Simultaneous Scalability and Security for Data-Intensive Web Applications

Amit Manjhi Anastassia Ailamaki Bruce M. Maggs
Todd C. Mowry Christopher Olston Anthony Tomasic

March 2006
CMU-CS-06-116

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

For Web applications in which the database component is the bottleneck, scalability can be provided by a third-party *Database Scalability Service Provider* (DSSP) that caches application data and supplies query answers on behalf of the application. Cost-effective DSSPs will need to cache data from many applications, inevitably raising concerns about security. However, if all data passing through a DSSP is encrypted to enhance security, then data updates trigger invalidation of large regions of cache. Consequently, achieving good scalability becomes virtually impossible. There is a tradeoff between security and scalability, which requires careful consideration.

In this paper we study the security-scalability tradeoff, both formally and empirically. We begin by providing a method for statically identifying segments of the database that can be encrypted without impacting scalability. Experiments over a prototype DSSP system show the effectiveness of our static analysis method—for all three realistic benchmark applications that we study, our method enables a significant fraction of the database to be encrypted without impacting scalability. Moreover, 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. 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.

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

1 Introduction

Applications deployed on the Internet are immediately accessible to a vast population of potential users. As a result, they tend to experience fluctuating and unpredictable load, especially due to events such as breaking news (e.g., 9/11) and sudden popularity spikes (e.g., the "Slashdot Effect"). If the application uses a commodity server, the (peak) customer load on the application may easily exceed the capacity of the server. The approach of over-provisioning—to invest in a server farm, and to hire skilled personnel to maintain it—is not only expensive but also risky because the expected customers might not show up. An appealing alternative is to contract with a *scalability service* that charges on a per usage basis. Content Delivery Networks (CDNs) [9] provide such service by maintaining a large, shared infrastructure to absorb load spikes that may occur for any individual application. However, CDNs currently do not provide a way for scaling the database component of a Web application. Hence the CDN solution is not sufficient when the database system is the bottleneck, as in several important Web applications like bulletin-boards and e-commerce applications.

To support applications where the database is the bottleneck, previous work [22] has proposed using a third-party *Database Scalability Service Provider* (DSSP). A DSSP caches application data and supplies query answers on behalf of the application. To be cost-effective, DSSPs will need to cache data from "home servers" of many applications (Figure 1 shows the resulting architecture), inevitably raising concerns about security[1]. Such concerns have been increasing lately, as borne by well-publicized instances of database theft [24] and the security legislation in the California Senate [7].

To use untrusted DSSPs with confidence, applications must:

- **Prevent the DSSP from tampering with master data**. The DSSP caches read-only copies, which are kept consistent with master copies maintained at application home servers via *invalidation*. All updates are applied to master copies directly. (In many Web applications, updates are infrequent; so the load on home servers due to updates is low.)

- **Prevent unauthorized reading of data that passes through the DSSP**. A straightforward way to secure data passing through the DSSP is to encrypt such data. The DSSP then stores encrypted data. To permit answering of queries, one can use encryption schemes that permit query processing on encrypted data. However, recent work [15] has shown that only weak encryption can be used if queries are to be executed efficiently on the DSSP. Therefore, with this option, security of all data might be compromised. The only remaining alternative then is to store (encrypted) query results in the form of *materialized views* at the DSSP. Answering a query requires only a lookup operation, permitting arbitrarily strong encryption.

Figure 2 shows the flow of queries, updates, and invalidations in the architecture implied by the above discussion. In the figure, diagonal shading denotes information that is subject to encryption. The DSSP maintains a cache of (encrypted) query results. (Encrypted) queries are answered from the cache when possible; cache misses result in queries being forwarded to the home organization. All updates are routed to the home organization via the DSSP (in encrypted form). The DSSP monitors completed updates, and invalidates cached query results as needed to ensure consistency. (In our architecture, the home organization is free from the overhead of participating in invalidation decisions.)

1.1 Security-Scalability Tradeoff

In this work, we study the security-scalability tradeoff that arises when DSSPs are employed. To illustrate the presence of such a tradeoff, we introduce a simple example application called SIMPLE-TOYSTORE, specified in Table 1. We focus on the application's database access *templates*—queries or updates missing zero or more parameter values. Table 2 lists the invalidations the DSSP 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

[1]By security, we mean that (1) the DSSP administrator's are unable to access an application's sensitive data, and (2) applications are unable to access each other's sensitive data via the DSSP.

