An Integrity Lock Architecture for Supporting Distributed Authorizations in Database Federations

Wei Li, Lingyu Wang, Bo Zhu, and Lei Zhang

Abstract In many data integration applications, a loosely coupled database federation is the underlying data model. This paper studies two related security issues unique to such a model, namely, how to support fine-grained access control of remote data and how to ensure the integrity of such data while allowing legitimate updates. For the first issue, we adapt the integrity lock architecture in multi-level database systems to a database federation. For the second issue, we propose three-stage procedure based on grids of Merkel Hash Trees. Finally, the performance of the proposed architecture and scheme is evaluated through experiments.

1 Introduction

Data integration and information sharing have attracted significant interests lately. Although web services play a key role in data integration as the main interface between autonomous systems, a loosely coupled database federation is usually the underlying data model for the integrated system. Among various issues in establishing such a database federation, the authorization of users requesting for resources that are located in remote databases remains to be a challenging issue in spite of many previous efforts. The autonomous nature of a loosely coupled federation makes it difficult to directly apply most centralized authorization models, including those

Wei Li, Lingyu Wang, and Bo Zhu
Concordia Institute for Information Systems Engineering
Concordia University
Montreal, QC H3G 1M8, Canada
e-mail: {w_li7, wang, zhubo}@ciise.concordia.ca

Lei Zhang
Center for Secure Information Systems
George Mason University
Fairfax, VA 22030-4444, USA
e-mail: lzhang8@gmu.edu

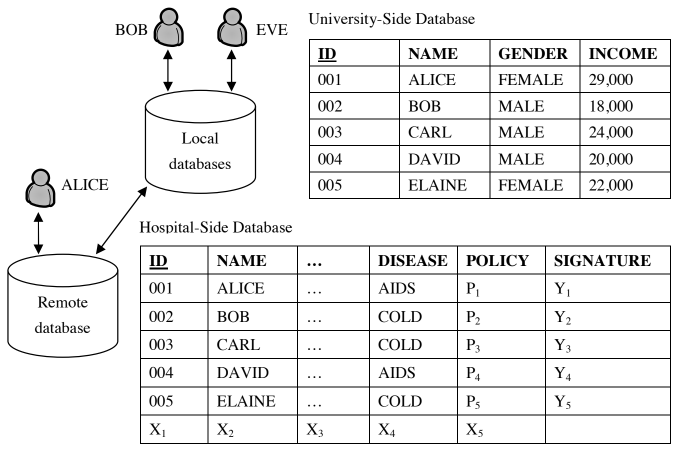
proposed for tightly coupled database federations. The subject and object in an access request may belong to different participating databases that are unaware of each other's user accounts, roles, or authorization policies. Duplicating such information among the members is generally not feasible due to the confidential nature of such information. In addition, participating members in a database federation usually lack full trust in each other, especially in terms of authorizations and data integrity.

In this paper, we propose to support the distributed authorization in database federations by adapting the *integrity lock architecture*, which is originally designed for building multi-level database systems from un-trusted DBMS. Although intended for a different purpose, the architecture has some properties that are particularly suitable for database federations. First, the architecture does not require the DBMS to be trusted for authorizations or data integrity. Instead, it supports *end-to-end* security between the creation of a record to the inquiry of the same record. This capability is essential to a database federation where members do not fully trust each other for authorizations or data integrity. Second, the architecture binds authorization polices to the data itself, which can avoid duplicating data or policy across the federation, and also allows for fine-grained and data-dependent authorizations. A database federation under the adapted integrity lock architecture has some similarity with outsourced databases (ODB), such as the lack of trust in the remote database. However, a fundamental difference is that data in a federation of operational databases is subject to constant updates. This difference brings a novel challenge for ensuring integrity while allowing legitimate updates.

Motivating Example Consider the toy example depicted in Figure 1 (we shall only consider two databases unless explicitly specified otherwise since extending our solutions to a federation with more members is straightforward). Suppose a fictitious university and its designated hospital employ an integrated application to provide the university's employees direct accesses to their medical records hosted at the hospital. Bob and Eve are two users of the university-side application, and Alice is a user of the hospital-side application (we do not show details of those applications but instead focus on the interaction between the underlying databases).

In Figure 1, consider the two tables in the university and hospital's database, respectively. The two tables are both about employees of the university, and they have two attributes *ID* and *NAME* in common. As a normal employee of the university, Bob is not supposed to have free accesses to other employees' *DISEASE* attribute values hosted at the hospital. On the other hand, another user at the university side, Eve, may be authorized to access records of a selected group of employees due to her special job function (for example, as a staff working at the university clinic or as a secretary in a department). At the hospital side, Alice is prohibited from accessing the *INCOME* attribute of any university employee. However, as a doctor designated by the university, Alice is authorized to modify (and access) the *DISEASE* attribute.

