst security (full exposure of all data). On the other extreme, the rightmost strategy, a minimal blind strategy (MBS), offers the best security (full encryption of all data), but the worst scalability. Figure 6 confirms the claim made in Section 1 that blanket encryption of all data (thereby requiring a blind invalidation strategy) significantly hinders scalability.

4.4.2 Achieving Simultaneous Scalability and Privacy

The basic tradeoff between security and scalability is illustrated quantitatively in Figure 7, which shows measurements of the BOOKSTORE application executed on our prototype DBSS system. The vertical axis plots scalability, measured as the number of concurrent users that can be supported while keeping response times within acceptable limits. The horizontal axis plots a simple measure of security: the number of query templates embedded in the bookstore application for which query results are encrypted as they pass through the DBSS. It is straightforward to achieve either good security or good scalability by encrypting either all data or no data. The outcome of applying our static analysis idea is shown in the upper-right point in Figure 7, labeled “our approach.” The data that can be encrypted using our approach does not need to be considered for the security-scalability tradeoff, thus greatly simplifying the task of managing the tradeoff. Hence, for the benchmark applications we have evaluated, our approach automatically achieves good security\(^9\) without compromising scalability.

4.4.3 Security Enhancement Achieved

In this section we show that for all three applications, the static analysis step of our scalability-conscious security design methodology enables significantly greater security without impacting scalability. Recall Figure 7 of Section 1.1.2, which plots scalability\(^10\) versus security, for a simple metric of security that counts the number of query templates for which results can be encrypted. Our static analysis identifies 21 out of the 28 query templates associated with the BOOKSTORE application, for which encrypting the results has no impact on scalability. While encouraging, that result does not tell the whole story. Here we examine in greater depth the degree of security afforded by our static analysis.

The outcome of our static analysis depends on the initial determination of what highly sensitive data absolutely must be encrypted (details in our paper [26]). To make this determination, we defer to the well-known California data privacy law [9], which, when applied to our applications, mandates securing all credit card information.

\(^9\)See Section 4.4.3 for details on what data is kept private under our approach.

\(^10\)Computational overhead of encryption and decryption is not taken into account. Optimizing the encryption and decryption process is beyond the scope of the paper.
Figure 5 plots the exposure levels of query and update templates both before and after our static analysis is invoked. The top three graphs correspond to the query templates of each application, and the bottom three graphs correspond to the update templates. The y-axis of each graph plots the possible exposure levels for a template (low exposure on the bottom; high exposure on top). The x-axis plots the query or update templates associated with an application, in increasing order of exposure. The dashed lines show the initial exposure levels mandated by the California data privacy law (only a little encryption is needed to comply); the solid lines show the final exposure levels resulting from the application of our static analysis. The area between the lines gives an idea of the reduction in exposure achieved using our approach.

Much of the data whose exposure level can be reduced due to our static analysis turns out to be moderately sensitive, and therefore the reduction in exposure would likely be a welcome security enhancement. To illustrate, we supply examples of moderately sensitive data that can be encrypted:

- **Auction application**: the historical record of user bids (i.e., user A bid B dollars on item C at time D).
- **BBoard application**: the ratings users give one another based on the quality of their postings (i.e., user A gave user B a rating of C).
- **Bookstore application**: book purchase association rules discovered by the vendor (i.e., customers who purchase book A often also purchase book B).

In all cases scalability is not affected—it remains the same as that of MVIS in Figure 6.

5 Invalidation Clues

In this section, we use the following default “no-clue” scenario. The DBSS knows the application’s database schema, including the schema’s keys and foreign keys. It also knows the application’s query and update templates. On a query or update, the DBSS is informed as to which template has been used, but not the instantiated parameters. We will consider various scenarios where clues are added on top of this default scenario.

We begin in Section 5.1 by providing an example that illustrates the benefits of clues. Section 5.2 then introduces a taxonomy for clues and shows how different types of clues can be used to achieve different precisions in invalidations. Section 5.3 then describes in detail how clues can be used for precise invalidation, and the overheads involved in computing the clues. Section 5.4 next discusses how clues can be used to achieve a balance between scalability and privacy. Finally, Section 5.5 presents our empirical findings. For full details on any aspect of this work, please refer to the full version of our paper [25].

In this section, we will use *privacy* as a short hand for both security and privacy.
Table 5. A simplified auction example, consisting of a query template $Q^T$ and an update template $U^T$ on a base relation, items, with attributes item_id, seller, category, and end_date. The question marks indicate parameters bound at execution time.

<table>
<thead>
<tr>
<th>Query Clue</th>
<th>Update Clue</th>
<th>Query Result invalidated if</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>none</td>
<td>any update occurs</td>
</tr>
<tr>
<td>query result (unencrypted)</td>
<td>20060316, 7</td>
<td>item_id=7 in query result</td>
</tr>
<tr>
<td>item_id values in query result</td>
<td>7</td>
<td>item_id=7 in query result</td>
</tr>
<tr>
<td>Bloom-filter of item_id values</td>
<td>Bloom-filter of {7}</td>
<td>item_id=7 present as per Bloom-filter</td>
</tr>
</tbody>
</table>

Table 6. Four clue scenarios and their effect on what the DBSS invalidates when an update $U^T$ with end_date=20060316 and item_id=7 occurs.