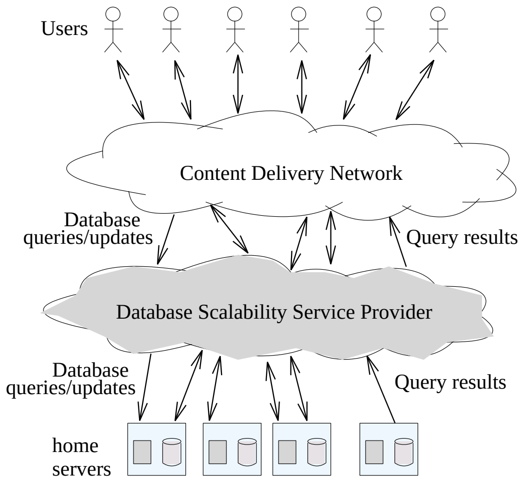


Figure 1: Scalable architecture for database-intensive Web applications. In this work, we focus on the Database Scalability Service Provider (DSSP), the shaded cloud.

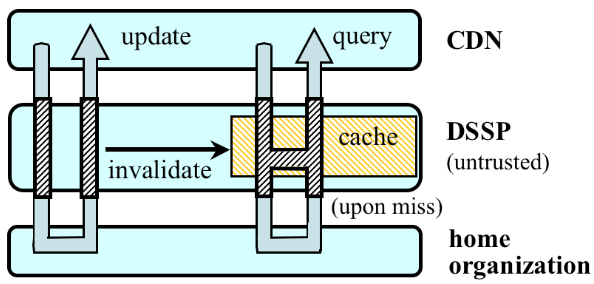


Figure 2: Query, update, and invalidation pathways.

the DSSP 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 DSSP increases (moving down the rows), the number of invalidations it needs to make decreases, thereby increasing scalability.

Encryption of queries, updates and data for security purposes limits the information available to the DSSP for making invalidation decisions. With limited information, the DSSP 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, which shows measurements of the TCP-W online bookstore benchmark executed on a prototype DSSP system we have built (details are provided in Section 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 DSSP. 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.

1.2 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

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 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?			Invalidation Condition
Templates	Parameters	Query Results	
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 2: Invalidations differ depending on the amount of information the DSSP can access. The table is for update U_1^T with parameter `5`.

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 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 paper 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, 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[2] without compromising scalability.

1.3 Related Work

Prior work related to ours can be partitioned into two categories: database services and view invalidation. We discuss each in turn.

1.3.1 Database Services

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

In the DO model, an application outsources all aspects of management of its database to a third party [12]. A key concern is to safeguard the application's sensitive data. Since the DO provider houses the application's

[2] See Section 5.4 for details on what data is kept private under our approach.

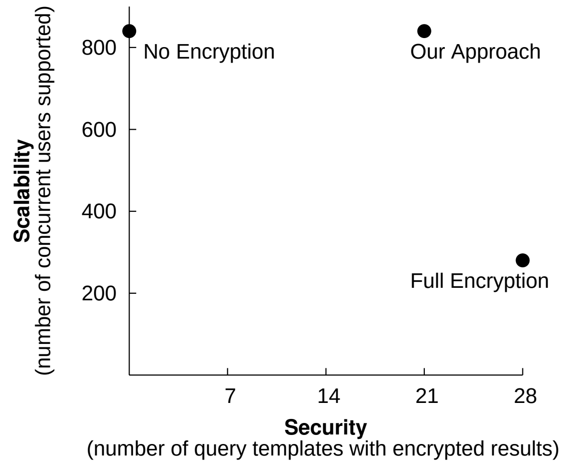


Figure 3: Security-scalability tradeoff (TPC-W BOOKSTORE benchmark).

entire database, one way to ensure security of an application's data is to store an encrypted database at the DO provider, and use encryption schemes that permit query processing on encrypted data [2, 11, 13]. Aggarwal et al. [1] suggest an alternative—distribute data across multiple independent providers that do not communicate with one another.

