Invalidation Clues for Database Scalability Services[1]

This is an updated version of the technical report CMU-CS-06-139. It supersedes the previous technical report.

Amit Manjhi Phillip B.Gibbons Anastassia Ailamaki
Charles Garrod Bruce M. Maggs Todd C. Mowry
Christopher Olston Anthony Tomasic Haifeng Yu

December 2006
CMU-CS-06-139R

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

For their scalability needs, data-intensive Web applications can use a Database Scalability Service (DBSS), which caches applications' query results and answers queries on their behalf. One way for applications to address their security/privacy concerns when using a DBSS is to encrypt all data that passes through the DBSS. Doing so, however, causes the DBSS to invalidate large regions of its cache when data updates occur. To invalidate more precisely, the DBSS needs help in order to know which results to invalidate; such help inevitably reveals some properties about the data. In this paper, we present *invalidation clues*, a general technique that enables applications to reveal little data to the DBSS, yet limit the number of unnecessary invalidations. Compared with previous approaches, invalidation clues provide applications significantly improved tradeoffs between security/privacy and scalability. Our experiments using three Web application benchmarks, on a prototype DBSS we have built, confirm that invalidation clues are indeed a low-overhead, effective, and general technique for applications to balance their privacy and scalability needs.

[1]This research is supported in part by National Science Foundation grants, including NeTS CNS–0520192, CNF–0433540, and NeTS CNF–0435382.

Keywords: Scalability Service, Scalability, View Invalidation, Web Applications

1 Introduction

Internet applications 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"). Investing in a server farm that can accommodate such high loads is not only expensive (particularly after factoring in the management costs) but also risky because the expected customers might not show up. An appealing alternative is to contract with a *scalability service* that charges based on usage. Content Delivery Networks (CDNs) [14] provide such service by maintaining a large, shared infrastructure to absorb the load spikes that may occur for any individual application. However, CDNs currently do not provide a way to scale the *database* component of a Web application. Hence the CDN solution is not sufficient when the *database system is the bottleneck*, as in many e-commerce applications.

To overcome this key bottleneck, *Database Scalability Services (DBSS)* can be used to extend the scaling benefits provided by CDNs to the database component of Web applications [24, 28]. As in CDNs, a third party (*Database Scalability Service Provider*) provides such service by maintaining a large, shared infrastructure to offload work from and to absorb load spikes for any individual database. Figure 1 depicts the resulting architecture, in which (1) a Web application's code is executed at trusted hosts (application "servers"), (2) the code in turn fires off database updates/queries that are handled by a DBSS, and (3) any updates and queries that cannot be answered by the DBSS are sent to backend databases on the application vendor's "home" servers.

A key challenge in the design of a DBSS is providing this shared scalability infrastructure while protecting each organization's sensitive data. The goals are (1) to limit the DBSS administrator's ability to observe or infer an application's sensitive data, and (2) to limit an application's ability to use the DBSS to observe or infer another application's sensitive data. Such concerns have been increasing in the past few years, as borne by well-publicized instances of database theft [31]. From the viewpoint of the home organization, these are *security* concerns; from the viewpoint of an individual user whose personal data may be revealed, these are *privacy* concerns.

Security and 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*. 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 [12, 33]. This scenario is similar to the standard security scenario of two trusted parties communicating over an untrusted channel. We consider the ciphertext-only attack [30] and the chosen-plaintext attack [30] in this scenario—details are in Section 3.3. When a data update occurs, to maintain consistency, the DBSS must invalidate (at least) all the cached query results that changed. Because the results are encrypted, the DBSS needs help from the application in order to know which results to invalidate; such help inevitably reveals some properties about the data. 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 and privacy concerns.