5.1 An Illustrative Example

Consider a simplified application called SIMPLEST-AUCTION, specified in Table 5. In this application, queries follow the template $Q^T$ (requesting information on items that are being auctioned by a particular seller) and updates follow the template $U^T$ (setting a new end date for a particular item). The DBSS caches the (encrypted) results of previous queries and uses any clues at hand to decide what to invalidate on an update. Table 6 presents four different clue scenarios, and what happens when an update occurs. The first row depicts a scenario with no clues; in such cases, the DBSS has no way of knowing which (encrypted) cache result is invalidated by this (encrypted) update and hence must invalidate the entire cache. The second row depicts the opposite extreme in which the DBSS sees unencrypted query results and updates and hence can perform precise invalidation. While the former provides maximum privacy but minimum scalability, the latter provides maximum scalability but no privacy. In contrast, the third row uses better clues—they reveal less information (e.g., the item_ids, but not the sellers, categories, or end_dates), yet enables precise invalidation as before. The last row uses Bloom-filters\textsuperscript{11} to hide even the item_ids, at a cost of a small probability of an unnecessary invalidation. This shows how clues can offer an application a fine-grained control—the size of the Bloom-filter in this case—to choose a desired balance of security/privacy and scalability.

5.2 Using Clues for Invalidations

Here we describe how clues can be used for invalidations. We use the query and update model, terminology, and notation described in Section 4.1.1. We start in Section 5.2.1 by formalizing the notion of precise invalidations. Finally, in Section 5.2.2 we present various types of clues and provide examples of when each type is useful.

\textsuperscript{11}A Bloom-filter \cite{7} encodes a set as a short bit vector. Each value $v$ in the set is represented by setting the $h_1(v)$'th, $h_2(v)$'th and $h_3(v)$'th bit in the bit vector, for three hash functions $h_1$, $h_2$, and $h_3$. A query result is invalidated if the three bits set in the update clue Bloom-filter are all set in the query clue Bloom-filter. A larger Bloom-filter reduces the number of unnecessary invalidations but reveals more about the data.
Table 7. A slightly more elaborate auction example, consisting of three query templates, two update templates, and two base relations: (1) items with attributes item_id, seller, category, and end_date, and (2) users with attributes user_id and region. Attribute items.seller is a foreign key into the users relation. The question marks indicate parameters bound at execution time.

<table>
<thead>
<tr>
<th>SIMPLE-AUCTION</th>
<th></th>
</tr>
</thead>
</table>
| $Q_T^1$ | SELECT item_id, category, end_date  
 | FROM items WHERE seller=? |
| $Q_T^2$ | SELECT user_id FROM users WHERE region=? |
| $Q_T^3$ | SELECT item_id FROM items, users  
 | WHERE items.seller=users.user_id  
 | AND items.category=?  
 | AND items.end_date>=?  
 | AND users.region=? |
| $U_T^1$ | UPDATE items SET end_date=end_date+? DAYS  
 | WHERE item_id=? |
| $U_T^2$ | INSERT INTO users (user_id, region)  
 | VALUES (?, ?) |

Table 8. Types of clues required to implement a DIS for template-pairs of the SIMPLE-AUCTION example in Table 7.

<table>
<thead>
<tr>
<th>Pair</th>
<th>(Query clue, Update clue)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle Q_T^1, U_T^1 \rangle$</td>
<td>(result, parameter)</td>
</tr>
<tr>
<td>$\langle Q_T^1, U_T^2 \rangle$</td>
<td>(, ) (never invalidates: different relations)</td>
</tr>
<tr>
<td>$\langle Q_T^2, U_T^1 \rangle$</td>
<td>(, ) (never invalidates: different relations)</td>
</tr>
<tr>
<td>$\langle Q_T^2, U_T^2 \rangle$</td>
<td>(parameter, parameter)</td>
</tr>
</tbody>
</table>
| $\langle Q_T^3, U_T^1 \rangle$ | (database, parameter) or  
 | (parameter, database) |
| $\langle Q_T^3, U_T^2 \rangle$ | (, ) (never invalidates: foreign key constraint) |

5.2.1 Database-Inspection Strategy

We formalize the notion of precise invalidation as the invalidation behavior of an idealized strategy that can inspect any portion of the database to determine which cached query results to invalidate for a given update. A cached query result for a query $Q$ must be invalidated if the update alters the answer to $Q$. We call such a strategy a Database-Inspection Strategy (DIS). A DIS invalidates the minimal number of query results—any other (correct) invalidation strategy invalidates at least the query results invalidated by a DIS. Thus a DIS is a useful lower bound, against which we can compare how successful particular clues are in helping the DBSS make invalidation decisions.

