Increasing the Scalability of Dynamic Web Applications

Amit Manjhi

CMU-CS-08-105

March 2008

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Bruce M. Maggs, Co-Chair
Todd C. Mowry, Co-Chair
Christopher Olston, Co-Chair
Mahadev Satyanarayanan
Michael J. Franklin, University of California at Berkeley

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Keywords: Scalability Service, Web Applications, Scalability, Optimization, View Invalidation, View Materialization.

To my parents and my wife Shruti.

Abstract

The continued growth of the Web and its increasing role in our daily life has created new technical and social challenges. On the technical side, applications deployed on the Internet suffer from unpredictable load, especially due to events such as breaking news (e.g., Hurricane Katrina) and sudden popularity spikes (e.g., the "Slashdot Effect"). A large number of these Web applications increasingly use a database to generate customized and personalized responses to users' requests. Because of the widely varying load, currently there is no economical way to provision infrastructure for many of these applications in which *the database system is the bottleneck*. On the social side, Web applications increasingly collect sensitive data, which must be kept private.

In this dissertation we address both these technical and social challenges. We design and implement a Database Scalability Service (DBSS), which can offer scalability to data-intensive applications as a plug-in subscription service with a per-usage charge. A DBSS works by caching applications' data and answering queries on their behalf. It uses a large shared infrastructure to absorb load spikes that may occur in any individual application. We address two key issues in designing a DBSS: (a) the privacy concerns of applications in allowing the DBSS to cache their data, and (b) the performance concerns due to the high latency applications face in accessing their data in a DBSS setting.

Simply encrypting all the data that passes through the DBSS is not a feasible solution to an application's privacy concerns. On an update, the DBSS must invalidate (at least) data from its cache that have changed . If an application encrypts all the data passing through the DBSS, the DBSS cannot discern any information about what data it is caching. The DBSS then is forced to invalidate large amounts of data from its cache on any update, which leads to poor scalability. In deciding how much data (that passes

through the DBSS) to encrypt, the application faces a tradeoff between privacy and scalability. On the one hand, encrypting *more* data means that the DBSS will invalidate far more than needed, decreasing scalability. On the other hand, encrypting *less* data raises privacy concerns. We study this tradeoff both formally and empirically. To simplify the task of managing this tradeoff, we devise a method for statically identifying segments of the database that can be encrypted without impacting scalability. Experiments with three realistic benchmark applications show that our static method is effective. For each application, it identifies a significant fraction of the database that can be encrypted without any scalability penalty. Moreover, most of the data that it identifies is "moderately" sensitive, which application designers will want to encrypt, if doing so has no performance overhead.

For some applications, extra information from the database, beyond the data passing through the DBSS, is useful in making invalidation decisions. We present invalidation clues, a framework that allows applications to provide this extra information to the DBSS. Clues also provide fine-grained control to the applications for disclosing any other information to the DBSS that reveals little, yet limits the number of unnecessary invalidations. Our experiments using three Web benchmark applications on our prototype DBSS confirm that invalidation clues are indeed a low-overhead, effective, and general technique for applications to balance their privacy and scalability needs.

To address the performance concerns due to the high latency an application faces in accessing its data in the DBSS setting, we devise compiler-driven transformations that reduce the number of times an application must access its data. Using our three benchmark applications, we show that our transformations apply widely and indeed reduce the number of times an application has to access its data. Finally, on our prototype DBSS, we confirm that this reduction significantly improves scalability.

Acknowledgments

I was fortunate to have three great advisors: Todd Mowry, Bruce Maggs, and Chris Olston. I learnt from Todd the value of patience, from Bruce the value of humor, and from Chris the value of time-management and the practice of setting and achieving goals. Thank you Todd, Bruce, and Chris for making me a better researcher and a better person.

I would like to thank the other members of my Ph.D. thesis committee, Mahadev Satyanarayanan and Mike Franklin, for their thoughtful comments and invaluable suggestions that have improved both the quality of the experimental results and the completeness of this thesis.

Other than my advisors, I discussed my thesis research with Anastassia Ailamaki, Anthony Tomasic, Charlie Garrod, Phil Gibbons, and Haifeng Yu. These discussions were invaluable in shaping this thesis.

I wrote a few other papers resulting from my course projects and summer internships. I learned a lot from my co-authors on those papers, Mukesh Agarwal, Nikhil Bansal, Srini Seshan, Kedar Dhamdhere, Vladislav Shkapenyuk, and Suman Nath, about how research is done and how to best present it. I would also like to thank members of the stampede group, particularly Chris Colohan and Shimin Chen, for always being present to help me with any research or non-research related issue.

I had two memorable summer internships at Intel Research Pittsburgh. Even after the internships, I had a cubicle for almost a year for being affiliated with the lab. I want to thank all the people in Intel Research Pittsburgh for making my stay enjoyable. In particular, I am greatly indebted to Phil Gibbons, my mentor during both summers, for making my internships productive and fun.

In my sixth year, with two other graduate students, I started Buxfer.com, a Web application that twenty-somethings could use to manage their expenses. It was a great experience that taught me how

to build reliable systems, and instilled in me the confidence, discipline, and determination to achieve seemingly impossible tasks. I would like to thank everyone who contributed to my Buxfer experience.

Seven years is a long time, but I never got bored of Pittsburgh, thanks to the Indian dinner gang – a bunch of mostly Indian graduate students who were often too lazy to cook at home. The composition of the group changed over the years as many of the seniors graduated and new students joined the group, but it was always fun to enjoy good, often half-priced, food in good company.

Finally, I express my deepest gratitude to my family. I owe a great deal to my parents, my sister, and my brother-in-law who were always affectionate and extremely supportive during the entire period. I am indebted to my wife, Shruti, herself a graduate student at Georgia Tech, for providing unwavering encouragement. Without her love, patience, encouragement, and support in the last few years of my Ph.D., it would have been difficult for me to finish.

Contents

List of Figures

List of Tables

Chapter 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., Hurricane Katrina) and sudden popularity spikes (e.g., the "Slashdot Effect"). Administrators currently face a "provisioning" dilemma: either (i) waste money by heavily overprovisioning the infrastructure, or (ii) risk being unavailable whenever the load increases suddenly. Both of these alternatives have a high economic cost and are undesirable.

This problem is largely addressed for static content (e.g., images) by Content Delivery Network (CDN) technology [23, 38, 71, 87], which offers on-demand scalability as a plug-in service. To serve a request for static content, an application just returns the requested file from its filesystem. By caching the static content and returning the appropriate file on a client request, CDNs can effectively offload the work an application needs to do for serving static content. To absorb load spikes (which may occur in any individual application) and yet be cost-effective, CDNs make use of a large infrastructure that is shared across multiple applications. Hence CDNs can offer good scalability and charge application providers on a per-usage basis.

The Web is increasingly becoming more "dynamic" – the content is produced by programs that execute at the time a request is made and is often customized based on several factors like a user's preferences and the previous content the user has viewed. Dynamic content allows creation of rich interactive

applications like social networks, bulletin boards, civic emergency management, and e-commerce applications, which represent the future landscape of Web applications. Since dynamic content is generated by programs and may depend on data not contained in the user's request, CDN technology cannot be used to serve dynamic content. Hence CDN technology is not sufficient for scaling dynamic applications.

With dynamic applications, application administrators face the same provisioning problem. **In this thesis we show that it is possible to build a subscription-oriented scalability service that provides on-demand scalability to dynamic Web applications.**

In the rest of this chapter we first illustrate the potential benefits of subscription-oriented scalability services for dynamic Web applications with two example scenarios in Section 1.1. Next we discuss the challenges in creating a scalability service for dynamic Web applications in Section 1.2. Section 1.3 presents related work. We present an overview of our approach in Section 1.4, our contributions in Section 1.5, and thesis organization in Section 1.6.

1.1 Example Scenarios

1.1.1 E-Commerce

Consider a relatively small-scale Web-based e-commerce operation whose customer base is expanding. Suppose the relatively low-cost equipment on which the e-commerce site was originally built is becoming saturated with load, and will soon be unable to serve all the customers. Standard solutions include upgrading to faster equipment on which to run the web, application, and/or database servers, or moving to a parallel cluster-based architecture as used by big e-commerce vendors. Unfortunately, these solutions require a large investment in equipment, and, perhaps more significantly, funding for staff with the expertise necessary to manage the more complex infrastructure. Moreover, transitioning to a new architecture will undoubtedly create new bugs and may lead to costly application errors or system downtime.

A better option would be to subscribe to a scalability service on a pay-by-usage basis. A cost curve proportional to usage could potentially save the company large sums of money, especially if demand plateaus or drops. In this scenario, the equipment and management costs are shifted to the scalability service provider, where they can be amortized across many subscribers.

1.1.2 Civic Emergency Management

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

This application lends itself naturally to a Web-based implementation, but there are several inherent difficulties. First, demand for the application is likely never to occur, but if it should occur, it will come very quickly and as a large spike. It would be costly for the city to invest in enough permanent infrastructure to satisfy the demand, and not cost-effective to keep this infrastructure idle. Second, it is critical to give all end users prompt and reliable access to information. It is even more important that data collected from end users requiring immediate assistance be recorded reliably. Third, the delivery of information customized to each end user, such as the generation of maps and directions, requires significant computational resources. This information cannot easily be conveyed through a telephone conversation, and in any case, the scale of the demand would make a call-center solution impractical.

The ability to tap into a scalability service would solve these difficulties. The city would have to prepare software in advance and maintain a modest amount of permanent infrastructure, but could rely on the scalability service to shoulder the network and computational load, when demand suddenly arrived.

```
$template := SELECT from_user_id
             FROM comments
             WHERE to_user_id = ?;
$query := set_params ($template, $to_id);
$result := execute ($query);
foreach ($row in $result) {
    $from_id := get_user_id ($row);
    $template := SELECT user_name
                 FROM users
                 WHERE user_id = ?;
    $query := set_params ($template, $from_id);
    $result2 := execute ($query);
}
```

Figure 1.1: A sample code fragment from the AUCTION application for finding names of users who have posted comments about a particular user. The code fragment shows how the declarative code, consisting of two query templates, is interspersed in the procedural code. 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`.

1.2 Challenges in Creating a Scalability Service for Dynamic Web Applications

The example scenarios in Section 1.1 illustrate the potential benefits of a plug-in scalability service for current and future dynamic Web applications on the Internet. Constructing such a service for dynamic applications is much more challenging than doing so for traditional static content.

Web applications are collections of programs commonly written in a procedural language like Java or PHP. On an HTTP request, one or more of these programs is run to generate the response. The programs interact with the application's database, which houses and manages the application's data, by issuing queries and updates. These queries and updates are (typically) constructed at runtime by setting the missing parameter values in the query and update *templates* – queries or updates, embedded in an application's code, missing zero or more parameter values. Figure 1.1 shows a sample code fragment from one of the benchmark applications we use. The code fragment finds the names of users who have

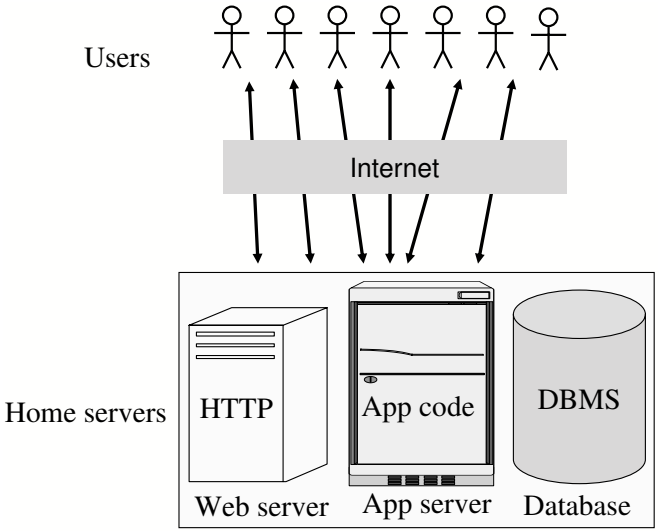


Figure 1.2: Traditional centralized architecture.

posted comments about a particular user. To achieve this task, the code uses two query templates, which find: (i) the user identifiers of all users who have posted comments about a particular user, and (ii) the username of a user given her identifier. The code first issues a query based on the first template. Then for each of the user identifiers returned in the result, it issues a query based on the second template.

Web applications are typically deployed on a three-tiered server-side architecture consisting of one or more instances of: a web server, an application server, and a database server. The web server manages the HTTP interactions, the application server runs the application code, and the database server houses the application's database. We call these server(s), maintained by the application, its *home server(s)*. (Figure 1.2 shows the resulting architecture.) The key to scalability is to ensure that the home server(s) remains lightly loaded even at high request rates. Because an application's web servers and application servers do not carry any persistent state, they can be replicated so that each replica remains lightly loaded even at high request rates. Alternatively, an application can use a CDN that executes application code to scale its web and application server. In fact, Akamai's EdgeSuite for Java service already provides this functionality [58].

The main challenge is to design the database part of the scalability service, or a *Database Scalabil-*

ity Service (DBSS), which can effectively offload at least some of the database work from the home infrastructure's database server(s). We describe these challenges below.

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

Maintaining privacy and security of data in the face of updates In both the example applications in Section 1.1, it is desirable that the DBSS, a third-party service, does not learn anything about the application's data. Privacy is critical more so due to the well-publicized instances of database theft [103] and the security legislation in the California Senate [24]. Guaranteeing privacy without affecting scalability is a challenge in itself; doing so in presence of distributed updates, common in dynamic applications, is even more difficult. Section 1.4.1 provides details on how updates increase the difficulty of simultaneously maintaining security and providing good scalability.

Guaranteeing data consistency One method to reduce the load on the database server(s) is to cache or replicate the data at multiple nodes. Because users of a Web application are geographically distributed, to reduce user latencies, often these caches or replicas are widely distributed as well [6, 8, 66]. Many Web applications require strong consistency for their most important data. For example, in our civic emergency management scenario (Section 1.1.2), inventory data for emergency supplies must be managed precisely – inconsistencies could cost lives. It is well-known that maintaining strong consistency among replicas in a distributed setting presents significant scalability challenges [48].

Keeping user latency acceptable Most Web applications are interactive. High user latencies drive customers away, nullifying any advantage that using a scalability service provides [59, 60]. Hence to be effective, a scalability service should meet another key requirement besides offloading work from the application's home server(s) – the final user latency of a response generated using the scalability service should be acceptable. Meeting this requirement in a scalability service, whose nodes are distributed all over the Internet, can be challenging.

1.3 Related Work

In this section we provide an overview of the related work. Prior work related to ours can be partitioned into four categories: database services, database caching and replication, privacy, and commercial efforts. We discuss each in turn.

1.3.1 Database Services

As we outlined in Section 1.2, the key challenge in building a scalability service is scaling the database component. Existing work on providing *database services* can be classified into two categories: the Database Outsourcing (DO) model and the in-house database scalability model. In contrast, we propose the Database Scalability Service (DBSS) model.

In the DO model, an application outsources all aspects of management of its database to a third party [55]. 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 [3, 54, 56]. Aggarwal et al. [2] suggest an alternative—distribute data across multiple independent providers that do not communicate with one another.

In the in-house database scalability model [6, 8, 12, 66, 70, 72], other machines within the application's organization are used to cache data and answer query results on behalf of the database server(s). While these approaches are cheaper than buying larger database server(s), they still suffer from the same fundamental provisioning problem, i.e., how much caching infrastructure to provision. Hence these approaches are uneconomical for Web applications, where the load is highly variable and unpredictable.

In contrast to work in the DO and the in-house database scalability models, we consider the DBSS model, in which only *database scalability*, and not full-fledged database management, is outsourced to a third party [84]. Under the DBSS model, application providers retain master copies of their data on their own systems, with the DBSS only caching and serving read-only copies on their behalf. In our DBSS 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).

1.3.2 Database Caching and Replication

Remote caching of database objects first received significant attention during the late 1980's as a technique that improves performance without sacrificing strong consistency or 1-copy serializability [17], in client-server object-oriented databases. The key issues addressed were: should consistency be maintained by propagating changed data or invalidating cached data at remote clients (e.g., [43]), which locking mechanisms should be used [26, 43], and whether "objects" or "data pages" should be shipped from the server to the clients [37, 44]. The work in this area did not explore update propagation methods or the security concerns due to caching of data.

More recently, database caching has been investigated as a means to scale dynamic Web applications. Compared to client-server object-oriented databases, Web applications see a wider variation in the user load, as discussed in Section 1.1. Because of this high variance in user load, we believe that the efforts in the in-house database scalability model [6, 8, 12, 66, 70, 72], will continue to suffer from the "provisioning" problem and will have limited applicability. Web applications may serve millions of users, who are geographically distributed. To improve scalability and user latency, many systems put database caches on the edge of the network [6, 8, 66]. However, in a wide-area network (where there is a possibility of network failures), strong consistency and good performance cannot be guaranteed simultaneously [16, 42, 48]. Most caching systems sacrifice consistency for performance – for example, in the DBCache [72] and DBProxy [8] projects at IBM and the MTCache [66] project at Microsoft Research, an authoritative copy is maintained at the back-end database(s) and caches are regularly updated to keep them consistent with the back-end database(s). In addition, techniques like specifying a freshness constraint for queries [5], pre-declaring the access patterns of all write transactions [11], handling the read and write transactions separately [89], and leveraging the statically available query and update templates [9, 32] have been proposed to lower the overhead of maintaining a desired consistency. With

our work, the application developer can specify the parts of the application for which she desires strong consistency; for everything else, best-effort consistency is provided.

Our approach of scaling the database involves caching materialized views and invalidating them when data updates render them obsolete. Levy and Sagiv [68] 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 [52] 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 [51] provide techniques to update views using a subset of the query statement, the update statement, and the updated base relation. Quass et al. [90] 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.

The works cited above are special cases of clues (Chapter 4). However, they do not address privacy concerns. Furthermore, we demonstrate the necessity and advantages of specially designed "database-derived" clues, in order to achieve precise invalidations (Section 4.4.2). The work closest to this in technique is by Candan et al. [25]. 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 file system for mobile computing environments, Mummert et al. [78] maintained cache entries at multiple levels of granularity which allowed trading precision of invalidation for speed of invalidation. Satya [95] 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. 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.

1.3.3 Privacy

A key challenge of a DBSS is providing this shared scalability infrastructure while protecting each organization's sensitive data. 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. [4] 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. [3] present order-preserving encryption schemes. Since the encryption schemes preserve order, these schemes could be used to enable order-comparisons over encrypted data like 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.

Much work has been done on privacy metrics, starting with the work on k-anonymity [101]. Under k-anonymity, each record is indistinguishable from at least k-1 other records with respect to some "quasi-identifying" attributes. There has been follow-up work on creating efficient algorithms for k-anonymity using generalization and tuple suppression techniques [14, 67, 102]. Several improvements over k-anonymity have also been proposed [69, 73]. Irrespective of which of these metrics is used to measure privacy, the privacy-scalability tradeoff exists in the DBSS setting. These different metrics simply influence the exact values in the privacy-scalability curve. For our experiments regarding equality-comparisons over clues, we use the simple metric of just measuring the number of distinct values revealed to the scalability service.

Hore et al. [57] study the privacy-utility tradeoff in the choice of the "coarseness" of the index on encrypted data. Our bucketization technique in Section 4.5.3 is similar. However, the different domain we consider requires different optimization objectives.

1.3.4 Commercial Efforts

Akamai Technologies, a leading CDN, has an "EdgeJava" product, which allows Web content providers to execute Java servlets on Akamai's proxy servers. For example, a significant use of EdgeJava was a widely advertised promotion staged by Logitech Corporation, in which peak demand exceeded 60,000

user requests per second. For each request, a Java servlet dynamically generated an HTML document, indicating whether the end user was a winner, which was then served to the end user's browser. Other Akamai customers have used EdgeJava to perform server-side transformations of XML to HTML. Akamai provides weak consistency for cached data via TTL-based protocols, with the option of associating "do-not-cache" directives with objects that require strong consistency.

In the Database Outsourcing (DO) model discussed in Section 1.3.1, there have been many recent commercial efforts [1, 7, 33, 93]. While these services promise on-demand database scalability and follow a pay-per-usage model, currently they only provide rudimentary database functionality, both in terms of the data model as well as the querying support. The most notable among these services, Amazon's SimpleDB service, supports a simple data model where each table is modeled as an independent hash table. Even the foreign key relationships between tables cannot be specified. As a result of the simple data model, the query functionality is rudimentary as well – limited to lookups in a single table based on a simple predicate. Therefore these services, in their current state, can not be used by any dynamic Web application that uses its database extensively. In addition, to use these services, applications need to trust the service providers with their data.

Neither Akamai nor any of these commercial efforts in the DO model has, as yet, addressed or even explored the security concerns of a third-party service storing data from several Web applications.

Avokia (www.avokia.com) is another notable commercial effort which aims to scale the database independently for each application running in a centralized setting. Since Avokia's solution is in the centralized setting, it will inevitably suffer from the provisioning problem.

1.4 Our Approach

Our approach exploits two basic properties of most Web applications: the underlying data workloads tend to (1) be dominated by reads[1], and (2) consist of a small, fixed set of query and update templates.

[1]As per [61], the "visits to media upload" ratio is 20:1 for `wikipedia.org`, 500:1 for `flickr.com`, and 600:1 for `youtube.com`, three popular dynamic content sites on the Internet today. The "reads to writes" ratio should be similar to

For the diverse set of benchmark applications we studied, the number of templates varied between 10 and 100 – details in Figure 2.6. The first property makes it feasible to handle all data modifications at each application's home server(s). With this approach, no data updating is performed outside of the home organization, and tight control over authentication of updates and overall data integrity is retained. We exploit the second property, i.e., predefined query and update templates, to (1) provide best-effort consistency that places almost no overhead on the home server(s) of an application, and (2) ensure privacy and security of data in face of updates. Levy and Sagiv [68] provide several heuristic methods for determining when query statements are independent of updates in many practical cases. This work can be easily extended to determine (at compile-time) the query templates that are independent of an update template. On an update, this analysis enables the DBSS node to not only quickly narrow down the candidate query results to consider for invalidation but also to be sure that the invalidation is precise (details in Chapter 3). Such an analysis can also be used by the DBSS to find data that is not useful for invalidation. Such data can be secured without affecting the scalability in the DBSS scenario (details in Chapter 3 and Chapter 4).

We start in Section 1.4.1 by presenting our overall architecture of a scalability service for dynamic content applications and our approach to data consistency. Section 1.4.2, Section 1.4.3, and Section 1.4.4 address the challenge of maintaining privacy and security in the face of data updates. Lastly, Section 1.4.5 addresses the challenge of keeping final user latency acceptable.

1.4.1 Overall Architecture

Figure 1.3 depicts the overall architecture of a scalability service, in which (1) a Web application's code is executed at trusted hosts (application "servers"), shown in Figure 1.3 as the CDN, (2) the code 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 back-end databases within the application's home organization. We have built a prototype DBSS with this general architecture, and used it to scale three

this ratio. Figure 2.11 lists the writes to reads ratio for our benchmark applications – reads still dominate writes, although the ratio of reads to writes for the benchmark applications is lower than [61].

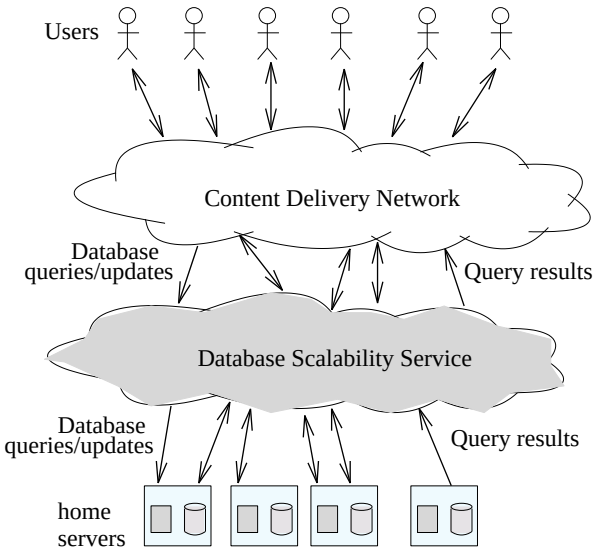


Figure 1.3: Scalable architecture for database-intensive Web applications. In this thesis, we focus on the Database Scalability Service (DBSS), the shaded cloud.

data-intensive Web benchmark applications. We provide more details about the overall design and implementation of our scalability service in Chapter 2.

Any distributed system must grapple with the issue of consistency. Rather than explore the wide space of potential solutions to the consistency problem, we adopt a simple model of consistency. Our model is based on the insight that only some data in Web applications require strong consistency. For example, only inventory data for emergency supplies in our civic emergency management example (Section 1.1.2), where inconsistencies could cost lives, require strong consistency. For other applications like bulletin-boards, strong consistency is not required. So we support two levels of consistency guarantees – the application developer can specify the parts of the application for which she desires strong consistency; for everything else, best-effort consistency is provided. Additionally, we exploit the fact that Web applications use a predefined set of query and update templates to provide best-effort consistency that places almost no overhead on the home server(s) of an application. Finally, we believe that recent work [15, 49, 50], which leverages the differing consistency needs of different data to guarantee serializability in an environment where each read operation carries a "freshness constraint", can be used to

improve the consistency guarantees in our scenario.

1.4.2 Guaranteeing Privacy and Security in a DBSS Setting

As we pointed out in Section 1.2, a key challenge 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 [103]. We use *privacy* to denote these concerns. We use *security* to denote a special case of privacy. While the goal of privacy is to limit the data from being observed or inferred, the goal of security is to just limit the data from being observed. In Chapter 3, we address the security concerns. In Chapter 4, we expand the discussion to 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, and the DBSS caches read-only (encrypted) copies of query results that are kept consistent via *invalidation*.

Security/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 1.4.3, or by sending invalidation clues, a more general technique we present in Section 1.4.4 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, decreasing scalability. On the other hand, revealing *more* about the data to the DBSS raises

security/privacy concerns.

1.4.3 Scalability-Conscious Security Design Methodology

We study this security-scalability tradeoff, both analytically and empirically, in Chapter 3. To help manage this tradeoff, we present a static analysis method in Section 3.3 for identifying segments of an application's database that are never useful for invalidation decisions. The application administrator can stop worrying about making such data available to the DBSS. Moreover, for all three benchmark applications we study (details in Section 2.6), 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[2].

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.

1.4.4 Invalidation Clues

We present *invalidation clues*, a general framework (the solution of Section 1.4.3 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.

[2]See Section 3.5.3 for details on what data is kept private using our static analysis method.

- *Limit revealed information:* Our clues enable the application to achieve a target level of 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 4.2 for an illustrative example of how clues enable applications to balance their security/privacy and scalability requirements.

We present all details on invalidation clues in Chapter 4. We show how invalidation clues offer applications a low overhead control to balance their privacy and scalability needs at a fine granularity. 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 benchmark applications we study (described in Section 2.6).

1.4.5 Holistic Transformations to Reduce User Latencies

As discussed in the previous sections, Web applications can use the services of a secure scalability service to get on-demand scalability, at an economical pay-per-usage rate. However, just being scalable is not sufficient. Web applications, in addition to being scalable, must be interactive. After clicking a web link or typing a URL into the browser's address bar, a user expects the content to appear within at most a few seconds. High user latencies can drive customers away, nullifying any scalability advantage that using a scalability service provides [59, 60].

To keep the end user latency low, it is necessary to understand the factors that contribute to this latency in a scalability service setting (Figure 1.3). A Web application (Figure 1.1 shows a fragment) is a collection of programs. On any HTTP request, one or more of these programs is executed. To obtain the data

these programs need to generate the response, they issue database queries. Frequently, these programs issue multiple queries for each HTTP request: for the benchmark applications we study, the average number of queries per HTTP request varies between 1.8 and 8.5, as in Table 5.3. Furthermore, for every database query that misses in the DBSS cache, the user must endure the long latency of accessing the home server database. Hence to keep user latencies low, it is desirable to either reduce the number of database requests or hide their latencies.

In Chapter 5 we present two transformations: one for reducing the number of database requests and the other for hiding the latency of database requests. These transformations change both the database queries as well as the code surrounding the queries. Web application code is typically in a procedural language like Java or PHP, while the database queries are in a declarative language like SQL. These transformations require an understanding of both the procedural and declarative languages. They treat the program as a whole. Therefore, we call these transformations *holistic*. In Chapter 5 we discuss why opportunities for applying these transformations will continue to exist in current Web application code, present algorithms for automating these transformations in a source-to-source compiler [39, 81], and evaluate the effects of applying these two transformations to Web applications, both in a traditional as well as a scalability service setting.

1.5 Contributions

The primary contributions of this thesis are the following:

- [Chapter 2] We design, build, and evaluate the *first* prototype of a Database Scalability Service (DBSS).
- [Chapter 3] We present a convenient shortcut to managing the security-scalability tradeoff in the DBSS setting. Our solution is to (statically) determine which data can be encrypted without any impact on scalability. We confirm the effectiveness of our static analysis method, by applying it to three realistic benchmark applications that use a prototype DBSS system we built. In all three cases, our static analysis identifies significant portions of data that can be secured without

impacting scalability. The security-scalability tradeoff does not need to be considered for such data, significantly lightening the burden on the application administrator managing the tradeoff.