Invalidation Clues. In this paper, 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 paper, 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

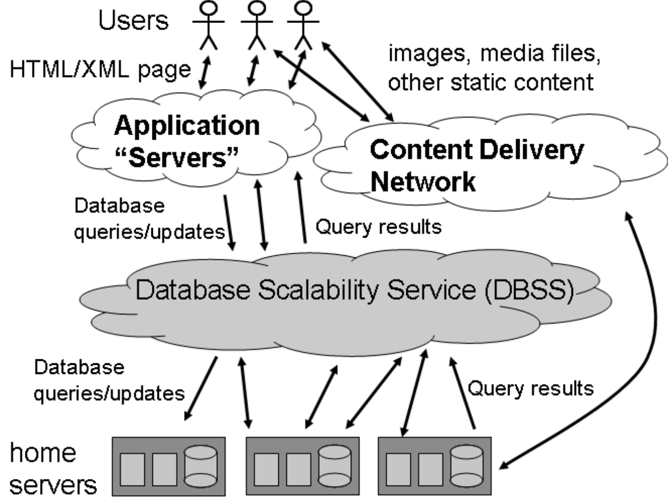

Figure 1: A scalable architecture for database-intensive Web applications.

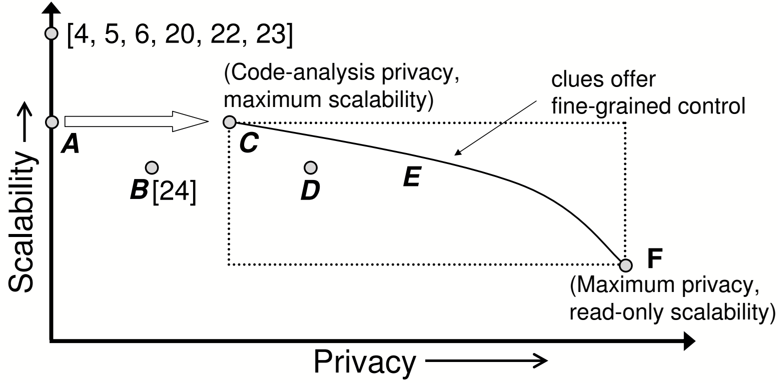

Figure 2: Privacy-Scalability tradeoff in the presence of clues. The dashed box shows the region in which an application can operate in our scheme. The six scenarios, *A*–*F*, are explained later in Table 2. Code-analysis privacy and read-only scalability are explained in Section 5.1.

rule out most unnecessary invalidations.
(2) *Limit revealed information:* Our clues enable the application to achieve a target security/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 [4, 5, 6, 20, 22, 23, 24, 28], invalidation clues provide applications significantly improved tradeoffs between security/privacy and scalability. This difference is demonstrated in Figure 2 (discussed in detail in Section 7), which compares prior work in database scaling technology to our scheme. Only our scheme enables the favorable tradeoffs inside the dashed box.

Our Contributions. The main contributions of this paper are as follows.

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

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

Table 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.

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

- We show how to keep application data secure/private under a more general attack model than previous work [24].
- We identify families of common query/update classes where extra information is needed from the database in order to perform precise invalidations. We show that generating these "database-derived" clues in response to an update typically requires accessing only one or two database rows. We present a strategy that uses such clues only when the scalability benefit from reduced invalidations outweighs the cost of computing the clue.
- Finally, using experiments with three Web application benchmarks—a bookstore (TPC-W), an auction (RUBiS), and a bulletin-board (RUBBoS)—running on our prototype DBSS, we demonstrate the scalability benefits of our proposed clues. We also use representative queries from these benchmarks to show the effectiveness of our configurable clues in providing an improved security/privacy versus scalability tradeoff.

Road Map. Section 2 provides an overview of invalidation clues. Section 3 and Section 4 show how different types of clues can be used to achieve different precisions in invalidations. Section 5 discusses how clues can be tailored to balance between privacy and scalability. Section 6 presents our empirical findings. Section 7 presents related work. Finally, Section 8 presents conclusions.

In the remainder of the paper, we will use *privacy* as a short hand for both security and privacy.

2 An Illustrative Example

This section introduces invalidation clues via an example. Consider the application SIMPLE-BBOARD, specified in Table 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 2 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 database scaling technology. Most of this work [4, 5, 6, 20, 22, 23] 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 [24] (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 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 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

	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 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.

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 2 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 [24] into the terminology of this paper. [24] 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 ids but not the result bodys), 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), 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 ids, 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**.