5.2.2 Types of Clues

We can classify clues based on what data are used to compute them. A clue might be a parameter clue, a result clue, or a database clue, based on whether it is computed from the query parameters, the query result, or the database itself. Note that the contents of different types of clues may overlap. Intuitively, parameter clues incur the smallest overhead, whereas database clues potentially incur the largest overhead. Recall that we also distinguish between query clues (attached to encrypted query results) and update clues (attached to encrypted updates).

Consider the simple-auction application shown in Table 7, a slightly more elaborate auction than our earlier example. For each of its query/update template pairs, Table 8 lists the different kind of clues required to implement a DIS. In the first row, it suffices to have result query clues and parameter update clues, in order to implement a DIS. For example, the set of item_id values in the query result together with the
item_id from the update statement suffice. Invalidation is ruled out in the second and third rows simply by examining the templates. It is also ruled out in the last row because of the foreign key relationship. In the fourth row, only the region attributes need to be matched for a DIS—so the query and updates clues are just a function of their instantiated parameters. For the fifth row, invalidation of cached results of any instance of the query template $Q_I^T$ in response to an update template $U_I^T$ cannot be ruled out just by inspecting the query result, query parameters, or update parameters. For example, increasing the end_date may mean that the item in $U_I^T$ now satisfies the cached $Q_I^T$ query—but only if the item has the appropriate associated category and region (information only available in the database). So parameter and result clues are insufficient to prevent wholesale invalidation. Database clues are needed.

5.3 Database Clues

The previous subsection motivated the use of database clues using the simple-auction example. In this section, we first identify (Section 5.3.1) families of common query/update classes where database clues are required for precise invalidation. Section 5.3.2 discusses the problems with achieving precise invalidations using query clues, and then presents our solution using update clues. Finally, Section 5.3.3 presents practical techniques that further reduce overheads and/or increase privacy by relaxing the precise invalidation requirement.

5.3.1 Templates Requiring Database Clues

Using the query and update model of Section 4.1.1, we enumerate the query/update classes for which database clues are required for precise invalidation. For brevity, we defer the details to our paper [25].

5.3.2 Implementing Database Clues

We now discuss how to implement database clues, in order to achieve the precise invalidations of a DIS, while minimizing both the overheads and the amount revealed about the data.

Problems with Using Database Query Clues. One way to achieve a DIS would be to use database query clues. The goal for a database query clue is to provide all the data from the database that could potentially help in deciding if a future update would affect the given query result. Self-maintaining view techniques [33] could be used to identify the minimal such data. For example, for query template $Q_I^T$ in Table 7, the techniques in [33] would suggest the DBSS caches two database fragments: (a) the seller, category, and end_date of each item in the items table, and (b) the region of each user in the users table. For Web applications, because the set of update templates is known in advance, the amount of data stored can sometimes be reduced. In the previous example, because of the limited update templates, it suffices to cache all item_ids that satisfy all but the end_date predicate of the instantiated $Q_I^T$ query; these are the only rows that can possibly become part of the query result as a result of $U_I^T$ updating the end_date for some item. However, in general, given many cached queries and a richer collection of update templates, the amount of data stored can be quite large. As a result, this approach suffers from two significant problems. First, the cached portions of the database must themselves be maintained, resulting in additional overhead and additional clues to enable the maintenance. For example, maintaining the region information would mean that instances of update $U_I^T$, which could previously be ignored for $Q_I^T$ (because attribute items.seller is a foreign key into the users relation), can no longer be ignored. Second, because the approach potentially reveals large portions of the database, it does not offer any reasonable privacy.

Our Solution. Instead, our approach is to achieve a DIS by generating the relevant database information on-the-fly as database update clues. Because all updates are centrally handled by our system, such clues are computed at the home organization. Database update clues make sense in our setting where the query templates are known. For example, for the update template $U_I^T$ in Table 7, knowing the query templates enables the clue to be computed from just four values: the category of the specific item being updated, the old and new end_dates of the item, and the region of the specific seller of the item. Together with parameter query clues stored with an instantiated query $Q_I$, these enable a DBSS to achieve a DIS, by checking whether these four values now satisfy $Q_I$ as a result of the update.

With database update clues, there is no overhead of keeping them consistent because the clue is generated on-the-fly with every update. However, generating them each time places extra load on the home server’s organization. Hence, it is not obvious whether the increase in scalability from precise invalidation outweighs
the decrease in scalability from generating the clues. Fortunately, for the templates in the three realistic benchmarks we study, the work to generate a database update clue is rather minimal. In particular, out of the over 1000 (query template, update template) pairs, only 21 require database clues (details are in Section 3.5). Of these 21, almost all of them require fetching a single row from a table and perhaps a single associated row from a joining table, as in the \( \langle Q^T_3, U^T_1 \rangle \) example above. Moreover, for these same reasons, the amount revealed about the data tends to be small. For further details on how we compute database update clues including an algorithm for computing database update clues for simple query/update template pairs, please refer our paper [25].

### 5.3.3 Beyond Precise Invalidations

Thus far, we have focused on the goal of matching DIS’s optimal number of invalidations, while minimizing overheads and maximizing privacy. However, because of the minimal invalidations requirement, we have sacrificed opportunities to further minimize overheads and maximize privacy. In this section, we present several simple techniques that further reduce overheads and/or increase privacy by relaxing the precise invalidation requirement.