- [Chapter 4] We propose *invalidation clues*, a general framework for enabling applications to reveal little data to the DBSS, yet provide sufficient information to limit unnecessary invalidations of results cached at the DBSS. Compared with previous approaches, our proposed invalidation clues provide increased scalability to the DBSS for a target security/privacy level, as well as more fine-grained control of this tradeoff. Using three realistic Web benchmark applications, we illustrate the issues and solutions for generating effective clues, e.g., by identifying categories requiring database clues, and then evaluate the effectiveness of our solutions on our DBSS prototype.

- [Chapter 5] We propose two holistic transformations to reduce the user latency of an application executing in a DBSS setting. We discuss why opportunities for applying these transformations will continue to exist in Web applications and present algorithms for automating these transformations in a source-to-source compiler. We finally evaluate the effect of these two transformations on three realistic Web benchmark applications, both in the traditional centralized setting and the DBSS setting.

1.6 Thesis Organization

Chapter 2 describes our overall design and implementation of a scalability service for applications where the *database system is the bottleneck*. The novel piece of such a service is the Database Scalability Service, and Chapter 2 describes it in detail. It also describes the benchmark applications we use to evaluate the DBSS, describes the setup and evaluation methodology we use for all our experiments, and presents results confirming that the database system is indeed the bottleneck in the benchmark applications we study.

Chapter 3 describes the security-scalability tradeoff in a DBSS setting. It provides a formal characterization of view invalidation strategies in terms of what data they need to access. Based on this characterization, it presents a static method for automatically identifying data that is not useful for in-

validation and thus can be encrypted without reducing scalability. Results on the three benchmarks we study confirm the effectiveness of our technique.

Chapter 4 describes invalidation clues, how different types of clues can be used to achieve different precisions in invalidations, how clues can be tailored to balance between privacy and scalability, and our empirical findings using clues.

Chapter 5 describes two transformations for reducing the latency experienced by users of a Web application executing in a DBSS setting. It argues why opportunities for applying these transformations exist, identifies the various code "patterns" where these transformations apply, discusses the consequences of these transformations other than the reduction in latency, and measures the effects of such transformations on three benchmark applications in a centralized as well as a DBSS setting.

Finally, Chapter 6 contains a summary of the important results in this thesis, and discusses their implications.

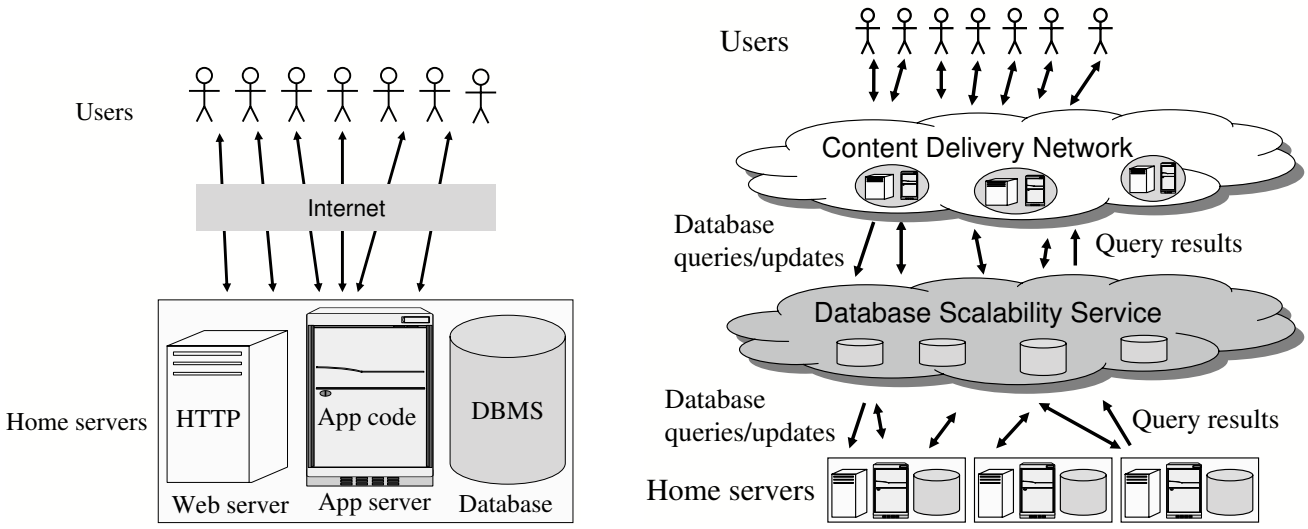
Chapter 2

Architecture of the Scalability Service

In this chapter we present the architecture of our scalability service. Section 1.2 laid out the challenges in designing a scalability service. In addition to these challenges, there are two goals that our scalability service architecture should meet.

First the scalability service architecture should aim to be general and powerful as possible, so that it can generate any form of dynamic content *as the application administrator intends it to be generated*. This approach is in contrast with that of trying to detect the structure of dynamic content [20, 92], and using more traditional static caching techniques to generate the pages [27, 36, 106]. So the scalability service architecture must be able to execute code, cache code, documents, and images, and process database requests. Second the scalability service architecture should place no additional burden on the Web application developers. Any application developed for the traditional centralized scenario should execute efficiently on the scalability service architecture, without requiring any additional effort from the application developer. For an alternative architecture to be easily adopted, it must not require developers to master a dramatically different and difficult methodology.

To eliminate all scalability bottlenecks, we explore an approach that replicates all three tiers (web server, application server, back-end database) of the traditional centralized architecture. Figure 2.1(a) illustrates the traditional architecture (here all three tiers are depicted as executing on a single home server; in general they may be spread across multiple physical servers). Figure 2.1(b) represents our



(a) Traditional centralized architecture.

(b) Distributed architecture.

Figure 2.1: Traditional versus distributed architecture.

distributed architecture, in which (1) a Web application's code is executed at trusted hosts (application "servers"), shown by the Content Delivery Network (CDN) in the figure, which provides the functionality of a web and application server, (2) the code in turn fires off database queries and updates that are handled by the Database Scalability Service (DBSS), and (3) queries that cannot be answered by the DBSS and updates are sent to back-end databases within the application's "home" organization. In our architecture, the home server for each application provider retains the capability to independently answer requests. We have built a prototype DBSS with this general architecture, and used it to scale three data-intensive Web benchmark applications.

Since our scalability service architecture replicates all three tiers of the traditional centralized architecture, it automatically generates the dynamic content as the application administrator intends it to be generated. Furthermore, no work is required to port the application to the new architecture – simply a different JDBC driver has to be loaded. However, as we discuss in Chapter 5, applications executing in a scalability service architecture may feel *less* responsive (to user's actions). Chapter 5 proposes two transformations that automatically increase the responsiveness of an application executing on the DBSS architecture.

Outline. In Section 2.1– 2.4, we first present the design of the four logical nodes in our system: home servers, CDN, DBSS, and the clients. Section 2.5 describes the flow of invalidations in our system. Section 2.6 describes the three benchmark applications we use for evaluating our prototype scalability service. Section 2.7 describes our methodology for carrying out the experiments in this thesis. In Section 2.8, we present detailed results to confirm that the database system is indeed the bottleneck in a traditional centralized architecture. We also confirm that our scalability service architecture is able to effectively offload work from the home server(s) of the benchmark applications. Finally, we summarize in Section 2.9.

2.1 Home Server

Each home server embodies the traditional three-tiered architecture, which enables it to generate Web content dynamically and serve directly as much dynamic content as it chooses. The top tier is a standard *web server*, which manages HTTP interactions with clients. The web server is augmented with a second tier, the *application server*, which can execute programs for generating responses to clients' request. Finally, the third tier consists of a *database server* that the application uses to manage all of its data.

In our prototype each home server is implemented as follows. We use Tomcat [62] 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 [79] as our back-end database management system and mm.mysql [75], a type IV JDBC driver, as our database driver.

To use the scalability service, the home server just has to run a light-weight module written in Java. On a cache miss or on an update, the DBSS contacts this module and provides it with the (encrypted) query or update. The module decrypts the query or update as necessary, generates the query result or the update acknowledgment, and does other required processing on the query result or the update acknowledgment before sending it back to the DBSS nodes. The other required processing includes encrypting the query result and computing "database-derived" clues for queries and updates ("Database-derived" clues provide additional information that the DBSS node can use for more precise invalidation;

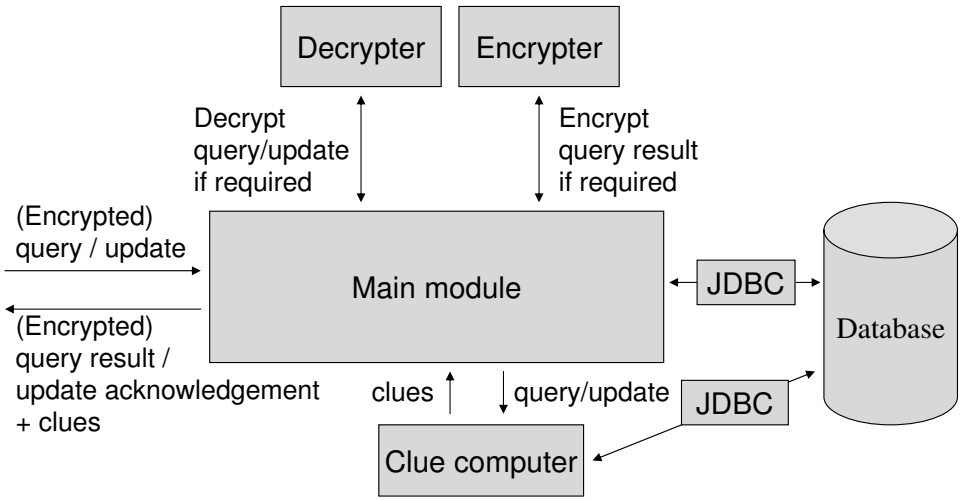


Figure 2.2: Architecture of the part of the home server used by the DBSS.

see Chapter 4 for details). The module reads a configuration file of the query and update templates in the application, and uses this knowledge to efficiently compute the clues. Figure 2.2 shows all the software modules at the home server and their interactions.

2.2 DBSS Node

The design of the DBSS nodes is our central contribution. The DBSS nodes cache data on behalf of the applications. We present the cache structure of the DBSS in Section 2.2.1. There are two main tasks that a DBSS node has to perform: handling database queries and handling database updates. We discuss these in Section 2.2.2 and Section 2.2.3, respectively. In Section 2.2.4 we present the consistency guarantees that our system provides. Finally, in Section 2.2.5 we provide other implementation details.

2.2.1 Cache Structure

In our design, the DBSS nodes cache the results of database queries (i.e., materialized views) rather than the tables of the database itself, or arbitrary subtables. We made this choice primarily due to privacy/security concerns. If the DBSS were to cache database tables, it would need to provide efficient

query processing capabilities over encrypted data. Recent work [63] has shown that only weak encryption can be used if queries are to be executed efficiently on encrypted data. Therefore, if the DBSS node caches database tables, privacy of all data might be compromised.

Caching query results also makes the DBSS independent of the back-end database implementation. This flexibility is required since different applications may choose different back-end databases. Lastly, caching query results has the advantage that complex queries need not be re-executed, and the DBSS does not have to implement full database functionality (e.g., it does not need a query optimizer).

The cache at the DBSS is partitioned to ensure better read/write concurrency. The partitions are based on query templates. Furthermore, the cache is maintained in memory to lower the read/write latency. Entries that do not fit into the memory are written onto the disk using a Greedy-dual-size-frequency (GDSF) policy [13]. In GDSF, each cache entry has an associated priority, which takes into account the size of the entry (i.e., the query result) and the cost of computing the entry. On each reuse, an entry's priority is increased. Whenever an entry needs to be evicted, the one with the least priority is chosen. In our preliminary evaluation, we found GDSF to perform better than LRU or other cache replacement algorithms that do not consider size and cost explicitly.

2.2.2 Handling Database Queries

On receiving an (encrypted) database query, a DBSS node first tries to answer the (encrypted) query from its store of cached query results. Queries that miss in its store are forwarded to the back-end database. Since web workloads have high reuse across database queries (for our benchmark applications, the hit rate varied from around 60% to 80% in typical runs), this architecture enables the DBSS node to effectively offload work from the database server(s) of the home organization. When the database server sends the query result, the DBSS node stores a copy of the (encrypted) query result along with any clues that the home server sent, and forwards the result to the CDN node.

If the query template corresponding to the (encrypted) query has been marked uncacheable, the DBSS simply forwards the query to the home server, and forwards the (encrypted) query result back to the CDN.

2.2.3 Handling Database Updates

On receiving an update, a DBSS node first sends it to the back-end database, and waits for an acknowledgment that the back-end database has applied the update. It then evicts from its cache (i.e., invalidates) the query results, which it believes have changed. Next, it must forward the update and the corresponding clue to those DBSS nodes whose caches are likely to be affected by the update. Charles Garrod, a group member, is exploring efficient techniques for carrying out this task [46].

For determining which query results to evict from its cache, the DBSS node can only use the query and update clues, since all other data that it sees is encrypted. This determination, or the precision with which the DBSS can carry out the invalidation, depends on what query and update clues the application has provided – more the information in the clues, more precisely the DBSS can invalidate. However, as the application provides more information in the clues, the DBSS nodes learn more information, and less is the privacy. Hence there is a privacy-scalability tradeoff in the DBSS setting. We study this tradeoff in Chapter 3 and propose solutions for managing this tradeoff in Chapters 3 and 4.

2.2.4 Consistency

Rather than explore the wide space of potential solutions to the consistency problem, our DBSS prototype currently supports a simple consistency model. By default, the DBSS provides non-transactional best-effort consistency for query results. The query and update templates are pre-analyzed to speed up the invalidation process. This guarantee suffices for most of an application's data. Examples of such data include the ten best sellers in the bookstore and the latest posting in the bulletin board. For data requiring strong consistency such as the number of copies of a book in stock in the bookstore application, or the inventory data for emergency supplies in our civic emergency management scenario in Section 1.1.2, applications can mark the corresponding query templates as uncacheable. The DBSS then does not cache query results for any such templates. Of course, marking objects as uncacheable may increase the load on the home server infrastructure, reducing the scalability of our scalability service architecture. For our benchmark applications, none of the query templates (out of a total of 94 query templates) were marked

as uncacheable.

2.2.5 Other Implementation Details

The entire DBSS code is implemented in Java. It reads a configuration file consisting of the query and update templates. It is multi-threaded, and uses a thread-pool and persistent connections for improved performance.

2.3 CDN Node

The CDN nodes provide the functionality of the web server and the application server. The trusted application "servers" are used to encrypt queries/updates and decrypt query results, as well as run application code. These hosts could either (1) be maintained by the application vendor—for many data-intensive Web applications, executing application code is not the real bottleneck and hence a modest number of hosts suffice, (2) be maintained by the CDN—if the vendor trusts the CDN, or (3) be users' machines—there are on-going efforts to guarantee secure execution of code on a remote machine [31, 105]. This scenario is similar to the standard security scenario of two trusted parties communicating over an untrusted channel. We consider the ciphertext-only attack [96] and the chosen-plaintext attack [96] in this scenario—details are in Section 4.3.3.

We used Tomcat [62] to provide the functionality of a web server and an application server, i.e., the ability to interact with a user running a web browser and the ability to run Web applications. Additionally, the application server loads our custom JDBC driver that connects to a DBSS node instead of the back-end database. Loading a different driver is the only change that an application must make to use the DBSS infrastructure. We modified our custom JDBC driver further so that it supports prefetching of database queries. On a prefetch request, the JDBC driver just notes the request and returns immediately. Another daemon thread, which periodically checks for any outstanding prefetch requests, issues the prefetch request and ensures that the query result is brought in the DBSS cache, if it is not already present there.

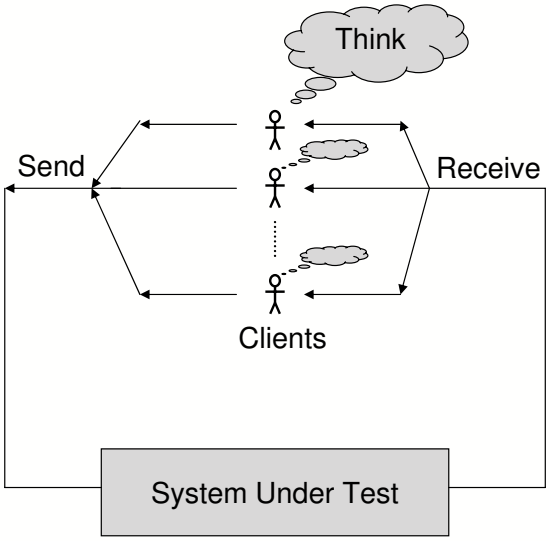


Figure 2.3: A closed system, in which there are a fixed number of users. A user sends a request only when it has waited for at least "think time" after receiving the response to its previous request.

2.4 Clients

To evaluate the traditional centralized architecture and our scalability service architecture, we use each application's workload generator. The workload generators use programs called *Emulated Browsers (EBs)* to emulate human users. These workload generators interact with the system under test, which is either the centralized architecture or the scalability service architecture, in a closed system model [97], i.e., at any time in an experiment, there are a fixed number of EBs and an EB issues a new request only after the application has responded to its previous request. Furthermore, EBs simulate human usage patterns by issuing an HTTP request, waiting for the response, and pausing for a *think time* of X seconds after receiving the response and before requesting another Web page—X is drawn from a negative exponential distribution with a mean of seven seconds. Figure 2.3 shows this closed system model.

An EB continually strives to model the behavior of a human trying to accomplish specific tasks such as ordering books and browsing the new arrivals. To model such behavior, an EB issues HTTP interactions as per a Markov chain, where the states are individual programs of the Web application and edges have

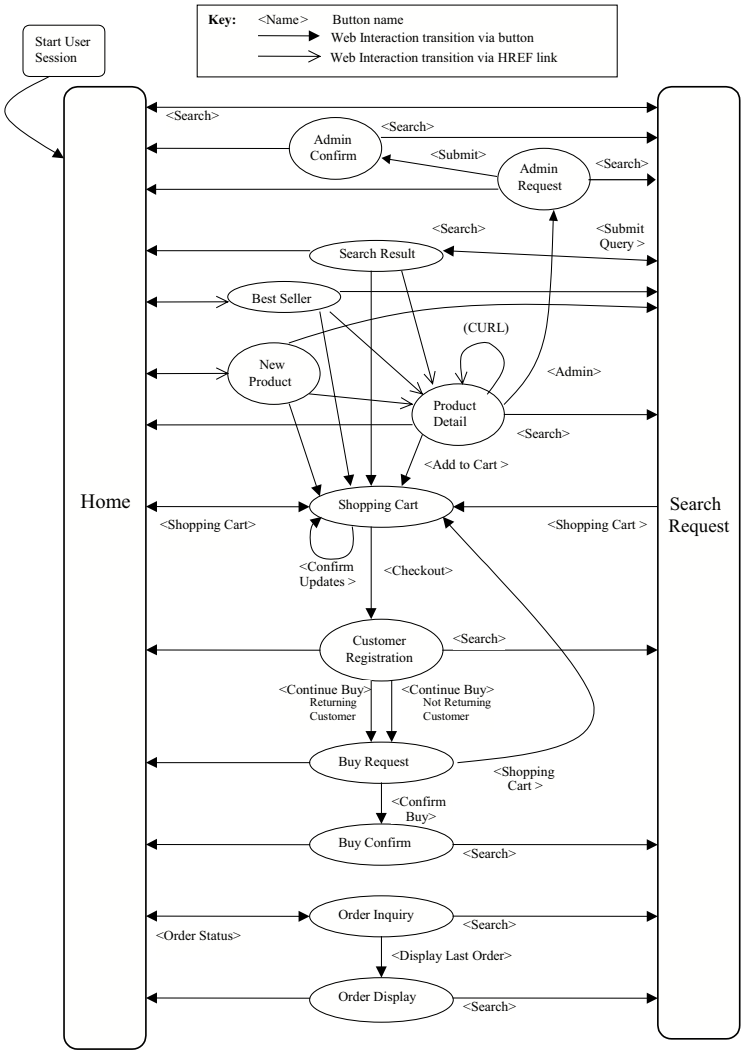


Figure 2.4: The transition graph for the BOOKSTORE application (reproduced from the TPCW [104] specification).

weights that denote the transition probability of moving from one state to another. Figure 2.4 shows the transition matrix for the BOOKSTORE application. The edges do not show the probabilities of the transitions; they just show the "action" that needs to be taken for that transition to happen.

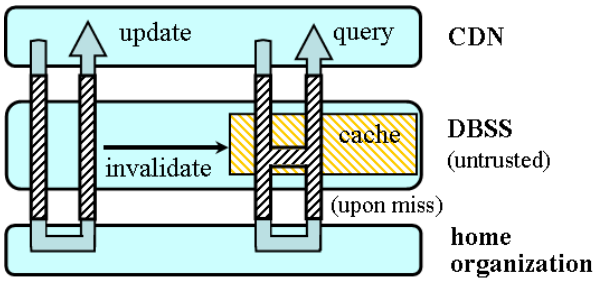


Figure 2.5: Query, update, and invalidation pathways.

2.5 Invalidation Flow

The flow of queries, updates, and invalidations in the system is shown in greater detail in Figure 2.5. 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.

2.6 Benchmark Applications

We sought Web benchmarks that make extensive use of a the database and are representative of real-world applications. We found three publicly available benchmark applications that met these criteria: RUBiS [82], an auction system modeled after `ebay.com`, RUBBoS [83], a simple bulletin-board-like system inspired by `slashdot.org`, and TPC-W [104], a transactional e-Commerce application that

Application	*Total number of programs*		*Number of templates*	
	Total	*Static*	*Query*	*Update*
AUCTION	25	5	28	11
BBOARD	21	3	38	13
BOOKSTORE	14	0	28	16

Figure 2.6: Total number of programs, and the number of query and update templates for our three benchmark applications.

captures the behavior of clients accessing an online bookstore[1]. We used Java implementation of these applications. We will henceforth refer to these applications as AUCTION, BBOARD, and BOOKSTORE, respectively.

Figure 2.6 lists the total number of programs, and the number of query and update templates in our three benchmark applications. The HTTP interactions for the AUCTION and BBOARD applications are significantly higher than the BOOKSTORE application, and they also have a few static HTML pages. BBOARD has the highest number of query templates, while BOOKSTORE has the highest number of update templates. Figure 2.7 provides the database configuration parameters we used in our experiments.

2.7 Methodology

We carefully optimized the performance of the centralized architecture by enabling the query caching feature of MySQL4, the back-end database, and by adding the indices necessary to make queries execute as quickly as possible. We also eliminated most static content from our workload by ensuring that the

[1]To 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. [22]. 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.

Application	*DB size*	*Parameters*
AUCTION	1 GB	33,667 items 100,000 registered users
BBOARD	1.5 GB	213,292 comments 500,000 registered users
BOOKSTORE	200 MB	10,000 items 86,400 registered users

Figure 2.7: Application configuration parameters.

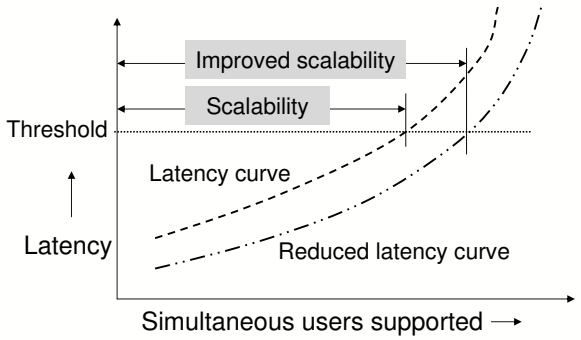


Figure 2.8: The figure shows (a) how scalability is computed as the number of simultaneous users supported within a latency threshold, (b) how a reduction in latency improves scalability.

emulated browsers did not request any images. This modification makes our results conservative since the static content of a normal workload can easily be cached by our system.

For the BOOKSTORE benchmark, we used the standard shopping mix. Each of our experiments started from a cold cache at the DBSS and ran for ten minutes.

2.7.1 Evaluation Metrics

The key evaluation metric is the maximum throughput (requests serviced per second) achieved by the scalability service architecture vs. the centralized architecture. A secondary goal is to reduce the delays

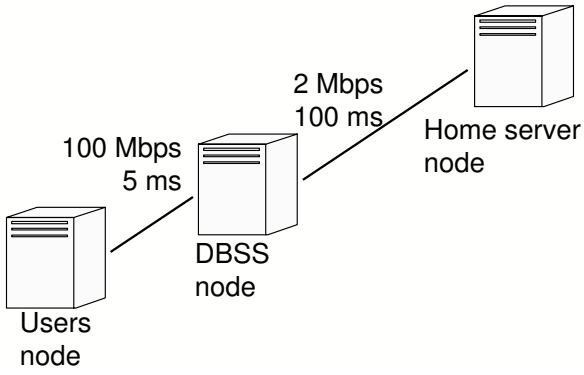


Figure 2.9: The SIMPLE scenario used in the experiments.

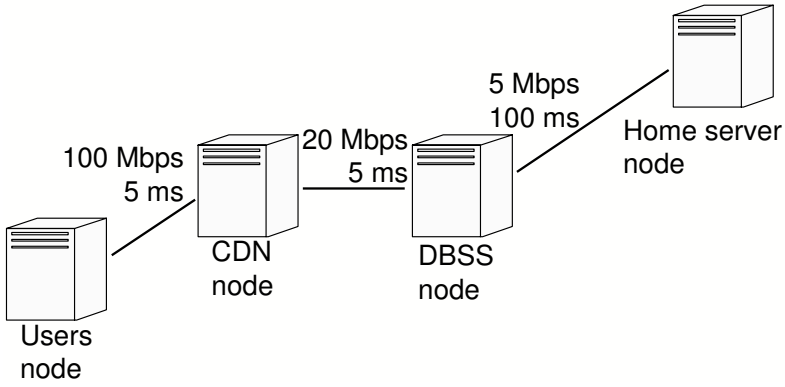


Figure 2.10: The SIMPLE_TC scenario used in the experiments.

experienced by end users, or at least to not increase delays dramatically. If a user suffers a long delay in receiving a response, the user might go away and the response becomes useless [59, 60]. It is therefore desirable to combine the two metrics. We combine the two metrics in a user perceived scalability metric, which is measured as the maximum number of users that could be supported while keeping the response time below a threshold. We refer the user perceived scalability metric as *scalability* in the rest of this thesis. Figure 2.8 shows how scalability is computed. It also shows how a reduction in latency improves the scalability metric. For our experiments, we use a latency threshold of two seconds and this latency threshold had to be met by 90% of the HTTP requests.

2.7.2 Scenarios

We use two scenarios, both running on Emulab [107] for evaluating our prototype: SIMPLE and SIMPLE_TC, which differed in whether the CDN node was co-located with the DBSS node or not and whether the bandwidth between the home server and the DBSS node was a bottleneck (at high scalability) or not. The SIMPLE scenario (Figure 2.9) had just two nodes–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. For simplicity, the DBSS node implemented the functionality of the CDN node. The SIMPLE_TC scenario (Figure 2.10) had three nodes–the SIMPLE scenario plus a separate CDN node. The CDN node and the DBSS were connected by a low latency, high bandwidth link (5 ms latency, 20 Mbps). In the SIMPLE experiment, the home server and DBSS node were connected by a high latency, low bandwidth duplex link (100 ms latency, 2 Mbps). However, we discovered that this bandwidth was proving to be a bottleneck in a traditional centralized scenario. Hence in the `SIMPLE_TC` setting, we increased the bandwidth to 5 Mbps from 2 Mbps.[2] In both scenarios, we used just one additional node to emulate all clients—the client was connected to the CDN 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.

2.8 Preliminary Evaluation

Figure 2.11 lists sample query and update numbers for a ten minute run. The updates to queries ratio is the lowest (1:50.9) for the BBOARD application and the highest (1:6.3) for the BOOKSTORE application. Figure 2.12 provides the cache hit rates measured for each of the three benchmark applications under a high load. The BBOARD benchmark achieves high hit rates, implying that the DBSS should be able to offload much of the database work from home servers of similar applications. The hit rates for the AUC-

[2] Figure 4.5 and Figure 5.9 confirm that this change increased the scalability for each of the three benchmark applications by over 20% in the centralized architecture.

Application	*Users*	*Queries*	*Updates*	*Update:Query ratio*
AUCTION	650	187.7k	13.3k updates (53.9% insertions, 46.1% modifications)	1:14.1
BBOARD	350	376.8k	7.4k updates (61.4% insertions, 2.4% deletions, 36.2% modifications)	1:50.9
BOOKSTORE	900	91.8k	14.5k updates (42.0% insertions, 3.9% deletions, 54.1% modifications)	1:6.3

Figure 2.11: Sample update rates for a ten-minute run.

Benchmark	Users	Cache hit rate
AUCTION	650	57.4%
BBOARD	350	75.5%
BOOKSTORE	900	66.4%

Figure 2.12: Cache hit rates for the three benchmark applications.

TION and BOOKSTORE benchmarks are less impressive. We need to explore TTL-based approaches [34], which are the norm in caching static content, to improve the cache hit rates for these two applications.