In contrast to work in the context of the DO model, we consider the DSSP model, in which only *database scalability*, and not full-fledged database management, is outsourced to a third party [22]. Under the DSSP model, application providers retain master copies of their data on their own systems, with the DSSP only caching and serving read-only copies on their behalf. In our DSSP approach, query execution on third party servers is not needed, so arbitrarily strong encryption of the remotely-cached data is possible. We contend that from a security and data integrity standpoint, the scalability provider model is more attractive than the DO model in the case of Web applications with read/write workloads (e.g., e-commerce applications).

Other ongoing efforts to create DSSP technology include the DBCache [3, 18] and DBProxy [4] projects at IBM Research, and the CachePortal project [17] at NEC Laboratories. To the best of our knowledge, no prior work has studied security issues in the DSSP model, which is the focus of this paper.

1.3.2 View Invalidation

There has been prior work pertaining to invalidation of cached materialized views. Choi and Luo [6] proposed a technique that leverages statically-available query/update templates to speed up runtime invalidation decisions. Candan et al. [8] introduced techniques for deciding whether to invalidate cached views in response to database updates. These techniques leverage "polling queries" to inspect portions of the database not available in the materialized view. The need to invalidate a view in response to a particular update can in some cases be ruled out by analysis of the view definition and update statements alone, without inspecting any other data. Levy and Sagiv [16] provide methods of ruling query statements (and hence view definitions) independent of updates in many practical cases, although the general query/update independence problem is undecidable.

With our work, the focus is not on developing new strategies for deciding whether to invalidate cached views. Rather, we develop a formal characterization of view invalidation strategies in terms of what data they access, as a basis for studying the tradeoff between data security and scalability.

1.4 Our Contributions

We develop a formal characterization of view invalidation strategies in terms of what data they access, and use the formal characterization to cleanly formulate the security-scalability management problem. We then present 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 DSSP system (detailed in Section 5) show 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.

1.5 Outline

To underpin our study of the security-scalability tradeoff, we begin in Section 2 by presenting our formal characterization of cache invalidation strategies, each of which represents a natural choice in the space of security-scalability options. Section 3 describes our methodology for management of the tradeoff, while Section 4 presents our main contribution: a static analysis method for determining which data can be encrypted without impacting scalability. In Section 5 we present our empirical findings, which point to the effectiveness of our technique. Plans for future work appear in Section 6. We summarize in Section 7.

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 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 paper. Then, in Section 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 2.3 we study the mixed invalidation strategies that arise when the information available for making invalidation decisions varies across queries and across updates.

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 1 shows an example). Let $\mathcal{Q}^T = \{Q_1^T, \ldots, Q_n^T\}$ 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 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 [5, 23], 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.

2.1.1 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.

2.2 Formal Characterization of View Invalidation Strategies

Recall that in our current design, the DSSP 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 2, may be defined as follows (The arguments to a strategy also list the information the strategy can access):

- **Blind Strategy[3] (BS) $\mathcal{S}()$:** No information is available to make the invalidation decision. Correctness requires that $(\exists U, Q, D : (Q[D] \neq Q[D + U])) \Rightarrow (\mathcal{S}() = I)$.

[3] In earlier work [22], the term *black-box strategy* was used to refer to the same concept.

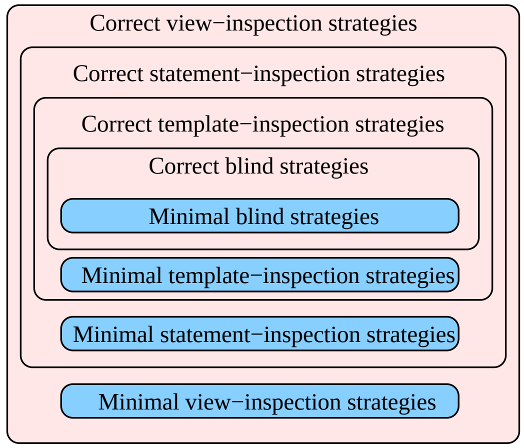


Figure 4: Relationships among classes of view invalidation strategies, in the general case.

- **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 [10] can be used to implement a view-inspection strategy. Similarly, the methods of [16] can be used to implement a template- or a statement-inspection strategy. Finally, implementing a blind strategy is simple: invalidate all cached query results on any update.

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

We now define minimality:

Minimality: A view invalidation strategy $\mathcal{S}$ 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 4 depicts the relationships among classes of view invalidation strategies as a Venn diagram.