**The Hybrid Strategy.** Although as argued above, the overheads of computing a database update clue are minimal, it turns out that, depending on the workload, the overhead of computing a particular database update clue can be higher than its benefit. In the three benchmarks we study, there are cases where most of the invalidation savings arise from a small subset of the database update clues. While generating these clues is worthwhile, generating the other clues (where the savings is small) costs more than the savings. To address such concerns, we use a simple Hybrid strategy that monitors the workload for invalidation savings and then generates database update clues only when the savings exceeds an estimated threshold of the (appropriately normalized) cost to generate the clue. Although more wholesale invalidations are needed whenever we do not generate a database update clue, the overall effect is an increase in scalability, as shown in Section 5.5.

**Increasing Privacy through Hashing and Bloom-filters.** As argued above, for most updates the amount of revealed data is small (e.g., four values in the update clue for the \( \langle Q^T_3, U^T_1 \rangle \) example). However, even revealing four values per update may be more than desired if there are thousands to millions of updates. Fortunately, in many cases, the revealed values are used solely for equality tests with query parameters, e.g., the category and region values in the \( \langle Q^T_3, U^T_1 \rangle \) clue. In such cases, the actual values can be obscured by using a one-way hash function.\(^\text{12}\) The equality test is assumed to succeed if the hashed values match. Such an approach will always invalidate when required for correctness, but it introduces a very small probability of an unnecessary invalidation due to a hash collision. Thus, for all practical purposes, it is as good as a DIS strategy, but with better privacy.

In other common cases, the revealed values are used for order comparisons with query parameters, e.g., the end_date value in the \( \langle Q^T_3, U^T_1 \rangle \) clue. In such cases, the actual values can be hidden to varying degrees as a trade-off against invalidation precision, as will be discussed in Section 5.4.

Finally, another common case involves testing whether a particular value in an update clue is in a set of values in a result query clue. For example, consider the SIMPLEST-AUCTION example in Table 5 and the corresponding result query clue and parameter update clue in the third row of Table 6. These clues enable precise invalidation but reveal all the item_id values in the query result. Instead, as shown in the last row of Table 6, we can obscure these item_id values by using Bloom-filters [7], as discussed in the introduction. Although Bloom-filters introduce a small probability of unnecessary invalidations (the probability is tunable by the number of hash functions used in the filter and the size of the bit vector), for all practical purposes, it is as good as a DIS strategy, but with better privacy.

Although using hash functions and Bloom-filters are known techniques for increasing privacy, we adapt them to a new scenario (DBSS) and demonstrate their importance and effectiveness.

### 5.4 Privacy-Scalability Trade-off

In this section we study privacy-scalability trade-offs in the DBSS setting. For simplicity, we focus on scenarios where queries are from a single query template and updates are from a single update template. We begin in Section 5.4.1 by describing our attack model of the DBSS. We next show in Section 5.4.2 that there is a fundamental trade-off between privacy and scalability in our DBSS setting. Section 5.4.3 then

\(^{12}\text{In Section 5.4 we consider a stronger attack model that requires a more sophisticated approach to effectively hide values.}\)
Figure 8. Privacy-Scalability trade-off in the presence of clues. The dashed lines show the region in which an application can operate.

Table 9. A query-update template pair from the BOOKSTORE benchmark.

<table>
<thead>
<tr>
<th>Query Template</th>
<th>Update Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>$QT$ SELECT i_stock FROM item WHERE i_id=?</td>
<td>$UT$ UPDATE item SET i_stock=? WHERE i_id=?</td>
</tr>
</tbody>
</table>

presents an overview of how applications could get extra privacy by having the DBSS carry out unnecessary invalidations. Next, in Sections 5.4.4 and 5.4.5, we study representative query and update template pairs from our application benchmarks, and present configurable clues for these pairs. Finally in Section 5.4.6, we discuss how our current work applies to entire applications, beyond a single query and update template pair.

5.4.1 The Attack Model of the DBSS

In our model, a DBSS has a honest-but-curious behavior—it invalidates correctly as per the query and update clues, but tries its best to infer the contents of the encrypted query results, encrypted queries, and encrypted updates. Furthermore, a DBSS can pose as a user. Posing as a user provides the DBSS the following additional capabilities:

- It enables the DBSS to correlate unencrypted query or update parameters to clues, by posing such queries or updates and observing what clues are generated.
- It enables the DBSS to learn about encrypted query results, by observing whether or not a particular query result is invalidated in response to an update it poses.

5.4.2 The Limit Cases

Figure 8 illustrates the privacy-scalability trade-off that an application faces in our DBSS setting, where (a) the DBSS has an attack model as described in Section 5.4.1 and (b) the home server does not track the state of the DBSS’s cache. We show in our paper [25] that if an application achieves the maximum scalability, it gets minimum privacy (the upper left corner of Figure 8), and if it achieves the maximum privacy, it gets minimum scalability (the lower right corner of Figure 8).