Figure 2.13 plots the average latency per dynamic HTTP request, at three different number of EBs (clients), for our three benchmark applications executing in a traditional centralized setting. The latency has three components: the client latency (the time spent by an average request in going from the client to the home server and by the response in returning back), the processing time spent in the web and application server, and the time spent in servicing database requests. As the number of EBs increases, time spent in each category increases, reflecting growing load. To understand these variations in latency, we also plot the average bandwidth usage and the maximum CPU usage of the the home server for these experiments in Figure 2.14 and Figure 2.15 respectively. The CPU usage is mainly due to two categories of processes: *mysqld* (processes for the database server) and *java* (processes for the web server and the application server). Since the home server CPU usage reaches 100% for the AUCTION and BOOKSTORE

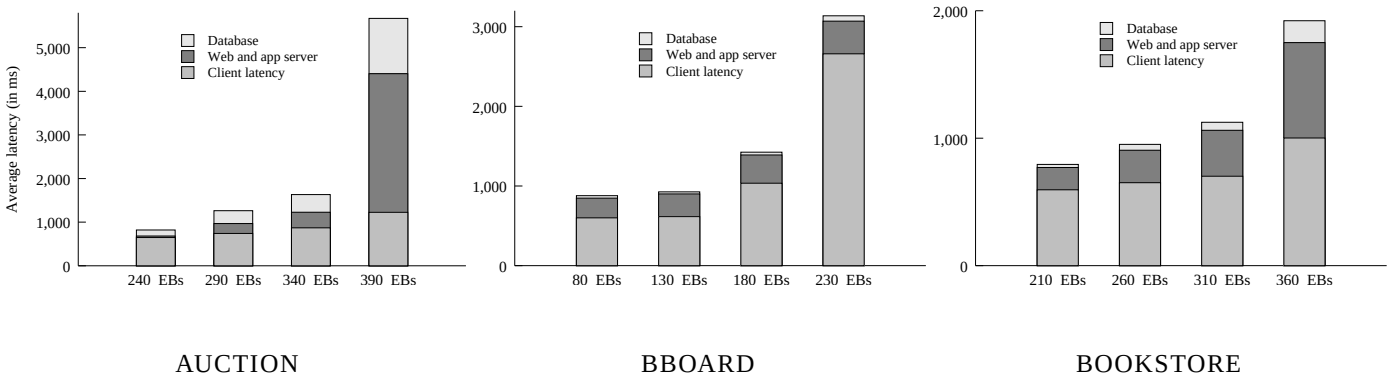


Figure 2.13: Average latency per dynamic HTTP request, at three different number of EBs, for the three benchmark applications executing in a traditional centralized setting.

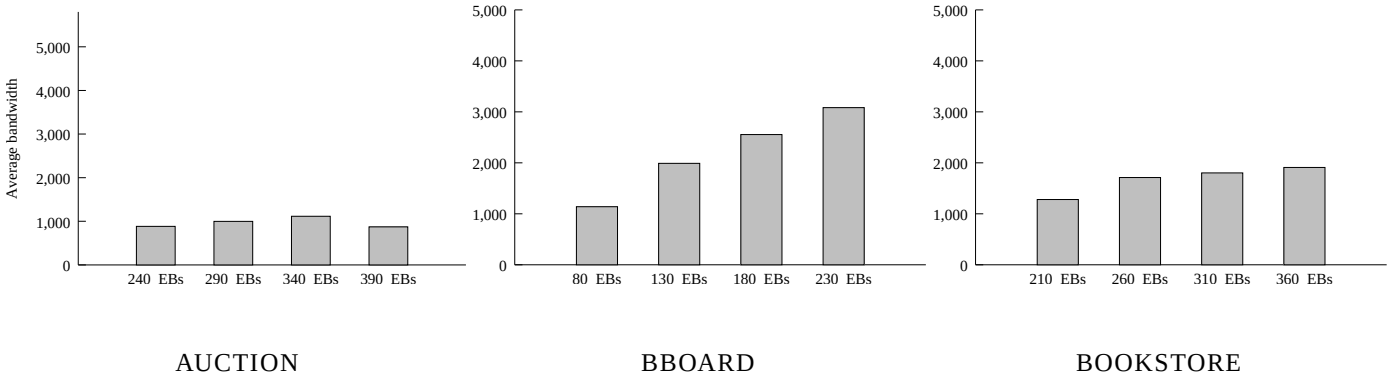


Figure 2.14: Average bandwidth usage of the home server, at three different number of EBs, for the three benchmark applications executing in a traditional centralized setting.

applications, it is evident that the home server CPU is the bottleneck for these applications. Furthermore, for both applications, the database server uses more CPU than the web and application server, indicating that the applications are database intensive. In contrast, for the BBOARD application, the application and web server use more CPU than the database server. This behavior is expected because the BBOARD application is presentation heavy.

We expect the average latency to increase sharply whenever a bottleneck is hit. For the AUCTION and BOOKSTORE applications, this increase happens at 390 EBs and 360 EBs respectively, when the CPU utilization reaches 100%. Note that most of the latency increase is due to increases in the time spent

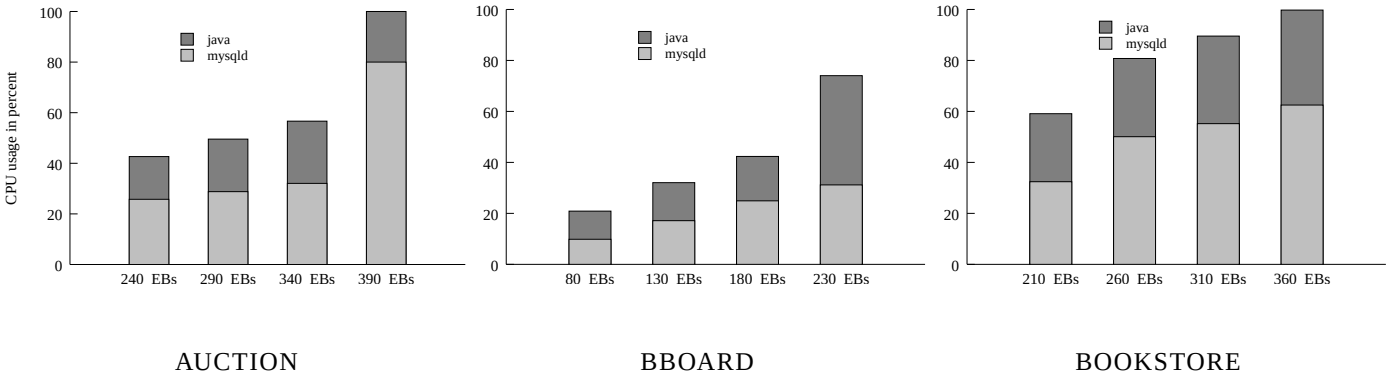
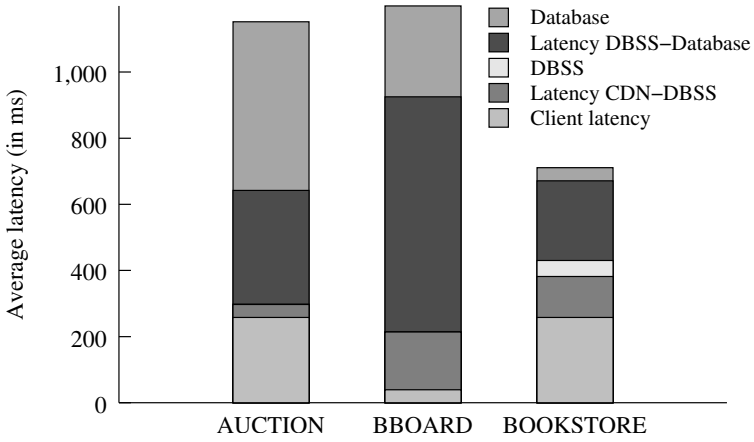


Figure 2.15: CPU usage at the home server, at three different number of EBs, for the three benchmark applications executing in a traditional centralized setting.

at the web, application, and database server, consistent with the fact that the home server CPU is the bottleneck. For the BBOARD application, the latency increases sharply at 230 EBs, indicating that some bottleneck has been hit. We know that the home server CPU is not the bottleneck in this case – since the CPU utilization remains below 80%. Instead, we believe it is the capacity of the link from the home server which proves to be a bottleneck. The maximum capacity of the link is 5Mbps, signifying that its average utilization over the duration of the entire experiment for 230 EBs, in case of the BBOARD application, is higher than 60%. That the link is the bottleneck is also consistent with how sharply the client latency increases in Figure 2.13.

Figure 2.13 plots the average latency per dynamic HTTP request, at three different number of EBs (clients), for our three benchmark applications executing in our scalability service setting. Note that for each application, at similar latencies, our DBSS architecture supports much higher number of simultaneous users than the traditional centralized scenario. Furthermore, both the CPU usage and the bandwidth usage of the home server are well within their maximum limits, thus indicating that the home server can handle even more load. (A higher load could not be supported because the single DBSS node became a bottleneck.) Figure 2.17, which plot the scalability metric defined in Section 2.7.1, confirms the scalability advantages of our DBSS architecture.



Application	*EBs*	*CPU usage (in %)* *mysqld*	*java*	*Bw usage* *in kbps*
AUCTION	650	40.2	7.6	329
BBOARD	350	26.5	7.4	535
BOOKSTORE	900	61.1	5.4	261

Figure 2.16: Average latency per dynamic HTTP request for the three benchmark applications executing in our scalability service setting. The adjoining table provides the number of EBs and the resource usage at the home server during the experiment.

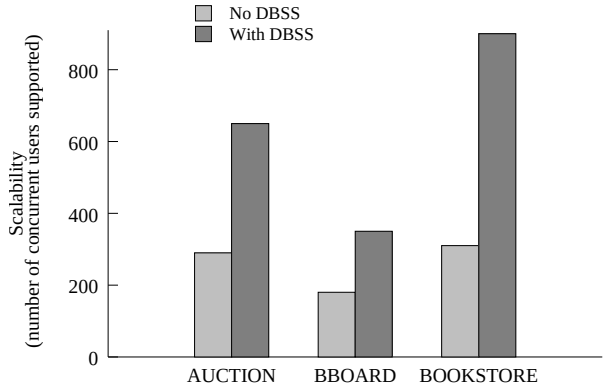


Figure 2.17: Scalability in the presence and absence of the DBSS.

The latency in Figure 2.16 consists of five components: the client latency including the execution time at the CDN, the latency from the CDN to the DBSS, the time spent at the DBSS, the latency from the DBSS to the database, and the time spent at the database. The latency from the DBSS node to the database is highest for the BBOARD application because the BBOARD application has the most number of database queries per HTTP request. Compared to Figure 2.13, the database load on the home server is substantially reduced (reflected in the lower average latency to service back-end requests in spite of the number of EBs in the scalability service setting being much higher than the maximum number of EBs in the centralized setting). This observation provides some evidence that in our scalability service

architecture, the DBSS is able to shield the home server database from increasing load.

2.9 Summary

In this chapter we presented the overall architecture of the scalability service, and the design of the major components. We have built a prototype scalability service with this architecture, and used it to scale three benchmark applications. We finished this chapter by describing the three benchmark applications, our methodology for carrying out the experiments in this thesis, a detailed analysis of the bottlenecks in a traditional centralized setting, and an analysis of our DBSS prototype.

Chapter 3

Simultaneous Scalability and Security for Data Intensive Web Applications

As argued in Section 1.4.3, there is an important tradeoff between security and scalability in the DBSS setting. Recall from Section 1.4.2 that the goals with security are (1) to limit the DBSS administrator's ability to observe an application's sensitive data, and (2) to limit an application's ability to use the DBSS to observer another application's sensitive data. It is not immediately clear how application administrators can manage this security scalability tradeoff. In this chapter, we present our work which greatly simplifies an application administrator's task of managing this tradeoff and achieving simultaneous scalability and security when using a DBSS.

We begin in Section 3.1 by providing an example that illustrates the security-scalability tradeoff in the DBSS setting and an overview of our approach. To underpin our study of the security-scalability tradeoff, we present our formal characterization of cache invalidation strategies in Section 3.2, each of which represents a natural choice in the space of security-scalability options. Section 3.3 describes our methodology for management of the tradeoff, while Section 3.4 presents our main contribution: a static analysis method for determining which data can be encrypted without impacting scalability. In Section 3.5 we present our empirical findings, which point to the effectiveness of our technique. We present the contributions this chapter makes in Section 3.6 and summarize in Section 3.7.

Q_1^T	`SELECT toy_id FROM toys WHERE toy_name=?`
Q_2^T	`SELECT qty FROM toys WHERE toy_id=?`
Q_3^T	`SELECT cust_name FROM customers WHERE cust_id=?`
U_1^T	`DELETE FROM toys WHERE toy_id=?`

Table 3.1: 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: `toys` with attributes `toy_id, toy_name, qty`, and `customers` with attributes `cust_id, cust_name`. The question marks indicate parameters bound at execution time.

Accessible?			
Templates	Parameters	Query Results	Invalidation Condition
No	No	No	All of Q_1^T, Q_2^T, Q_3^T
Yes	No	No	All Q_1^T, all Q_2^T
Yes	Yes	No	All Q_1^T, Q_2^T if `toy_id=5`
Yes	Yes	Yes	Q_1^T if `toy_id=5`, Q_2^T if `toy_id=5`

Table 3.2: Invalidations differ depending on the amount of information the DBSS can access. The table is for update U_1^T with parameter `5`.

3.1 Security-Scalability Tradeoff

To illustrate the presence of the security-scalability tradeoff when DBSSs are employed, we introduce a simple example application called SIMPLE-TOYSTORE, specified in Table 3.1. We focus on the application's database access *templates*—queries or updates missing zero or more parameter values. Table 3.2 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 information the DBSS is able to access. For example, if no information is accessible, i.e., all data is encrypted, as in the first row, then all cached query results are invalidated on seeing an instance of update U_1^T. However, if the template information is accessible, as in the second row, then cached query results of all instances of only Q_1^T and Q_2^T are invalidated. As the information available to a DBSS increases (moving down the rows), the number of invalidations it needs to make decreases, thereby increasing scalability.

There is an important tradeoff between security and scalability in the DBSS scenario. Encryption of queries, updates and data for security purposes limits the information available to the DBSS for making invalidation decisions. With limited information, the DBSS is forced to employ conservative invalidation strategies to maintain consistency, resulting in excess invalidations and reduced scalability. This basic tradeoff between security and scalability is illustrated quantitatively in Figure 3.1, which shows measurements of the TPC-W online bookstore benchmark executed on a prototype DBSS system we have built (details are provided in Section 3.5). 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. Achieving good scalability and adequate security simultaneously requires more thought.

3.1.1 Managing the Security-Scalability Tradeoff

There is often room to maneuver with respect to what data needs to be encrypted. Flexibility arises because in most Web applications, not all data is equally sensitive. It may range from highly-sensitive data such as credit card information, to moderately sensitive data such as inventory records, to completely insensitive data such as the weekly best-seller list, which is made public anyway.

In general, management of the security-scalability tradeoff requires careful assessment of data sensitivity, weighed against scalability goals. Unfortunately, it is nontrivial to assess the scalability implications of ensuring the security of a particular portion of the database. Furthermore, for data that is not

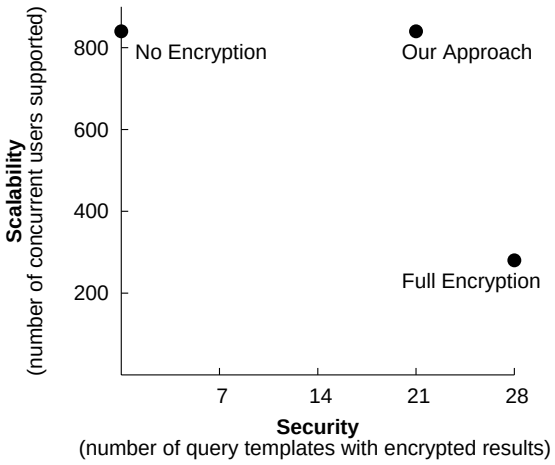


Figure 3.1: Security-scalability tradeoff (TPC-W BOOKSTORE benchmark).

entirely insensitive, it can be difficult to quantify sensitivity in a meaningful way. Therefore it is not immediately clear how to best approach the task of managing the security-scalability tradeoff.

In this chapter we present a convenient shortcut, which simplifies the task substantially while avoiding undesirable compromises with respect to security or scalability. The idea is to identify portions of the database that can be encrypted while incurring no additional penalty to scalability. The outcome of applying this idea is shown in the upper-right point in Figure 3.1, 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[1] without compromising scalability.

3.2 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 3.2.1 by providing the details of our basic query and update model, and introducing the terminology and notation we use in the rest of the chapter.

[1] See Section 3.5.3 for details on what data is kept private under our approach.

Then, in Section 3.2.2 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. Finally, in Section 3.2.3 we study the mixed invalidation strategies that arise when the information available for making invalidation decisions varies across queries and across updates.

3.2.1 Query and Update Model

The database components of a Web application consist of a fixed set of query templates, and a fixed set of update templates (Table 3.1 shows an example). Let $Q^T = \{Q_1^T, \ldots, Q_n^T\}$ and $\mathcal{U}^T = \{U_1^T, \ldots, U_m^T\}$ denote the set of query and update templates, respectively. A query Q is composed of a query template Q^T to which parameters Q^P are attached at execution time. Formally, $Q = Q^T(Q^P)$. Likewise, $U = U^T(U^P)$. Let $Q[D]$ denote the result of evaluating query Q over database D. Let $(D+U)$ denote the database state resulting from application of update U. A sequence of queries and updates issued at runtime constitutes a *workload*.

Based on our study of three benchmark applications (details in Section 3.5.1), 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 work [18, 90], 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 modifies non-key attributes of the row (of a relation) that satisfies an equality predicate over the primary key of the relation.

Assumptions for simplifying the presentation of our analysis

To simplify the presentation of our analysis (Section 2.3 and Section 4) of which information can be encrypted without impacting scalability, we make three assumptions about the update and query templates: First, each selection predicate either compares attribute values across two relations or compares a value with a constant. Second, no constants that might aid in invalidations are embedded in a query or update template. Third, no queries compute Cartesian Products, i.e., each query has a non-empty selection predicate. The above assumptions always hold for two of three benchmark applications we study, and are violated in less than 3% of the update/query template pairs for the third benchmark. Whenever the assumptions do not hold, no encryption is recommended for the given update/query template pair. This conservative strategy ensures that our analysis never recommends encrypting any data, for which encryption impacts scalability.

To simplify the presentation further, we make two additional assumptions about the execution of updates and queries: First, no query whose result is subject to invalidation by either an insertion or a deletion statement in the workload returns an empty result set. Second, each update has some effect on the database, i.e., for each update U, $D \neq [D+U]$. In our experiments with all three of the benchmark applications we study, these assumptions always hold, and cause no loss of scalability.

3.2.2 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 $\mathcal{S}$ 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 $\mathcal{S}$ is correct iff for any query Q, database D, and update U, $(Q[D] \neq Q[D+U]) \Rightarrow (\mathcal{S}(U,Q,\ldots) = I)$.

(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 3.2, may be defined as follows (The arguments to a strategy also list the information the strategy can access):

- **Blind Strategy**[2] **(BS)** $\mathcal{S}()$**:** No information is available to make the invalidation decision. Correctness requires that $(\exists U, Q, D : (Q[D] \neq Q[D+U])) \Rightarrow (\mathcal{S}() = I)$.

- **Template-Inspection Strategy (TIS)** $\mathcal{S}(U^T, Q^T)$**:** Only the update template U^T and query template Q^T may be used to make the invalidation decision. Correctness requires that $(\exists U^P, Q^P, D : (Q^T(Q^P)[D] \neq Q^T(Q^P)[D+U^T(U^P)])) \Rightarrow (\mathcal{S}(U^T, Q^T) = I)$.

- **Statement-Inspection Strategy (SIS)** $\mathcal{S}(U, Q)$**:** Only the update U and query statement Q may be used to make the invalidation decision. Correctness requires that $(\exists D : (Q[D] \neq Q[D+U])) \Rightarrow (\mathcal{S}(U, Q) = I)$.

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

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 [51] can be used to implement a view-inspection strategy. Similarly, the methods of [68] 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.

[2] In earlier work [84], the term *black-box strategy* was used to refer to the same concept.

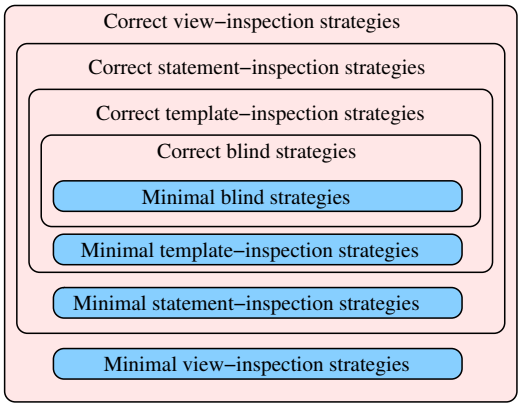


Figure 3.2: Relationships among classes of view invalidation strategies, in the general case.

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 3.2.

We now define minimality:

Minimality: A view invalidation strategy $\mathcal{S}$ belonging to class $\mathcal{C}$ is *minimal* if and only if it is correct and there exists no query statement Q, update statement U, and database D such that $\mathcal{S}$ invalidates the view $Q[D]$ in response to U, while another correct view invalidation strategy in class $\mathcal{C}$ 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 3.2 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 ex-

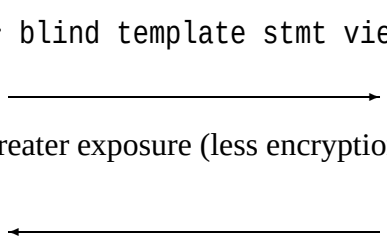


Figure 3.3: Security gradient.

treme, 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.[3]

3.2.3 Mixed Invalidation Strategies

Typically, not all of an application's data is equally sensitive. An administrator may wish to control encryption of information at a per-template granularity. To control what information to encrypt, the administrator chooses an *exposure level* $E(U^T) \in \{\texttt{blind}, \texttt{template}, \texttt{stmt}\}$ for each update template $U^T \in \mathcal{U}^T$, and an exposure level $E(Q^T) \in \{\texttt{blind}, \texttt{template}, \texttt{stmt}, \texttt{view}\}$ for each query template $Q^T \in \mathcal{Q}^T$. Each exposure level exposes some information of a query or an update; all information not exposed can then be encrypted. The `blind` exposure level exposes nothing; `template` exposes the template; `stmt` exposes the entire query or update statement (i.e., template and parameters); and `view` (only for query templates) exposes the query statement and the result of executing the query. Figure 3.3 shows the range of exposure level options.

Figure 3.4 shows the possible exposure level combinations for a given U^T/Q^T pair (the contents of the boxes may be ignored for now). When exposure level choices are made independently for every update

[3] Note that deterministic encryption is required for correct caching mechanics. 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.

		Query			
		`blind`	`template`	`stmt`	`view`
Update	`blind`	1	1	1	1
	`template`	1	A_{ij}	A_{ij}	A_{ij}
	`stmt`	1	A_{ij}	B_{ij}	C_{ij}

Figure 3.4: An *Invalidation Probability Matrix* IPM(U_i^T, Q_j^T).

and query template, the invalidation strategy to use may be determined at the granularity of update/query template pairs. In Figure 3.4, the shaded boxes correspond to the four classes of invalidation strategies introduced in Section 3.2.2. (We discuss the unshaded boxes shortly.)

Invalidation Probabilities

In our approach, exposure level choices determine the mix of invalidation strategies employed. Given a workload, the invalidation strategy used for a given U^T/Q^T pair in turn determines the *invalidation probability*—the likelihood that the invalidation strategy invalidates (the result of) an instance of the query template on seeing an instance of the update template (where probability distribution over template instances are derived from the workload). Invalidation probabilities also depend on the database, and may change over time. In general it is difficult to estimate these (dynamic) quantities accurately, but as we will see we can find useful invariant relationships among them using static analysis alone. For the purpose of our static analysis, we represent the invalidation probabilities for different choices of exposure levels as a matrix. An *Invalidation Probability Matrix* IPM (U_i^T, Q_j^T), illustrated in Figure 3.4, contains invalidation probability values for each combination of exposure levels for U_i^T and Q_j^T. (A_{ij}, B_{ij}, and C_{ij} are placeholders for invalidation probabilities that depend on workload and database characteristics.)

IPM's obey the following properties:

Property 1: The invalidation probability equals 1 if either exposure level is `blind`. Clearly, whenever no information is available about either update U or query Q, for correctness, the cached result of Q

must be invalidated whenever any update U occurs.

Property 2: The invalidation probability is the same for all cases in which one exposure level is `template` and the other is some exposure level other than `blind`. (We denote this invalidation probability by $A_{ij} \in [0,1]$.) Recall from Section 3.2.1 our assumptions that the selection predicates cannot compare two database values of the same relation and there are no constants in the update (query) templates. Under these assumptions, knowledge of the query (update) parameters but not the update (query) parameters does not aid in reducing invalidations because the query (update) parameters cannot be compared to anything. Similarly, knowledge of the query result but not the update parameters does not aid in reducing invalidations. (We omit formal proofs for brevity.)

Property 3: The invalidation probabilities constitute a gradient as we move from top-left to bottom-right in Figure 3.4, i.e., $1 \geq A_{ij} \geq B_{ij} \geq C_{ij} \geq 0$. Clearly, under minimal invalidation strategies, invalidations cannot increase if more information is available for making invalidation decisions.

From the above discussion, it follows that invalidation strategy classes corresponding to unshaded boxes in Figure 3.4 are of no interest since they are *dominated* by those corresponding to shaded boxes, i.e., the shaded boxes permit lower exposure while offering the same invalidation probability. In certain instances, additional domination relationships can be found. First, for certain update/query template pairs U_i^T / Q_j^T, it can be shown that $A_{ij} = 1$ (meaning minimal template inspection invalidation strategies are equivalent to minimal blind strategies for such update/query template pairs). Similarly, in some cases $B_{ij} = A_{ij}$ (meaning minimal statement inspection strategies are equivalent to minimal template inspection strategies for such update/query template pairs), and in some cases $C_{ij} = B_{ij}$ (meaning minimal view inspection strategies are equivalent to minimal statement inspection strategies for such update/query template pairs). We examine how to identify and exploit such cases in Section 3.4. Before we approach this topic, we first describe our overall approach to managing the security-scalability tradeoff while meeting scalability requirements.

3.3 Overview of Approach

In this section we outline our approach for managing the security-scalability tradeoff, given scalability requirements. As Figure 3.3 shows, one may control security by adjusting the exposure level of an application's update and query templates. We first provide our approach in Section 3.3.1, and then present a brief example in Section 3.3.2 that illustrates the approach.

3.3.1 Our Approach

A natural approach to solve the security-scalability management problem is to model it as a constrained optimization problem where each potential solution, i.e., an assignment of an exposure level to every template of the application, has an "overhead" and a "security" value; the objective is to maximize the "security" value while keeping the "overhead" below a given threshold. However, the approach is impractical because assigning meaningful security values to, and predicting overhead values of, each potential solution is virtually impossible.

We advocate a new scalability-conscious security design methodology, which uses the following practical three-step approach for managing the security-scalability tradeoff, given a scalability requirement:

1. Beginning with maximum exposure for all templates, i.e., exposure level `stmt` for each update template and exposure level `view` for each query template, reduce exposure levels (i.e., move to the left in Figure 3.3) based on cases in which data absolutely must be encrypted. Such requirements may be decided in an ad-hoc manner, or based on a data privacy law such as [24].

2. Using our static analysis techniques (described shortly), reduce exposure level of each template for which doing so does not impact scalability.

3. Prioritize remaining exposure level reduction possibilities based on security considerations and adjust with respect to the tradeoff with scalability.

Step 2 is the focus of our work. We divide Step 2 into two sub-steps:

Step 2(a): Characterize IPM domination relationships. Determine for each U_i^T/Q_j^T pair whether (a) $A_{ij} = 1$, (b) $B_{ij} = A_{ij}$, and (c) $C_{ij} = B_{ij}$. Identifying these relationships is a challenge; Section 3.4 is dedicated to this task.

Step 2(b): Eliminate high-exposure options whenever possible without hurting scalability. The inputs to this step include: (a) IPM tables with the information from IPM characterization (Step 2a) plugged in, and (b) the initial exposure levels of templates based on requirements that certain data must absolutely be encrypted (Step 1). The goal of Step 2b is to maximally reduce the exposure level for each template without impacting scalability. Since scalability is impacted whenever invalidation probabilities change, the key idea in achieving maximal reduction of exposure levels is to ensure that the invalidation probability of no update/query template pair (as given by the IPM table) changes due to a reduction in the exposure level of a template.

Algorithm MinExposure can be used to find the minimal exposure levels that offer the same scalability as the initial exposure levels of the templates. It repeatedly lowers the exposure level of a template if doing so does not increase any invalidation probability. Lines 4–12 use this idea for lowering the exposure level of query templates, and Lines 13–21 use this idea for lowering the exposure level of update templates. Furthermore, since the algorithm lowers the exposure level of a template if and only if doing so does not increase any invalidation probability, the final exposure levels are independent of the order in which the templates in Line 4 and Line 13 are selected for exposure level reduction.

We next provide an example that illustrates our approach.

3.3.2 Example

Consider the TOYSTORE application shown in Table 3.3, an extension of our earlier SIMPLE-TOYSTORE application of Table 3.1. As Step 1, the administrator may well decide that credit card numbers are not to be exposed, and accordingly reduce the exposure level of U_2^T to `template`. Using the notation introduced in Section 3.2.3, $E(U_2^T) =$ `template`.

