Ensuring Correctness over Untrusted Private Database*

Sarvjeet Singh
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA.
sarvjeet@cs.purdue.edu

Sunil Prabhakar
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA.
sunil@cs.purdue.edu

ABSTRACT

In this paper we address the problem of ensuring the correctness of query results returned by an untrusted private database. The database owns the data and may modify it at any time. The querier is allowed to execute queries over this database; however it may not learn anything more than the result of these legal queries. The querier does not necessarily trust the database and would like the owner to furnish proof that the data has not been modified in response to recent events such as the submission of the query. We develop two metrics that capture the correctness of query answers and propose a range of solutions that provide a trade-off between the degree of exposure of private data, and the overhead of generation and verification of the proof. Our proposed solutions are tested on real data through implementation using PostgreSQL.

1. INTRODUCTION

Consider the case of the food supply chain which is made up of multiple entities: farms, processing plants, distribution centers, warehouses, and retailers. These entities are typically independent, each with its own database that keeps track of its operations. Each entity would like to prevent other entities from learning the details of its operations as this may yield an advantage to a competitor. However, there are instances where it is necessary to provide access to some of this private data in order to enhance public safety, and comply with regulations. For example, if a packet of beef sold at a given store is found to be contaminated, it is necessary to recall all other packages that may also be infected. This entails searching through the private databases of various entities in the supply chain, beginning with the retailer that sold the package that has been found to be contaminated, and working backwards (i.e. to the distributors, packagers, etc.) to locate the source of the problem and then working forwards to track all possibly infected packages. The current solutions to this problem are manual involving paperwork and result in long delays, sometimes days. An automated solution to this problem would essentially remove these long delays and result in almost instantaneous detection.

Automatic detection requires interaction between multiple private databases involved in the food supply chain. To support such queries, the database owners can provide a limited interface into their databases that can be used either by other entities in the supply chain, or a federal organization. There is however, one major problem: since the databases are under the control of individual organizations, there is no guarantee that changes are not made to the database in order to produce misleading results. Providing incorrect results can help an organization protect itself or a partner from blame, and shift the blame to another innocent party. Once again, it is desirable that the external querier be able to obtain proof that the results returned by the database do reflect the correct evaluation of the submitted query over an uncorrupted version of the database.

More generally, consider the case where a law enforcing agency (e.g. the FBI) wants to query a corporate database for the purpose of ensuring compliance with regulations. The entity owning the data may be concerned about privacy and not willing to reveal its entire database to this agency. At the same time, there is an issue of trust. The federal agency cannot blindly trust the corporation to provide uncorrupted results, and would like to receive some proof from the database (which the agency cannot contest) that it has provided the correct result. This proof must handle the case that the database owner could have changed the data once the investigation has begun to mislead the agency.

A similar problem exists with virtually any situation where mutually distrusting entities need to exchange some data while preserving the privacy of the rest of the database. Emerging and recent regulations such as Sarbanes Oxley, and CFR 22 part 11 also impose constraints on the handling of data owned by corporations. Solutions that can provide guarantees of correctness of queries over these databases without exposing the entire contents of the database are highly desirable. In all these examples, it is in the interest of the database owner to share data with business partners or regulatory agencies.

Thus in practice, there is a strong need for providing guarantees of correctness of query results executed over a private database not under the control of the querier. One possible solution to this problem is to involve an external entity that is trusted (willingly or by law), e.g. the USDA in the food

*This work was supported by NSF grant 0534702

EDBT'08, March 25–30, 2008, Nantes, France.

supply example. Each database owner then sends a copy of their database (and updates) to this *trusted third party* which can verify that the queries are executed correctly. In fact, it could execute the queries itself. There are several problems with this solution: 1) This is a very expensive solution with respect to the volume of traffic to the third party and also the requirements of storage at the third party; 2) this third party is a potential weakness in the system – if it is compromised, then too much private data may be exposed; 3) the trusted party is now liable for the privacy of the data – it may be subjected to lawsuits claiming that it has leaked (willingly or unwillingly) private data of one organization to another; and 4) such solutions would be resisted by privacy advocates since there is too much of a "Big Brother" flavor.

To the best of our knowledge, this problem has not been addressed earlier. Existing solutions for tamper proofing audit trails [15], or privacy-preserving database access [9, 1], and authentic third-party data publication [4, 6, 11] are not applicable in this domain as discussed in the related work section. In this paper we propose scalable solutions for the privacy-preserving query result verification problem and develop a number of solutions that provide a tradeoff between the overhead for the owner, the efficiency of the verification, and the degree of exposure of the owner's database in order to prove the correctness of a query. However, it should be pointed out that our solution is directly applicable to the authentic third party data publication and the tamper proofing of audit trail problems too (with no modification whatsoever) with the added advantage that we do not need to trust the owners of these databases.