5.4.3 Trading Off Scalability for Privacy

The results in the previous section show that applications cannot hope for both very good scalability as well as very good privacy. To achieve privacy, the applications have to sacrifice scalability—by allowing needless invalidations. Through representative query and update template pairs from our applications, we next show how clues provide applications with a convenient knob to balance their privacy and scalability needs. We consider two cases, depending on whether invalidations involve equality comparisons (Section 5.4.4) or order comparisons (Section 5.4.5).

5.4.4 Equality Comparisons

Consider an actual template pair, shown in Table 9, from the BOOKSTORE benchmark (details in Section 3.5) where the invalidation decision involves an equality comparison. For precise invalidation, the DBSS needs the attribute value $i_id$ in the query and the update. However, in creating a clue, applications want to limit the information that is revealed and may not want to reveal the exact $i_id$ value.
One natural way to do so is to map parameter values to some space of place-holders and then only reveal place-holders as clues to a DBSS. Let \( \{a_1, \ldots, a_n\} \) be the parameter values and \( \{e_1, \ldots, e_m\} \) be the place-holders. Let \( f \) be the function that determines the mapping. The mapping can be represented by a bipartite graph as in Figure 9. Computing the query or the update clue then just involves finding the place-holder corresponding to the parameter value. The DBSS invalidates a cached query result if the values of the place-holders in the query and update clue match. An example is the hash function discussed in Section 5.3.3.

In this setting, all that the DBSS can see is the place-holders. Using its capabilities, it can at most infer the mapping \( f \) used to generate the place-holders. A metric of privacy in this setting then is the number of place-holders \( m \) that the application chooses. The lower this number is, the better the privacy is. In the extreme, if there is just one place-holder, the DBSS can not learn anything about the parameters. On the other extreme, a higher \( m \) means the DBSS can more precisely infer the parameter values that get mapped to an encrypted value.

Because the query results of all constituent parameter values that are mapped to a single place-holder get invalidated whenever an update with any of the constituent values is issued, the value of \( m \) has an opposite effect on the scalability. A higher \( m \) usually means that there are less unnecessary invalidations, and the scalability is higher. Thus an application can tune the value of \( m \) to balance its privacy and scalability requirements.

Furthermore, an application can use knowledge of the frequency distribution of parameters to further choose clues that maximize its scalability for a given privacy value. We show in our paper [25] how applications can devise such clues. We call the mapping used to create such clues as the EQUAILITY-OPTIMAL mapping.

In Section 5.5.2 we show that in the common case, an EQUAILITY-OPTIMAL mapping reduces the number of invalidations by around 20%, when compared to a simplistic mapping which maps an equal number of parameter values to each place-holder. Thus if applications know the probability distribution with which parameters are chosen when issuing updates, they can choose clues that maximize their scalability for a target privacy.

### 5.4.5 Order Comparisons

Consider the template pair shown in Table 10. This pair is from the AUCTION benchmark (details in Section 3.5), and the invalidation decision involves an order comparison on the end date of an item being auctioned. For precise invalidations, the DBSS needs the attribute value \textit{end\_date} in the query and the update. However, the application may not want to reveal the exact \textit{end\_date} value.

As with equality comparisons, we can apply an approach based on mapping parameter values to some space of place-holders and then revealing only place-holders in the clues. Assume parameter values \( \{a_1, \ldots, a_n\} \) with \( a_1 < a_2 < \ldots < a_n \) and place-holders \( \{e_1, \ldots, e_m\} \) with \( e_1 < e_2 < \ldots < e_m \). Let \( f \) be the function that determines the mapping. The application can use an Order-Preserving-Encryption-Scheme (OPES) [2] to...

---

\[ Q^T \text{ SELECT * FROM items WHERE end\_date} \geq? \]
\[ U^T \text{ INSERT INTO items VALUES (?, ..., ?)} \]

Table 10. A simplified query-update template pair from the AUCTION benchmark.

---

\[ ^{13} \text{In general, the discussion here applies to all attribute values used in invalidation equality comparisons, not just parameter values.} \]
map the parameter values to place-holders such that the order is preserved. Use of an OPES ensures that if \(a_i < a_j\) then \(f(a_i) < f(a_j)\). Over time, a honest-but-curious DBSS can learn a total ordering on the place-holders. Under attack models where the DBSS can not initiate inputs to the system, privacy is preserved because the DBSS can not associate place-holders to actual parameter values. However, with our attack model, the DBSS posing as a user can initiate queries with known parameter values, and hence correlate place-holders to actual parameter values. In particular, it can use binary search to find the parameter value corresponding to a place-holder.

The ability to use binary search quickly allows the DBSS to find the parameter value(s) for any place-holder. For place-holders \(e_i\) and \(e_j\) with \(e_i < e_j\) in query clues, let \(a_k\) be the maximum value that gets mapped to \(e_i\) and \(a_l\) be the minimum value that gets mapped to \(e_j\). Formally, \(a_k = \max_{f(a_i)=e_i} a_i\) and \(a_l = \min_{f(a_i)=e_j} a_i\). The DBSS can use binary search because in all of the above formulations, \(e_i < e_j\) implies \(a_k < a_l\), i.e., the order is preserved when mapping parameter values of query. Thus any place-holder corresponds to a disjoint range of parameter values, whose end-points can be determined by binary search.