The next step is Step 2a, in which the IPM domination relationships are characterized. The results for

Q_1^T	`SELECT toy_id FROM toys WHERE toy_name=?`
Q_2^T	`SELECT qty FROM toys WHERE toy_id=?`
Q_3^T	`SELECT cust_name FROM customers, credit_card WHERE cust_id=cid and zip_code=?`
U_1^T	`DELETE FROM toys WHERE toy_id=?`
U_2^T	`INSERT INTO credit_card (cid, number, zip_code) VALUES (?, ?, ?)`

Table 3.3: A more elaborate example TOYSTORE application having three query templates Q_1^T, Q_2^T, Q_3^T, two update templates U_1^T, U_2^T and three base relations: `toys` with attributes `toy_id`, `toy_name`, `qty`, `customers` with attributes `cust_id`, `cust_name`, and `credit_card` with attributes `cid`, `number`, `zip_code`. Attribute `credit_card.cid` is a foreign key into the `customers` relation. The question marks indicate parameters bound at execution time.

the TOYSTORE application are provided in Table 3.4. To understand intuitively how these relationships are determined, let us focus on the first row, i.e., entries corresponding to U_1^T. Since no instance of U_1^T can affect the result of any instance of Q_3^T, no instance of U_1^T will trigger invalidation of the result of any instance of Q_3^T, so $A_{13} = 0$. However, since an instance of U_1^T can affect the result of an instance of Q_2^T or Q_1^T, $A_{12} > 0$ and $A_{11} > 0$. As we show in Section 3.4, whenever $A_{ij} > 0$, $A_{ij} = 1$. Hence, $A_{11} = A_{12} = 1$. Further, using our analysis in Section 3.4, it can be inferred that $B_{11} = A_{11}$, i.e., knowledge of the parameters of U_1^T and Q_1^T does not aid in reducing invalidations. Also $C_{12} = B_{12}$, i.e., additional knowledge of the content of the result of an instance of Q_2^T, when the parameters of U_1^T and Q_2^T are already known, does not aid in reducing invalidations. Finally, since $A_{13} = 0$, $A_{13} = B_{13} = C_{13}$ holds trivially due to Property 3 (Section 3.2.3).

Step 2b, in which Algorithm MinExposure is invoked, follows the IPM characterization step. When invoked on the TOYSTORE application (Table 3.3) with inputs as $E(U_2^T) =$ `template` (Step 1) and Ta-

	Q_1^T (j=1)	Q_2^T (j=2)	Q_3^T (j=3)
U_1^T (i=1)	$A_{11} = 1$ $B_{11} = A_{11}$ $C_{11} < B_{11}$	$A_{12} = 1$ $B_{12} < A_{12}$ $C_{12} = B_{12}$	$A_{13} = 0$ $B_{13} = A_{13}$ $C_{13} = B_{13}$
U_2^T (i=2)	$A_{21} = 0$ $B_{21} = A_{21}$ $C_{21} = B_{21}$	$A_{22} = 0$ $B_{22} = A_{22}$ $C_{22} = B_{22}$	$A_{23} = 1$ $B_{23} < A_{23}$ $C_{23} = B_{23}$

Table 3.4: Summary of IPM characterization for the example TOYSTORE application.

ble 3.4 (Step 2a), the algorithm used for Step 2b reduces exposure level of query template Q_3^T from `view` to `template`, and of query template Q_2^T from `view` to `stmt`. By reducing the exposure level in this way, the inventory (quantity of toys in stock) and the customer demographic (customers in an area) are no longer exposed. An application provider may prefer not to expose this moderately sensitive information, all else being equal. Further, we confirm that the additional security this reduction in exposure enables does not impact scalability. As before, cached results of instances of Q_2^T are only invalidated by instances of U_1^T if the `toy_id` match, and cached results of all instances of Q_3^T are invalidated by any instance of U_2^T.

Having presented our overall approach, we next describe how to determine IPM domination relationships using static analysis (Step 2a).

3.4 IPM Characterization

Recall from Section 3.3.1 that IPM characterization entails: determining statically for each U_i^T/Q_j^T pair, whether (a) $A_{ij} = 1$, (b) $B_{ij} = A_{ij}$, and (c) $C_{ij} = B_{ij}$. We discuss in Sections 3.4.2 – 3.4.4, how to determine for a given U^T/Q^T pair whether each of these relationships holds. Then, in Section 3.4.5 we discuss how additional information, beyond those considered up to now, affect IPM values. But, first in Section 3.4.1, we introduce some terminology for classifying query and update templates in a way that

Symbol	*Meaning*
$S(U^T)$	Attributes used in any of the selection predicates (i.e., selection and join conditions) of U^T
$M(U^T)$	Attributes modified by U^T
$S(Q^T)$	Attributes used in selection predicates or order-by constructs of Q^T
$P(Q^T)$	Attributes retained in the result of Q^T

Table 3.5: Notation for aspects of templates.

is useful for our analysis.

3.4.1 Query and Update Classification

Define *selection attributes* of update template U^T (denoted $S(U^T)$) to be attributes used in any selection predicate (i.e., a selection or a join condition) of U^T. (If U^T is an insertion, $S(U^T) = \{\}$.) Further define *modified attributes* ($M(U^T)$) of U^T, *selection attributes* ($S(Q^T)$) of query template Q^T, and *preserved attributes* ($P(Q^T)$) of Q^T as in Table 3.5. If U^T is an insertion or a deletion, $M(U^T)$ is defined to be the set of all attributes in the table in which the insertion or deletion takes place. For the TOYSTORE application (Table 3.3), $S(Q_1^T) = \{\texttt{toys.toy\_name}\}$, $P(Q_1^T) = \{\texttt{toys.toy\_id}\}$, $S(U_1^T) = \{\texttt{toys.toy\_id}\}$, $M(U_1^T) = \{\texttt{toys.toy\_id}, \texttt{toys.toy\_name}, \texttt{toys.qty}\}$.

Recall from Section 3.2.1 that queries are restricted to be Select-Project-Join (SPJ) queries having conjunctive selection predicates, augmented with optional order-by, and top-k constructs. Further define two (possibly overlapping) classes of queries: ones with only equality joins or no joins (denoted $\mathcal{E}$ for equality), and ones with no top-k constructs ($\mathcal{N}$). As before, there are three classes of updates: insertions (denoted $\mathcal{I}$), deletions ($\mathcal{D}$), and modifications ($\mathcal{M}$). We say an update (query) template belongs to a particular update (query) class if any instance of the update (query) template belongs to the class.

For our static analysis, it is important to know whether any instance of an update template can ever affect the result of any instance of a query template. Following the terminology of [90], an update

$Q^T \in \mathcal{E}$	Q is a query with only equality joins
$Q^T \in \mathcal{N}$	Q is a SPJ query with no top-k constructs
$U^T \in I$	U is an insertion
$U^T \in \mathcal{D}$	U is a deletion
$U^T \in \mathcal{M}$	U is a modification
U^T is *ignorable* for Q^T ($\langle U^T, Q^T \rangle \in \mathcal{G}$)	$\langle U^T, Q^T \rangle \in \mathcal{G} \Leftrightarrow M(U^T) \cap (P(Q^T) \cup S(Q^T)) = \{\}$
Q^T is *result-unhelpful* for U^T ($\langle U^T, Q^T \rangle \in \mathcal{H}$)	$\langle U^T, Q^T \rangle \in \mathcal{H} \Leftrightarrow S(U^T) \cap P(Q^T) = \{\}$

Table 3.6: Query and update classes.

template U^T is *ignorable* with respect to a query template Q^T if and only if no attributes modified by the update template are either preserved by the query template, or used in the selection predicate of the query template. Let $\mathcal{G}$ denote the set of all such update/query template pairs, i.e., $\langle U^T, Q^T \rangle \in \mathcal{G} \Leftrightarrow M(U^T) \cap (P(Q^T) \cup S(Q^T)) = \{\}$. For example, in the TOYSTORE application (Table 3.3), update template U_1^T is ignorable with respect to query template Q_3^T.

It is also important to know whether a query result has any information that aids in reducing invalidations. A query template Q^T is *result-unhelpful* with respect to an update template U^T if and only if none of the selection attributes of the update template are preserved by the query template. Let $\mathcal{H}$ denote the set of all such update/query template pairs, i.e., $\langle U^T, Q^T \rangle \in \mathcal{H} \Leftrightarrow S(U^T) \cap P(Q^T) = \{\}$. For example, in the TOYSTORE application (Table 3.3), query template Q_3^T is result-unhelpful for update template U_2^T.

In Table 3.6, we summarize the different classes of templates and properties of update/query template pairs.

3.4.2 Blind vs. Template-Inspection (Does $A_{ij} = 1$?)

Begin by considering the case in which both update and query templates are exposed. If any instance of update template U_i^T could cause invalidation of cached results of all possible instances of query template Q_j^T, then $A_{ij} = 1$. Hence, there is no advantage to using a minimal template-inspection strategy instead of a minimal blind strategy, i.e., knowledge of the query or update templates does not aid in decreasing invalidations. For example, A_{11} equals 1 in the TOYSTORE application (Table 3.4).

Furthermore, if A_{ij} is greater than 0, then A_{ij} equals 1, i.e., $A_{ij} > 0 \Rightarrow A_{ij} = 1$. The implication holds because the invalidation behavior of a template-inspection strategy is the same for all instances of an update/query template pair. So if there exists some instance of U_i^T that causes invalidation of cached results of some instance of Q_j^T, then 'any' instance of U_i^T causes invalidation of cached results of 'all' instances of Q_j^T. Thus, A_{ij} either equals 0 or 1.

Lemma 1 provides the necessary and sufficient conditions for determining if A_{ij} equals 0.

Lemma 1. *With assumptions as in Section 3.2.1, invalidation probability A_{ij} equals 0 if and only if the update template U_i^T is ignorable with respect to the query template Q_j^T. Formally, $A_{ij} = 0 \Leftrightarrow \langle U_i^T, Q_j^T \rangle \in \mathcal{G}$. Otherwise, $A_{ij} = 1$.*

Proof. We only prove the "if" part of this Lemma, and omit the proof of the "only if" part for brevity.

An instance of an update template can only change the values of the attributes in $M(U_i^T)$. Further, the result of an instance of a query template Q_j^T changes only if values of any of the attributes in the union of $P(Q_j^T)$ and $S(Q_j^T)$ changes. If the two sets, $M(U_i^T)$ and $P(Q_j^T) \cup S(Q_j^T)$, don't intersect, then no instance of U_i^T can invalidate the cached query result of any instance of Q_j^T, i.e., $M(U_i^T) \cap (P(Q_j^T) \cup S(Q_j^T)) = \{\} \Rightarrow A_{ij} = 0$. Using the definition of when U_i^T is ignorable with respect to Q_j^T, we get $\langle U_i^T, Q_j^T \rangle \in \mathcal{H} \Rightarrow A_{ij} = 0$. □

3.4.3 Template-Inspection vs. Statement-Inspection (Does $B_{ij} = A_{ij}$?)

For a given update/query template pair, if whenever a minimal template-inspection strategy (MTIS) evaluates to invalidate (denoted I), a minimal statement-inspection strategy (MSIS) also evaluates to I, then $B_{ij} = A_{ij}$, i.e., knowledge of update and query parameters in addition to the update and query template does not aid in decreasing invalidations. Since A_{ij} can take only two possible values, 0 or 1, if $B_{ij} = A_{ij}$, then either $B_{ij} = A_{ij} = 0$ or $B_{ij} = A_{ij} = 1$.

Case 1 ($\mathbf{B_{ij} = A_{ij} = 0}$): Property 3 (Section 3.2.3) implies that the equality $B_{ij} = A_{ij} = 0$ holds if and only if $A_{ij} = 0$. Furthermore, from Lemma 1, we know the necessary and sufficient conditions for A_{ij} being 0. Combining the two statements, $B_{ij} = A_{ij} = 0$ holds if and only if the update template is ignorable with respect to the query template, i.e., $B_{ij} = A_{ij} = 0 \Leftrightarrow \langle U_i^T, Q_j^T \rangle \in \mathcal{G}$.

Case 2 ($\mathbf{B_{ij} = A_{ij} = 1}$): The equality $A_{ij} = 1$ is a necessary condition for $B_{ij} = A_{ij} = 1$. Using Lemma 1, the previous statement can be rewritten as: update template U_i^T must not be ignorable with respect to query template Q_j^T for the equality $B_{ij} = A_{ij} = 1$ to hold. This necessary condition for $B_{ij} = A_{ij} = 1$ is however not a sufficient condition since a MSIS also has knowledge of the parameters of the update and the query statement. This knowledge may allow the MSIS to infer that an instance of U_i^T does not affect the cached query result of some instance of Q_j^T. For example, $A_{12} = 1$ but $B_{12} < 1$ in the TOYSTORE application (Table 3.4).

However, if $S(U_i^T) \cap S(Q_j^T) = \{\}$, then knowing the parameters in addition to the update and query templates cannot aid in decreasing invalidations. Hence a sufficient condition for $B_{ij} = A_{ij} = 1$ is: If no attribute is common to the selection predicates of both the update and query template, and the update template is not ignorable with respect to the query template, then $B_{ij} = A_{ij} = 1$, i.e., $(S(U_i^T) \cap S(Q_j^T) = \{\}) \wedge (\langle U_i^T, Q_j^T \rangle \notin \mathcal{G}) \Rightarrow B_{ij} = A_{ij} = 1$.

3.4.4 Statement-Inspection vs. View-Inspection (Does $C_{ij} = B_{ij}$?)

For a given update/query template, if whenever a minimal statement-inspection strategy (MSIS) evaluates to invalidate (denoted I), a minimal view-inspection strategy (MVIS) also evaluates to I, then

$C_{ij} = B_{ij}$, i.e., knowledge of the query result in addition to the update and query statement does not aid in decreasing invalidations. From Property 3 (Section 3.2.3), $C_{ij} \leq B_{ij}$. In this subsection we provide several sufficient conditions for the equality $C_{ij} = B_{ij}$ by identifying important classes of update/query pairs for which the equality holds. For other classes, we provide an example instance of U_i^T and Q_j^T for which $C_{ij} < B_{ij}$. Next, we consider the three classes of updates in turn: insertions, deletions, and modifications.

Insertions. This paragraph applies if the update is an insertion. If queries are limited to SPJ queries having conjunctive selection predicates, with equality as the join operator, augmented by optional order-by constructs, then whenever a MSIS evaluates to I, a MVIS also evaluates to I, i.e., $(U_i^T \in I) \wedge (Q_j^T \in \mathcal{E} \cap \mathcal{N}) \Rightarrow C_{ij} = B_{ij}$. We prove this result as Lemma 4 in Appendix A.1. This result is our most significant contribution in finding sufficient conditions for $C_{ij} = B_{ij}$. For example, C_{23} equals B_{23} for the TOYSTORE application (Table 3.4), as predicted by this result. However, when the query template either has one or more of $\{<, \leq, >, \geq\}$ appearing in the join predicate ($Q_j^T \notin \mathcal{E}$), or has a top-k construct ($Q_j^T \notin \mathcal{N}$), C_{ij} may be less than B_{ij}, as illustrated when the update `INSERT INTO toys (toy_id, toy_name, qty) VALUES (15, 'toyB', 10)` is paired with either of the following queries:

a)
```
SELECT t1.toy_id, t1.qty, t2.toy_id, t2.qty
FROM toys as t1, toys as t2
WHERE t1.toy_name='toyA' AND t2.toy_name='toyB'
  AND t1.qty > t2.qty
```

Suppose the query result has just one tuple `(10, 3, 12, 2)`. A minimal statement-inspection strategy will invalidate the cached query result, since a `'toyA'` with `qty` > 10 might exist in the database. However, a minimal view-invalidation strategy, with the knowledge of the cached query result, which implies that there is no `'toyA'` with `qty > 3`, will not invalidate the query result.

b)
```
SELECT MAX(qty) FROM toys
```

Suppose the result of this top-k query is `15`. A minimal statement-inspection strategy will necessarily invalidate the cached query result, since the current `max(qty)` might be less than `10`. However, a

minimal view-invalidation strategy, with the knowledge of the query result, will not invalidate the cached query result.

Deletions. This paragraph applies if the update is a deletion. If the query template is result-unhelpful with respect to the update template, then whenever a MSIS evaluates to invalidate (I), a MVIS also evaluates to I, i.e., $\langle U_i^T, Q_j^T \rangle \in \mathcal{H} \Rightarrow C_{ij} = B_{ij}$. We prove this result formally as Lemma 5 in Appendix A.1. For example, the equalities $C_{12} = B_{12}$ and $C_{13} = B_{13}$ hold for the TOYSTORE application (Table 3.4), as predicted by this result. Moreover, the U_1^T/Q_1^T pair of the TOYSTORE application is an example where the precondition of this result is not met and $C_{11} < B_{11}$.

Modifications. This paragraph applies if the update is a modification. If either the update template is ignorable with respect to the query template or the query template is result-unhelpful with respect to the update template, then whenever a MSIS evaluates to invalidate (I), a MVIS also evaluates to I, i.e., $\langle U_i^T, Q_j^T \rangle \in \mathcal{G} \cup \mathcal{H} \Rightarrow C_{ij} = B_{ij}$. We prove this result formally as Lemma 6 in Appendix A.1. Moreover, if the precondition of this result is not met, C_{ij} may be less than B_{ij}, as with the following update/query pair:

```
UPDATE toys SET qty=10 WHERE toy_id=5
```

```
SELECT toy_id FROM toys WHERE qty > 100
```

Let the toy with `toy_id=5` be absent from the cached query result. A minimal statement-inspection strategy will necessarily invalidate the cached query result, because the cached result could contain the toy with `toy_id = 5`. A minimal view-inspection strategy will not invalidate it.

3.4.5 Database Integrity Constraints

So far the IPM values are based on the DBSS's (optional) knowledge of the update statement, the query statement, and the query result. The DBSS can further lower the values of the invalidation probabilities A_{ij}, B_{ij}, and C_{ij}, i.e., increase the precision of invalidation decisions, by using database integrity constraints. Database *integrity constraints* are conditions on the database that must be satisfied at all times,

i.e., all instances of the database must satisfy the constraints. We expect the DBSS to know the basic database integrity constraints[4], and thus use them for providing greater scalability to the applications. We list two such basic database integrity constraints below, and show, using the TOYSTORE application (Table 3.3), how knowledge of the constraints can affect values of the IPM:

1. **Primary key constraint**: Consider the query template Q_2^T. If `toy_id` is the primary key of the `toys` relation, then the `toys` table cannot have more than one tuple with the same value of `toy_id`. As a result, no insertion into the `toys` relation affects the cached query result of any instance of the query template Q_2^T.

2. **Foreign key constraint**: Consider the query template Q_3^T. We already assume that attribute `cid` of the `credit_card` relation is a foreign key into `customers` relation, i.e., the value of the `cid` attribute for any tuple of the `credit_card` relation should be the same as the value of the attribute `cust_id` for some tuple in the `customers` relation. Further, any insertion into the `customers` relation inserts a new `cust_id`, which cannot join with any tuple in the `credit_card` relation. As a result, no insertion into the `customers` relation affects the cached query result of any instance of Q_3^T.

For any update/query template pair, if either of the two integrity constraints applies, A_{ij} becomes zero. Furthermore, as Property 3 (Section 3.2.3) implies, if $A_{ij} = 0$, then the equality $A_{ij} = B_{ij} = C_{ij} = 0$ holds.

3.5 Evaluation

Using our prototype DBSS (described in Chapter 2), we evaluated how well our scalability-conscious security design methodology works in practice. Before presenting these results, we first describe in Section 3.5.1 how the templates of our benchmark applications, described in Section 2.6, differ from the assumptions outlined in Section 3.2.1. We also present in Section 3.5.1 the IPM characterization results

[4]For all three benchmark applications that we study (details in Section 3.5.1), database integrity constraints fall into the category of insensitive data, and so revealing it to the DBSS does not compromise security.

Application	*Number of U^T/Q^T pairs for which*				
	$A=B=$	$A=1$			
		$B<A$		$B=A$	
	$=C=0$	$C<B$	$C=B$	$C<B$	$C=B$
AUCTION	267	2	25	14	0
BBOARD	488	0	25	25	2
BOOKSTORE	405	0	22	18	3

Table 3.7: IPM characterization results for the three applications. The table entries denote the number of update/query template pairs for which particular IPM relationships hold.

of applying our static analysis to the benchmark applications. Next, in Section 3.5.2 we confirm that blanket encryption of all data passing through the DBSS greatly hurts scalability. Finally, in Section 3.5.3 we find that our scalability-conscious security design methodology enables significantly greater security without impacting scalability.

3.5.1 IPM Characterization Results

The update/query templates of the benchmark applications we used (Section 2.6) differ from the assumptions outlined in Section 3.2.1 in one significant way: between 7% and 11% of the query templates for each application have aggregation or group-by constructs. *Aggregation* is one of `min, max, count, sum, avg,` and *group-by* allows application of aggregation functions to tuples clustered by some attribute. Our current model does not handle aggregation and group-by queries. For our evaluation, we separately consider each update/query template pair, where the query has an aggregation or group-by construct, and manually determine the behavior of each of the four classes of minimal invalidation strategies of Section 3.2.2.

Table 3.7 summarizes the IPM characterization results for the three applications, assuming the DBSS

has knowledge of the two types of database integrity constraints mentioned in Section 3.4.5. Each row of Table 3.7 corresponds to an application. The table entries denote the number of update/query template pairs for which particular IPM relationships hold. The first column lists the number of update/query template (U^T/Q^T) pairs for which the equality $A = B = C = 0$ holds. For each application, the majority of U^T/Q^T pairs fall in this category. For the remaining U^T/Q^T pairs, invalidation probability A equals 1. These U^T/Q^T pairs are further divided into four categories, represented by the next four columns of Table 3.7, depending on whether $B < A$ or $B = A$, and whether $C < B$ or $C = B$. As Table 3.7 shows, equalities $B = A$ and/or $C = B$ hold for the majority of the template pairs. Accordingly, for these template pairs, reducing the exposure of templates does not increase invalidations. Thus, the analysis presented in Section 3.4 applies to the applications we studied.

3.5.2 Magnitude of Security-Scalability Tradeoff

We performed our experiments in the SIMPLE scenario and the methodology described in Section 2.7. Figure 3.6 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 2.7. On the x-axis, we consider an instance of each of the four classes of invalidation strategies introduced in Section 3.2.2. (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 2.7.

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 3.6 confirms the claim made in Section 3.1 that blanket encryption of all data (thereby requiring a blind invalidation strategy) significantly hinders scalability.

3.5.3 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 3.1 of Section 3.1.1, which plots scalability[5] 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.

As discussed in Section 3.3.1, the outcome of our static analysis (Step 2) depends on the initial determination of what highly sensitive data absolutely must be encrypted (Step 1). To make this determination, we defer to the well-known California data privacy law [24], which, when applied to our applications, mandates securing all credit card information.

Figure 3.5 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:

[5]Computational overhead of encryption and decryption is not taken into account. Optimizing the encryption and decryption process is beyond the scope of this work.

- 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 3.6.

3.6 Chapter Contributions

We developed a formal characterization of view invalidation strategies in terms of what data they access, and used the formal characterization to cleanly formulate the security-scalability management problem. We then presented a method for automatically identifying data that can be encrypted without reducing scalability at all. Our method is based on static analysis of the data access templates of a given Web application. It determines which query results, query statements, and update statements associated with the application can be encrypted without impacting scalability.

Our experiments over a prototype DBSS system showed that several Web applications can encrypt the majority of query results, as well as a substantial fraction of parameters to query and update statements, with no scalability penalty. Furthermore, much of the data that is secured at no cost, falls into the moderately sensitive category. This type of data would not tend to be classified as compulsory for encryption, yet application designers may well choose to encrypt it, if armed with the knowledge that doing so does not impact scalability.

Our static analysis method enables a new scalability-conscious security design methodology that greatly simplifies the task of managing the security-scalability tradeoff: First, an administrator identifies highly-sensitive data (perhaps by applying a security law) and sets it aside for compulsory encryption. Second, our static analysis method is invoked to determine which of the remaining data can be encrypted

without impacting scalability. As a result, the administrator only needs to weigh the security-scalability tradeoff over the substantially reduced set of data items for which encryption may have scalability implications.

3.7 Summary

In this chapter we explored ways to secure the data of Web applications that use the services of a shared DBSS to meet their database scalability needs. At the heart of the problem is the tradeoff between security and scalability that occurs in this framework. When updates occur, the DBSS needs to invalidate data from its cache. The amount of data invalidated varies depending on the information exposed to the DBSS. The less information exposed to the DBSS, the more invalidations required, and the lower the scalability.

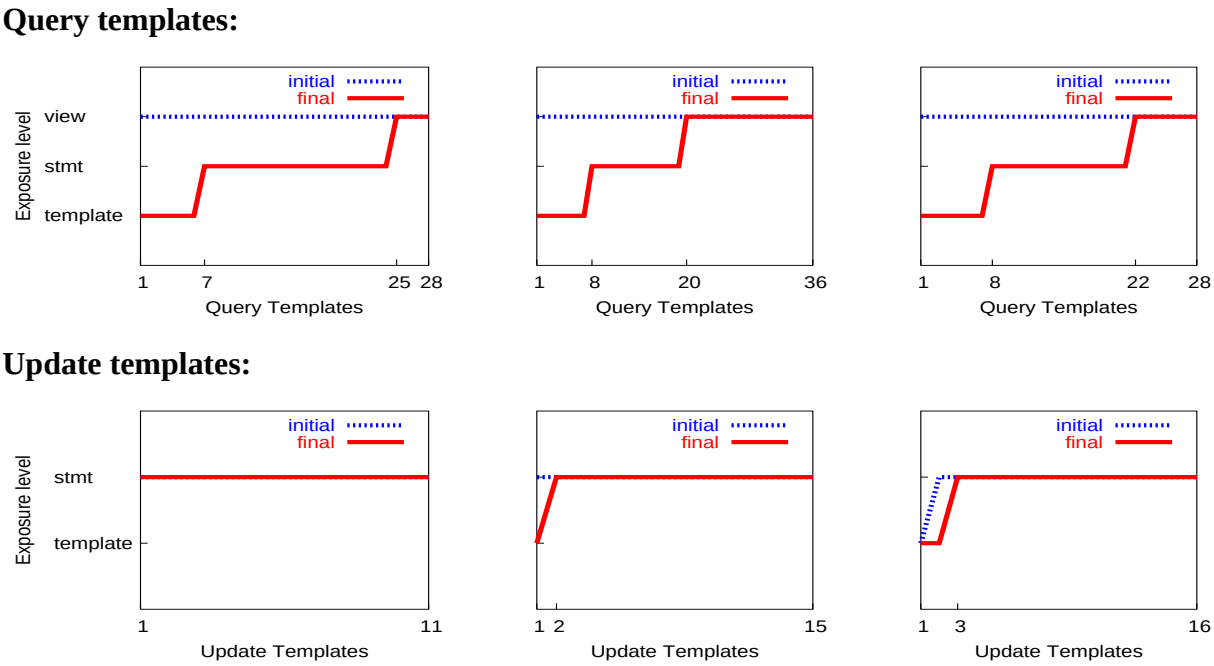
We presented a convenient shortcut to manage the security-scalability tradeoff. Our solution is to (statically) determine which data can be encrypted without any impact on scalability. We confirmed the effectiveness of our static analysis method, by applying it to three realistic benchmark applications that use a prototype DSSP system we built. In all three cases, our static analysis identified significant portions of data that could be secured without impacting scalability. The security-scalability tradeoff did not need to be considered for such data, significantly lightening the burden on the application administrator managing the tradeoff.

Algorithm MinExposure**: Reduce exposure levels of application templates.** (We assign numeric values corresponding to each exposure level as follows: `blind`=1, `template`=2, `stmt`=3, `view`=4. Let $p_{k,l}$ represent the value in the k^{th} row, l^{th} column of IPM (U^T, Q^T).)

Inputs: $\mathcal{U}^T$, Q^T, $IPM(U^T, Q^T)$ *for each* $(U^T, Q^T) \in \mathcal{U}^T \times Q^T$, *initial exposure levels* $E(U^T)$ *and* $E(Q^T)$ *for each template in* $\mathcal{U}^T \cup Q^T$

Output: updated exposure levels $E(U^T)$ *and* $E(Q^T)$ *for each template in* $\mathcal{U}^T \cup Q^T$

01 $done \leftarrow$ **false**
02 **while** $done =$ **false**
03 $done \leftarrow$ **true**
04 **for each** $Q^T \in Q^T$ where $E(Q^T) > 1$
05 $l \leftarrow E(Q^T)$
06 **for each** $U^T \in \mathcal{U}^T$
07 $k \leftarrow 1$
08 **while** $p_{k,l} = p_{k,(l-1)}$ and $k < E(U^T)$
09 $k \leftarrow k+1$
10 **if** $p_{k,l} = p_{k,(l-1)}$
11 $done \leftarrow$ **false**
12 $E(Q^T) \leftarrow l-1$
13 **for each** $U^T \in \mathcal{U}^T$ where $E(U^T) > 1$
14 $k \leftarrow E(U^T)$
15 **for each** $Q^T \in Q^T$
16 $l \leftarrow 1$
17 **while** $p_{k,l} = p_{(k-1),l}$ and $l < E(Q^T)$
18 $l \leftarrow l+1$
19 **if** $p_{k,l} = p_{(k-1),l}$
20 $done \leftarrow$ **false**
21 $E(U^T) \leftarrow k-1$



(a) AUCTION (b) BBOARD (c) BOOKSTORE

Figure 3.5: Starting with the California data privacy law, additional exposure reduction for query and update templates.