The above scenario demonstrates the need for a federation of databases. We can certainly store the *DISEASE* attribute in the university-side database and thus completely eliminate the hospital-side table. However, such attribute (and other related medical data) will most likely be accessed and updated more frequently from the hospital side, so storing it at the hospital is a more natural choice. The above sce-



University-Side Database

ID	NAME	GENDER	INCOME
001	ALICE	FEMALE	29,000
002	BOB	MALE	18,000
003	CARL	MALE	24,000
004	DAVID	MALE	20,000
005	ELAINE	FEMALE	22,000

Hospital-Side Database

ID	NAME	…	DISEASE	POLICY	SIGNATURE
001	ALICE	…	AIDS	P_1	Y_1
002	BOB	…	COLD	P_2	Y_2
003	CARL	…	COLD	P_3	Y_3
004	DAVID	…	AIDS	P_4	Y_4
005	ELAINE	…	COLD	P_5	Y_5
X_1	X_2	X_3	X_4	X_5	

Fig. 1 An Example Database Federation

nario also shows the need for fine-grained and data-dependent access control of remote data. Row-level (or attribute-level) access control is clearly needed since Bob should normally only access his own records. Eve's job function may entitle her to only access records that satisfy certain conditions, such as *DISEASE* not equal to *AIDS*. That is, the access control policy may depend on actual data. Finally, the scenario shows the need for verifying the legitimacy of updates of remote data, such as that only a doctor designated by the university can modify the *DISEASE* attribute.

In the special setting of a database federation, we assume the university still *owns*, and is responsible for, its employees' medical records, even though the records are stored in the hospital. This is different from the case of two separate organizations where the university has no responsibility for its employees' interaction with a hospital. From this point of view, we can regard the university as *outsourcing* their employees' medical records to the hospital. However, different from the outsourced database (ODB) architecture where outsourced data are relatively static, the database federation we consider comprises of operational databases in which data are constantly being updated. As we shall show, existing solutions for ensuring the integrity of outsourced data in ODB are not sufficient for database federations.

The rest of the paper is organized as follows. Section 2 adapts the integrity lock architecture to database federations for fine-grained access control of remote data. Section 3 proposes a three-stage procedure for supporting legitimate updates of remote data while ensuring their integrity. Section 4 shows experimental results to evaluate different caching schemes. Section 5 reviews previous work. Section 6 concludes the paper.

2 Adapting The Integrity Lock Architecture to Database Federations

The *Integrity Lock architecture* is one of the *Woods Hole architectures* originally proposed for multi-level databases [19]. The integrity lock architecture depends on a trusted front end (also called a filter) to mediate accesses between users and the un-trusted DBMS (the original model also has an un-trusted front end, which is omitted here for simplicity) [5, 6, 8, 15]. Each tuple in tables has two additional attributes, namely, a security level and a cryptographic stamp. The stamp is basically a message authentication code (MAC) computed over the whole tuple excluding the stamp itself using a cryptographic key known to the trusted front end only.

The trusted front end determines the security level of the new tuple and computes the stamp to append it to the query when a tuple is to be inserted or updated. The query is then forwarded to the DBMS for execution. When users submit a legitimate selection query, the trusted front end will simply forward the query to the DBMS. Upon receiving the query result from the latter, the trusted front end will verify all tuples in the result and their security levels by recomputing and matching the cryptographic stamps. If all the data check out, the trusted front end will then filter out prohibited tuples based on their security levels, the user's security level, and the security policy. The remaining tuples are then returned to the user as the query result. The main objective of the integrity lock architecture is to reduce costs by building secure databases from un-trusted off-the-shelf DBMS components.

As described above, the cryptographic stamps provide *end-to-end integrity* from the time a record is created (or modified) to the time it is returned in a query result. The un-trusted DBMS cannot alter the record or its associated security level without being detected. Such a capability naturally fits in the requirements of a database federation illustrated before. More specifically, in Figure 1, we can regard the university-side database as the trusted front end, and the hospital-side database as an un-trusted DBMS in the integrity lock architecture. Suppose a user Eve of the university-side database wants to insert or update some records in the table stored at the hospital The university-side database will compute and append a cryptographic stamp to the tuple to be inserted or updated. When a user of the university-side database wants to select tuples in the hospital-side database, the university database will enforce any policy that is locally stored through either rejecting or modifying the original query posed by the user. Upon receiving query results from the latter, the university database will then verify the integrity of each returned tuple in the results through the cryptographic stamp in the tuple. It then filters out any tuple that Bob is not allowed to access according to the access control policy.