The specific problem considered in this paper is as follows. We have two entities - the database owner (Bob) and the querier (Alice). The two entities do not entirely trust each other. Bob allows Alice to execute certain queries over his private database. He is willing to reveal as little information as possible to Alice, apart from the results of the query. Alice, on the other hand, is not necessarily confident of the results she receives and may want a *guarantee* from Bob that he has returned the correct results to Alice, without modifying the database (e.g., *after* receiving Alice's query). Figure 1 shows the high-level model of the problem with the possibility of a trusted third party. Alice can ask Bob to commit to his database (while preserving its privacy) before issuing a query. Alternatively, Bob can periodically commit the database. In this paper, the notion of "commit" is that Bob ensures that he can prove the authenticity of these data at a later point in time. The important parameters of the problem are: 1) providing a guarantee for correctness; 2) the overhead on Bob and Alice; and 3) the degree of exposure of Bob's data over and above the query results.

In its full generality this problem is very hard to solve. Note that the database owner can legitimately modify any value in its database (e.g. the number of cans of soup sold today). Thus we would need some means of distinguishing valid modifications from invalid ones. This problem is very hard to solve, and we believe that it is impractical to provide guarantees about dynamic attributes – i.e. those that change over time as part of the operations of the database. We therefore begin by making the following assumptions to limit the scope of the problem. We assume that guarantees can only be provided for data that is not modified after a given point in time (e.g. the number of cans of soup sold yesterday or earlier.). Bob *freezes* the values of some data items

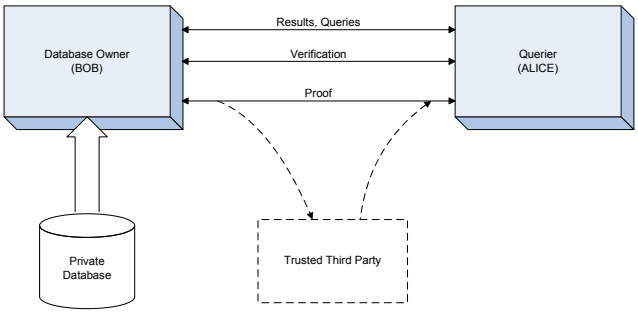

Figure 1: Interaction between the entities.

periodically (e.g. daily or every few hours), after which no modification or deletion of this data is allowed. (To be precise, modifications are allowed, but their authenticity cannot be guaranteed.)

The owner generates a proof that it has frozen the database at regular intervals and ships that proof to an external entity. This could be Alice, or a third party (note that we do not need to fully trust this third party). The only requirement from the third party (if it is used) is that it does not modify the proof. The inclusion of the third party is only a minor issue and does not impact any of the details of our solution. Consequently, throughout this paper we will assume that Alice receives the proof. The contributions of this paper are as follows:

1. Identification and formal statement of the untrusted private database verification problem;

2. A range of solutions that vary in the degree of privacy and the overhead of generating and verifying the proof; and

3. Details of a prototype implementation using PostgreSQL and experimental validation.

The rest of this paper is organized as follows. Section 2 discusses the related work. Section 3 gives a formal definition of the problem and the model used in this paper. Section 4 provides a brief background on some relevant tools that will be used in our solutions. Section 5 presents our proposed solutions. Section 6 discusses implementation details and experimental results are presented in Section 7. Section 8 concludes the paper.

2. RELATED WORK

There has been a recent surge in interest in privacy concerns for databases [1, 9, 2]. Several efforts have focused on generating data mining results over multiple databases while preserving privacy [9, 2, 5]. These protocols are highly tailored to the mining algorithm and are not general purpose. Furthermore, they trust each of the owners of the individual databases to provide correct data. A malicious participant can mislead the group. Consequently, they are inapplicable for the problem studied in this paper.

A recent paper [15] studied the problem of detecting malicious modifications of data by an external intruder. This is achieved through tamper detection of an audit log of the database that records all changes. The database is

treated as an append-only database (in particular, a temporal database). This work does not address privacy concerns of the database and assumes that the database owner is a trusted entity. Our solution (with slight modifications) can be applied to this problem.