**Defeating binary search.** Our key observation is that for correct invalidations, the order has to be preserved only between parameters of queries and parameters of updates, and not across the parameters of queries and updates. This flexibility allows us to use two mapping functions \(f_q\) (to map query parameters) and \(f_u\) (to map update parameters) so that if \(a_i\) is a query parameter and \(a_j\) is an update parameter with \(a_i < a_j\), then \(f_q(a_i) < f_u(a_j)\).

One family of such mappings is where a non-negative number is subtracted from each query parameter and a non-negative number is added to each update parameter. Formally, \(f_q(a_i) = a_i - r_q(a_i)\) and \(f_u(a_j) = a_j + r_u(a_j)\), where \(r_q(a_i)\) and \(r_u(a_j)\) are always non-negative, but can even be randomly generated. With such a mapping, the DBSS can no longer use binary search to quickly find the parameters corresponding to a place-holder.

**A mapping with a provable guarantee.** Next, we show how an application can use the two mappings for greater privacy. Assume \(f_u\) is the identity function, i.e., \(r_u(a_j)\) is always zero. The choice of \(r_q\) allows the application to control its privacy-scalability trade-off. For parameter values \(a_1 < \ldots < a_n\), an application not wanting to let the DBSS learn the order information can measure privacy leak as the number of pairs for which the DBSS can figure out the correct ordering. Privacy \(p\) can then be measured simply by normalizing the privacy leak and subtracting it from 1. Formally, \(\text{privacy}(p) = 1 - \frac{2}{n(n+1)} \sum_{i<j} P(f_q(a_i) < f_q(a_j))\), where \(P(a_i < a_j) = 1\) if \(f_q(a_i) < f_q(a_j)\), \(1/2\) if \(f_q(a_i) = f_q(a_j)\), and 0 otherwise.

Under such a definition, we show in our paper [25] how for a fixed number of invalidations, an application can choose \(r_u\) values that maximize its privacy. We call such a mapping the ORDER-OPTIMAL mapping. We also assume that the probability of choosing any parameter while issuing the update is constant.

Section 5.5.2 shows that for a given scalability value, this mapping enables twice the privacy of an OPES.

### 5.4.6 Discussion

For our query and update model, any invalidation decision in an application fundamentally involves either an equality comparison (or its generalization to a set membership test) or an order comparison. Our above results can then be easily applied to the entire application. However, care must be taken in treating all the subcases as independent, because the query or update can enforce a relationship between attribute values.

### 5.5 Evaluation

We summarize our results from our paper [25]. We executed our benchmark applications (Section 3.5) on our prototype DBSS, described in Section 3 as per our experimental methodology, described in Section 3.6. We present results from our implementation of database clues in Section 5.5.1. Finally, in Section 5.5.2 we measure the effectiveness of our techniques in helping an application manage its privacy-scalability trade-off.

#### 5.5.1 Scalability Benefits of Invalidation Clues

Figure 10 plots the scalability of an application as a function of the invalidation strategy used by the DBSS, for all three applications. The y-axis plots scalability, measured as specified in Section 3.6. On the x-axis, we consider four cases: one corresponding to not using a DBSS, and the other three corresponding to DBSS strategies based on different classes of clues: Clues (excl. DB clues)\(^{14}\), which uses only parameter and

---

\(^{14}\)In Section 4 we denote this strategy as MVIS.
result clues, Clues (incl. DB clues), which uses parameter, result, and database update clues (as presented in Section 5.3.2), and Hybrid, which uses the HYBRID strategy presented in Section 5.3.3.

In all applications, using a DBSS with invalidation clues significantly increased scalability. This agrees with previous work [26], which can be viewed as having considered specific types of (non-database) clues. Because the rightmost strategy, Hybrid, heuristically uses database update clues only when the increase in scalability is higher than the overhead, it offers the most scalability, for all three applications. For the BBOARD application, the overhead of computing database update clues is high relative to the decrease in invalidations and, hence, using database update clues whenever required for precise invalidations results in worse scalability. Figure 10 confirms the claim made in Section 5.3.3 that the use of database update clues must be carefully weighed against the expected benefit.

5.5.2 Privacy Experiments

Figure 11 shows the reduction in the number of invalidations when we use our EQUALITY-OPTIMAL mapping algorithm described in Section 5.4.4. The y-axis plots the percentage reduction in invalidations in using our EQUALITY-OPTIMAL mapping over a simplistic mapping which maps an equal number of parameter values to each place-holder. (The percentage reduction is a crude estimate of how much scalability improvement an application can achieve by switching to an EQUALITY-OPTIMAL mapping.) On the x-axis, we plot the number of place-holders, which we increase in steps of three. The parameter values were chosen according to the Zipf-distribution we used in BOOKSTORE, over a domain of 100 possible parameter values. As expected, when all parameter values are either mapped to a single place-holder or are mapped to separate place-holders, any algorithm is as good as the EQUALITY-OPTIMAL algorithm. Similarly, when the number of place-holders is large (right part of the graph), all mapping algorithms result in almost the same number of invalidations. In other cases, however, the EQUALITY-OPTIMAL algorithm reduces invalidations by around 20%. Preliminary sensitivity analysis shows that the benefits increase as the parameter distribution becomes more skewed.