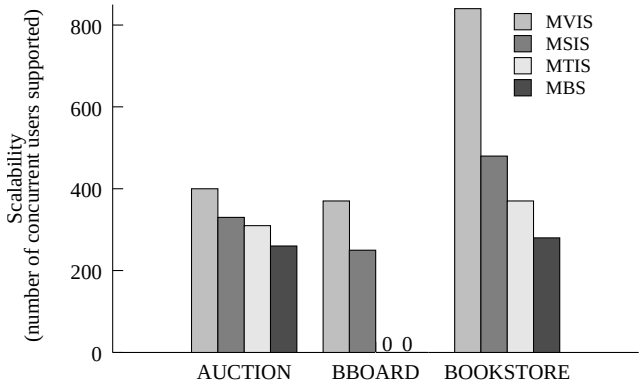


Figure 3.6: Tradeoff between security and scalability, as a function of coarse-grain invalidation strategy.

Chapter 4

Invalidation Clues for Database Scalability Services

Recall from Section 1.4.4 that invalidation clues present a general framework for applications to reveal little data to the DBSS, yet prevent wholesale invalidations. For completeness, we reproduce the description of invalidation clues from Section 1.4.4 in Section 4.1. Section 4.2 provides an overview of invalidation clues using an example. Section 4.3 and Section 4.4 show how different types of clues can be used to achieve different precisions in invalidations. Section 4.5 discusses how clues can be tailored to balance between privacy and scalability. Section 4.6 presents our empirical findings. Section 4.7 summarizes the contributions this chapter makes. Finally, Section 4.8 presents a summary of the chapter.

In this chapter, we focus on *privacy*. Note that our notion of privacy encapsulates the notion of security. Recall from Section 1.4.2 that the goals with privacy 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.

4.1 Introduction

Invalidation Clues. We present *invalidation clues*, a general framework for enabling 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. In this chapter, we show how specially designed clues can achieve three desirable goals:

(1) *Limit unnecessary invalidations:* Our clues provide relevant information to the DBSS that enable it to rule out most unnecessary invalidations.

(2) *Limit revealed information:* Our clues enable the application to achieve a target privacy by hiding information from the DBSS.

(3) *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.

Compared with previous approaches [6, 8, 12, 66, 70, 72, 74, 84], invalidation clues provide applications significantly improved tradeoffs between privacy and scalability. This difference is demonstrated in Figure 4.1 (discussed in detail in Section 1.3.1), which compares prior work in database scaling technology to our scheme. Only our scheme enables the favorable tradeoffs inside the dashed box.

4.2 An Illustrative Example

This section introduces invalidation clues via an example. Consider the application SIMPLE-BBOARD, specified in Table 4.1. In this application, queries follow the template Q^T (requesting information on comments, with rating above a threshold, made on a particular story) and updates follow the template U^T (changing a comment's rating). The DBSS caches the (encrypted) results of previous queries and uses any clues at hand to decide what to invalidate on an update.

Figure 4.1 plots six different scenarios of clues that illustrate the privacy-scalability tradeoff an application faces with various schemes, using SIMPLE-BBOARD as an example. It also plots prior work in

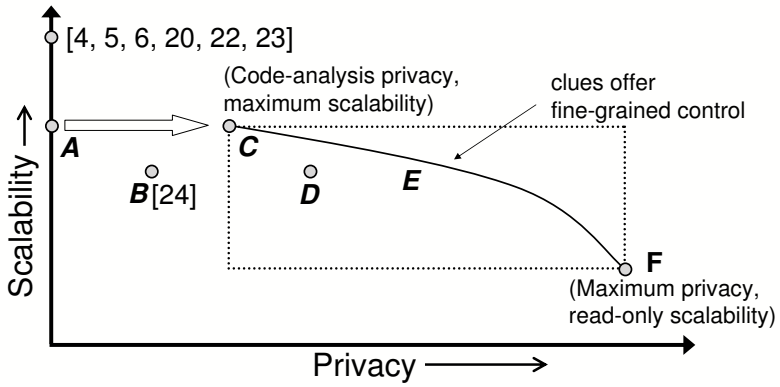


Figure 4.1: Privacy-Scalability tradeoff in the presence of clues. The dashed box shows the region in which an application can operate in our scheme. The six scenarios, ***A***–***F***, are explained later in Table 4.2. Code-analysis privacy and read-only scalability are explained in Section 4.5.1.

SIMPLE-BBOARD	
Q^T	`SELECT id, body FROM comments WHERE story=? AND rating>=?`
U^T	`UPDATE comments SET rating=rating+? WHERE id=?`

Table 4.1: A simplified bulletin-board example, consisting of a query template Q^T and an update template U^T on a base relation `comments` with attributes `id`, `story`, `rating`, and `body`. The question marks indicate parameters bound at execution time.

database scaling technology. Most of this work [6, 8, 12, 66, 70, 72] does not address privacy concerns, and as a result, can attain more scalability than our architecture (e.g., by not encrypting data, cached query results may be incrementally maintained at the caches, instead of just invalidated). Our previous work [74] (plotted as **B** in the figure) showed how to encrypt data that is not useful for invalidation. Without the general notion of clues introduced here, however, the previous work was unable to achieve the favorable tradeoffs in the figure's dashed box, even under a weaker attack model.

Table 4.2 summarizes the clue scenarios and what happens when an update occurs. Scenario **A** depicts a scenario in which the DBSS gets a copy of the entire database and sees the updates (`id` value of 123 and `rating` increment of 1 in the example update) and hence can perform precise invalidation (we formalize the notion in Section 4.3.4). Because the increase in rating by U^T can never cause `id`=123 to drop out

of a query result, the only case where the result is invalidated is when `id`=123 is not in the query result but its `story` matches Q's `story` and its new `rating` now exceeds Q's `rating` parameter. Scenario **F** depicts the other extreme—a scenario with no clues; in such cases, the DBSS has no way of knowing which (encrypted) cache result for an earlier encrypted query is invalidated by this (encrypted) update. Hence, it must invalidate the entire cache on an update. As Figure 4.1 shows, while the former provides maximum scalability (for invalidation based approaches) but no privacy, the latter provides maximum privacy but minimum scalability.

Scenario **B** translates the solution proposed in [74] into the terminology of this chapter. [74] did not have a notion of clues and privacy was "all-or-nothing"—the different attributes in parameters or the query results could not be encrypted independently. In this scenario, the DBSS does not know the `story` and `rating` of `id`=123, so if the `id` is not in the unencrypted query result, then the DBSS does not know whether the `id` should now be added and hence it must invalidate.

Because our clues can be arbitrarily fine-grained, our scheme enables better choices than previous schemes. Scenario **D**, for example, has the same invalidations as scenario **B**, but additionally encrypts the `body` of comments—only the `id` field is revealed, in order to enable checking for a particular `id`. Scenario **C** uses better clues than scenario **A**—they reveal less information (e.g., the `story`, `rating`, result `id`s but not the result `body`s), yet enable precise invalidation as in Scenario **A**. Including the `story` and `rating` of `id`=123 in the update clue is an example of a "database-derived" clue (discussed in Section 4.4), because these attributes are not in the update and hence need to be looked-up in the database.

Finally, scenario **E** uses Bloom-filters[1] to hide even the `id`s, at a cost of a small probability of an unnecessary invalidation. This example illustrates how clues offer fine-grained control to an application—the size of the Bloom-filter in this case—to choose a desired balance of privacy and scalability, as depicted by the range of choices in the curved line for scenario **E**.

[1] A Bloom-filter [19] 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 longer Bloom-filter reduces the number of unnecessary invalidations but reveals more about the data.

	Query Clue for Q	*Update Clue*	*Query Q Result invalidated*
A	entire database; *Q*'s `story&rating`	123, 1	if `id`=123 should be added given its `story` & `rating`
B	entire query result (unencrypted)	123, 1	if `id`=123 is absent from query result
C	*Q*'s `story&rating`, `id` values in result	123, and its `story&rating`	as in scenario A
D	`id` values (only) in query result	123	as in scenario B
E	*Q*'s `story&rating`, Bloom-filter of `id` values in result	Bloom-filter of {123}, and 123's `story&rating`	scenario A, with some false positives due to Bloom-filter
F	none	none	if any update occurs

Table 4.2: Six clue scenarios **A**–**F** and their effect on what the DBSS invalidates when an update U^T with `id`=123 and `rating=rating+1` occurs.

4.3 Using Clues for Invalidations

In this section we describe how clues can be used for invalidations. We begin in Section 4.3.1 by describing the architecture that is the context for our work. Section 4.3.2 provides the details of our basic query and update model, and introduces the terminology and notation we use in the rest of the chapter. Section 4.3.3 describes the attack model of the DBSS. Then, in Section 4.3.4, we formalize the notion of precise invalidations. Finally, in Section 4.3.5 we present various types of clues and provide examples of when each type is useful.

4.3.1 Architecture

The overall system architecture is as described in Chapter 2. The invalidation flow is described in Section 2.5. Depending on how the query clues and update clues are computed, this general formulation can emulate any invalidation strategy in the DBSS setting. In particular, the application, via clues, can send relevant data (about the rest of the database) to the DBSS, which may enable the DBSS to achieve

more precise invalidation.

4.3.2 Query and Update Model

Our query and update model is based on our study of three benchmark Web applications (details in Section 4.6.1). In our model there are a fixed set of query templates and a fixed set of update templates. A query is composed of a query template to which parameters are attached at execution time. Likewise, an update is composed of an update template to which parameters are attached at execution time. (Examples are in Tables 4.1, 4.4, 4.7, and 4.8.) 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 (except Cartesian product). As in previous related work [18, 74, 90], 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 semantics; 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 attributes of a relation. Rows satisfying the predicate are deleted. Each *modification* statement modifies non-key attributes of a row selected according to an equality predicate on the relation's primary key.

4.3.3 The Attack Model of the DBSS

In this chapter we use the following default "no-clue" scenario. The DBSS knows the application's database schema, including the primary keys and foreign keys, and 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.

When considering privacy, we assume that a DBSS can pose as a user "on top of" being honest-but-curious. An honest-but-curious DBSS invalidates correctly as per the query and update clues, but tries to infer the contents of the encrypted query results, encrypted queries, and encrypted updates, i.e., the DBSS is limited to ciphertext-only attacks [96]. Additionally, posing as a user enables the DBSS to issue queries and updates, observe which clues are generated, and correlate values in unencrypted queries and updates to clues, i.e., the DBSS can perform chosen-plaintext attacks [96].

4.3.4 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 and only 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.

4.3.5 Types of Clues

Recall that we distinguish between *query clues* (attached to encrypted query results) and *update clues* (attached to encrypted updates). We further classify query and update clues based on what data are used to compute them. A query clue might be a *parameter* query clue, a *result* query clue, or a *database* query clue, based on whether it is computed from the query parameters, the query result, or the database itself. Similarly, an update clue might be a *parameter* update clue or a *database* update clue based on whether it is computed from the update parameters or the database itself. Note that the contents of different types

	Computed from		
Attached to	*Parameters*	*Result*	*Database*
query result	parameter query clue	result query clue	database query clue
update	parameter update clue		database update clue

Table 4.3: A taxonomy of clues (The various clue types are in normal font). Clues differ based on whether they are attached to query results or updates, and whether they are computed from parameters, result, or database.

of clues may overlap. Table 4.3 summarizes the taxonomy of clues.

Consider the SIMPLE-AUCTION application shown in Table 4.4. For each of its query/update template pairs, Table 4.5 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. In other words, 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_3^T in response to an update template U_1^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_1^T now satisfies the cached Q_3^T query—but only if the item has the appropriate `category` and `region` (information available only in the database). So parameter and result clues are insufficient to prevent wholesale invalidation. Database clues are needed.

4.4 Database Clues

The previous section motivated the use of database clues using the SIMPLE-AUCTION example. We begin this section by identifying in Section 4.4.1 families of common query/update classes where database

SIMPLE-AUCTION	
Q_1^T	`SELECT item_id, category, end_date` `FROM items WHERE seller=?`
Q_2^T	`SELECT user_id FROM users WHERE region=?`

Q_3^T	`SELECT item_id FROM items, users`
	`WHERE items.seller=users.user_id`
	`AND items.category=?`
	`AND items.end_date>=?`
	`AND users.region=?`

U_1^T	`UPDATE items SET end_date=end_date+? DAYS` `WHERE item_id=?`
U_2^T	`INSERT INTO users (user_id, region)` `VALUES (?, ?)`

Table 4.4: A simple 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.

clues are required for precise invalidation. Section 4.4.2 discusses the problems with achieving precise invalidations using *database query* clues, and then presents our solution using *database update* clues. Finally, while database clues enable precise invalidation, for some workloads the overhead of computing them can be higher than their savings. Section 4.4.3 presents practical techniques that further reduce overheads and/or increase privacy by relaxing the precise invalidation requirement.

Pair	⟨*Query clue, Update clue*⟩
$\langle Q_1^T, U_1^T \rangle$	⟨ result, parameter ⟩
$\langle Q_1^T, U_2^T \rangle$	⟨ , ⟩ *(never invalidates: different relations)*
$\langle Q_2^T, U_1^T \rangle$	⟨ , ⟩ *(never invalidates: different relations)*
$\langle Q_2^T, U_2^T \rangle$	⟨ parameter, parameter ⟩
$\langle Q_3^T, U_1^T \rangle$	⟨ database, parameter ⟩ or ⟨ parameter, database ⟩
$\langle Q_3^T, U_2^T \rangle$	⟨ , ⟩ *(never invalidates: foreign key constraint)*

Table 4.5: Types of clues required to implement a DIS for template-pairs of the SIMPLE-AUCTION example in Table 4.4.

4.4.1 Templates Requiring Database Clues

We begin by introducing some terminology for classifying query and update templates in a way that is useful for our analysis. Then, we enumerate the query/update classes for which database clues are required for precise invalidation.

Query and Update Classification

Define the *selection attributes* of an update template U^T (denoted $S(U^T)$) to be the attributes used in any selection predicate (i.e., a selection or a join condition in the `where` clause) of U^T. (If U^T is an insertion, $S(U^T) = \{\}$.) Further define the *modified attributes* ($M(U^T)$) of U^T, the *selection attributes* ($S(Q^T)$) of a query template Q^T, and the *preserved attributes* ($P(Q^T)$) of Q^T as in Table 4.6. If U^T is an insertion or a deletion from a relation, $M(U^T)$ is defined to be the set of all attributes in the relation. For the SIMPLE-BBOARD application (Table 4.1), $S(U^T) = \{\texttt{comments.id}\}$, $M(U^T) = \{\texttt{comments.rating}\}$, $S(Q^T) = \{\texttt{comments.story}, \texttt{comments.rating}\}$, and $P(Q^T) = \{\texttt{comments.id}, \texttt{comments.body}\}$.

Symbol	*Meaning*
$S(U^T)$	Attributes used in the selection/join predicates of U^T (i.e., in the `where` clause)
$M(U^T)$	Attributes modified by U^T
$S(Q^T)$	Attributes used in the selection/join predicates or order-by constructs of Q^T
$P(Q^T)$	Attributes preserved in the result of Q^T (i.e., in the `select` clause)

Table 4.6: Notation for aspects of templates.

Enumeration of Classes

We identify important classes of update/query template pairs, for which database clues are necessary for achieving the invalidation behavior of a DIS. For all the other classes in the query and update model we consider, described in Section 4.3.2, database clues are not necessary. (We omit proofs for brevity.)

For ease of understanding, we divide the classes into three main categories. A common condition across all three categories is that the update not be "ignorable" with respect to the query. We say an update template is *ignorable* with respect to a query template if and only if none of the attributes modified by the update template belong to either the selection or preserved attributes of the query template. Formally, an update is ignorable if and only if $M(U^T) \cap (S(Q^T) \cup P(Q^T))$ is empty. For simplicity in the discussion below, we assume that there are no foreign key constraints. The discussion can easily be extended to handle foreign keys. Next, we enumerate the three categories. For each category, if applicable, we provide separate examples for insertion, deletion, and modification templates.

Category I. The rules for the first category are: (a) the update might add at least one row to the query result, and (b) there is at least one attribute belonging to the query's selection attributes whose final value is not specified in the update. The intuition behind this rule is that as long as there is at least

one attribute whose value needs to be examined in the database in order to determine whether or not the update affects the query result, a database clue is required. For example, in Table 4.4, consider the query Q_3^T with either modification template U_1^T, or the following insertion and modification templates:

```
INSERT INTO items (item_id, seller, category, end_date) VALUES (?, ?, ?, ?)
UPDATE items SET end_date=?  WHERE item_id=?
```

Category II. The rules for the second category are: (a) the query involves a top-k predicate, and (b) the query fails to preserve at least one of its order-by attributes that is modified by the update. The intuition behind this rule is that because of the top-k predicate, even when an update affects some tuple in the database that is absent from the query result, it might affect the query result. For example, consider the query template `SELECT item_id FROM items WHERE category=? ORDER BY end_date FETCH` 11^{th} `to` 21^{st} `rows`[2] paired with any of the following templates:

```
INSERT INTO items (item_id, seller, category, end_date) VALUES (?, ?, ?, ?)
DELETE FROM items WHERE item_id=?
UPDATE items SET category=?  WHERE item_id=?
```

Category III. The rule for the third category is: there is at least one attribute in the selection predicate of the update template that is not preserved by the query template. The intuition behind this rule is that the query result does not contain sufficient information to determine whether the update affects the query result or not. For example, consider the query template `SELECT end_date FROM items WHERE category=?` paired with either of the following:

```
DELETE FROM items WHERE item_id=?
UPDATE items SET end_date=?  WHERE item_id=?
```

[2]Such a query arises, e.g., when the application wants to fetch and display the second page of query results.

4.4.2 Implementing Database Clues

We now discuss how to implement database clues, so as to achieve as precise invalidations as 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 is 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 [90] could be used to identify the minimal such data. For example, for query template Q_3^T in Table 4.4, the techniques in [90] 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_3^T query; these are the only rows that can possibly become part of the query result as a result of U_1^T updating the `end_date` for some item.

In general, given many cached queries and a richer collection of update templates than in the SIMPLE-AUCTION example, the amount of auxiliary data stored to maintain the views 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_2^T, which could previously be ignored for Q_3^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 at runtime 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_1^T in Table 4.4, 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, these enable a DBSS to achieve a DIS, by checking whether these four values now satisfy Q 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 $\langle$ query template, update template $\rangle$ pairs, only 21 require database clues (details are in Section 4.6.1). 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 $\langle Q_3^T, U_1^T \rangle$ example above. Moreover, for these same reasons, database update clues achieve better privacy.

We use the following procedure for determining clues. Most of the work is precomputed offline given the set of templates for an application. For our three applications, we performed this precomputation by hand; however, it would not be difficult to automate much of this process. For example, precomputing which update templates are ignorable by which query templates can be automated by extracting $S(U_i^T)$ and $M(U_i^T)$ for each update template U_i^T and $S(Q_j^T)$ and $P(Q_j^T)$ for each query template Q_j^T, and then testing whether $M(U^T) \cap (S(Q^T) \cup P(Q^T))$ is empty. Similarly, there are simple, easily automated, rules for determining pairs made ignorable by foreign key constraints. The precomputed results are stored in a table for fast reference during execution. For those pairs using database update clues, a script is generated and stored in the table for computing the clue. Figure 4.2 shows how a database update clue is computed for single table SPJ queries. (Note that if U^T is a modification template, the algorithm in Figure 4.2 must be called twice, once *before* and once *after* applying the update. If U^T is a deletion template, the algorithm must only be called *before* applying the update.) This algorithm can readily be extended to handle top-k and join queries. After the extension, there are only a few pairs in our benchmarks, which fall outside the query and update model we consider, that we currently only know

Algorithm: For update template U^T and SPJ query template Q^T, find the database-update clue.

Inputs: update template U^T, query template Q^T

Output: database-update clue C as an associative array

1 If U^T is an insertion, **return**

2 $X \leftarrow M(U^T) \cap (P(Q^T) \cup S(Q^T))$

3 If $X = \{\}$, **return** `/* ignorable update */`

4 if U^T is a deletion

5 $\quad X \leftarrow S(Q^T)$

6 **for each** *attr* a $\in X$,

7 $\quad C\{\texttt{a}\} \leftarrow$ "value of a in the row being updated"

8 **return C**

Figure 4.2: Pseudo code for computing a database update clue when query templates are restricted to a single table.

how to do by hand.

4.4.3 Beyond Precise Invalidations

Thus far, we have focused on the goal of matching DIS's optimal number of invalidations. 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.

Opportunistic Database Clues. Although the overheads of computing database clues are minimal, depending on the workload, their overheads can still be higher than their savings in some cases. 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 OPPORTUNISTIC 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 4.6.

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 $\langle Q_3^T, U_1^T \rangle$ 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 $\langle Q_3^T, U_1^T \rangle$ clue. In such cases, the actual values can be obscured by using a one-way hash function. 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 $\langle Q_3^T, U_1^T \rangle$ clue. In such cases, the actual values can be hidden to varying degrees as a tradeoff against invalidation precision, as will be discussed in Section 4.5.

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 SIMPLE-BBOARD example in Table 4.1 and the corresponding result query clue and parameter update clue in Scenario **C** of Table 4.2. These clues enable exact matching of `id`s but reveal all the `id` values in the query result. Instead, as shown in Scenario **E** of Table 4.2, we can obscure these `id` values by using Bloom-filters [19], as discussed in Section 4.2. 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 exact matching, but with better privacy.

4.5 Privacy-Scalability Tradeoffs

In this section we study privacy-scalability tradeoffs in the DBSS setting, considering the attack model of Section 4.3.3. We begin in Section 4.5.1 by showing that there is a fundamental tradeoff between privacy and scalability in our DBSS setting. Section 4.5.2 then presents an overview of how applications could get extra privacy by having the DBSS carry out unnecessary invalidations. Next, in Sections 4.5.3 and 4.5.4, we study representative query and update template pairs from our benchmark applications, and present configurable clues for these pairs. Finally in Section 4.5.5, we discuss how our current work applies to entire applications, beyond a single query and update template pair.

4.5.1 The Limit Cases

Recall the dashed box in Figure 4.1 from Section 4.1, which illustrates the privacy-scalability tradeoff that an application faces in our DBSS setting, where (a) the DBSS has an attack model as described in Section 4.3.3 and (b) the home server does not track the state of the DBSS's cache. We denote as *code-analysis privacy* the level of privacy that an application can attain by encrypting the data not

useful for invalidation (determined statically by analyzing the application code as in [74]). On the other hand, minimal scalability is achieved when the DBSS invalidates all its cache entries on any update, i.e., queries can only be answered from the cache as long as the workload remains read-only. We call this level of minimal scalability *read-only scalability*.

As we show next, if an application achieves the maximum scalability, it gets code-analysis privacy (the upper left corner of the dashed box in Figure 4.1), and if it achieves the maximum privacy, it gets read-only scalability (the lower right corner of the dashed box in Figure 4.1). Thus, applications cannot hope for both good scalability and good privacy.

Maximum privacy implies read-only scalability. An application achieves the maximum privacy if the DBSS it is using cannot distinguish between any two encrypted query results in its cache. Because the DBSS can pose as a user and issue updates, on any update, either all or none of an application's query results should be invalidated. Otherwise, the DBSS can distinguish between query results that were invalidated and those that were not invalidated. Furthermore, for any non-trivial workload, it is likely that an update invalidates some query result. Because the home server does not track what the DBSS's cache contains, for privacy and correctness, it requires the DBSS to invalidate all query results on every update. Thus the application achieves read-only scalability.

Maximum scalability implies code-analysis privacy. An application achieves maximum scalability when the invalidation behavior of the DBSS resembles a Database Inspection Strategy (Section 4.3.4). We focus on two representative cases: (a) the invalidation decision involves an equality comparison, and (b) the invalidation decision involves an order comparison. In case (a), the DBSS can repeatedly issue updates till the query result is invalidated. Since the invalidation is precise and the DBSS is issuing the updates, the DBSS learns the value of the data in the query result used for invalidation. In case (b), the DBSS first computes an ordering between encrypted query results. It can do so easily, based on the frequency with which a query result is invalidated. (Note that cache evictions do not affect the maintenance of the frequency count, because (i) the DBSS can always store the query result just for the purposes of maintaining this frequency count, and (ii) the home server does not track the contents of a DBSS's cache.) It can then pose as a user and do a binary search on the ordered query results to find the

Q^T	`SELECT i_stock FROM item WHERE i_id=?`
U^T	`UPDATE item SET i_stock=? WHERE i_id=?`

Table 4.7: A query-update template pair from the BOOKSTORE benchmark.

value corresponding to an encrypted query result. Thus in both cases, equality and order comparisons, maximum scalability results in the code-analysis privacy.

4.5.2 Trading Off Scalability for Privacy

In order to increase privacy, 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 4.5.3) or order comparisons (Section 4.5.4).

4.5.3 Equality Comparisons

Consider an actual template pair, shown in Table 4.7, from the BOOKSTORE benchmark (details in Section 4.6.1) 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[3] 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

[3]In general, the discussion here applies to all attribute values used in invalidation equality comparisons, not just parameter values.

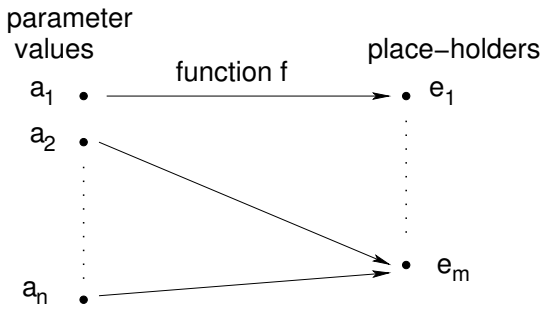


Figure 4.3: An example mapping of parameter values to place-holders.

by a bipartite graph as in Figure 4.3. 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 4.4.3.

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.

Next, we show that an application can use knowledge of the frequency distribution of parameters to further choose clues that maximize its scalability for a given privacy value. Before proceeding, we introduce some notation.

Let p_j denote the probability with which an update with parameter a_j is issued. Formally, $\sum_j p_j = 1$. For each of the place-holder values e_i, let domain-size n_i and cumulative probability P_i denote the

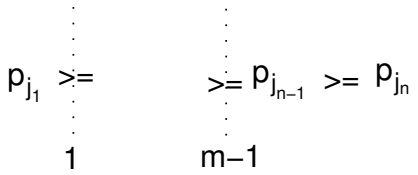


Figure 4.4: The solution implied by Lemma 2. $j_i \in \{1,\ldots,n\}$ is such that the parameter value a_{j_i} is the ith most frequently occurring.

number of parameter values mapped to a place-holder e_i and the sum of their probabilities, respectively. Formally, for $i \in \{1,\ldots,m\}$, $n_i = |\{a_j | f(a_j) = e_i\}|$, and $P_i = \sum_{f(a_j)=e_i} p_j$. Also $\sum_{i=1}^{m} n_i = n$, and $\sum_{i=1}^{m} P_i = 1$.

If the application knows the p_j values, for a given fixed privacy value m, we show how it can choose a mapping that minimizes the total number of invalidations (the term $\sum_{i=1}^{m} n_i P_i$ represents the total number of invalidations). Formally, the constrained optimization problem is to find the EQUALITY-OPTIMAL mapping that minimizes $\sum_{i=1}^{m} n_i P_i$ given the constraints $\sum_{i=1}^{m} n_i = n$ and $\sum_{i=1}^{m} P_i = 1$. Lemma 2 provides the key insight required to find the EQUALITY-OPTIMAL mapping.

Lemma 2. *For a given privacy value, the minimum number of invalidations is achieved when: for any two place-holders e_i and e_j with domain-size n_i less than domain-size n_j, the probability with which an update using a value mapped to e_i is issued is higher than the probability with which an update using a value mapped to e_j is issued.*

Proof. Suppose the number of invalidations is minimum, and yet there are two place-holders e_i and e_j with $n_i < n_j$ such that for value x mapped to e_i ($f(x) = e_i$) and value y mapped to e_j ($f(y) = e_j$), $p_x \leq p_y$.

In the expression for the number of invalidations, the contribution of terms in which p_x and p_y appear is $n_i p_x + n_j p_y$. By swapping x and y, this contribution is reduced, thereby reducing the total number of invalidations. Hence, the original mapping was not minimum, a contradiction. □

Lemma 2 implies that the final solution has a form as shown in Figure 4.4, where the parameter values are arranged in a sorted order of the probabilities with which they are issued, and only parameter values with consecutive ranks can map to the same place-holder. Another implication of Lemma 2 is that the

Q^T	`SELECT * FROM items WHERE end_date>=?`
U^T	`INSERT INTO items VALUES (?, ..., ?)`

Table 4.8: A simplified query-update template pair from the AUCTION benchmark.

problem of finding an EQUALITY-OPTIMAL mapping has the *optimal sub-structure* property, i.e., parts of the mapping are themselves optimal solutions to parts of the problem. Dynamic Programming, which uses memoization to get rid of repeated computations, can be used to solve this problem in $O(nm)$ space and $O(n^2m)$ time.

In Section 4.6.3 we show that in the common case, an EQUALITY-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.

4.5.4 Order Comparisons

Consider the template pair shown in Table 4.8. This pair is from the AUCTION benchmark (details in Section 4.6.1), 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 `end_date` in the query and the update. However, the application may not want to reveal the exact `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) [3] to 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 $f(a_i) < f(a_j)$. An honest-but-curious DBSS can learn a total ordering on the place-holders either immediately (if it can observe the execution of the invalidation code), or over

time (if it can only observe which results are invalidated). However, privacy is still preserved since the DBSS cannot associate place-holders to actual parameter values (as in [3]). In contrast to an honest-but-curious DBSS, use of an OPES provides little privacy with our attack model. The DBSS by posing as a user can initiate queries with known parameter values, observe the clues generated, and correlate place-holders to the parameter values. Moreover, since it can learn a total ordering on the place-holders (as mentioned above), it can use binary search to quickly find the parameter value(s) corresponding to a 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, $a_k = \max_{f(a_k)=e_i}$ and $a_l = \min_{f(a_l)=e_j}$. 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.