The adapted architecture also faces other issues. First, the original architecture requires a whole tuple to be returned by the un-trusted DBMS [8, 5], because the cryptographic stamp is computed over the whole tuple (excluding the stamp itself). This limitation may cause unnecessary communication overhead between databases in the federation. A natural solution to remove this limitation is to use a Merkle Hash Tree (MHT) [16]. Second, the integrity lock architecture can only detect modified tuples but cannot detect the omission of tuples in a query result. That is, the completeness of query results is not guaranteed. A similar issue has recently been addressed in outsourced databases (ODB) [7, 20, 14]. Two approaches can address this issue. A signature can be created on every pair of adjacent tuples (assuming the tuples are sorted in the desired order), and this chain of signatures is sufficient to prove that all tuples in the query result are contiguous and no tuple has been omitted. To reduce communication overhead and verification efforts, the signatures can be aggregated using techniques like the Condensed RSA [18]. Another approach is to build a MHT on the stamps of all tuples based on a desired order, so omitting tuples from query results will be detected when comparing the signature of the root to the stamp. However, applying the above solutions in ODB to the integrity lock architecture in database federations is not practical. A fundamental difference between ODB and database federations is that the former usually assumes a relatively static database with no or infrequent updates [1]. Data updates usually imply significant computational and communication costs. Such an overhead is not acceptable to database federations, because the members of such a federation are operational databases and data are constantly updated. We shall address such issues in the rest of this paper.

3 Supporting Frequent Updates While Ensuring Data Integrity

3.1 Overview

The previous section left open the issue of ensuring the integrity of data in remote databases while allowing for updates made by authorized users. First of all, we describe what we mean by *authorized users*. For simplicity, we shall refer to the database hosting data as *remote database* and the other database *local database*. We assume the federation provides each member the capability of authenticating users of a remote database. Such a capability should be independent of the remote database since we assume it to be un-trusted for authorizations. Our solutions will not depend on specific ways of authenticating remote users, although we shall consider a concrete case where a remote user possesses a public/private key pair and (queries issued by) the user is authenticated through digital signatures created using his/her private key.

Two seemingly viable approaches are either to verify the update queries, or to verify the state of remote data immediately after each update. First, in Figure 1, whenever Alice attempts to update a record, the hospital-side database can send the query and records to be updated, which are both digitally signed by Alice, to the university-side database for verification. The latter will verify the legitimacy of the update by comparing Alice's credential to the access control policies stored in the

[1] One exception is the recent work on accommodating updates while ensuring data confidentiality [4], which is parallel to our work since we focus more on data integrity.

records. However, this approach is not valid because the hospital-side database must be trusted in forwarding all update queries for verification and in incorporating all and only those legitimate updates after they are verified. Second, the university-side database can choose to verify the state of remote data after every update made to the data. However, this approach faces two difficulties. First of all, it is difficult to know about every update, if the remote database is not trusted (it may delay or omit reporting an update). Moreover, the approach may incur unnecessary performance overhead. For example, during a diagnosis, a doctor may need to continuously make temporary updates to a medical record before a final diagnosis conclusion can be reached. The university-side database should not be required to verify all those temporary updates.

We take a three-stage approach, as outlined below and elaborated in following sections.

- First, referring to the example in Figure 1, the university-side database will adopt a *lazy* approach in detecting modifications. More precisely, when Bob or Eve issues a selection query and the hospital-side database returns the query result, the university-side database will attempt to detect and localize any modifications related to tuples in the query result based on a two-dimensional grid of MHTs.
- Second, if a modification is detected and localized, then the local database will request the remote database for proofs of the legitimacy of such updates. The remote database then submits necessary log entries containing digitally signed update queries corresponding to those updates. The local database will check whether the queries are made by those users who are authorized for such updates and whether those queries indeed correspond to the modified data.
- Third, the local database will then disregard any tuples in the query result for which no valid proof can be provided by the remote database. To accommodate legitimate updates, the local database will incrementally compute the new MHTs and send them back to the remote database who will incorporate those new MHTs into the table.

3.2 Detecting and Localizing Modifications

We compute a two-dimensional grid of MHTs on a table to detect and localize any update to tuple or attribute level (a grid of watermarks is proposed for similar purposes in [10]). In Figure 2, $A_i (1 \leq i \leq n+1)$ are the attributes, among which we assume A_1 is the primary key and A_n the access control policy for each tuple. The MHT is built with a collision-free hash function $h()$ and $sig()$ stands for a public key signature algorithm. Each $y_i (1 \leq i \leq m)$ is the signature of the root w_i of a MHT built on the tuple $(v_{i,1}, v_{i,2}, \ldots, v_{i,n})$. Similarly, each x_i is a signature of the root u_i of the MHT built on the column $(v_{1,i}, v_{2,i}, \ldots, v_{m,i})$. Referring to Figure 1, for the hospital-side table, the signatures will be created by the university-side database using its private key. If a table includes tuples that are *owned* by multiple databases, then multiple signatures can be created and then aggregated (for example, using the

Condensed RSA scheme [18]) as one attribute value, so any involved database can verify the signature.