The choice of invalidation strategy determines what information can be encrypted. On the one extreme, if a view-inspection strategy is used, neither queries, nor updates, nor cached query results can be encrypted.

Exposure levels: `blind` `template` `stmt` `view`

greater exposure (less encryption) $\longrightarrow$

$\longleftarrow$ greater security

Figure 5: Security gradient.

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

On the other extreme, if a blind strategy is used, all queries, updates, and cached query results can be encrypted.[4]

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* $E(U^T) \in \{$`blind`, `template`, `stmt`$\}$ for each update template $U^T \in \mathcal{U}^T$, and an exposure level $E(Q^T) \in \{$`blind`, `template`, `stmt`, `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 5 shows the range of exposure level options.

Figure 6 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 and query template, the invalidation strategy to use may be determined at the granularity of update/query template pairs. In Figure 6, the shaded boxes correspond to the four classes of invalidation strategies introduced in Section 2.2. (We discuss the unshaded boxes shortly.)

2.3.1 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 6, contains invalidation probability values for

[4] 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 DSSP 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.

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 2.1.1 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 6, 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 6 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 4. Before we approach this topic, we first describe our overall approach to managing the security-scalability tradeoff while meeting scalability requirements.

3 Overview of Approach

In this section we outline our approach for managing the security-scalability tradeoff, given scalability requirements. As Figure 5 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 3.1, and then present a brief example in Section 3.2 that illustrates the approach.

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 5) 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 [7].

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

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

```
done ← false
while done = false
    done ← true
    for each Q^T ∈ 𝒬^T where E(Q^T) > 1
        l ← E(Q^T)
        for each U^T ∈ 𝒰^T
            k ← 1
            while p_{k,l} = p_{k,(l−1)} and k < E(U^T)
                k ← k + 1
            if p_{k,l} = p_{k,(l−1)}
                done ← false
                E(Q^T) ← l − 1
    for each U^T ∈ 𝒰^T where E(U^T) > 1
        k ← E(U^T)
        for each Q^T ∈ 𝒬^T
            l ← 1
            while p_{k,l} = p_{(k−1),l} and l < E(Q^T)
                l ← l + 1
            if p_{k,l} = p_{(k−1),l}
                done ← false
                E(U^T) ← k − 1
```

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 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

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: 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.

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.

3.2 Example

Consider the TOYSTORE application shown in Table 3, an extension of our earlier SIMPLE-TOYSTORE application of Table 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 2.3, $E(U_2^T) =$ `template`.

The next step is Step 2a, in which the IPM domination relationships are characterized. The results for the TOYSTORE application are provided in Table 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 4, whenever $A_{ij} > 0$, $A_{ij} = 1$. Hence, $A_{11} = A_{12} = 1$. Further, using our analysis in Section 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 2.3).

Step 2b, in which Algorithm MinExposure is invoked, follows the IPM characterization step. When invoked on the TOYSTORE application (Table 3) with inputs as $E(U_2^T) =$ `template` (Step 1) and Table 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).

	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 4: Summary of IPM characterization for the example TOYSTORE application.

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 5: Notation for aspects of templates.

4 IPM Characterization

Recall from Section 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 4.2 – 4.4, how to determine for a given U^T/Q^T pair whether each of these relationships holds. Then, in Section 4.5 we discuss how additional information, beyond those considered up to now, affect IPM values. But, first in Section 4.1, we introduce some terminology for classifying query and update templates in a way that is useful for our analysis.

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 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), $S(Q_1^T) = \{$`toys.toy_name`$\}$, $P(Q_1^T) = \{$`toys.toy_id`$\}$, $S(U_1^T) = \{$`toys.toy_id`$\}$, $M(U_1^T) = \{$`toys.toy_id`, `toys.toy_name`, `toys.qty`$\}$.

Recall from Section 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 [23], an update 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), 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

$Q^T \in \mathcal{E}$ $Q^T \in \mathcal{N}$	Q is a query with only equality joins Q is a SPJ query with no top-k constructs
$U^T \in \mathcal{I}$ $U^T \in \mathcal{D}$ $U^T \in \mathcal{M}$	U is an insertion U is a deletion 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 6: Query and update classes.

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), query template Q_3^T is result-unhelpful for update template U_2^T.