Figure 12 plots the improvement in privacy due to using two mappings instead of one mapping, as described in Section 5.4.5. The x-axis plots normalized privacy, measured as per the definition in that section. The y-axis plots normalized scalability, measured as \( \frac{\text{max} - I_j}{\text{max} - \text{min}} \), where \( I_j \) is the number of invalidations for the \( j \)th data point and max and min are the maximum and minimum, respectively, of the \( I_j \) over all data points \( j \). For the one mapping approach, we use an order-preserving encryption scheme in which multiple values could be mapped to a single value. For the two mappings approach, we use an identity mapping, and the ORDER-
Figure 12. Improvement in privacy on using two mappings instead of one mapping.

Figure 13. Increase in cache hit ratio due to use of a more precise invalidation strategy.

OPTIMAL mapping described in Section 5.4.5. For a given scalability, with our two mapping approach, the privacy is almost twice that of a one-mapping approach. Although we believe that these results are skewed by the specific privacy measure we use, we argue that the factor of two gap between the curves demonstrates a significant opportunity for two-mapping approaches tailored to more comprehensive privacy measures.

6 Proposed Work

Our proposed work can be divided into two categories: (a) a more realistic evaluation of the scalability benefits of different invalidation clues (Section 6.1), and (b) optimization of an application’s interaction with its database (Section 6.2).

6.1 Invalidation Clues

We intend to measure the scalability advantages of using different invalidation clues in the more realistic setting when (a) an application uses multiple DBSS nodes (so far in our experiments, an application uses a single DBSS node), and (b) each DBSS node starts with a warm cache (so far in our experiments, each DBSS node starts with a cold cache). We discuss these in detail next.

6.1.1 Using Multiple DBSS Nodes

Till now in our experiments, an application uses a single DBSS node. However, in a real deployment, an application is likely to use a network of DBSS nodes.

The key difference between using a single DBSS node and using multiple DBSS nodes is that in the latter, each DBSS node’s cache needs to reflect not only the updates that arrive at the DBSS, but also updates (for the application) that arrive at other DBSS nodes. From the perspective of a DBSS node, this difference means that as the number of DBSS nodes increases, its update rate increases almost linearly whereas its query rate remains constant. Depending on the query rate and when the queries are issued, this increase in the update rate can alter the relative advantages of using a more precise invalidation strategy at the DBSS.

For example, if the query rate is much higher than the update rate, then the relative scalability advantage of using a more precise invalidation strategy is likely to increase with increasing number of DBSS nodes. Consider Figure 13 which plots the effect of the number of simulated proxies against the increase in cache hit ratio of a more precise invalidation strategy for the BBOARD benchmark confirms this behavior. The X-axis plots the number of simulated DBSS nodes and the Y-axis plots the increase in cache hit ratio due to use of database clues. For this experiment, by partitioning the cache of a DBSS node, we could use the DBSS node to simulate multiple DBSS nodes. We plan to get similar graphs for the other two benchmarks.

The above experiment only shows the trend of how different invalidation strategies are likely to perform with increasing number of DBSS nodes. However, it does not provide a way to measure the actual scalability
benefit, because a single DBSS node is used to simulate multiple DBSS nodes. So we intend to obtain scalability numbers by having applications use a real setup of multiple DBSS nodes. One key challenge in doing so is in ensuring that a single DBSS node sees all updates in the system. To address this issue, we are adapting an existing publish-subscribe infrastructure to efficiently distribute updates among the network of DBSS nodes [13], an effort led by another group-member of our S3 project. We intend to use this infrastructure to obtain the final scalability numbers.

6.1.2 Using a Warm Cache

Till now, we started all our experiments from a cold cache. As a first cut, this method was fine since we were only studying the relative differences in scalability of the different invalidation strategies. However, now we want to provide measurements of the absolute scalability benefit. Since the application will use the DBSS continuously for extended periods, we need to start from a warm cache for measuring the absolute scalability.

The main difficulty in using a warm cache is that we currently do not have an indexing structure to quickly locate the cache entries to invalidate in response to an update. Instead, we currently identify such cache entries by doing a linear scan. The linear scan does not scale, and as the cache grows larger, more work needs to be done to identify such cache entries. In particular, in many cases, the scalability benefits of increased hit rate due to a large cache is offset by the increased overhead of identifying which cache entries to invalidate on an update.

Our plan is to build a simple indexing structure over the cache entries. Since invalidation decisions either involve equality comparisons or less than (greater than is equivalent to less than) comparisons, our indexing structure should support both these operations. Equality comparisons are easy to support. However, indexing support for less than comparisons is still an area of active research. Fortunately, less than comparisons are much less frequently required for making invalidation decisions than equality comparisons. So for our purposes, it would be fine even if the index for less than comparisons has a high-overhead.