$\underline{A_1}$	A_2	A_3	A_4	A_5	…	A_n	A_{n+1}
$v_{1,1}$	$v_{1,2}$	…				$v_{1,n}$	y_1
$v_{2,1}$	$v_{2,2}$	…				$v_{2,n}$	y_2
…	…	…				…	…
$v_{m,1}$	$v_{m,2}$	…				$v_{m,n}$	y_m
x_1	x_2	x_3	x_4	x_5	…	x_n	

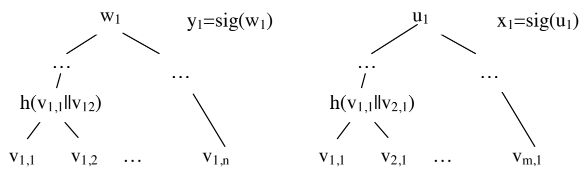


Fig. 2 A Grid of Merkel Hash Trees on Tables

Suppose Bob poses a selection-projection query whose result includes a set of values $V \subseteq \{v_{i,j} \mid 1 \leq i \leq m, 1 \leq j \leq n-1\}$. Then the hospital-side database needs to return the set V, the policy $v_{i,n}$ and the signatures x_i and y_j for each $v_{i,j} \in V$. Moreover, the siblings needed for computing the root of the MHTs from which the signatures have been computed should also be returned. Upon receiving the query result, the university-side database will first verify the signatures and the values in V by re-computing the root of the corresponding MHTs. If all the signatures are valid, then university database is ensured about the integrity of the query result. It will then examine the access control policies and filter out those tuples that are not allowed to be accessed by the user, and check the completeness of the query result based on the MHTs using techniques in [7, 20, 14]. If everything checks out, the query will be answered.

If some of the recomputed signatures do not match the ones included in the query result, then modified data must first be localized based on following observations [10]. If a value $v_{i,j}$ is updated, then the signatures y_i and x_j will both mismatch. The insertion of a new tuple $(v_{i,1}, v_{i,2}, \ldots, v_{i,n})$ will cause the signature $x_1, x_2, \ldots, x_n$ and y_i to mismatch, while all the $y_j (j \neq i)$ will still match. The deletion of a tuple $(v_{i,1}, v_{i,2}, \ldots, v_{i,n})$ will cause the signature $x_1, x_2, \ldots, x_n$ to mismatch, while all the $y_i (1 \leq i \leq n-1)$ will still match. The localization of modifications helps to reduce the amount of proofs that need to be provided (and thus the communication and computational costs) in the later verification phase. However, this mechanism does not guarantee the precise identification of every update made to the data. Fortunately, as we shall show, the verification phase does not rely on this localization mechanism.

3.3 Verifying the Legitimacy of Updates

Before we discuss the protocol for verifying updates, we need to describe how a remote database is supposed to handle updates. A remote database will need to record all the following into a log file: The update query, the signature of the query created with the user's private key, the current time, the current value before the update for deletion, and the current signatures involved by the update. Such information in the log file will allow the remote database to be rolled back to the last valid state. The information will thus act as proofs for the legitimacy of updates. When updates are detected and localized, the local and remote databases will both follow the protocol shown in Figure 3 to automatically verify the legitimacy of those updates and to accommodate legitimate updates by updating signatures stored in the table.

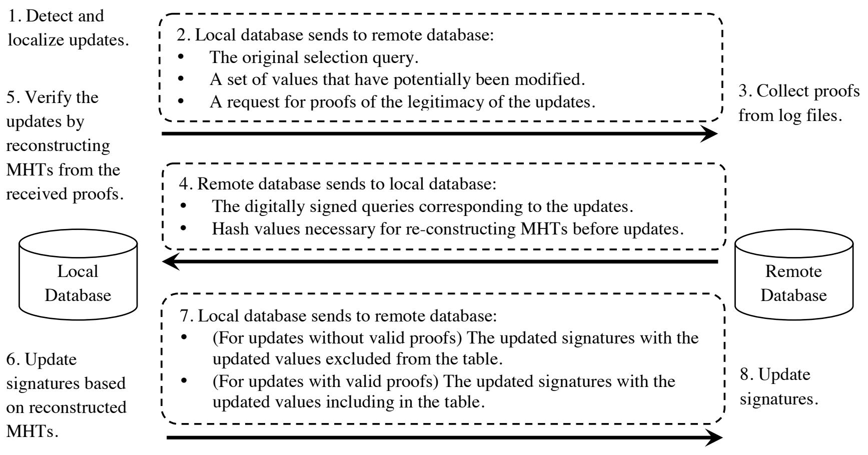

1. Detect and localize updates.
2. Local database sends to remote database:
• The original selection query.
• A set of values that have potentially been modified.
• A request for proofs of the legitimacy of the updates.
3. Collect proofs from log files.
4. Remote database sends to local database:
• The digitally signed queries corresponding to the updates.
• Hash values necessary for re-constructing MHTs before updates.
5. Verify the updates by reconstructing MHTs from the received proofs.
Local Database
Remote Database
6. Update signatures based on reconstructed MHTs.
7. Local database sends to remote database:
• (For updates without valid proofs) The updated signatures with the updated values excluded from the table.
• (For updates with valid proofs) The updated signatures with the updated values including in the table.
8. Update signatures.