3 Using Clues for Invalidations

In this section we describe how clues can be used for invalidations. We begin in Section 3.1 by describing the architecture that is the context for our work. Section 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 paper. Section 3.3 describes the attack model of the DBSS. Then, in Section 3.4, we formalize the notion of precise invalidations. Finally, in Section 3.5 we present various types of clues and provide examples of when each type is useful.

[1] A Bloom-filter [9] 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.

3.1 Architecture

The overall system architecture is as depicted in Figure 1 (see Section 1). 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 encrypted updates are 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.

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.

3.2 Query and Update Model

Our query and update model is based on our study of three benchmark Web applications (details in Section 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 1, 4, 7, and 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 [8, 24, 29], 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.

3.3 The Attack Model of the DBSS

In this paper 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 [30]. 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 [30].

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

Table 3: A taxonomy of clues (The various clue types are in italics). Clues differ based on whether they are attached to query results or updates, and whether they are computed from parameters, result, or database.

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.

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 of clues may overlap. Table 3 summarizes the taxonomy of clues.

Consider the SIMPLE-AUCTION application shown in Table 4. For each of its query/update template pairs, Table 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 Database Clues

The previous section motivated the use of database clues using the SIMPLE-AUCTION example. In this section, we first identify (Section 4.1) families of common query/update classes where database clues are required for precise invalidation. Section 4.2 discusses the problems with achieving precise invalidations using *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: 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.

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 5: Types of clues required to implement a DIS for template-pairs of the SIMPLE-AUCTION example in Table 4.

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 4.3 presents practical techniques that further reduce overheads and/or increase privacy by relaxing the precise invalidation requirement.

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.

4.1.1 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 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

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 6: Notation for aspects of templates.

SIMPLE-BBOARD application (Table 1), $S(U^T) = \{$`comments.id`$\}$, $M(U^T) = \{$`comments.rating`$\}$, $S(Q^T) = \{$`comments.story`, `comments.rating`$\}$, and $P(Q^T) = \{$`comments.id`, `comments.body`.

4.1.2 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 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, 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` 11th `to` 21st `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=?
```

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

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=?
```

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 [29] could be used to identify the minimal such data. For example, for query template Q_3^T in Table 4, the techniques in [29] 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_id`s 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, 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_date`s 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 ⟨ query template, update template ⟩ pairs, only 21 require database clues (details are in Section 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

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 $\quad C\{$`a`$\} \leftarrow$ "value of `a` in the row being updated"
8 **return C**

Figure 3: Pseudo code for computing a database update clue when query templates are restricted to a single table.

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 3 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 3 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 how to do by hand.

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 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 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 1 and the corresponding result query clue and parameter update clue in Scenario **C** of Table 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 2, we can obscure these `id` values by using Bloom-filters [9], as discussed in Section 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.

5 Privacy-Scalability Tradeoffs

In this section we study privacy-scalability tradeoffs in the DBSS setting, considering the attack model of Section 3.3. We begin in Section 5.1 by showing that there is a fundamental tradeoff between privacy and scalability in our DBSS setting. Section 5.2 then presents an overview of how applications could get extra privacy by having the DBSS carry out unnecessary invalidations. Next, in Sections 5.3 and 5.4, we study representative query and update template pairs from our application benchmarks, and present configurable clues for these pairs. Finally in Section 5.5, we discuss how our current work applies to entire applications, beyond a single query and update template pair.

5.1 The Limit Cases

Recall the dashed box in Figure 2 from Section 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 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 [24]). 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 2), and if it achieves the maximum privacy, it gets read-only scalability (the lower right corner of the dashed box in Figure 2). 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 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,

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

Table 7: A query-update template pair from the BOOKSTORE benchmark.

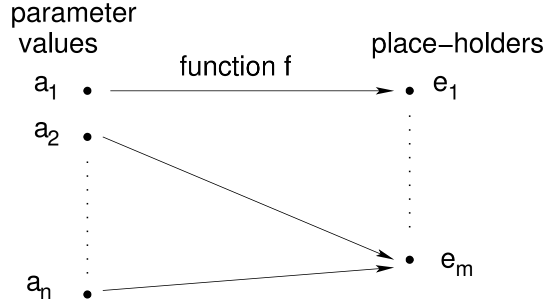

Figure 4: An example mapping of parameter values to place-holders.