To defeat 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. Formally, for two query (or update) parameter values a_i and a_j with $a_i < a_j$ and mapping f, $f(a_i) < f(a_j)$ need not be true. This flexibility enables 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 because even if $a_i < a_j$, neither $f_q(a_i) < f_q(a_j)$ nor $f_u(a_i) < f_u(a_j)$ may be true.

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 tradeoff. 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, 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 and assuming that all parameter values are equi-probable, we show 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.

Lemma 3. *In an* ORDER-OPTIMAL *mapping, for any two parameter values a_i and a_j with $a_i < a_j$, if $r_q(a_i)$ and $r_q(a_j)$ are non-zero, then $f_q(a_i) > f_q(a_j)$.*

Proof. **By contradiction.** Assume in an ORDER-OPTIMAL mapping, there exist two values a_i and a_j with $a_i < a_j$, for which $r_q(a_i) > 0$ and $r_q(a_j) > 0$. If $r_q(a_j)$ is increased by 1 and $r_q(a_i)$ is decreased by 1, the total number of invalidations remain the same, but the privacy increases. Hence contradiction. □

An implication of Lemma 3 is that for any given number of invalidations *i*, to find ORDER-OPTIMAL, the following two steps should be carried out: (1) Find values $a_{-n} < a_{-n+1} < \ldots a_{-1}$ so that $a_{-1} = a_1$. (2) Starting with the maximum a_i, map each a_i to a_{-i} till the invalidation limit is reached. If the invalidation limit is reached in an a_i getting to a_{-i}, allow the a_i to reach whatever value is reachable.

Section 4.6.3 shows that for a given scalability value, this mapping enables twice the privacy of an OPES.

4.5.5 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. Thus, our above results can be applied to the entire application. Note, however, that care must be taken

in treating queries or updates with conjunctions between arithmetic predicates that share attributes (e.g., `WHERE end_date > ? AND end_date < ? + 30 DAYS`).

4.6 Evaluation

We evaluated our proposed clues by implementing them in our prototype DBSS and then measuring the scalability advantages of using various types of invalidation clues. In this section, we describe characteristics of our benchmark applications in Section 4.6.1 and our scalability results in Section 4.6.2. Finally, in Section 4.6.3 we measure the effectiveness of our techniques in helping an application manage its privacy-scalability tradeoff.

4.6.1 Characteristics of the Benchmark Applications

We used the benchmark applications described in Section 2.6. There were a few queries in these benchmark applications (12 out of 94 templates) that did not conform to our query model (Section 4.3.2), e.g., aggregate queries. For these queries, we use parameter and result clues but not database clues.

Table 4.9 provides, for each of the three applications, the number of template pairs which require database clues for precise invalidations, and classifies them according to the categories introduced in Section 4.4.1. As the table shows, only 21 (out of the over 1000) pairs require database clues, and all but 2 of these fall into Category I.

4.6.2 Scalability Benefits of Invalidation Clues

We performed our experiments in the SIMPLE scenario and the methodology described in Section 2.7. Figure 4.5 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 2.7. On the x-axis, we consider five cases: one corresponding to not using a DBSS, one corresponding to not

Application	*Number of $\langle U^T, Q^T \rangle$ pairs in category*		
	Category I	Category II	Category III
AUCTION	9	1	0
BBOARD	7	0	0
BOOKSTORE	3	1	0

Table 4.9: Number of template pairs in the three applications which require database clues for precise invalidations, classified as per the categories introduced in Section 4.4.1.

using clues[4], and the other three corresponding to DBSS strategies based on different classes of clues: *Clues (excl. DB clues)*, which uses only parameter and result clues[5], *Clues (incl. DB clues)*, which uses parameter, result, and database update clues (as presented in Section 4.4.2), and *Opportunistic*, which uses the OPPORTUNISTIC strategy presented in Section 4.4.3.

In all applications, using a DBSS with invalidation clues significantly increased scalability. This agrees with previous work [74], which can be viewed as having considered specific types of (non-database) clues. Because the rightmost strategy, Opportunistic, 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. As the figure shows, the results for the BBOARD application differ from the others in two respects. First, when no clues are used, not even a small number of clients can be supported within the response time threshold specified in Section 2.7. This is because each HTTP request results in about ten database requests, most of which suffer cache misses (due to no clues being used). Second, the overhead of computing database update clues is high relative to the decrease in invalidations. Hence, as Figure 4.5 shows, using database update clues whenever required for precise invalidations results in worse scalability. Figure 4.5 thus confirms the claim made in Section 4.4.3 that the use of database update clues must be carefully weighed against the expected benefit.

[4]The scalability of this strategy is the same as the Minimal Template-Inspection Strategy (MTIS) of [74].

[5]The scalability of this strategy is the same as the Minimal View-Inspection Strategy (MVIS) of [74].

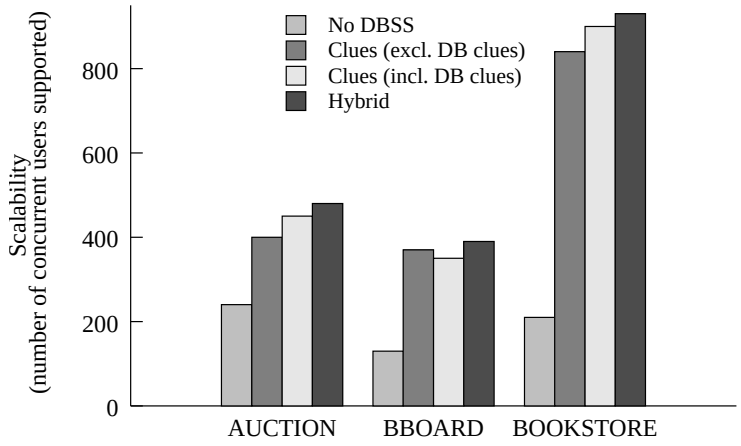


Figure 4.5: Impact of invalidation clues on scalability. For comparison, we include the scalability numbers without a DBSS.

4.6.3 Privacy Experiments

Figure 4.6 shows the reduction in the number of invalidations. The workload used is the template pair in Table 4.7, with the parameter values chosen according to the Zipf distribution in BOOKSTORE, over a domain of 100 values. The y-axis plots the percentage reduction in invalidations in using our EQUALITY-OPTIMAL mapping (Section 4.5.2), over a simplistic mapping which maps an equal number of parameter values to each place-holder. (The percentage reduction is a crude estimate of the scalability improvement an application can achieve by switching to an EQUALITY-OPTIMAL mapping.) On the x-axis, we plot the number of place-holders. (Recall from Section 4.5.2 that fewer place-holders implies greater privacy.) As expected, when all parameter values are mapped to a single place-holder or most are mapped to separate place-holders (right part of the graph), both mapping algorithms result in almost the same number of invalidations. In other cases, however, the EQUALITY-OPTIMAL algorithm reduces invalidations by around 20%. The benefits increase as the distribution over the parameters becomes more skewed.

Figure 4.7 plots the improvement in privacy due to using two mappings instead of one mapping, as described in Section 4.5.4. The workload used is the template pair in Table 4.8, with parameter

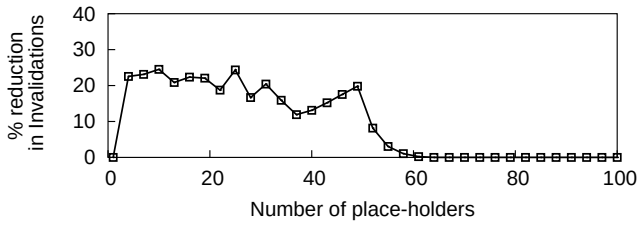


Figure 4.6: Reduction in invalidations due to our EQUALITY-OPTIMAL mapping algorithm.

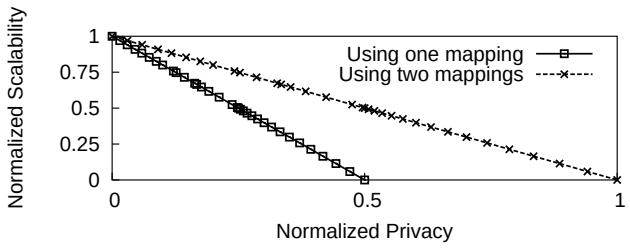


Figure 4.7: Improvement in privacy on using two mappings instead of one mapping.

values chosen uniformly-at-random, over a domain of 100 values. The x-axis plots normalized privacy, measured as per the definition in that section. The y-axis plots normalized scalability, measured as $\frac{\max - I_j}{\max - \min}$, where I_j is the number of invalidations for the jth 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, augmented so that multiple adjacent values could be mapped to a single place-holder. For the two mappings approach, we use an identity mapping, and the ORDER-OPTIMAL mapping described in Section 4.5.4. For a given scalability, with our two mapping approach, the privacy is almost twice that of a one-mapping approach. Although these results are skewed by the specific privacy measure we use, we believe that the factor of two gap between the curves demonstrates a significant opportunity for using two-mapping approaches.

4.7 Chapter Contributions

In this chapter, we made the following contributions:

- We presented invalidation clues, a general framework that offers applications a low overhead, fine-grained control to balance their privacy and scalability needs, and provides better tradeoffs than

previous approaches. We also provided examples of several configurable invalidation clues.

- We showed how to keep application data secure/private under a general attack model where the DBSS can pose as a user, issuing queries and update, and then observing the invalidations in an attempt to learn other user's data.

- We identified families of common query/update classes where extra information is needed from the database in order to perform precise invalidations. We showed that generating these "database-derived" clues in response to an update typically requires accessing only one or two database rows. We presented a strategy that uses such clues only when the scalability benefit from reduced invalidations outweighs the cost of computing the clue.

- Using experiments with three Web benchmark applications—a bookstore (TPC-W), an auction (RUBiS), and a bulletin-board (RUBBoS)—running on our prototype DBSS, we demonstrated the scalability benefits of our proposed clues. We also used representative queries from these benchmarks to show the effectiveness of our configurable clues in providing an improved privacy versus scalability tradeoff.

4.8 Summary

Database scalability services (DBSSs) are an extension of CDNs that offload work from and absorb load spikes for individual application databases, thereby removing a key bottleneck for many Web applications without the expense/headaches of an over-provisioned server farm. This chapter presented *invalidations clues*, a general framework and techniques for enabling applications to reveal little data to the DBSS, yet provide sufficient information to limit unnecessary invalidations of results cached at the DBSS. Compared with previous approaches, our proposed invalidation clues provide increased scalability to the DBSS for a target privacy level, as well as more fine-grained control of this tradeoff. Using three realistic Web benchmark applications, we illustrated the issues and solutions for generating effective clues, e.g., by identifying categories requiring database clues, and then we demonstrated the solutions on our DBSS prototype.

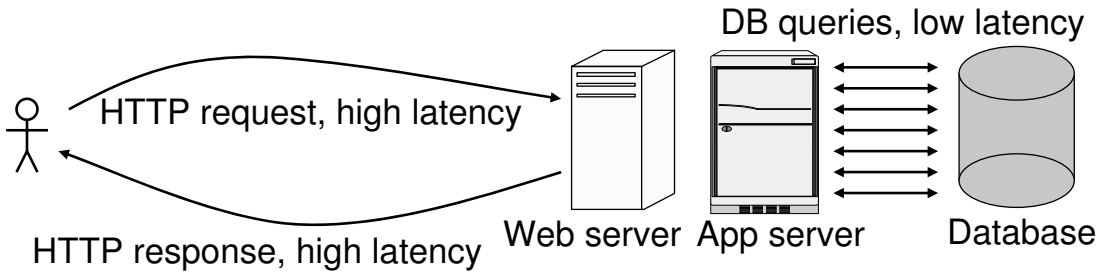
Chapter 5

Holistic Query Transformations for Dynamic Web Applications

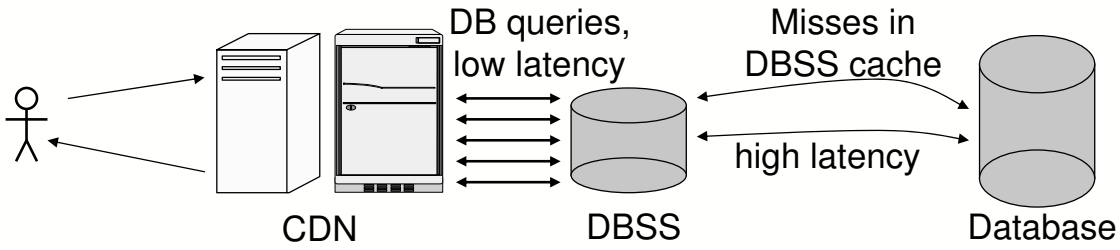
Web applications are interactive. User studies [59, 60] have shown that high user latencies drive customers away. Therefore it is important that we preserve low user latencies with a scalability service, even under high load. For our experiments throughout this thesis, we take this into account by requiring responses to have a latency of two seconds or less.

To ensure low user latencies, it is important to understand how this latency arises. A Web application is a collection of programs. On an HTTP request, an application server runs one or more of these programs to generate the response. To access the data these programs need to generate the response, they issue database queries. Frequently, the programs issue multiple database queries for each HTTP interaction: e.g., for the benchmark applications we study, the average number of queries per dynamic HTTP request varies between 1.8 and 9.1 (Table 5.3). In a traditional centralized setting, these database queries are answered by a database server, which is in the same administrative domain and connected to the application server(s) by a high bandwidth, low latency link. As a result, these multiple round-trips have little impact on the overall latency a user experiences. The user latency is dominated by the high latency of reaching the web server of the application. Figure 5.1(a) shows the different latency components in a traditional centralized setting.

In a scalability service setting, the DBSS first tries to answer a database request from its cache. If



(a) Latency in a traditional centralized architecture.



(b) Latency in a distributed architecture.

Figure 5.1: Latency in a traditional versus distributed architecture.

the request hits in the DBSS cache, the delay in obtaining the query result is minimal. However, if the request misses in the cache, the user must endure the delay in getting the response back from the home server database. This delay is typically long because the scalability service nodes are geographically distributed. Figure 5.1(b) shows the different latency components in a scalability service setting. Even after methods to boost the cache hit rate are employed by scalability service nodes, users are likely to experience a high latency in the scalability service setting if multiple database requests miss the cache on an HTTP request.

To reduce the user latency, it is desirable to either eliminate database requests or hide their latencies. There are several reasons why opportunities to do so appear in current Web applications. First, these applications are typically written for a traditional centralized setting, in which there is minimal overhead of issuing multiple database requests. So application developers frequently do not optimize for the number of database requests the application issues. Second, application developers find it convenient to

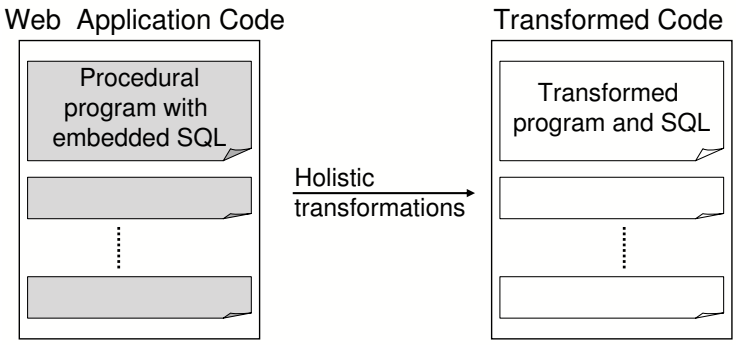


Figure 5.2: The holistic transformations, when applied to a Web application, reduce the number of database queries that the Web application issues per HTTP request at runtime.

abstract database values as objects in the program, a paradigm that is also adopted by Object Relational Mapping tools [41, 85]. If they need multiple values, they just issue multiple queries. Third, there are instances where it is easier for developers to express their main logic in the procedural language and issue multiple, short queries because it is closer to how the data is actually presented to the user, as in the example in Figure 5.5.

In this work we propose two transformations that rewrite the application code to either eliminate database requests or hide their latencies. Our first transformation, the MERGING transformation, eliminates queries by clustering related queries. Our second transformation, the NONBLOCKING transformation, hides the long latency in fetching query results, by overlapping the execution of queries.

Both transformations that we propose change the database queries as well as the application code surrounding them. Web applications are commonly written in a procedural language like Java or PHP whereas they issue database queries in a declarative language, typically SQL. Applying these transformations requires an understanding of the procedural language as well as the declarative language. These transformations affect the program as a whole. Therefore we call these transformations *holistic* (Figure 5.2). To evaluate the effectiveness of these transformations, we have applied it to three benchmark applications. While we currently applied them manually, we believe that the algorithms (described in Section 5.1.3 and Section 5.2.1) should be straightforward to automate in a source-to-source compiler [39, 81]. We next discuss the transformations individually in the next two sections.

```
$template:=SELECT from_user_id
           FROM comments
           WHERE to_user_id = ?;
$query:=set_params ($template, $to_id);
$result:=execute ($query);
foreach ($row in $result) {
   $from_id:=get_user_id ($row);
   $template:=SELECT user_name
              FROM users
              WHERE user_id = ?;
   $query:=set_params ($template, $from_id);
   $result2:=execute ($query);
}
```

(a) Original code

```
$template:=SELECT from_user_id, user_name
           FROM comments, users
           WHERE from_user_id = user_id
             AND to_user_id = ?;
$query:=set_params ($template, $to_id);
$result:=execute ($query)
```

(b) After the MERGING transformation

Figure 5.3: A code fragment from the AUCTION application, showing the original code on the left, and the code after applying the MERGING transformation on the right. The code, an example of the Loop-to-join pattern, finds the names of users who have posted comments about a particular user. 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`.

5.1 The MERGING Transformation: Clustering Related Queries

We explain the MERGING transformation using an illustrative example. Consider the left-hand side code fragment of Figure 5.3 which is taken from the AUCTION benchmark. The program issues several short inter-related queries and the procedural code combines their results. In a DBSS setting, for each query that results in a cache miss at the DBSS node, the user must endure the long delay of accessing the home server database. Assuming a constant hit rate at the DBSS cache, the latency observed by the end-user is proportional to the number of queries issued in an HTTP interaction. The MERGING transformation transforms the left-hand side to the equivalent right-hand side code, merging all the short inter-related queries into one join query. The program then needs to issue just one query instead of the previous $N+1$ queries, assuming the loop is repeated N times.

5.1.1 Impact on the Total Work in the System

While it is certainly possible for the MERGING transformation to either decrease or increase the total amount of work done in the system, we do not expect it to affect the total amount of work in the system. We use the term *work* to mean the use of any resources like disk I/O or CPU in the system. In most cases, like the example in Figure 5.3, we simply expect it to change the division of work between the application and the database server(s). The example, which involves a simple one-to-one join operation, is likely to require the same amount of work even after the MERGING transformation. For a complete understanding of the consequences of this optimization, we however discuss instances in which applying this transformation might decrease or increase the total amount of work in the system.

Applying the MERGING transformation can decrease the total amount of work if the database is able to execute the queries more efficiently after applying the transformation. This happens when the database can execute a larger task more efficiently than executing several small tasks: e.g., if the application code in Figure 5.3 involved a many-to-many join instead of an one-to-one join. The left hand side code will still implement a nested loop join in the application with a pre-determined outer table whereas on the right hand side, the database optimizer will be able to do a much better task at optimizing the join based on the cardinality and selectivity estimations of the tables involved and the available indices. See [29] for an overview of query optimization in relational databases. This reduction in work would be reflected in improved scalability in both the centralized as well as the DBSS setting. Since deployed application code is unlikely to have such inefficiencies, we do not expect such opportunities to exist in a deployed application code. As expected, we did not find any opportunities for significantly reducing work in the three benchmark applications we used. As a result, we did not see any measurable scalability improvement due to the MERGING transformation in any of the benchmark applications in the centralized setting.

Applying the MERGING transformation can increase the total amount of work in the system when one or more of the queries being merged is issued conditionally. For example, consider a slightly modified version of Figure 5.3. Imagine that the loop is executed only if the returned result contained at least ten rows. Then applying the transformation can increase the total amount of work if most of the returned

```
$template1:=SELECT MAX(bid) FROM bids
            WHERE item_id = ?;
$query1:=set_params ($template1, $item_id);
$result1:=execute ($query1);
$template2:=SELECT COUNT(*) FROM bids
            WHERE item_id = ?;
$query2:=set_params ($template2, $item_id);
$result2:=execute ($query2);
```

(a) Original code

```
$template:=SELECT MAX(bid), COUNT(*)
            FROM bids
            WHERE item_id = ?;
$query:=set_params ($template, $item_id);
$result:=execute ($query);
```

(b) After the MERGING transformation

Figure 5.4: An example of the merge-projection-predicates pattern, showing the original code on the left, and the code after applying the MERGING transformation on the right. The code fragment is a simplified version of the code from the AUCTION application, and finds the current maximum bid and the total number of bids for an item. We focus on the `bids` relation with the `bid` and the `item_id` attributes.

results had fewer than ten rows. To decide whether to apply this transformation in such "speculative" situations or not, we use estimates of the relative costs of evaluating the query result and the frequencies with which the different queries are issued. In practice, we will not apply this transformation when doing so increases the total amount of work. Only once in our benchmark applications, we had to decide whether to apply this transformation speculatively or not – it occurred in the BBOARD benchmark and we decided to speculatively apply the transformation.

In the example in Figure 5.3, the MERGING transformation converted a loop in the application code to a database join. We call this pattern the "loop-to-join" pattern. In the next section we list all the patterns that we found in the three benchmark applications.

5.1.2 Code Patterns Where the MERGING Transformation Applies

Based on our study of the three benchmark applications, we found three code patterns where the MERGING transformation applies. In Table 5.2 of Section 5.3 we list how frequently each of these patterns exist in the benchmark applications.

Loop-to-join: In this pattern the application first issues a query to get multiple values and then for each

```
$template:=SELECT id, body FROM comments
           WHERE parent = 0 AND story = ?;
$query:=set_params ($template, $sid);
$result:=execute ($query);
push_comments (Stack, $result);
while (($comment:=pop (Stack)) != NULL) {
   print ($comment);
   $cid:=get_id ($comment);
   $template:=SELECT id, body FROM comments
              WHERE parent = ?  AND story = ?;
   $query:=set_params ($template, $cid, $sid);
   $result := execute ($query);
   push_comments (Stack, $result);
}
```

(a) Original code

```
$template:=SELECT id, body, parent
           FROM comments
           WHERE story = ?;
$query:=set_params ($template, $cid, $sid);
$result:=execute ($query);
```

(b) After the MERGING transformation

Figure 5.5: An example of the merge-selection-predicates pattern, showing the original code on the left, and the code after applying the MERGING transformation on the right (We just show the database queries on the right). The simplified code fragment is from the BBOARD application, and shows all the comments on a story in a tree format. We focus on the `comments` relation with the `id`, `body`, `parent`, and `story` attributes.

value (using a loop structure), issues another database query. This code pattern can be transformed to a single database join query, as in the example in Figure 5.3. Out of the three patterns we found, this pattern occurs the most frequently. In fact, it occurs in all three benchmark applications that we study.

Merge-projection-predicates: In this pattern the application issues multiple queries in succession that are identical except in the attributes they project. This code pattern can be transformed to a single database query where the projection clause is a union of the projection clauses of the original queries. For instance, in the AUCTION benchmark example in Figure 5.4, a query to find the maximum bid on an item is followed by another query to find the number of bids for the same item.

For any merge-projection-predicates pattern, the MERGING optimization reduces the total work in the system. For the example in Figure 5.4, after the `MERGING` transformation, the database must

lookup just one row instead of looking up the row twice, saving on both the disk I/O and CPU costs. Of course, these costs will be reduced only if the database is "row" oriented, which is true of most general purpose databases today. While this pattern exists in the AUCTION and BBOARD benchmarks, we do not expect this pattern to occur frequently for a deployed Web application. This pattern might exist only when it is difficult to optimize away the pattern: e.g., (1) there is a significant time gap among when the different component queries being issued, and (2) some component queries are issued speculatively.

Merge-selection-predicates: In this pattern the application issues multiple queries in succession that are identical except in a selection clause. This code pattern can be transformed to a single database query where the differing selection clause is dropped and the attribute used in the dropped selection clause is added to the projection attributes. For example, in the BBOARD benchmark, when a user views a story, all the comments for the story are to be displayed in a tree format (The comments on a story can be viewed as a tree with the story being the root, each comment being a node of the tree, and comments which are replies to a particular comment, determining the children-parent relationships in the tree). In the original code, to achieve this task, a tree traversal is done, and at each tree node, a new query is issued to fetch the children comments. The issued query does a selection on the `parent` and the `story` attribute. Applying this transformation, all the comments on the story can be obtained using a single query: the issued query simply does a selection on the `story` attribute, as illustrated in Figure 5.5. We found this pattern only in the BBOARD application where it existed because the original code more closely reflected how the data is actually presented to the user.

5.1.3 Algorithm for Automating the MERGING Transformation

In this section we present an algorithm for automating the MERGING transformation, which should be straightforward to implement in a source-to-source compiler [39, 81]. As with any compiler transformation, the algorithm can bail out if it does not completely understand the program. Additionally, for ease of exposition, we assume that this transformation modifies a program of the application code only up

to the first update statement. The algorithm works by identifying the three code patterns: loop-to-join, merge-projection-predicates, and merge-selection-predicates, and then making appropriate changes in each case.

Loop-to-join pattern: For the loop-to-join pattern, we build on the work done in optimizing nested queries over 25 years ago [64]. We first identify loops in the program. We then check if: (1) the loop iterates using the result of a previous query, (2) the loop issues a query in each iteration, and (3) the previous query is executed whenever the loop executes. Moreover, to avoid issuing speculative queries, we check if the loop is executed whenever the previous query is executed.

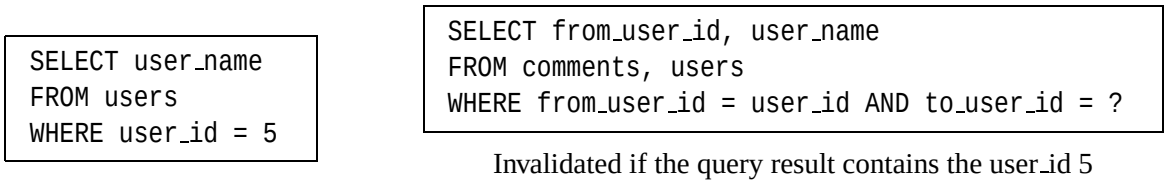
Once the pattern is identified, we can use work done by [64] to replace the small queries by a merged query. Additionally, variables that used the result of any of the small queries must be reinitialized to use the result of the merged query.

Merge-projection-predicates pattern: For the merge-projection-predicates, we check if (1) the second query is executed whenever the first query is executed (using control-flow-analysis [77]), and (2) the queries are identical except the projection predicates. If these pre-conditions are satisfied, the two queries can be merged as in Figure 5.4. Finally, variables that used the result of any of the queries being merged must be reinitialized to use the result of the merged query.

Merge-selection-predicates pattern: For the merge-selection-predicates, we check if: (1) the outer query is executed whenever the loop is executed and vice versa (using control-flow-analysis), and (2) the queries are identical, except one selection clause. If these pre-conditions are met, the query is transformed as per the example in Figure 5.5. Finally, variables that used the result of any of the queries being merged must be reinitialized to use the result of the merged query.

5.1.4 Other Tradeoffs

There are other advantages and disadvantages of applying the MERGING transformation, beyond just reducing latencies.



(a) Invalidations in the original code (b) Invalidations after applying the MERGING transformation

Figure 5.6: Query results that are invalidated on an update with template as `UPDATE users SET user_name = ? WHERE user_id = ?` and `user_id` as 5, before and after applying the MERGING transformation. Since the MERGING transformation increases caching granularity, it leads to more invalidations, and consequently, less reuse of work.

Interactions with query result caching For our DBSS setting, in which the query results are cached, the MERGING transformation, which merges short related queries into a long query, increases the caching granularity. Increasing the caching granularity implies that on an invalidation, a larger cache entry, which is more expensive to compute, is invalidated. For example, in Figure 5.6, rather than invalidating the result of a simple lookup query, in the transformed code, the update `UPDATE users SET user_name = ? WHERE user_id = 5` invalidates the result of the join query, a query result which is more expensive to compute.

Because of the possibility of increased invalidation overhead, the latency reduction due to this transformation must be weighed carefully against the increased invalidation, before applying this transformation in a setting that caches query results. For our benchmark applications, the increase in invalidations was minimal, and so we always decided to apply this transformation.

Impact on privacy In Chapter 3 and Chapter 4, we discussed how applications can ensure the privacy of their data in a DBSS setting. The MERGING transformation, by making query results larger, lowers the number of distinct query results in an application's workload. On the one hand, if only query results, parameters, or templates can be clues (as in Chapter 3), the MERGING transformation lowers the number of distinct privacy levels at which the application can operate. On the other hand, if arbitrary clues are possible (as in Chapter 4), the MERGING transformation has no effect on the number of distinct privacy levels at which the application can operate.