Fig. 3 The Protocol for the Verification of Updates

In step 1, the local database detects mismatches in signatures and localizes the updates to a set of values that may have been updated (recall that the localization does not guarantee the precise set of modified values). The local database will then send to the remote database the potentially updated values and related information, such as the original selection query in step 2. In step 3, the remote database examines its log files to find each update query that involves the received values. For each such query, the remote database will attempt to reconstruct the mismatched signatures using values and signatures found in the log file, which are supposed to be before the update. If a state is found in which all the mismatched signatures match again, then the involved queries will be collected as proofs and sent to the local database in step 4. Otherwise, the remote database will send to the local database a response indicating no proof for the updates is found.

In step 5, the local database will verify the signatures of the received update queries and ensure those queries are made by users who are authorized for such updates. The local database then attempts to reconstruct from the received queries a previous valid state in which all mismatched signatures match again. If such a state is found and all the update queries until that state are made by authorized users, then the detected updates are legitimate so the local database will create signatures by including the updated values (the details will be given in the next section) in step 6. Otherwise, the updates are unauthorized, so signatures are created by excluding the updated values in step 6. Upon receiving the updated signatures in step 7, the remote database will then update the received signatures in the table in step 8. The local database will only answer the original selection query if all the involved values are successfully verified.

3.4 Accommodating Legitimate Updates

To accommodate updates that are successfully verified to be made by authorized users, the local database needs to compute new signatures by including the updated values so the remote database can update the signatures in the table. Similarly, updates of signatures are also required for newly inserted tuples. Recomputing signatures for each record does not incur a significant performance overhead because the number of attributes in a table is limited. However, the signature of a column may be computed over a large number of records, and its computation is thus costly. Moreover, any insertion or update of a record will cause at least one of the signatures of columns to be updated. To reduce the computational cost of such updates, an obvious solution is to divide the table into smaller sub-tables with fewer records, and then apply the aforementioned grid of MHTs to each sub-table independently (instead of actually dividing the table, it is more convenient to simply change the way the grid of MHTs is computed).

However, upon a closer look, dividing the table does not solve all the problems. First, the table may need to be divided differently based on the ordering of tuples by different attributes. For example, in Figure 1, suppose we divide the table based on *ID*, then a query asking for tuples with a certain age may involve all the sub-tables, which essentially diminishes the value of dividing the table (diving the table will also cause more storage cost due to more signatures). Second, a more severe issue lies in the fact that even for a smaller sub-table, the local database cannot recompute signatures from all the values stored in the table simply because it does not have such values. Sending those values from the remote database will incur too much communication cost. Even to let the remote database compute the root will still incur high computational cost, considering that each insertion of a new tuple will cause the whole sub-table to be sent over.

Fortunately, a MHT can be incrementally updated. As illustrated in Fig 4, to update the hash value 3, the local database only needs the hash values 1, 2 in the MHT of each column, instead of all the leaves. To balance the MHT over time, for insertion of new tuples, we should choose to insert each value at an existing hash value that has the shortest path to the root (this may not be feasible for ordered attributes where the order of MHT leaves is used for ensuring the completeness of query results). The next question, however, is where to obtain the required hash values 1 and 2, given that recomputing them from the leaves is not an option. One possibility is to keep a cache of all or part of the non-leaf hash values in the MHT. If we keep all the non-leaf values in a cache, then a direct lookup in the cache will be sufficient for computing the root, which has a logarithm complexity in the cardinality of the table (or sub-table).

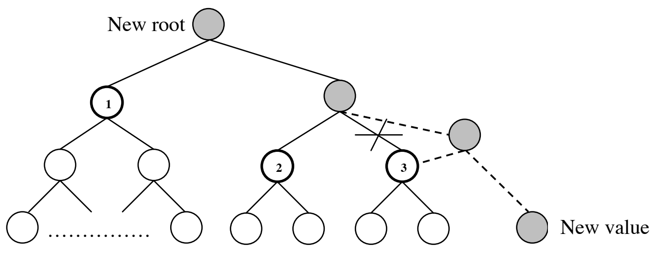


Fig. 4 Update the Root of a MHT

Considering the fact that the number of all non-leaf values is comparable to the number of leaves, the storage overhead is prohibitive. Instead, we can choose to cache only part of the MHT based on available storage. Two approaches are possible. First, we can use a static cache for a fixed portion of the MHT. If we assume a query will uniformly select any tuple, then clearly the higher a hash value is in the MHT, the more chance it will have to be useful in recomputing the new root of the MHT. For example, in Fig 4, the value 1 will be needed in the update of twice as much values as the value 2 will. Given a limited storage, we thus fill the cache in a top-down manner (excluding the root).