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

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 5.3) or order comparisons (Section 5.4).

5.3 Equality Comparisons

Consider an actual template pair, shown in Table 7, from the BOOKSTORE benchmark (details in Section 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 by a bipartite graph as in Figure 4. 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 4.3.

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

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

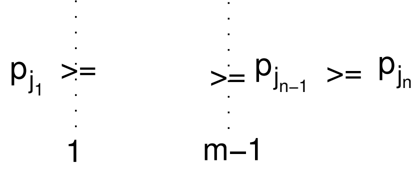

Figure 5: The solution implied by Lemma 1. $j_i \in \{1, \ldots, n\}$ **is such that the parameter value** a_{j_i} **is the** i**th most frequently occurring.**

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 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 1 provides the key insight required to find the EQUALITY-OPTIMAL mapping.

Lemma 1 *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 1 implies that the final solution has a form as shown in Figure 5, 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 1 is that the 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 $O(n^2 m)$ time.

In Section 6.4 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.

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

Table 8: A simplified query-update template pair from the AUCTION benchmark.

5.4 Order Comparisons

Consider the template pair shown in Table 8. This pair is from the AUCTION benchmark (details in Section 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) [2] 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 [2]). 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 2 *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 2 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 6.4 shows that for a given scalability value, this mapping enables twice the privacy of an OPES.

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`).

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 our benchmark applications (Section 6.1), our experimental methodology (Section 6.2), and our scalability results (Section 6.3). Finally, in Section 6.4 we measure the effectiveness of our techniques in helping an application manage its privacy-scalability tradeoff.

6.1 Benchmark Applications

We used three publicly available Web benchmark applications that extensively use a database and represent real-world applications: RUBiS [26], an auction system modeled after `ebay.com`, RUBBoS [27], a simple bulletin-board-like system inspired by `slashdot.org`, and TPC-W [32], a transactional e-Commerce application that captures the behavior of clients accessing an online book store. [4]. We used Java implementations of these applications. We will henceforth refer to these applications as AUCTION, BBOARD, and BOOKSTORE, respectively.

There were a few queries in these benchmarks (12 out of 94 templates) that did not conform to our query model (Section 3.2), e.g., aggregate queries. For these queries, we use parameter and result clues but not database clues.

Table 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 4.1. 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] 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. [10]. 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	*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 9: Number of template pairs in the three applications which require database clues for precise invalidations, classified as per the categories introduced in Section 4.1.

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 6: Application configuration parameters.

6.2 Experimental Methodology

We use the same methodology that we used in [24]. We report results for a simple two-node configuration—a home server that runs MySQL4 [25] as its database management system, and a DBSS node that provides answers to database queries using its store of the cached query results, running on Emulab [34]. (To keep the configuration simple, the DBSS node also provided the functionality of an application "server", i.e., the ability to run Web applications and to interact with a user running a Web browser. We used Tomcat [7] to provide both functionalities.) Cached query results were kept consistent with the home server's database using non-transactional invalidation of cached query results.

The home server machine had an Intel P-III 850 MHz processor with 512 MB of memory, while the DBSS node had an Intel 64-bit Xeon processor with 2GB of memory. In all experiments, the home server and DBSS node were connected by a high latency, low bandwidth duplex link (100 ms latency, 2 Mbps). Each client was connected to the DBSS node by a low latency, high bandwidth duplex link (5 ms latency, 20 Mbps).

Because the overhead for emulating clients is low, a single additional Emulab node was used to emulate all clients. As in the TPC-W [32] specification, clients simulate human usage patterns by issuing an HTTP request, waiting for the response, and pausing for a *think time* before requesting another Web page—the think time is drawn from a negative exponential distribution with a mean of seven seconds.

Figure 6 provides the configuration parameters we used in our experiments. Each experiment ran for ten minutes, and the DBSS node started with a cold cache each time. *Scalability* was measured as the maximum number of users that could be supported while keeping the response time below two seconds for 90% of the HTTP requests.