5.2 The NONBLOCKING Transformation: Prefetching Query Results

After issuing a database query, a Web application waits for the query result. In many cases, this wait is unnecessary since the the next database query does not depend on the answer to the current query. In such cases, the user latency can be greatly reduced by overlapping the query executions. In this section we present the NONBLOCKING transformation, which can overlap executions of multiple queries that do not depend on each other by "prefetching" query results.

To illustrate how this transformation can be applied to a code fragment, consider Figure 5.7, which shows two functionally-equivalent code fragments from the BOOKSTORE application. The program on the right shows the code after applying the NONBLOCKING transformation. For *query*2, we execute both the methods: *execute_non_blocking* and *execute*. While the method *execute_non_blocking* does not block and only serves to populate the cache with the query result, the *execute* method fetches the query result to be used in the program. If the latency of the first database request is t_a and the latency of the second request is t_b, this transformation reduces the overall latency from $t_a + t_b$ to $\max\{t_a, t_b\}$.

Ideally, whenever the program that dynamically generates the HTTP response starts running, we would like to issue prefetch requests for all queries that the program will issue during its execution. However, issuing a prefetch request for each query, at the start of the program's execution, might not always be possible because: (1) one of the parameters of the query is the result of a previous query, (2) the query is conditionally issued and the condition uses the result of a previous query, and (3) there is an update statement before the query that may affect the query result.

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. We call this graph the *database dependence graph* of a program. Given this directed acyclic graph, a database access can be issued as soon as all database accesses that are its ancestors in the directed acyclic graph have completed. With this formulation, the latency that a user sees can be brought down significantly.

Note that this transformation normally does not change the amount of work that must be done, it just

```
$template1:=SELECT item_name
            FROM items i1, items i2
            WHERE i1.id = i2.related
              AND i2.id = ?;
$query1:=set_params ($template1, $id);
$result1:=execute ($query1);
$template2:=SELECT user_name FROM users
            WHERE user_id = ?;
$query2:=set_params ($template1, $user_id);
$result2 := execute ($query2);
```

(a) Original code

```
$template2:=SELECT user_name FROM users
            WHERE user_id = ?;
$query2:=set_params ($template2, $user_id);
execute_non_blocking ($query2);
$template1:=SELECT item_name
            FROM items i1, items i2
            WHERE i1.id = i2.related
              AND i2.id = ?;
$query1:=set_params ($template1, $id);
$result1:=execute ($query1);
$result2:=execute ($query2);
```

(b) After the NONBLOCKING transformation

Figure 5.7: A simplified code fragment from the BOOKSTORE application, which finds the name of an item related to the item the user is viewing and the name of the user, given her id. 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 code after applying the NONBLOCKING transformation.

improves the scheduling of the work. However, if a prefetch is issued for a query that is conditionally executed, the result of the prefetch will not always be used. While issuing such "speculative" prefetches increases the total work in the system, it allows a scalability service to trade off reduced latency for extra work done in the system. For our evaluation, we issued speculative prefetches whenever possible, since the queries were *not* issued only in case of error conditions – an infrequent occurrence for any application.

Application of this transformation can be automated – we outline an algorithm for automatically applying this transformation in Section 5.2.1. Finally, in Section 5.2.2 we discuss other issues relating to this transformation.

5.2.1 Algorithm for Automating the NONBLOCKING Transformation

In this section we present an algorithm for automating the NONBLOCKING transformation, which should be straight-forward to implement in a source-to-source compiler [39, 81]. As with any compiler transformation, the algorithm can bail out if it does not completely understand the program. For ease of exposition, we make two assumptions. First, similar to the assumption in Section 5.1.3, we assume that the algorithm modifies a program of the application code only up to the first update statement. The work on query-update-independence [68] can be used to remove this restriction. Second, we assume that there are no edges in the database dependence graph of the program, as defined before. The dependence graph for a program can be computed using data-flow techniques [77]. After allowing for the assumptions, the algorithm is:

1. Let Q be the list of all the queries in the program that appear before any database modification. The goal is to place a *non-blocking-execute* function call to every query q appearing in the list Q at the beginning of the program.

2. For every query q, put a copy of all variable initializations that query q uses directly or indirectly (through some other variable) at the beginning of the program. Next, put a *non-blocking-execute* function call after all these variable initializations. Since the database dependence graph has no edges, the order in which the queries are selected from the list Q does not matter.

5.2.2 Implementation Issues

We now describe three issues regarding the implementation of the prefetch mechanism that we evaluate later in Section 5.3.5.

Prefetching support in the runtime layer For this transformation to work, the runtime layer must support the execution of non-blocking queries. In our implementation, as an admission control mechanism to avoid overloading the database, the DBSS node maintains a fixed number of connections to the home server database. The runtime layer must decide on the number of connections to

allocate to fulfilling prefetch requests. For our implementation, we dynamically allocated the connections used for fulfilling prefetch requests, depending on how many were available after fulfilling the regular database requests. Of course, if the object that the prefetch was requesting was already present in the cache, we just filtered the prefetch.

Timing of prefetches For hiding latencies due to a miss, it is critical that the prefetches be issued at the right time. On the one hand, if the prefetch is issued late, it will not be able to hide the latency due to the miss. On the other hand, if it is issued well in advance, its result might be invalidated by a later update and it becomes useless. From our experiments, we found out that none of our prefetches were "early" even if we issued them at the earliest possible time – when the program generating the HTTP response started its execution. So we just used this policy in our implementation.

5.3 Evaluation

We evaluate the two transformations–MERGING and NONBLOCKING–by applying them to the three benchmark applications (i.e., AUCTION, BBOARD, and BOOKSTORE) described earlier in Section 2.6 and then measuring the resulting scalability improvements. Note that we measure scalability as the number of simultaneous users that can be supported with latency remaining under a threshold. We performed our experiments in the SIMPLE_TC scenario using the methodology described in Section 2.7.

We start in Section 5.3.1 and Section 5.3.2 by evaluating the effects of these transformations on scalability and latency, both in the traditional centralized setting as well as the DBSS setting. Next, we list the frequencies with which the two transformations apply to our benchmark applications in Section 5.3.3. We finally present the detailed "coverage" results of the two transformations in Section 5.3.4 and Section 5.3.5.

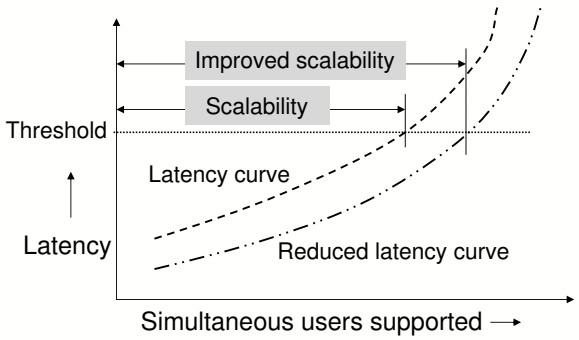


Figure 5.8: The figure shows how a reduction in latency improves scalability.

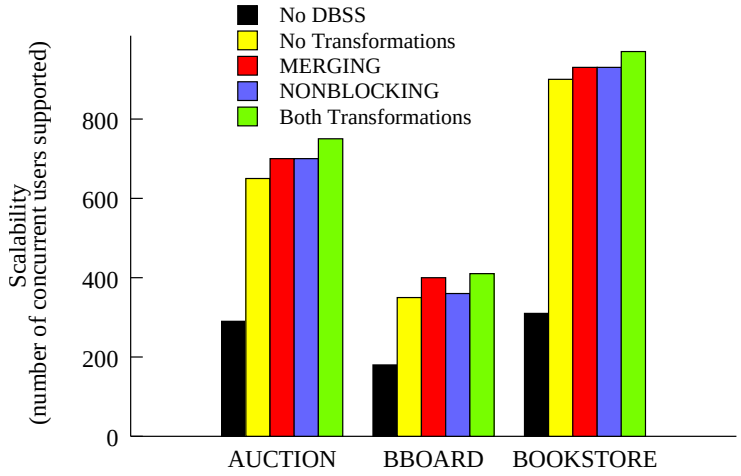


Figure 5.9: Scalability impact of the transformations. For comparison, we include the scalability numbers without a DBSS, the leftmost bar for each application.

5.3.1 Scalability Impact of the Transformations

So far, we have focused on how the MERGING and NONBLOCKING transformations reduce the latency of an HTTP request in the DBSS setting. However, we use scalability as the single unifying metric in this thesis. We measure scalability as the number of simultaneous users that can be supported with latency remaining under a threshold. Figure 5.8 shows how a reduction in latency improves the scalability metric. Because of the reduced latency, the scalability in the figure increases from "scalability" to "improved scalability."

Figure 5.9 plots the scalability of an application as a function of the code transformations used, for all three benchmark applications. The y-axis plots scalability, measured as specified in Section 2.7. On the x-axis, we consider five cases: one corresponding to not using the DBSS, one corresponding to using the DBSS but no transformations, and the other three corresponding to using either or both the transformations.

For clarity, we did not plot the bar for using the transformations in the centralized setting. The results were identical to not using the transformation in the centralized setting, showing that these transformations do not have any effect on the performance in a centralized setting. In all applications, using a DBSS significantly increased scalability. Turning on the transformations further improved scalability. The MERGING transformation has the most effect on the BBOARD application and the least effect on the BOOKSTORE application. On the other hand, the NONBLOCKING transformation has the most effect on the BOOKSTORE benchmark and the least effect on the BBOARD benchmark.

The two transformations: MERGING and NONBLOCKING, are complementary. While the MERGING transformation can be applied only when the queries are *related*, the NONBLOCKING transformation can be applied only when the queries are *not related*. Consequently, we expect that both transformations must be applied for the best scalability. Figure 5.9 shows that the scalability indeed increases the most when both transformations are applied simultaneously.

5.3.2 Latency Impact of the Transformations

Even though a single unifying metric like 'scalability' is helpful in comparisons, it is not always able to correctly portray the magnitude of a change. The scalability improvements due to these transformations, at around 10%, seem minor. To understand the results better, we plot the average latencies for two popular interactions in the BBOARD application. (We chose BBOARD because the latency effects of the transformations on BBOARD is the highest.)

Figure 5.10 shows the effect of the transformations on the average latency, for two dynamic interactions of the BBOARD application, executing in a DBSS setting. Applying both transformations reduces latency by over 50%. Of the two transformations, the MERGING transformation causes a greater reduc-

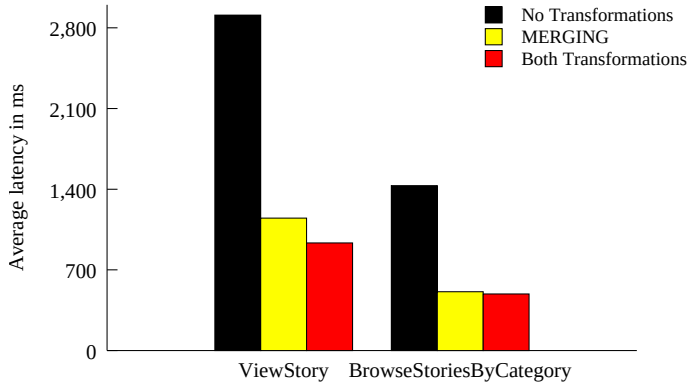


Figure 5.10: Impact of the MERGING and NONBLOCKING transformations on latency. We show the average latency for two dynamic interactions in the BBOARD benchmark. The graph shows that the MERGING transformation has a significant impact on the average latency.

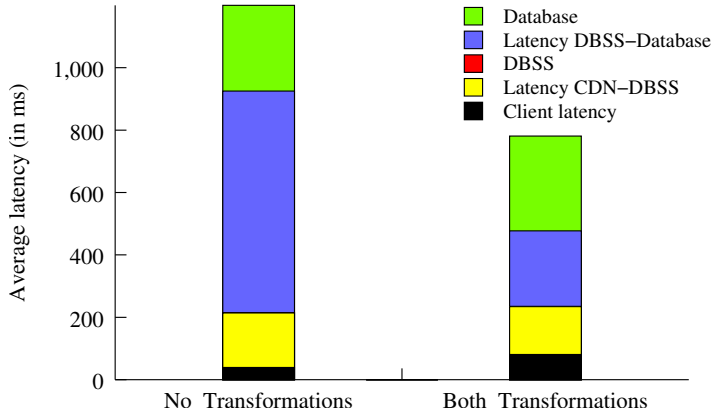


Figure 5.11: Impact of the two transformations on the average latency of a dynamic interaction in the BBOARD application, executing in a DBSS setting.

tion in latency. The larger impact of the MERGING transformation agrees with the scalability results in Figure 5.9. Figure 5.10 also shows that both transformations are complementary: applying both reduces the latency more than applying either of them.

Figure 5.11 evaluates the impact of the two transformations on the average latency of a dynamic interaction in the BBOARD application, executing in a DBSS setting. The latency consists of five components: the client latency including the execution time at the CDN, the network latency from the CDN to the DBSS, the time spent at the DBSS, the latency from the DBSS to the database, and the time spent at the

Application	Percentage of runtime HTTP interactions			
	Static	MERGING transformation	NONBLOCKING transformation	Either
AUCTION	15.9%	15.2%	3.8%	15.7%
BBOARD	7.4%	69.8%	28.5%	70.1%
BOOKSTORE	0.0%	0.8%	58.6%	59.4%

Table 5.1: Runtime HTTP interactions in which the MERGING and NONBLOCKING transformation apply. The "either" column represents interactions in which at least one of the two transformations apply. The "static" column represents interactions in which a static HTML file is returned. Clearly, neither transformation can apply to such interactions.

database. almost all the latency decrease is due to a reduction in the network latency from the CDN to the DBSS.

A latency decrease often does not result in a commensurate scalability increase. For example, while these two transformations reduce the average latency by 38% for the BBOARD application (Figure 5.11), they increase the scalability by only about 10% (Figure 5.9). To understand this difference, we need to refer back to Figure 5.8. In the figure, the latency decrease and the scalability increase are related by the slope of the latency-users curve. This slope governs how much the scalability will increase due to a decrease in latency. Steeper the curve, more is the slope, and less is the impact on scalability due to any reduction in latency. There is another factor that contributes to why a latency decrease does not result in a commensurate scalability increase. These transformations sometimes tend to impact low-latency interactions more than high-latency interactions. For the ViewStory interaction of the BBOARD application, while the latency reduction for low-latency interactions[1] was 78%, the latency reduction for the high-latency interactions was only 58%.

Application	Total query templates	Percentage of query templates where the patterns apply		
		Loop-to-join	*Merge-projection-predicates*	*Merge-selection-predicates*
AUCTION	28	25.0%	3.6%	0.0%
BBOARD	38	26.3%	13.2%	5.3%
BOOKSTORE	28	7.1%	0.0%	0.0%

Table 5.2: Frequency of occurrence of different patterns in which the `MERGING` transformation applies.

5.3.3 Applicability of the Transformations

Table 5.1 lists the percentage of runtime HTTP interactions in which these transformations apply. The "either" column represents interactions in which at least one of the two transformations apply. The "static" column represents interactions in which a static HTML page is returned. Clearly, neither transformation can apply to such interactions. Even after including the static interactions (interactions which return a HTML file), one of these transformations applied to over 15%, 70%, and 59% of all runtime HTTP interactions for the AUCTION, BBOARD, and the BOOKSTORE benchmarks, respectively. For the BBOARD application, the MERGING transformation applies to over 69% of all HTTP interactions – this high percentage is one of the reasons why the MERGING transformation is particularly effective in increasing scalability (Figure 5.9) and reducing latency (Figure 5.10) of the BBOARD application. A similar argument can be made for the NONBLOCKING transformation and the BOOKSTORE application.

5.3.4 Coverage of the MERGING Transformation

Table 5.2 lists the total number of query templates per benchmark application and the number of times we could find the different "patterns" described in Section 5.1.2. We found the maximum number of patterns in the BBOARD benchmark where the MERGING transformation could be applied to almost half of the query templates. In contrast, the BOOKSTORE benchmark had the fewest opportunities for

[1] All interactions that had a latency below the threshold were categorized as low-latency interactions. The latencies were measured after the two transformations were applied.

Application	*Cache hit ratio*	Average number of database queries per dynamic HTTP interaction		*% decrease*
		original code	*after the* MERGING *transformation*	
AUCTION	57.4%	2.6	2.1	19%
BBOARD	75.5%	9.1	1.9	79%
BOOKSTORE	66.4%	1.78	1.77	1%

Table 5.3: Average number of database queries per dynamic HTTP interaction for the three benchmarks. For our benchmark applications, the MERGING transformation does not affect the cache hit ratio.

applying this transformation. As for the patterns, the most frequently occurring pattern was loop-to-join, while the most uncommon pattern was merge-selection-predicates.

Table 5.3 lists the average number of database queries per dynamic HTTP interaction for all three benchmark applications both before and after applying the MERGING transformation, and computes the percentage decrease due to the transformation. The maximum decrease, 79%, occurs for the BBOARD benchmark and the minimum decrease, 1%, occurs for the BOOKSTORE benchmark. These results are in line with the results in Table 5.2 where the MERGING transformation applies most to the query templates of BBOARD benchmark, and least to the query templates of the BOOKSTORE benchmark. The table also provides the cache hit rates for the benchmark applications. From the cache hit rates, the average number of round trips from the DBSS node to the back-end database node that the MERGING transformation saves can be easily computed – 0.21, 1.76, and 0.003 round-trips are saved for the AUCTION, BBOARD, and BOOKSTORE benchmarks. These huge savings for the BBOARD application is reflected in Figure 5.11 – the 'latency DBSS-Database' decreases by almost 400ms.

5.3.5 Coverage of the NONBLOCKING Transformation

Figure 5.12 plots how effective the `NONBLOCKING` transformation is in hiding the cache misses. The y-axis plots the misses and prefetches for each of the benchmarks (The original misses have been nor-

Application	Original misses normalized to 100%			Filtered Pfs	Wasted Pfs
	Still miss	Partial Pfs	Useful Pfs		
AUCTION	87.9	11.0	1.1	7.4	0.1
BBOARD	94.7	1.7	3.6	8.2	0.4
BOOKSTORE	88.8	6.2	5.0	68.4	2.2

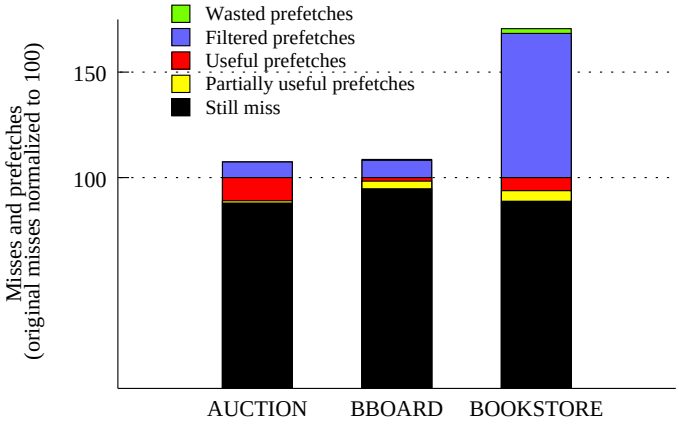


Figure 5.12: Impact of the NONBLOCKING transformation on the total number of misses, for the three benchmark applications. We use 'pfs' as a short-hand for prefetches.

malized to 100). After prefetches are issued, these misses either still remain a miss (*still miss*), meaning no prefetch was issued for it, or the prefetches were completely (*useful prefetches*) or partially useful (*partially useful prefetches*) in hiding the latency of a miss. Some prefetches were filtered by the caching layer because the object was present in the cache (*Filtered prefetches*). The final category was of those prefetches that were issued speculatively, and never used (*wasted prefetches*). This category of prefetches wastes bandwidth and CPU cycles. As the figure shows, the fraction of such prefetches is fairly small, which justifies our decision for issuing speculative prefetches in Section 5.2.

From Table 5.1 we expect many prefetches to be issued for the BOOKSTORE application. Figure 5.12 shows that while this is true, most of the prefetches turn out to be useless since they are filtered by the caching layer. Still, this transformation is able to hide latency for around 10% of the misses. For the AUCTION application, even though fewer prefetches are issued, it is still able to hide latency for around 10% of the misses. For the BBOARD application, fewest prefetches are issued, and the transformation is able to hide latency for only about 4% of the misses. The scalability improvement due to this transformation depends to some degree on the percentage of misses that the prefetches are able to hide: whereas we see a small impact of this transformation on the scalability of the AUCTION and the BOOKSTORE applications, the impact is almost zero on the scalability of the BBOARD application (Figure 5.9).

5.4 Related Work

Related research can be classified into two main areas: (1) prior work related to the NONBLOCKING transformation, and (2) prior work related to the MERGING transformation. We discuss each in turn.

5.4.1 Work Related to the NONBLOCKING Transformation

The NONBLOCKING transformation aims to hide the latency of a miss in the DBSS cache by prefetching the query result. A lot of prior work has been done on prefetching. A commonly used technique is to issue prefetches by detecting patterns in misses. This technique has been used widely for hiding latency of page faults in virtual memory systems [35], reducing access times of static web pages [40, 80, 86], and improving the overall performance of file systems [65]. Of course, for this technique to work, a pattern must be established. No prefetches can be issued while the patterns are being established. Second, this technique no longer remains useful when the access pattern changes.

Patterson et al. [88] propose an alternative approach to detecting patterns in misses. Applications must be manually modified to generate hints about their access patterns. Follow-up work by Chang et al. [28] automates this process of generating the access hints. Similarly, Mowry et al. [76] and Brown et al. [21], show how compiler analysis integrated with simple OS support and a runtime layer, to adjust to dynamic conditions, can be used to effectively manage physical memory for out-of-core applications. The compiler analysis was used to insert prefetch and release instructions in the application code. While the goal of using application-specific knowledge to hide latencies is the same in these efforts as in our system, we focus on a different domain than virtual memory references and file reads and writes. Their work did not require analysis of SQL code embedded in a program.

5.4.2 Work Related to the MERGING Transformation

The work closest to our MERGING transformation is Cassyopia [91], a vision paper that proposes the use of compiler techniques for clustering system calls so that the overhead of crossing address spaces

is reduced. Our technique is fundamentally similar to Cassyopia: we want to reduce the latency due to multiple database queries by clustering these database queries. However, there are significant differences. First, the domains are different – our work seeks to hide the overhead of network latency whereas their work seeks to hide the overhead of context switching between processes. Second, we identify important patterns where the MERGING transformation can be applied. Third we argue why such patterns will continue to exist in future Web applications.

Most database vendors support stored procedures [100] which allow applications to invoke a block of procedural and declarative code at the database. Our approach of merging queries has several key advantages over using stored procedures. First, it is significantly harder for the database to optimize the execution of queries that use stored procedures than to optimize the execution of SQL queries [29, 30]. Second, it is significantly harder to maintain the consistency of a cache containing results of stored procedures. (If the results of stored procedures are not cached, no work is offloaded from the home server database.)

Work on optimizing the execution of nested queries [47] has mostly focused on decorrelation techniques [45, 64], which try to transform a given nested query into a form that does not use the nested subquery construct. Guravannavar et al. [53] propose improved nested iteration methods as an alternative to decorrelation. Decorrelation techniques enable the query optimizer to use better plans such as hash join for evaluating the nested query. The MERGING transformation for the loop-to-join pattern essentially performs decorrelation. Compared to nested query optimization, the differences are: (1) the transformation is carried out by the compiler instead of the database optimizer, and (2) the primary motivation for the transformation is to reduce the number of round-trips an application needs to make to access its data instead of improving the performance.

Some database optimizers implement multi-query optimization [94, 98], where they identify common sub-expressions in a sequence of queries to speed up all the queries. However, to be applicable, these optimizers need to see a batch of queries at once – a model different from how the Web applications normally work, where they have at most one outstanding query. Moreover, in a multi-query optimization setting, the database does most of the work, while for the MERGING transformation, the compiler does

most of the work – it needs to understand database queries as well as the procedural code surrounding them, identify patterns in the application code, and then transform the patterns accordingly.

Object relational mapping tools like Java's Hibernate [41] and Ruby-on-Rails's Active Records [85] result in Web application code that issue several simple, inter-related queries, all of which can be merged into a single query. Both Hibernate and Active Records provide hooks to replace these inter-related queries by a single query – Hibernate users can write queries in HQL, the Hibernate Query Language, where as Active Record users can write explicit SQL queries. Our work seeks to automate this process of merging inter-related queries.

The MERGING transformation can be viewed as a repartitioning of work between the application and database server. Yang et al. [108] present techniques to automatically partition a Web application into client and server parts, in order to optimize the application's response time. To be applicable, the Web application must be written in a custom language Hilda [109]. Similarly, the Abacus system [10] automates the placement of objects written in a custom language. In contrast, the MERGING transformation does not require applications to be rewritten in a custom language; it can be applied directly to legacy applications.

5.5 Summary

A single HTTP request in a dynamic Web application typically issues multiple database queries. In a DBSS setting, database queries that miss in the cache of the DBSS node, have to endure the high latency of accessing the home server database. In this chapter we proposed two holistic transformations – MERGING and NONBLOCKING – which can be implemented in a source-to-source compiler [39, 81]. These transformations reduce the latency by either clustering related queries or overlapping query execution. By manually inspecting our application code, we found opportunities to apply these transformations in over 15%, 49%, and 74% of all dynamic interactions for the AUCTION, BBOARD, and the BOOKSTORE, respectively. These transformations had almost no impact on the scalability in a centralized setting. However, in a DBSS setting, these transformations increase scalability by over 10%.

These transformations are useful in any setting where the latency of accessing the query results from the machine executing the application is non-negligible. For example, in a shared web-service hosting scenario where the application and the database server typically run on separate clusters of machines, latencies are often between 16ms and 20ms [99], and therefore these transformations will be useful.

We believe that these two transformations will continue to be useful. First, as users of Web applications become more demanding, these applications will increasingly be deployed in environments where there is a significant latency in accessing their data from their code. Second, as Web applications become functionally more complex, they will issue more database requests per HTTP requests. Finally, just like queries, we believe that even updates could be coalesced to reduce the number of round-trips an application has to make to access its data.

Chapter 6

Conclusions

The world is gradually moving to a service oriented architecture, as is evident by the increasing popularity of Web services offered by `google.com` and traditional software services offered by `salesforce.com`. The growing popularity of CDNs is another facet of the same trend. CDNs allow Web applications to outsource the delivery of content at an economical pay-per-usage model, freeing application administrators from the difficult task of creating and maintaining infrastructure for delivering content. However, CDN technology is not sufficient for dynamic Web applications where the database is the bottleneck. In this thesis we proposed a DBSS which can offload the work from an application's database server(s). The DBSS forms the key component of a scalability service, which can provide the same benefit as CDNs to dynamic Web applications. We addressed two key issues in designing a DBSS: (a) the privacy concerns in caching applications' data, and (b) the performance concerns due to the high latency applications face in accessing their data in a DBSS setting.

In assuaging applications' privacy concerns, we discovered that there is an important privacy-scalability tradeoff in the scalability service setting. We studied this tradeoff both formally and empirically. We provided two solutions for managing this tradeoff: (i) An algorithm for identifying data that can be kept private without any scalability penalty, (ii) A more general approach called invalidation clues for fine-grained control over this privacy-scalability tradeoff. We verified the effectiveness of both our solutions by executing three benchmark applications on our prototype scalability service system.

To address the performance concerns arising due to the high latency applications face in accessing their data in a DBSS setting, we proposed two transformations that reduce the number of times applications must access their data. We confirmed that these transformations apply widely and are indeed effective in reducing the number of times applications must access their data. We verified the effectiveness of our transformations by executing the three benchmark applications in a traditional centralized setting as well as our prototype scalability service system.

6.1 Contributions

This thesis made the following contributions:

- We designed, built, and evaluated the *first* prototype of a DBSS, which comprised the centerpiece of our scalability service architecture for dynamic applications. We used our prototype scalability service to scale three benchmark applications (an AUCTION, a BBOARD, and a BOOKSTORE benchmark) – for each application, we increased the scalability by a factor of at least 2. To identify the bottlenecks in the system, we also presented a breakdown of the latency a user experiences in a centralized setting as well as a DBSS setting.

- We presented a convenient shortcut to managing the security-scalability tradeoff that appears in the DBSS setting. Our solution is to (statically) determine which data can be encrypted without any impact on scalability. We confirmed the effectiveness of our static analysis method, by applying it to three benchmark applications. In all three cases, our static analysis identified significant portions of the data that could be secured without impacting scalability. Moreover, a large part of this identified data was "moderately sensitive," which application administrators would want to encrypt, if they knew that doing so did not have a scalability penalty. The security-scalability tradeoff did not need to be considered for the data that was identified by our static analysis, significantly lightening the burden on the application administrator managing the tradeoff.

- We presented *invalidations clues*, a general framework that enabled applications to reveal little data to the DBSS, yet provide sufficient information to limit unnecessary invalidations of results

cached at the DBSS. Compared with previous approaches, invalidation clues provided increased scalability to the DBSS for a target security/privacy level, as well as more fine-grained control of this tradeoff. Using three realistic Web benchmark applications, we illustrated the issues and solutions for generating effective clues, e.g., by identifying categories requiring database clues, and then evaluated the solutions on our DBSS prototype.

- We described two complementary compiler-driven holistic transformations that lowered the total delay an application code had to endure to access its data, on an HTTP request. We presented algorithms for automating these transformations in a source-to-source compiler [39, 81]. Using an AUCTION, a BBOARD, and a BOOKSTORE benchmark, we confirmed that these transformations: (i) applied to around 25%, 75%, and 50% of the runtime interactions for the AUCTION, BBOARD, and the BOOKSTORE benchmark applications, and (ii) indeed reduced the user latency in a DBSS setting. Our results showed that applying both transformation simultaneously improves scalability the most – the scalability improved by over 10% for each application benchmark.