The assumption that queries uniformly select tuples may not hold in many cases. Instead, subsequent queries may actually select adjacent tuples in the table. In this case, it will lead to better performance to let the queries to drive the caching of hash values. We consider the following dynamic caching scheme. We start with the cache of a top portion of the MHT. Each time we update one tuple, we recompute the new root with the updated value using as much values as possible from the cache. However, for each non-leaf value we need to recompute due to its absence in the cache, we insert this value into the cache by replacing a value that is least recently used (other standard caching schemes can certainly be used). Among those that have the same timestamp for last use, we replace the value that has the longest path from the root.

3.5 Security Analysis

We briefly describe how the proposed scheme prevent various attacks using the previous example. Suppose in the hospital-side database, a malicious user inserts/deletes medical records or modifies some values. Such modifications will cause mismatches between recomputed MHT roots and those stored in the table, by which the university-side database will detect modifications. The hospital-side database, controlled by the malicious user, cannot avoid such a detection due to the security of MHT. The malicious user may attempt to modify the log entries to hide his activities by masquerading as users authorized for the updates. However, we assume the university-side database can authenticate remote users' queries through their signatures, so such signatures cannot be created by the malicious user without the private key of an authorized user. The malicious user can prevent the hospital-side database from sending proofs or reporting the absence of proofs, but this does not help him/her to avoid detection (a timeout scheme can be used for the case of not receiving proofs in a timely fashion). The malicious user can also reorder or mix up updates made by authorized users with his/her unauthorized updates. However, this will also be detected when the university-side database attempts to rebuild a previous valid state of data but fails. The only damage that can be caused by malicious users is a denial of service when too many tuples are excluded due to unauthorized modifications. However, as mentioned before, a database member may request the remote database to initiate an investigation when the number of such tuples exceeds a threshold. Ultimately, the use of signatures computed over the grid of MHTs provides the end-to-end integrity guarantee between the time of creating or updating (by both authorized users from the university or at the hospital) to the time of inquiry.

4 Experimental Results

We have implemented the proposed techniques in Java running on systems equipped with the Intel Pentium M 1.80GHz processor, 1024G RAM, Windows, and Oracle 10g DBMS. The main objective of the experiments is to compare the performance of different caching schemes, namely, a static cache of all the non-leaf values of each MHT, a static caches of partial MHTs of different sizes, and a dynamic cache of fixed size based on queries.

The left-hand side chart in Figure 5 shows the computation cost of updating a tuple in different size of databases when all non-leaf values are cached. We can see that at the cost of storage, there is only a relatively small difference between updating tuples without recomputing signatures (that is, ignoring the security requirement) and re-computing signatures from static cache. On the other hand, recomputing MHTs from scratch is very costly. The right-hand side chart in Figure 5 shows both the storage requirement and the performance of static caches of different sizes, which all hold a top portion of the MHT. We update one tuple in a database with 15,000 records. We reduce the cache size by removing each level of the MHT in a bottom-up fashion. The curve with square dots shows the number of values in the cache, that is, the storage requirement for caching. The other curve shows the computational cost. We can see that the overall performance is good in the range of (the hash tree height) -3 and -10 where both the storage requirement and the computational cost are acceptably low.

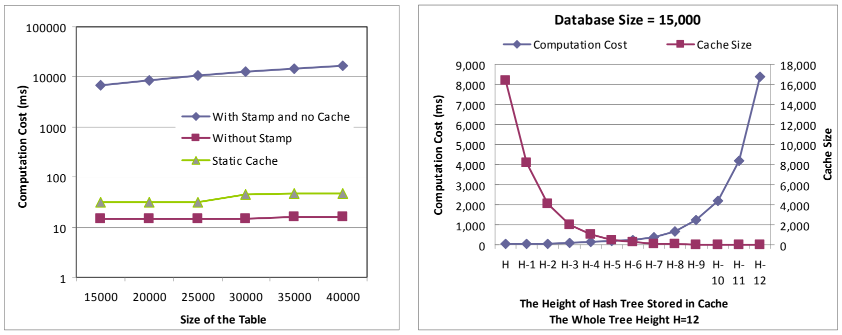


Fig. 5 The Performance of Static Cache

Figure 6 compares the computational cost of dynamic caching with that of the static caching under the same storage limitation. The database size is 15,000 records, and the cache is limited to store only 500 hash values in the MHT. To simulate queries that select adjacent tuples, we uniformly pick tuples within a window of different sizes. In Figure 6, *n* is the size of the window, *m* is the number of records involved by a query, the horizontal axis is the percentage of updated values within the window. We can see that as more and more values are updated, the performance of dynamic caching will improve since the cache hit rate will increase. The window size has a small effect on this result, which indicates that the dynamic cache is generally helpful as long as subsequent queries focus on adjacent tuples.