6.3 Scalability Benefits of Invalidation Clues

Figure 7 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 6.2. On the x-axis, we consider five cases: one corresponding to not using a DBSS, one corresponding to not using clues[5], 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[6], *Clues (incl. DB clues)*, which uses parameter, result,

[5]The scalability of this strategy is the same as the Minimal Template-Inspection Strategy (MTIS) of [24].

[6]The scalability of this strategy is the same as the Minimal View-Inspection Strategy (MVIS) of [24].

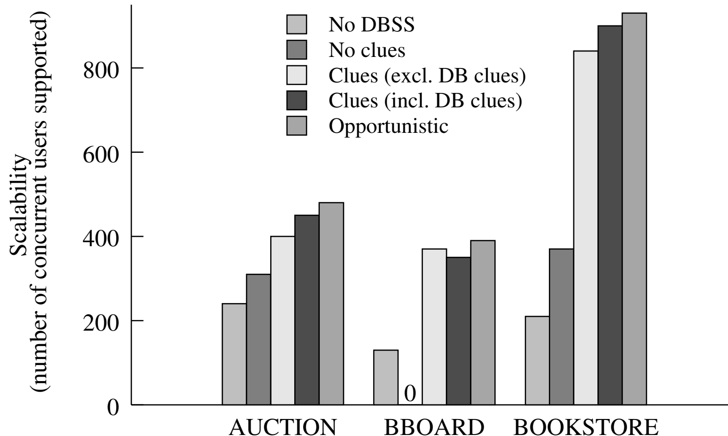

Figure 7: Impact of invalidation clues on scalability. For comparison, we include the scalability numbers without a DBSS.

and database update clues (as presented in Section 4.2), and *Opportunistic*, which uses the OPPORTUNISTIC strategy presented in Section 4.3.

In all applications, using a DBSS with invalidation clues significantly increased scalability. This agrees with previous work [24], 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 6.2. 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 7 shows, using database update clues whenever required for precise invalidations results in worse scalability. Figure 7 thus confirms the claim made in Section 4.3 that the use of database update clues must be carefully weighed against the expected benefit.

6.4 Privacy Experiments

Figure 8 shows the reduction in the number of invalidations. The workload used is the template pair in Table 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 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 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 9 plots the improvement in privacy due to using two mappings instead of one mapping, as described in Section 5.4. The workload used is the template pair in Table 8, with parameter 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

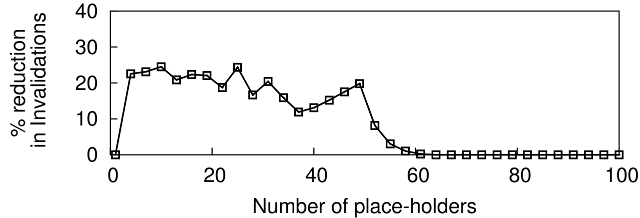

Figure 8: Reduction in invalidations due to our EQUALITY-OPTIMAL mapping algorithm.

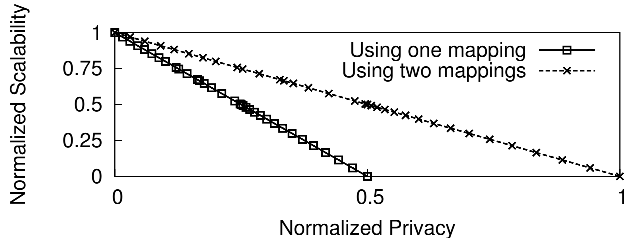

Figure 9: Improvement in privacy on using two mappings instead of one mapping.

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 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.

7 Related Work

We now discuss other related work in database services, view invalidation, and privacy.