In Table 6, we summarize the different classes of templates and properties of update/query template pairs.

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 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 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 omit a formal proof for brevity.

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 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 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., $(S(U_i^T) \cap S(Q_j^T) = \{\}) \wedge (\langle U_i^T, Q_j^T \rangle \notin \mathcal{G}) \Rightarrow B_{ij} = A_{ij} = 1$.

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 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., $(U_i^T \in \mathcal{I}) \wedge (Q_j^T \in \mathcal{E} \cap \mathcal{N}) \Rightarrow C_{ij} = B_{ij}$. We prove this result as Lemma 2 in Appendix A. 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 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 3 in Appendix A. For example, the equalities $C_{12} = B_{12}$ and $C_{13} = B_{13}$ hold for the TOYSTORE application (Table 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 4 in Appendix A. 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.

4.5 Database Integrity Constraints

So far the IPM values are based on the DSSP's (optional) knowledge of the update statement, the query statement, and the query result. The DSSP 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 DSSP to know the basic database integrity constraints[5], 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), 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 2.3) implies, if $A_{ij} = 0$, then the equality $A_{ij} = B_{ij} = C_{ij} = 0$ holds.

5 Evaluation

We have built a prototype DSSP to gain a better understanding of the magnitude of the security-scalability tradeoff, and to see how well our scalability-conscious security design methodology works in practice. Before presenting these results, we describe our benchmark applications in Section 5.1, and our experimental methodology in Section 5.2. Then, in Section 5.3 we confirm that blanket encryption of all data passing through the DSSP greatly hurts scalability. Finally, in Section 5.4 we find that our scalability-conscious security design methodology enables significantly greater security without impacting scalability.

[5]For all three benchmark applications that we study (details in Section 5.1), database integrity constraints fall into the category of insensitive data, and so revealing it to the DSSP does not compromise security.

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

Table 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.

5.1 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 [20], an auction system modeled after `ebay.com`, RUBBoS [21], a simple bulletin-board-like system inspired by `slashdot.org`, and TPC-W [25], a transactional e-Commerce application that captures the behavior of clients accessing an online book store[6]. We used Java implementation of these applications. We will henceforth refer to these applications as AUCTION, BBOARD, and BOOKSTORE, respectively.

The update/query templates of these applications differ from the assumptions outlined in Section 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 2.2.

5.1.1 IPM Characterization Results

Table 7 summarizes the IPM characterization results for the three applications, assuming the DSSP has knowledge of the two types of database integrity constraints mentioned in Section 4.5. Each row of Table 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 7, depending on whether $B < A$ or $B = A$, and whether $C < B$ or $C = B$. As Table 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 4 applies to the applications we studied.

5.2 Experimental Methodology

We performed our experiments with a simple two-node configuration—a home server that runs MySQL4 [19] as its database management system, and a DSSP node that provides answers to database queries using its store of the cached query results, running on Emulab [26]. (To keep the configuration simple, the DSSP

[6] 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. [6]. 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 7: Application configuration parameters.

Application	*Users*	*Updates*
AUCTION	400	3.9k updates (56% insertions, 44% modifications)
BBOARD	370	6.8k updates (60% insertions, 3% deletions, 37% modifications)
BOOKSTORE	760	11k updates (42% insertions, 4% deletions, 54% modifications)

Figure 8: Sample update rates.

node also provided the functionality of a CDN node, i.e., the ability to run Web applications and to interact with a user running a Web browser. We used Tomcat [14] 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 DSSP node had an Intel 64-bit Xeon processor with 2048 MB of memory. In all experiments, the home server and DSSP node were connected by a high latency, low bandwidth duplex link (100 ms latency, 2 Mbps). Each client was connected to the DSSP node by a low latency, high bandwidth duplex link (5 ms latency, 20 Mbps). These network settings model a deployment in which a DSSP node (because there are many of them) is "close" to the clients, most of which are "far" from any single home server.

Since the overhead for emulating clients is low, one additional Emulab node was used to emulate all clients. As in the TPC-W [25] specification, clients simulate human usage patterns by issuing an HTTP request, waiting for the response, and pausing for a *think time* of X seconds before requesting another Web page—X is drawn from a negative exponential distribution with a mean of seven seconds.