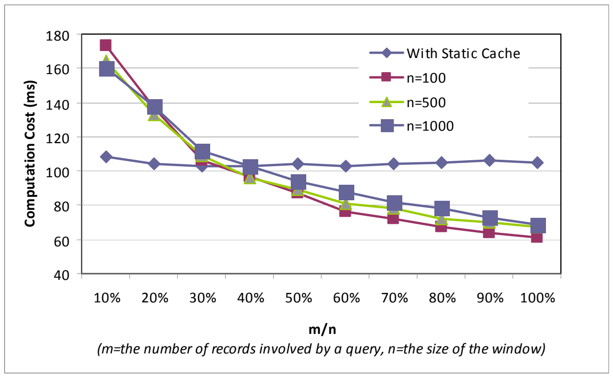


Fig. 6 The Performance of Static Cache and Dynamic Cache

5 Related Work

A Federated Database System (FDBS) is a collection of cooperating yet autonomous member database systems[21]. Member databases are usually heterogeneous in many aspects such as data models, query languages, authorization policies, and semantics (which refers to the fact that the same or similar data items may have different meanings or distinct intended usages among member databases). According to the degree of integration, FDBSs are mainly classified as *loosely coupled FDBS* and *tightly coupled FDBS*. A loosely coupled FDBS is rather like a collection of interoperable database systems. Most research efforts have focused on a tightly coupled FDBS, where the federation is created at design time and actively controls accesses through the federation. Although designing a tightly coupled FDBS from scratches has obvious advantages, in many cases it may not be feasible due to the implied costs. Our study assumes the loosely coupled FDBS model, and we do not require major modifications to existing DBMSs. This makes our approach more attractive to data integration applications. *Metadirectories* and *virtual directories* technology have similarity with our studied problem. They both can access data from different repositories by using directory mechanisms such as *Lightweight Directory Access Protocol (LDAP)*. When data in source directories changes frequently, it is a big headache to keep data updated. Which will have much more storage and computation cost when updating. However, our approach is based on the assumption that the remote database is untrusted to the local database, there is no authentication between the two databases.

Access control in FDBS is more complicated than in centralized databases due to the autonomy in authorization [2, 3, 9, 13, 23], which allows member databases to have certain control over shared data. Depending on the degree of such control, access control can be divided into three classes. For *full authorization autonomy*, member databases authenticate and authorize federation users as if they are accessing member databases directly. In the other extreme, *low authorization autonomy* fully trusts and relies on the federation to authenticate and authorize federation users. The compromise between the two, namely *medium authorization autonomy*, provides member databases with partial control on shared resources. Existing techniques, such as *subject switching*, usually requires members to agree on a loose mapping between user accounts and privileges in both databases such that one can help the other on making authorization decisions. Our approach does not require such a predefined mapping between databases but instead filters the result before giving it to the user. Several database recovery mechanisms based on trusted repair algorithms are adopted in commercial database systems. Each repair algorithm has static and dynamic version. There are various possibilities when maintaining read-from dependency information [1]. The survivability model extended from the class availability model is developed by using a state transition graph to model a ITDB (Intrusion Tolerant Database system), and it can provide essential services in the presence of attacks [22]. These works are similar to our approach in that they both need to isolate and recover from modified tuples. However, we focus more on the interaction between local and remote databases.

Multilevel databases have received enormous interests in the past, as surveyed in [19, 11, 12]. Various architectures have been proposed for building multilevel databases from un-trusted components [19]. The *polyinstantiation* issue arises when a relation contains records with identical primary key but different security levels [11]. A solution was given to the polyinstantiation problem based on the distinction between users and subjects [12]. The next section will review one of the architectures for multilevel databases in more details. More recently, outsourced database security has attracted significant interests [7, 18, 20, 17, 14]. One of the major issues in outsourced databases is to allow clients to verify the integrity of query results, because the database service provider in this model is usually not trusted. Various techniques based on cryptographic signatures and Merkle hash trees [16] have been proposed to address the integrity and completeness of query results. We have discussed the limitations in directly applying existing techniques in outsourced databases to the federation of operational databases in the paper. Parallel to our work, a recent effort is on accommodating updates while ensuring data confidentiality in ODB, The over-encryption model presents a solution for outsourced database to enforce access control and evolving polices using keys and tokens without the need for decrypting the resource to retrieve the original data and re-encryption [4].