Database Services. Existing work on providing *database services* can be classified into Database Outsourcing (DO) services [1, 16, 17, 18] and Database Scalability Services (DBSS) [4, 5, 6, 20, 22, 23, 24, 28]. With DO services, an application outsources all aspects of management of its database to a third party [17]. Guaranteeing privacy of applications' data is a key challenge in this setting [1, 16, 18]. With DBSS, only *database scalability* is outsourced to a third party: application providers retain master copies of their data on their own systems, with the DBSS caching and serving read-only copies on their behalf. This approach is more attractive from a privacy and data integrity standpoint than the DO approach, particularly for Web applications with read/write workloads (e.g., e-commerce applications). As discussed in Section 2, previous DBSS technology efforts [4, 5, 6, 20, 22, 23, 28] other than [24] did not address privacy concerns.

View Invalidation and Maintenance. Many papers have studied invalidation strategies for cached materialized views [11, 13, 21], but none of these study the privacy implications of using a particular invalidation strategy, the focus of our work. Likewise, many papers [15, 29] have studied techniques for view maintenance—how to change a view to reflect an update.

The view invalidation and view maintenance works cited above are special cases of clues. However, they do not address privacy concerns. Furthermore, we demonstrate the necessity and advantages of specially designed "database-derived" update clues, in order to achieve precise invalidations. The work closest to this in technique is by Candan et al. [11]. 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 just as a heuristic to get better invalidations. They neither implemented precise invalidations using polling queries, nor addressed privacy issues arising from the use of polling queries.

Privacy. There has been a lot of recent interest in keeping data private, yet allowing the computation of several functions on the data (e.g., [3]). Agrawal et al. [2] present order-preserving encryption schemes (OPES). As argued in Section 5.2, however, these schemes do not preserve privacy under our attack model. Hore et al. [19] study the privacy-utility tradeoff in the choice of the "coarseness" of the index on encrypted data. Our bucketization technique in Section 5 is similar. However, the resulting optimization problems are different because different privacy metrics apply.

8 Conclusion

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 paper 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 security/privacy level, as well as more fine-grained control of this tradeoff. Using three realistic Web application benchmarks, we illustrated the issues and solutions for generating effective clues, e.g., by identifying categories requiring database clues, and then we demonstrated the scalability benefits of these solutions on our DBSS prototype.

Acknowledgments. This research is supported in part by National Science Foundation grants, including NeTS CNS–0520192, CNF–0433540, and NeTS CNF–0435382.

References

[1] 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.

[2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *Proc. SIGMOD*, 2004.

[3] R. Agrawal, R. Srikant, and D. Thomas. Privacy preserving OLAP. In *SIGMOD Conference*, 2005.

[4] 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.

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

[6] C. Amza, G. Soundararajan, and E. Cecchet. Transparent caching with strong consistency in dynamic content web sites. In *International Conference on Supercomputing*, 2005.

[7] Apache Tomcat. `http://tomcat.apache.org`.

[8] J. A. Blakeley, N. Coburn, and P. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM TODS*, 14(3), 1989.

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

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

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

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

[13] C. Y. Choi and Q. Luo. Template-based runtime invalidation for database-generated web contents. In *APWeb*, 2004.

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

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

[16] H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database service provider model. In *Proc. SIGMOD*, 2002.

[17] H. Hacigumus, B. Iyer, and S. Mehrotra. Providing database as a service. In *Proc. ICDE*, 2002.

[18] H. Hacigumus, B. Iyer, and S. Mehrotra. Efficient execution of aggregation queries over encrypted relational databases. In *DASFAA*, 2004.

[19] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *VLDB*, 2004.

[20] P.-A. Larson, J. Goldstein, and J. Zhou. MTCache: Transparent mid-tier database caching in sql server. *Proc. ICDE*, 2004.

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

[22] 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.

[23] 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.

[24] A. Manjhi, C. Olston, A. Ailamaki, B. M. Maggs, T. C. Mowry, and A. Tomasic. Simultaneous scalability and security for data-intensive web applications. In *Proc. SIGMOD*, 2006.

[25] MySQL AB. MySQL database server.

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

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

[28] C. Olston, A. Manjhi, C. Garrod, A. Ailamaki, B. M. Maggs, and T. C. Mowry. A scalability service for dynamic web applications. In *Proc. CIDR*, 2005.

[29] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *PDIS*, 1996.

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

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

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

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

[34] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, 2002.