6.2 Optimizing an Application’s Interaction with its Database

We intend to optimize an application’s interaction with its database. In the next two sections, we first focus on how we can optimize the collection of database requests that a program issues. Then, we discuss how to optimize the schedule of database requests issued by a program.

6.2.1 Optimizing the Collection of Database Requests Issued

As shown in Table 3, if there is no caching, there are many advantages to issuing one complicated query over many simple queries. In the three applications we studied, we found many such instances in which a single query could replace multiple inter-related queries. We believe that such instances exist because:

- Application programmers often do not know enough about databases to optimize the partition of computation between the procedural and declarative languages. They just write queries and updates that are convenient for them to write.

- Programmers are more comfortable expressing the logic in the procedural language of CGI, Java Servlets, PHP, etc. in which the application code is written than in the declarative language of SQL.

- Sometimes it is easier for programmers to express the logic in the procedural language, because it is closer to how the data is actually presented to the user. For example, in the BBOARD application, when displaying all comments made on a particular story in a tree format, it is more natural for programmers to perform a tree traversal and issue one query at each tree node than to get all comments at once and then display the data in a tree format.

For our proposed work, we first intend to evaluate the benefit of pushing the logic in the procedural code to declarative code (Table 3 was an example) in a centralized setting. Finally we plan to explore how we can combine the unique advantages of the two approaches of Table 3 in the DBSS setting.
Table 11. Prefetching results for three different load levels.

<table>
<thead>
<tr>
<th>Load Level</th>
<th>I</th>
<th>II</th>
<th>III</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coverage (C)</td>
<td>9.4%</td>
<td>7.6%</td>
<td>8.5%</td>
</tr>
<tr>
<td>% Filtered (F)</td>
<td>55.0%</td>
<td>65.2%</td>
<td>64.8%</td>
</tr>
<tr>
<td>Hits (H = C*(1-F))</td>
<td>4.2%</td>
<td>2.6%</td>
<td>2.9%</td>
</tr>
<tr>
<td>% Avg savings (A)</td>
<td>49.4%</td>
<td>52.8%</td>
<td>50.2%</td>
</tr>
<tr>
<td>Latency reduction (A*H)</td>
<td>2.0%</td>
<td>1.3%</td>
<td>1.5%</td>
</tr>
</tbody>
</table>

Prototype and three benchmarks
Security−Scalability Tradeoff
Invalidation Clues
Scheduling database requests
Optimizing the collection of database requests
Writing

Figure 14. Thesis timeline

6.2.2 Optimizing the Schedule of Database Requests

Recall Table 4 from Section 1.2.2, which shows how the schedule of database accesses can be optimized. We refer to the `execute_non_blocking` call as a `prefetch` since the call prefetches the query result. For a database access for which a prefetch was issued, we refer to the time the program waits for database access to be completed as the `late time`. Ideally, `late time` should be zero so that the latency of the database access is completely hidden. We refer to the time saved due to prefetching as the `saved time`.

Table 11 shows detailed prefetching results for the `BOOKSTORE` application based on our preliminary work. The first row shows the coverage, i.e., the number of prefetches issued per 100 HTTP requests. Some of the prefetches are filtered because the query result is in transit or is already present in the cache. The second row shows the filtered prefetches as a percentage of total prefetches issued. Since our prefetching algorithm is not speculative, all the prefetches that are not filtered are hits (third row), i.e., the result is later used. The fourth row lists the average savings due to a prefetch as a percentage of the average time spent by a HTTP request in the system. The last row shows the overall latency reduction possible due to prefetches.

Since our compiler pass did not speculatively prefetch a database query, there is no extra load on the origin server due to prefetches. At the same time, this conservative attitude reduces the coverage of our technique. It applies only in 3 out of 14 interactions defined by `BOOKSTORE` application, which together account for about 10% dynamic interactions at run-time in our experiment.

While the prefetching technique is effective, it suffers because of low coverage. To fix the coverage problem, we intend to make our technique less conservative. Formally, each program of a Web application can be represented as a directed acyclic graph, where the nodes are database accesses, and there is an edge between two nodes if one node has to be executed after the other node for correctness. Given this directed acyclic graph, a database accesses `i` can be issued as soon as all database accesses that are ancestors of `i` in the directed acyclic graph have completed. With this formulation, the client latency can be reduced significantly. For our proposed work, we intend to evaluate how this less conservative strategy affects the scalability of the three applications.

7 Timeline

Figure 14 shows our proposed schedule to complete the thesis. We plan to spend the next three months on getting complete results for invalidation clues in a multi-DBSS setting where each DBSS node starts from a warm cache. The subsequent three months, we plan to work on optimizing the schedule of database
requests. Finally, we plan to spend the last six months on optimizing the collection of database requests a Web application uses and on writing the thesis.

References


[27] Mark Matthews. Type IV JDBC driver for MySQL.