6 Conclusion

We have addressed the issue of distributed authorization in a loosely coupled database federation. We revisited the integrity lock architecture for multi-level databases and showed that the architecture provides a solution to the authorization of accesses to remote data in database federations. We then proposed a novel three-stage scheme for the integrity lock architecture to ensure data integrity while allowing for legitimate updates to the data. We also devised a procedure for members of a database federation to update integrity stamps for legitimate updates. Our future work include the study of more efficient ways for handling concurrent updates made by multiple databases and the implementation and evaluation of a prototype based on the proposed techniques.

Acknowledgements The authors thank the anonymous reviewers for their valuable comments. This material is based upon work supported by Natural Sciences and Engineering Research Council of Canada under Discovery Grant N01035.

References

1. Paul Ammann, Sushil Jajodia, and Peng Liu. Recovery from malicious transactions. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1167–1185, 2002.
2. Jonscher D. and K.R. Dittrich. Argos - a configurable access control system for interoperable environments. In *IFIP Workshop on Database Security*, pages 43–60, 1995.

3. S. Dawson, P. Samarati, S. De Capitani di Vimercati, P. Lincoln, G. Wiederhold, M. Bilello, J. Akella, and Y. Tan. Secure access wrapper: Mediating security between heterogeneous databases. In *DARPA Information Survivability Conference and Exposition (DISCEX)*, 2000.
4. S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Over-encryption: Management of access control evolution on outsourced data. In *VLDB*, 2007.
5. D.E. Denning. Cryptographic checksums for multilevel database security. In *Proc. of the 1984 IEEE Symposium on Security and Privacy*, pages 52–61, 1984.
6. D.E. Denning. Commutative filters for reducing inference threats in multilevel database systems. In *Proc. of the 1985 IEEE Symposium on Security and Privacy*, pages 134–146, 1985.
7. Premkumar T. Devanbu, Michael Gertz, Chip Martel, and Stuart G. Stubblebine. Authentic third-party data publication. In *IFIP 11.3 Working Conference on Database Security*, pages 101–112, 2000.
8. R. Graubart. The integrity-lock approach to secure database management. In *Proc. of the 1984 IEEE Symposium on Security and Privacy*, page 62, 1984.
9. E. Gudes and M.S. Olivier. Security policies in replicated and autonomous databases. In *Proc. of the IFIP TC11 WG 11.3 Twelfth International Conference on Database Security*, pages 93–107, 1998.
10. H. Guo, Y. Li, A. Liu, and S. Jajodia. A fragile watermarking scheme for detecting malicious modifications of database relations. *Information Sciences*, 176(10):1350–1378, 2006.
11. S. Jajodia and R.S. Sandhu. Toward a multilevel secure relational data model. In M.D. Abrams, S. Jajodia, and H.J. Podell, editors, *Information Security An Integrated Collection of Essays*, pages 460–492. IEEE Computer Society Press, 1995.
12. S. Jajodia, R.S. Sandhu, and B.T. Blaustein. Solutions to the polyinstantiation problem. In M.D. Abrams, S. Jajodia, and H.J. Podell, editors, *Information Security An Integrated Collection of Essays*, pages 493–530. IEEE Computer Society Press, 1995.
13. D. Jonscher and K.R. Dittrich. An approach for building secure database federations. In *Proc. of the 20th VLDB Very Large Data Base Conference*, pages 24–35, 1994.
14. Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 121–132, New York, NY, USA, 2006. ACM Press.
15. C. Meadows. The integrity lock architecture and its application to message systems: Reducing covert channels. In *Proc. of the 1987 IEEE Symposium on Security and Privacy*, page 212, 1987.
16. R.C. Merle. A certified digital signature. In *Proc. of the Advances in Cryptology (CRYPTO'89)*, pages 218–238, 1989.
17. E. Mykletun and G. Tsudik. Aggregation queries in the database-as-a-service model. In *Proc. of the 2006 IFIP 11.3 Working Conference on Database Security*, 2006.
18. Einar Mykletun, Maithili Narasimha, and Gene Tsudik. Authentication and integrity in outsourced databases. *ACM Transactions on Storage (TOS)*, 2(2):107–138, 2006.
19. L. Notargiacomo. Architectures for mls database management systems. In M.D. Abrams, S. Jajodia, and H.J. Podell, editors, *Information Security An Integrated Collection of Essays*, pages 439–459. IEEE Computer Society Press, 1995.
20. HweeHwa Pang, Arpit Jain, Krithi Ramamritham, and Kian-Lee Tan. Verifying completeness of relational query results in data publishing. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 407–418, New York, NY, USA, 2005. ACM Press.
21. A.P. Sheth and J.A. Larson. Federated database system for managing distributed, hetergeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, 1990.
22. Hai Wang and Peng Liu. Modeling and evaluating the survivability of an intrusion tolerant database system. In *ESORICS*, 2006.
23. J. Yang, D. Wijesekera, and S. Jajodia. Subject switching algorithms for access control in federated databases. In *Proc. of 15th IFIP WG11.3 Working Conference on Database and Application Security*, pages 199–204, 2001.