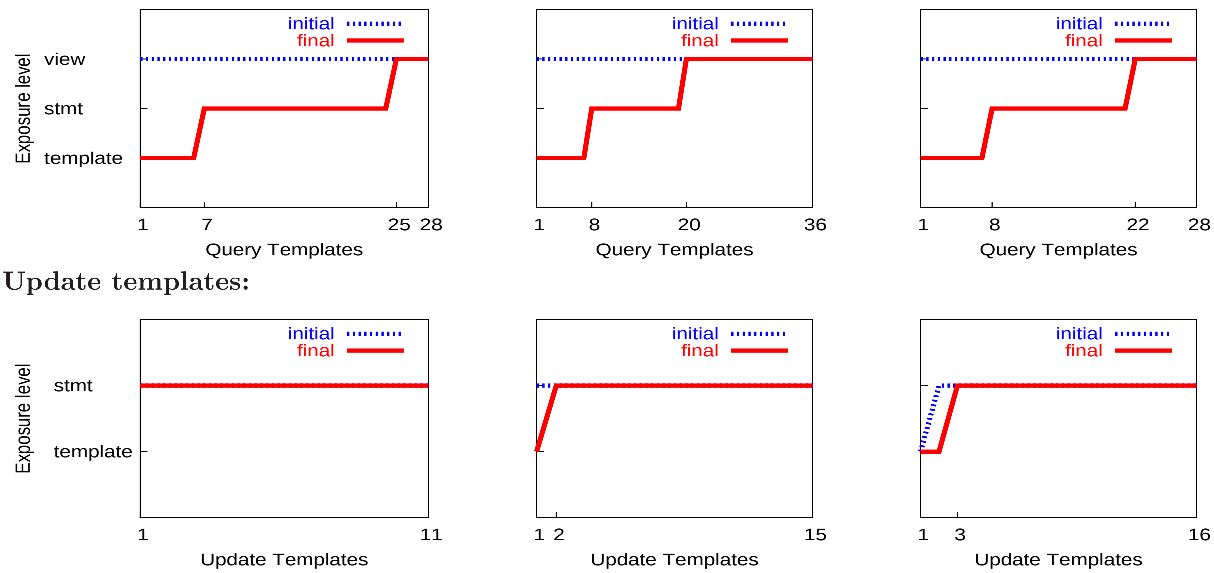
Each experiment ran for ten minutes, and the DSSP node started with a cold cache each time. Figure 7 provides the database configuration parameters we used in our experiments and Figure 8 lists sample update numbers for a ten minute run. *Scalability* was measured as the maximum number of users that could be supported while keeping the response time below two seconds for 90% of the HTTP requests.

5.3 Magnitude of Security-Scalability Tradeoff

Figure 10 plots the scalability of an application as a function of the invalidation strategy used by the DSSP, for all three applications. The y-axis plots scalability, measured as specified in Section 5.2. On the x-axis, we consider an instance of each of the four classes of invalidation strategies introduced in Section 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 5.2.

For each application, the leftmost strategy, a minimal view inspection strategy (MVIS), offers the best

Query templates:



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

Figure 9: Starting with the California data privacy law, additional exposure reduction for query and update templates.

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 10 confirms the claim made in Section 1 that blanket encryption of all data (thereby requiring a blind invalidation strategy) significantly hinders scalability.

5.4 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 of Section 1.2, which plots scalability[7] 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 3.1, 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 [7], which, when applied to our applications, mandates securing all credit card information.

Figure 9 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);

[7]Computational overhead of encryption and decryption is not taken into account. Optimizing the encryption and decryption process is beyond the scope of the paper.

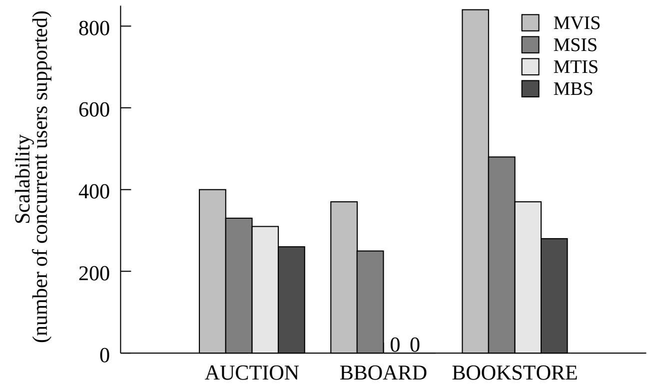


Figure 10: Tradeoff between security and scalability, as a function of coarse-grain invalidation strategy.

the solid lines show the final exposure levels resulting from the application of our static analysis. The area between the lines gives an idea of the reduction in exposure achieved using our approach.