6.2 Future Work

DBSSs and scalability services for dynamic content applications are natural extensions of CDNs. In this thesis we introduced scalability services and addressed two key problems that applications using a scalability service would encounter. Our system is not ready for real world deployment yet. A large-scale evaluation of the DBSS is needed to understand the performance of the scalability service in the real world. There is another promising avenue for future work. As we saw in Chapter 5, multiple round-trips over the wide-area network can result in significantly higher user latency. A compiler-analysis based tool coupled with a runtime component that can automatically place code and data together in the scalability service infrastructure will be extremely useful in this framework.

Appendix A

Proofs for Chapter 3

A.1 Proofs for Section 3.4.4

Lemma 4. *Let the update template be an insertion, and the query template be a SPJ query having conjunctive selection predicates, with equality as the join operator, augmented by an optional order-by construct. For the update/query template pair, whenever a minimal statement-inspection strategy (MSIS) evaluates to invalidate (denoted I), a minimal view-inspection strategy (MVIS) also evaluates to I, i.e.,* $(U_i^T \in I) \wedge (Q_j^T \in \mathcal{E} \cap \mathcal{N}) \Rightarrow C_{ij} = B_{ij}$.

Proof. See Appendix A.2. □

Lemma 5. *If the update template is a deletion, and the query template is result-unhelpful with respect to the update template, then for the update/query template pair, whenever a minimal statement-inspection strategy (MSIS) evaluates to invalidate (denoted I), a minimal view-inspection strategy (MVIS) also invalidates to I, i.e.,* $(U_i^T \in \mathcal{D}) \wedge (\langle U_i^T, Q_j^T \rangle \in \mathcal{H}) \Rightarrow C_{ij} = B_{ij}$.

Proof. If the query template is result-unhelpful with respect to the update template, then by definition, no attribute used in the selection conditions of the update is preserved by the query, i.e., $\langle U_i^T, Q_j^T \rangle \in \mathcal{H} \Rightarrow S(U_i^T) \cap P(Q_j^T) = \{\}$. Therefore there is no information in the query result that can aid in reducing invalidations. Hence C_{ij} equals B_{ij}. □

Lemma 6. *If the update template is a modification and either the update template is ignorable with respect to the query template, or, the query template is result-unhelpful with respect to the update template, then for the update/query template pair, whenever a minimal statement-inspection strategy (MSIS) evaluates to invalidate (denoted I), a minimal view-inspection strategy (MVIS) also evaluates to I, i.e.,* $(U_i^T \in \mathcal{M}) \wedge (\langle U_i^T, Q_j^T \rangle \in \mathcal{G} \cup \mathcal{H}) \Rightarrow C_{ij} = B_{ij}$.

Proof. Lemma 6 can be proved in two parts:

Part 1 ($\langle \mathbf{U_i^T}, \mathbf{Q_j^T} \rangle \in \mathcal{G} \Rightarrow \mathbf{C_{ij} = B_{ij}}$): If the update template is ignorable with respect to the query template, then Lemma 1 states that A_{ij} equals 0, i.e., $\langle U_i^T, Q_j^T \rangle \in \mathcal{G} \Rightarrow A_{ij} = 0$. Further, property 3 (Section 3.2.3) implies that if $A_{ij} = 0$, then the equality $A_{ij} = B_{ij} = C_{ij} = 0$ holds. Hence C_{ij} equals B_{ij}.

Part 2 ($\langle \mathbf{U_i^T}, \mathbf{Q_j^T} \rangle \in \mathcal{H} \Rightarrow \mathbf{C_{ij} = B_{ij}}$): If the query template is result-unhelpful with respect to the update template, then by definition, no attribute used in the selection conditions of the update is preserved by the query, i.e., $\langle U_i^T, Q_j^T \rangle \in \mathcal{H} \Rightarrow S(U_i^T) \cap P(Q_j^T) = \{\}$. Therefore there is no information in the query result that can aid in reducing invalidations. Hence C_{ij} equals B_{ij}. □

A.2 Proof of Lemma 4

In this section we prove Lemma 4. To keep the proof simple, we restrict the query language so that no tuple of the result uses more than one tuple from any single base relation. (This restriction, for example, rules out self-joins.) Our proof can, however, be extended so that this assumption is not needed. We start by providing background on evaluation of a SPJ query in Appendix A.2.1. Then, in Appendix A.2.2, we describe additional database operations we use in our proof. In Appendix A.2.3, we discuss under what conditions the result of a query changes because of an insertion. Finally, we formulate intermediate results as lemmas and prove Lemma 4 in Appendix A.2.4.

A.2.1 Evaluation of a query

Let there be n relations $R_1, \ldots, R_n$ over which query Q is defined. Any query Q that meets our assumptions can be evaluated in the following four steps:

1. Evaluate R_{CP} as the Cartesian Product of $R_1, \ldots, R_n$, i.e., $R_{CP} = R_1 \times \ldots \times R_n$.
2. Keep tuples of the Cartesian Product that satisfy all selection predicates.
3. Order the tuples according to the order-by construct, if present.
4. Prune the attributes of the tuple, according to the projection operation. (Note that duplicates are not eliminated because of the multi-set semantics.)

Recall that for any database D, we use $Q[D]$ to denote the result of evaluating Q over D. Let a tuple t in the Cartesian Product R_{CP} be a cross product of tuples $t_1, \ldots, t_n$ belonging to relations $R_1, \ldots, R_n$, respectively, i.e., $t_{CP} = t_1 \times \ldots \times t_n$, where $t_{CP} \in R_{CP} \wedge t_i \in R_i, \forall 1 \leq i \leq n$. If t_{CP} satisfies the selection predicates of query Q, then some tuple t', same as t_{CP} but perhaps with fewer attributes, is present in $Q[D]$. Now, consider a database D' with the same schema as D, and only one tuple t_i in each relation R_i. Then, $Q[D'] = \{t'\}$. In fact, for any tuple t' in $Q[D]$, a database D' with the same schema as D, but only one tuple per relation can be constructed so that $Q[D'] = \{t'\}$. We call such a database D' as a *Single-Tuple-Per-Relation (STPR) database* and denote the set of such databases for a database/query pair as $\mathcal{D}_S(D, Q)$.

We next introduce database operations permitted in our framework.

A.2.2 Additional Database Operations

We define the following three additional database operations:

1. *Subset relation for databases (denoted $\subseteq$)*: For given database instances D_1 and D_2, D_1 is a *subset* of D_2 (denoted $D_1 \subseteq D_2$) if D_1 has the same schema as D_2 and each relation in D_1 is a subset of corresponding relation in D_2.

2. *Union function for databases (denoted $\cup$)*: For given database instances D_1 and D_2 with the same schema, the union of D_1 and D_2 is a database where each relation in $D_1 \cup D_2$ is the set union of the corresponding relations in D_1 and D_2.

3. *Minimize function for databases* (denoted m): For a given database D and query Q, the output, which we call min-Database and denote $m(D,Q)$, of the minimize function is database that satisfies the following two properties: a) The result of evaluating the query on the database is the same, irrespective of whether the evaluation is done before or after applying the minimize function , i.e., $Q[D] = Q[m(D,Q)]$, and b) The query when evaluated on any subset of the min-Database that is not the min-Database itself, yields a result other than $Q[m(D,Q)]$, i.e., $\forall D_1 \subseteq m(D,Q)\ (m(D,Q) \subseteq D_1 \vee Q[D_1] \neq Q[m(D,Q)])$. To evaluate a minimize function, we use the following result, which we state without proof: A tuple t is present in a relation R of the min-Database if and only if the tuple is present in relation R of any of the STPR databases corresponding to the database and query. Using our union function, $m(D,Q) = \cup_{D' \in \mathcal{D}_S(D,Q)} D'$.

A.2.3 Does the result of a query change on an insertion?

We start by defining two terms: local selection predicates and join-attributes, which are relevant for the discussion of whether query result $Q[D]$ changes on insertion U or not. Assume that insertion U adds a tuple t to relation R_i.

Local selection predicates of a query Q with respect to an insertion U are selection predicates of the query that do not involve attributes of any relations other than R_i. For example, "toy_name = ?" would be a local selection predicate of Q_1^T for any insertion to the `toys` relation of the TOYSTORE application (Table 3.3).

Join-attributes of a query Q with respect to an insertion U are attributes of relation R_i that occur in any selection predicate that also involves attributes of any relation other than R_i. For example, attribute *cid* is a join attribute of query Q_3^T with respect to insertion U_2^T of the TOYSTORE application.

Insertion U affects the result of query Q if and only if:

1. The inserted tuple t satisfies the local selection predicates.

2. Tuple t joins with other tuples of the database. Whether or not the tuple can join with other tuples depends only on the tuple's values for the join attribute.

Next for an insertion/query pair, we define a special class of databases for which the insertion changes the result of a query.

Complementary database: A complementary database for an insertion U, query Q pair, denoted D_C, is a database which becomes a STPR database after update U is applied to it, i.e., $D_C + U \in \mathcal{D}_S(D_C + U, Q)$. It is easy to see that a complementary database exists only if the inserted tuple satisfies the local selection predicates.

A.2.4 Intermediate Lemmas and Proofs

Lemma 7. *For any given database D, insertion U, and query Q, the results of evaluating the query before and after applying the insertion to the database are different, if and only if a complementary database D_C corresponding to the insertion/query pair is a subset of the database, i.e.,* $Q[D] \neq Q[D+U] \Leftrightarrow \exists D_C\,(D_C + U \in \mathcal{D}_S(D_C + U, Q))$.

Proof. **Proof of the "if" part**: Let tuple t_{CP} represent the Cartesian Product of tuples of the complementary database with insertion U. Tuple t_{CP} satisfies all selection predicates of the query and so, some tuple t', same as t_{CP} but perhaps with fewer attributes, is present in the result of the query evaluated after the insertion. Further, since duplicates are not eliminated when projection is applied, the result of the query evaluated over the database containing the inserted tuple has one tuple more than the result of the query evaluated over the database not containing the inserted tuple.

Proof of the "only if" part by construction: If the result of query Q changes because of insertion U, then because of the evaluation process in Appendix A.2.1, there cannot be fewer tuples in the result after the insertion. In fact, $Q[D+U]$ will have one tuple more than $Q[D]$, which means the set $\mathcal{D}_S(D+U, Q)$ will have one more member than the set $\mathcal{D}_S(D, Q)$. Let insertion U inserts tuple t in relation R_i. Database

D_C can be constructed by removing t from the database that is present in $\mathcal{D}_S(D+U,Q)$, but not in $\mathcal{D}_S(D,Q)$. It can be verified that D_C is indeed a complementary database. □

Lemma 8. *Assume there exists a complementary database D_C for an insertion/query pair. Then for any database, either the results of evaluating the query on the minimal database before and after applying insertion U are different, or the results of evaluating the query on the min-Database before and after the union with the complementary database are same, i.e.,* $(Q[m(D,Q)] \neq Q[m(D,Q)+U]) \vee (Q[m(D,Q)] = Q[m(D,Q) \cup D_C])$.

Proof. If the results of evaluating the query on the minimal database before and after applying insertion U are different, then the proof is complete. Otherwise, the result of evaluating the query on the minimal database before and after applying insertion U are the same, i.e.,

$$Q[m(D,Q)] = Q[m(D,Q)+U] \tag{A.1}$$

From Appendix A.2.3, we know that for logic expression (A.1) to hold, either the inserted tuple fails to a) satisfy the local selection predicates, or b) join with other tuples. Because of the assumption in Lemma 8 about existence of complementary database, the inserted tuple must fail to join with other tuples for logic expression (A.1) to hold.

Assume the insertion adds tuple t to relation R_i. Also assume v is the value of the inserted tuple's join attributes. Since t does not join with other tuples, and min-Database has no tuples that do not contribute to the result, there can be no tuple in R_i with value of join attributes equal to v. So when union of complementary database D_C is taken with $m(D,Q)$, the result of the query does not change, i.e., $Q[m(D,Q)] = Q[m(D,Q) \cup D_C]$. □

We are now ready to prove Lemma 4.

Proof. **Proof of Lemma 4 by contradiction.** Assume to the contrary that there exists query Q, an instance of Q_i^T, insertion U, an instance of U_i^T, current database instance D_P (before the application of update U), and a current view V_p so that insertion U causes a minimal statement-inspection strategy

(MSIS) $\mathcal{S}_1$ to invalidate the cached result of the query but does not cause a minimal view-inspection strategy (MVIS) $\mathcal{S}_2$ to invalidate the cached result of the query.

By definition of a MSIS (Section 3.2.2), it follows that there exists database instance D such that applying insertion U changes the result of evaluating query Q on the database instance, i.e.,

$$\exists D\,(Q[D] \neq Q[D+U]) \tag{A.2}$$

Applying Lemma 7 to logic expression (A.2) implies that there exists complementary database D_C corresponding to the insertion/query pair.

Further, from the definition of a MVIS, it follows that on any database D, if query Q evaluates to V_p, then applying update U to the database does not affect the result of evaluating query Q on the database, i.e.,

$$\neg\exists D\,((Q[D] = V_p) \wedge (Q[D] \neq Q[D+U])) \tag{A.3}$$

$$\textit{or},\quad \forall D\,((Q[D] = V_p) \Rightarrow (Q[D] = Q[D+U])) \tag{A.4}$$

We now construct a database D' that does not obey logic expression (A.4), i.e., $Q[D'] = V_p \wedge Q[D'] \neq Q[D'+U]$.

Recall that database D_C is a complementary database corresponding to the U/Q pair and database D_P is the current database instance before application of the update. We claim $D' = m(D_p, Q) \cup D_C$. Mathematically, $Q[D_P] = V_p$. By definition of a minimize function (Appendix A.2.2), we know that the result of evaluating the query on the minimal database $m(D_P, Q)$ is also V_p, i.e., $Q[m(D_P, Q)] = V_p$. Further, because a complementary database exists for the insertion/query pair and logic expression (A.4) applies, the second condition of Lemma 8 holds, and the result of evaluating the query on the minimal database both before and after the union with a complementary database D_C remains the same, i.e.,

$$Q[m(D_P, Q)] = Q[m(D_P, Q) \cup D_C] \tag{A.5}$$

Using Lemma 7 in conjunction with logic expression (A.5) yields that the result of the query changes on applying the insertion to the database $m(D_P, Q) \cup D_C$, i.e., $Q[m(D_P, Q) \cup D_C] \neq Q[m(D_P, Q) \cup D_C + U]$.

Moreover, since $Q[m(D_P, Q)] = V_p$, logic expression (A.5) implies $Q[m(D_P, Q) \cup D_C] = V_p$. So $Q[D'] = V_p \wedge Q[D'] \neq Q[D' + U]$ for $D' = m(D_P, Q) \cup D_C$. Hence contradiction. □

Bibliography

[1] AdventNet Inc. Zoho Creator. `http://creator.zoho.com`.

[2] G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-Molina, K. Kenthapadi, R. Motwani, U. Srivastava, D. Thomas, and Y. Xu. Two can keep a secret: A distributed architecture for secure database services. In *Proc. CIDR*, 2005.

[3] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In *Proc. SIGMOD*, 2004.

[4] Rakesh Agrawal, Ramakrishnan Srikant, and Dilys Thomas. Privacy preserving OLAP. In *Proc. SIGMOD*, 2005.

[5] Fuat Akal, Can Türker, Hans-Jörg Schek, Yuri Breitbart, Torsten Grabs, and Lourens Veen. Fine-grained replication and scheduling with freshness and correctness guarantees. In *Proc. VLDB*, 2005.

[6] M. Altinel, C. Bornhvd, S. Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald. Cache tables: Paving the way for an adaptive database cache. In *Proc. VLDB*, 2003.

[7] Amazon Web Services. `http://aws.amazon.com`.

[8] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A dynamic data cache for Web applications. In *Proc. ICDE*, 2003.

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

[10] Khalil Amiri, David Petrou, Gregory R. Ganger, and Garth A. Gibson. Dynamic function placement for data-intensive cluster computing. In *USENIX Annual Technical Conference*, 2000.

[11] C. Amza, A. Cox, and W. Zwaenepoel. Conflict-aware scheduling for dynamic content applications. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[12] Cristiana Amza, Gokul Soundararajan, and Emmanuel Cecchet. Transparent caching with strong consistency in dynamic content web sites. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, New York, NY, USA, 2005.

[13] Martin Arlitt, Ludmilla Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evaluating content management techniques for Web proxy caches. In *Proceedings of the Workshop on Internet Server Performance (WISP99)*, 1999.

[14] Roberto J. Bayardo and Rakesh Agrawal. Data privacy through optimal k-anonymization. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, Washington, DC, USA, 2005.

[15] Philip A. Bernstein, Alan Fekete, Hongfei Guo, Raghu Ramakrishnan, and Pradeep Tamma. Relaxed-currency serializability for middle-tier caching and replication. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 2006.

[16] Philip A. Bernstein and Nathan Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database Systems*, 9(4), 1984.

[17] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[18] José A. Blakeley, Neil Coburn, and P. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems*, 14(3), 1989.

[19] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7), 1970.

[20] Daniel Brodie, Amrish Gupta, and Weisong Shi. Accelerating dynamic web content delivery using keyword-based fragment detection. *Journal of Web Engineering*, 2005.

[21] Angela Demke Brown and Todd C. Mowry. Taming the memory hogs: using compiler-inserted releases to manage physical memory intelligently. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, Berkeley, CA, USA, 2000.

[22] Erik Brynjolfsson, Yu (Jeffrey) Hu, and Michael D. Smith. Consumer surplus in the digital economy: Estimating the value of increased product variety at online booksellers. *Management Science*, 49(11), 2003. http://www.heinz.cmu.edu/~mds/cs.pdf.

[23] CacheFly. http://www.cachefly.com.

[24] California Senate. Bill SB 1386. http://info.sen.ca.gov/pub/01-02/bill/sen/sb_1351-1400/sb_13 2002.

[25] K. Candan, D. Agrawal, W. Li, O. Po, and W. Hsiung. View invalidation for dynamic content caching in multitiered architectures. In *Proc. VLDB*, 2002.

[26] Michael J. Carey, Michael J. Franklin, Miron Livny, and Eugene J. Shekita. Data caching tradeoffs in client-server dbms architectures. In *Proc. SIGMOD*, 1991.

[27] Jim Challenger, Arun Iyengar, Karen Witting, Cameron Ferstat, and Paul Reed. A publishing system for efficiently creating dynamic web content. In *Proc. INFOCOM*, 2000.

[28] Fay Chang and Garth A. Gibson. Automatic i/o hint generation through speculative execution. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, Berkeley, CA, USA, 1999.

[29] Surajit Chaudhuri. An overview of query optimization in relational systems. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, New York, NY, USA, 1998.

[30] Surajit Chaudhuri and Kyuseok Shim. Optimization of queries with user-defined predicates. *ACM Transactions on Database Systems*, 24(2), 1999.

[31] B. Chen and R. Morris. Certifying program execution with secure processors. In *USENIX HotOS Workshop*, 2003.

[32] Chun Yi Choi and Qiong Luo. Template-based runtime invalidation for database-generated web contents. In *Asia Pacific Web Conference*, 2004.

[33] Coghead Inc. http://www.coghead.com.

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

[35] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. Practical prefetching via data compression. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, 1993.

[36] Anindya Datta, Kaushik Dutta, Helen Thomas, Debra VanderMeer, Suresha, and Krithi Ramamritham. Proxy-based acceleration of dynamically generated content on the world wide web: an approach and implementation. In *Proc. SIGMOD*, New York, NY, USA, 2002.

[37] David J. DeWitt, Philippe Futtersack, David Maier, and Fernando Velez. A study of three alternative workstation-server architectures for object oriented database systems. In *Proc. VLDB*, 1990.

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

[39] Laurie J. Hendren et al. Soot: a java optimization framework. `http://www.sable.mcgill.ca/soot/`.

[40] Li Fan, Pei Cao, Wei Lin, and Quinn Jacobson. Web prefetching between low-bandwidth clients and proxies: potential and performance. In *SIGMETRICS '99: Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 1999.

[41] Hibernate: Relational Persistence for Java and .NET. `http://www.hibernate.org`.

[42] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Symposium on Operating Systems Principles*, 1997.

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

[44] Michael J. Franklin, Michael J. Carey, and Miron Livny. Global memory management in client-server database architectures. In *Proc. VLDB*, San Francisco, CA, USA, 1992.

[45] Richard A. Ganski and Harry K. T. Wong. Optimization of nested sql queries revisited. *SIGMOD Record*, 16(3), 1987.

[46] Charlie Garrod, Amit Manjhi, Anastassia Ailamaki, Phillip B. Gibbons, Bruce M. Maggs, Todd C. Mowry, Christopher Olston, and Anthony Tomasic. Scalable consistency management for web database caches. Technical report, Carnegie Mellon University, 2006, `http://www.cs.cmu.edu/~manjhi/scalableConsistency.pdf`.

[47] G. Graefe. Executing nested queries. In *Conference on Database Systems for Business, Technology and the Web*, 2003.

[48] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. SIGMOD*, Montreal, Canada, June 1996.

[49] Hongfei Guo, Per-Ake Larson, and Raghu Ramakrishnan. Caching with "good enough" currency, consistency, and completeness. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, 2005.

[50] Hongfei Guo, Per-Ake Larson, Raghu Ramakrishnan, and Jonathan Goldstein. Relaxed currency and consistency: how to say "good enough" in SQL. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 2004.

[51] Ashish Gupta and Jose A. Blakeley. Using partial information to update materialized views. *Information Systems*, 20(9), 1995.

[52] Ashish Gupta and Iderpal Singh Mumick, editors. *Materialized views: techniques, implementations, and applications*. MIT Press, Cambridge, MA, USA, 1999.

[53] Ravindra Guravannavar, H. S. Ramanujam, and S. Sudarshan. Optimizing nested queries with parameter sort orders. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, 2005.

[54] Hakan Hacigumus, Bala Iyer, Chen Li, and Sharad Mehrotra. Executing SQL over encrypted data in the database service provider model. In *Proc. SIGMOD*, 2002.

[55] Hakan Hacigumus, Bala Iyer, and Sharad Mehrotra. Providing database as a service. In *Proc. ICDE*, 2002.

[56] Hakan Hacigumus, Bala Iyer, and Sharad Mehrotra. Efficient execution of aggregation queries over encrypted relational databases. In *9th International Conference on Database Systems for Advanced Applications*, 2004.

[57] Bijit Hore, Sharad Mehrotra, and Gene Tsudik. A privacy-preserving index for range queries. In *Proc. VLDB*, 2004.

[58] Akamai Technologies Inc. Akamai and IBM unveil edge computing solution. `www.akamai.com/html/about/press/releases/2002/press_050802.html`.

[59] Akamai Technologies Inc. and Jupiter Research Inc. Akamai and Jupiter Research identify '4 seconds' as the new threshold of acceptability for retail web page response times. http://www.akamai.com/html/about/press/releases/2006/press_110606.html.

[60] Akamai Technologies Inc. and Quocirca. Akamai and quocirca identify '4 second' performance threshold for european web-based enterprise applications. http://www.edgejava.net/html/about/press/releases/2007/press_110707.html.

[61] Time Inc. Who's really participating in web 2.0. http://www.time.com/time/business/article/0,8599,1

[62] Jakarta Project. Apache Tomcat.

[63] Murat Kantarcioglu and Chris Clifton. Security issues in querying encrypted data. Technical Report TR-04-013, Purdue University, 2004.

[64] Won Kim. On optimizing an sql-like nested query. *ACM Transactions on Database Systems*, 7(3), 1982.

[65] Thomas M. Kroeger and Darrell D. E. Long. Predicting file system actions from prior events. In *ATEC'96: Proceedings of the Annual Technical Conference on USENIX 1996 Annual Technical Conference*, Berkeley, CA, USA, 1996.

[66] Per-Ake Larson, Jonathan Goldstein, and Jingren Zhou. Mtcache: Transparent mid-tier database caching in sql server. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, Washington, DC, USA, 2004.

[67] Kristen LeFevre, David J. DeWitt, and Raghu Ramakrishnan. Incognito: efficient full-domain k-anonymity. In *Proc. SIGMOD*, New York, NY, USA, 2005.

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

[69] Ninghui Li, Tiancheng Li, and Suresh Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and ℓ-diversity. In *Proc. ICDE*, 2007.

[70] W. Li, O. Po, W. Hsiung, K. S. Candan, D. Agrawal, Y. Akca, and K. Taniguchi. CachePortal II: Acceleration of very large scale data center-hosted database-driven web applications. In *Proc. VLDB*, 2003.

[71] Limelight Networks. http://www.limelightnetworks.com.

[72] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton. Middle-tier database caching for e-business. In *Proc. SIGMOD*, 2002.

[73] Ashwin Machanavajjhala, Johannes Gehrke, Daniel Kifer, and Muthuramakrishnan Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. In *22nd IEEE International Conference on Data Engineering*, 2006.

[74] Amit Manjhi, Anastassia Ailamaki, Bruce M. Maggs, Todd C. Mowry, Christopher Olston, and Anthony Tomasic. Simultaneous scalability and security for data-intensive web applications. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 2006.

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

[76] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted i/o prefetching for out-of-core applications. In *OSDI '96: Proceedings of the second USENIX symposium on Operating systems design and implementation*, New York, NY, USA, 1996.

[77] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[78] L. Mummert and M. Satyanarayanan. Large granularity cache coherence for intermittent connectivity. In *USTC'94: Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference*, Berkeley, CA, USA, 1994.

[79] MySQL AB. MySQL database server.

[80] Alexandros Nanopoulos, Dimitrios Katsaros, and Yannis Manolopoulos. A data mining algorithm for generalized web prefetching. *IEEE Transactions on Knowledge and Data Engineering*, 15(5), 2003.

[81] ObjectWeb Consortium. ASM. `http://asm.objectweb.org`.

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

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

[84] Christopher Olston, Amit Manjhi, Charles Garrod, Anastassia Ailamaki, Bruce M. Maggs, and Todd C. Mowry. A scalability service for dynamic web applications. In *Proc. CIDR*, 2005.

[85] Ruby on Rails. `http://www.rubyonrails.org`.

[86] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using predictive prefetching to improve world wide web latency. *SIGCOMM Comput. Commun. Rev.*, 26(3), 1996.

[87] Panther Express. `http://www.pantherexpress.com`.

[88] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, New York, NY, USA, 1995.

[89] Christian Plattner and Gustavo Alonso. Ganymed: Scalable replication for transactional web applications. In *Middleware*, 2004.

[90] Dallan Quass, Ashish Gupta, Inderpal Singh Mumick, and Jennifer Widom. Making views self-maintainable for data warehousing. In *Proc. Fourth International Conference on Parallel and Distributed Information Systems*, 1996.

[91] Mohan Rajagopalan, Saumya K. Debray, Matti A. Hiltunen, and Richard D. Schlichting. Cassyopia: compiler assisted system optimization. In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, 2003.

[92] Lakshmish Ramaswamy, Arun Iyengar, Ling Liu, and Fred Douglis. Automatic detection of fragments in dynamically generated web pages. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, New York, NY, USA, 2004.

[93] Relationals, Inc. `http://www.longjump.com`.

[94] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. *SIGMOD Record*, 29(2), 2000.

[95] Mahadev Satyanarayanan. Fundamental challenges in mobile computing. In *Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, 1996.

[96] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, 1996.

[97] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: a cautionary tale. In *NSDI'06: Proceedings of the 3rd conference on 3rd Symposium on Networked Systems Design & Implementation*, Berkeley, CA, USA, 2006.

[98] Timos K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1), 1988.

[99] Simple measurements on the infrastructure of Dreamhost, a leading Web-hosting company. `http://www.dreamhost.com/`.

[100] Michael Stonebraker, Jeff Anton, and Eric Hanson. Extending a database system with procedures. *ACM Transactions on Database Systems*, 12(3), 1987.

[101] L. Sweeney. k-anonymity: A model for protecting privacy. *International journal of uncertainty, fuzziness, and knowledge-based systems*, 2002.

[102] Latanya Sweeney. Achieving k-anonymity privacy protection using generalization and suppression. *International journal of uncertainty, fuzziness, and knowledge-based systems*, 10(5), 2002.

[103] The Washington Post. Advertiser charged in massive database theft. `http://www.washingtonpost.com/wp-dyn/articles/A4364-2004Jul21.html`, July, 2004.

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

[105] Trusted Computing Group. Trusted Platform Module Main Specification, Version 1.2. `http://www.trustedcomputing.org`.

[106] Mark Tsimelzon, Bill Weihl, Joseph Chung, Dan Frantz, John Basso, Chirs Newton, Mark Hale, Larry Jacobs, Conleth O'Connell, and Mark Nottingham (editor). ESI Language Specification 1.0. `http://www.w3.org/TR/2001/NOTE-esi-lang-20010804`.

[107] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, 2002.

[108] Fan Yang, Nitin Gupta, Nicholas Gerner, Xin Qi, Alan Demers, Johannes Gehrke, and Jayavel Shanmugasundaram. A unified platform for data driven web applications with automatic client-server partitioning. In *WWW*, New York, NY, USA, 2007.

[109] Fan Yang, Jayavel Shanmugasundaram, Mirek Riedewald, and Johannes Gehrke. Hilda: A high-level language for data-driven web applications. In *Proc. ICDE*, 2006.