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

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

In all cases scalability is not affected—it remains the same as that of MVIS in Figure 10.

6 Future Work

We view this work as an encouraging first step, and we plan to extend our research in the following directions. First, we intend to devise invalidation strategies that use additional data—in the form of auxiliary views [23] or cached query results of other queries—to make more precise invalidation decisions. Such invalidation strategies have the potential to increase scalability without compromising security by much. Second, we plan to study the effect of reducing the granularity of encryption, from query results to columns, on the security-scalability tradeoff. Third, we intend to improve our static analysis to find more classes of update/query template pairs for which two or more minimal invalidation strategies always produce the same outcome. Discovering such classes is key to enabling greater security without impacting scalability. Finally, we plan to develop a tool that provides estimates about the cost of encrypting a particular data item, which the administrators can then use to decide whether to encrypt a data item or not.

7 Summary

In this paper we explored ways to secure the data of Web applications that use the services of a shared DSSP 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 DSSP needs to invalidate data from its cache. The amount of data invalidated varies depending on the information exposed to the DSSP. The less information exposed to the DSSP, 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.

Acknowledgments

We thank Mukesh Agrawal, Charles Garrod, Phillip B. Gibbons, Bradley Milan, and Haifeng Yu for their valuable feedback and suggestions.

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] 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.

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

[5] J. A. Blakeley, N. Coburn, and P. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM TODS*, 14(3):369–400, 1989.

[6] 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`.

[7] California Senate. Bill SB 1386. `http://info.sen.ca.gov/pub/01-02/bill/sen/sb_1351-1400/sb_1386_bill_200%20926_chaptered.html`, 2002.

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

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

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

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

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

[13] H. Hacigumus, B. Iyer, and S. Mehrotra. Efficient execution of aggregation queries over encrypted relational databases. In *9th International Conference on Database Systems for Advanced Applications*, 2004.

[14] Jakarta Project. Apache Tomcat.

[15] M. Kantarcioglu and C. Clifton. Security issues in querying encrypted data. Technical Report TR-04-013, Purdue University, 2004.

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

[17] 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.

[18] 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.

[19] MySQL AB. MySQL database server.

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

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

[22] 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.

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

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

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

[26] 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.

A Proofs for Section 4.4

Lemma 2 *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.,* $(U_i^T \in \mathcal{I}) \wedge (Q_j^T \in \mathcal{E} \cap \mathcal{N}) \Rightarrow C_{ij} = B_{ij}$.

Proof: See Appendix B. □

Lemma 3 *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 4 *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 4 can be proved in two parts:

Part 1 $(\langle \mathbf{U_i^T}, \mathbf{Q_j^T} \rangle \in \mathcal{G} \Rightarrow \mathbf{C_{ij}} = \mathbf{B_{ij}})$**:** 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 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 $(\langle \mathbf{U_i^T}, \mathbf{Q_j^T} \rangle \in \mathcal{H} \Rightarrow \mathbf{C_{ij}} = \mathbf{B_{ij}})$**:** 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}. □

B Proof of Lemma 2

In this section we prove Lemma 2. 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 B.1. Then, in Appendix B.2, we describe additional database operations we use in our proof. In Appendix B.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 2 in Appendix B.4.

B.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.

B.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, $m(D, Q) = \cup_{D' \in \mathcal{D}_S(D,Q)} D'$.

B.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).

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.

B.4 Intermediate Lemmas and Proofs

Lemma 5 *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 B.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 6 *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.,

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

From Appendix B.3, we know that for logic expression (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 6 about existence of complementary database, the inserted tuple must fail to join with other tuples for logic expression (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 2.

Proof: **Proof of Lemma 2 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 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.,

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

Applying Lemma 5 to logic expression (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.,

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

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

We now construct a database D' that does not obey logic expression (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 B.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 (4) applies, the second condition of Lemma 6 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.,

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

Using Lemma 5 in conjunction with logic expression (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 (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. $\square$