Devanbu et al. [4] recently proposed a solution to the problem of authentic third-party database publication. This problem deals with a database owner that wishes to use a third party to host his data. The owner does not entirely trust the third party and would like to ensure that the values stored in the database are not modified by the host. Their solution does not address the problem of privacy of the database. Furthermore, the solution relies upon complete trust of the database owner. Although their solution bears a superficial resemblance to ours (in terms of the use of merkle trees), as discussed in Appendix A it is not applicable to our problem. On the other hand, our solution can be directly applied to the problem of authentic third-party database publication. Our solution is much more efficient in terms of storage and computation as compared to the solution in [4]. The solution in [4] needs to specially address each join and selection that may be executed by the querier. Our solution does not suffer from this limitation. A related paper [3] addresses similar issues for XML.

The problem of private outsourcing of a database has also been studied [7] wherein a semi-trusted third party is used to host a database. To protect privacy, the data is encrypted by the owner and stored only in encrypted format at the host. This introduces challenges for efficient execution of queries and creation of indexes [8].

To the best of our knowledge the problem of ensuring correctness over untrusted private database has not been addressed earlier.

3. ASSUMPTIONS AND MODEL

In this section, we describe the assumptions and trust model for the entities involved in the protocol.

3.1 Querier (Alice)

The database is located in a remote location over which Alice has no control. The database owner (Bob) determines what types of queries are allowed to be executed. In order to prove that the results of allowed queries are correct, the database periodically commits ("freezes") its current state. Subsequent queries must be guaranteed to return results computed over this committed state. Any change to the committed values should be detected. Alice may demand proof of correctness for a given result and ask Bob to commit the database before submitting a query. Correctness of the results is only guaranteed over the committed data.

In case of an update over the committed data, Alice will either detect this modification or Bob should send Alice the value of the tuple at the time it was frozen. The application semantics determines which of the above actions is more appropriate. In either case, this does not restrict the applicability of the results of this paper.

3.2 Database owner (Bob)

The database owner controls the private database. He has unconditional read and write access to the database. He can intercept all the queries posed to the database and their results, and may even modify the results. In order to prove the correctness of the query results, Bob explicitly generates a proof by freezing the data. Note that since Alice does not trust Bob, some proof of the freezing must be shipped outside of the database where it cannot be modified by Bob. This can be achieved by either sending some information to Alice, to a semi-trusted third party, or by using an independent authentication authority. We assume that whatever data is frozen by Bob is correct. In practical settings, this is the same as recording entries in an accounts ledger – since the entries can be audited, the owner is discouraged from recording incorrect data. Similarly, for the current problem, there needs to be in place a mechanism that enables a random audit of frozen data in order to ensure that Bob does not freeze incorrect data. Once a data item is frozen, the protocol should not allow him to modify it – i.e. Bob's hands should be tied with respect to the frozen data. This is a reasonable assumption given that the event which causes Bob to become malicious and skew the results to his favor does not happen before Bob generates and sends the proof. Nevertheless, we do not trust Bob to follow the algorithm correctly. He may try to find loopholes in the protocol to generate a proof that does not tie his hands completely (e.g. as discussed in Appendix A, instead of freezing a tuple to one value, Bob may try to freeze it in such a way that allows him to report two or more possible values for that tuple).

There is no restriction on how the query results can be modified. Further, Bob is concerned about the privacy of the database. He wishes to reveal only a minimal amount of information to Alice, in addition to the query results.

3.3 Definition of Correctness

There are two aspects of correctness of query results. We now present two requirements for the correctness of query results returned by a private database. Without loss of generality, we can denote the result tuples of a SPJ query (queries involving only select, projects and joins) as:

$$R_i = R_{i_1}^{T_1} \| R_{i_2}^{T_2} \ldots \| R_{i_q}^{T_q}$$

where $\|$ denotes concatenation, R_i is the i^{th} tuple of the result and $R_{i_j}^{T_j}$ refers to the value of projected attributes of tuple i_j of table T_j. We divide the correctness requirements into α- and β-correctness as defined below:

α-correctness: This refers to the *correctness of the result values*, i.e. the validity of the tuple values returned by the query. Formally, this implies that the values returned, $R_{i_j}^{T_j}$, must match the values that were frozen when the proof was generated.

β-correctness: This refers to the *correctness of query execution*. It implies that joins and selections were performed correctly and the i^{th} tuple of the result should in fact consist of data from tuples $i_1, i_2, \ldots i_q$ of tables $T_1, T_2, \ldots T_q$ respectively. This definition also checks for absence of valid tuples from the result set.

To understand β-correctness intuitively, it is helpful to picture the database table as collection of tuple ids only. E.g. for a selection query β-correctness only tests whether the correct tuple ids are part of the result – it does not check whether the data corresponding to these tuple ids is unmodified. In case of joins, it checks that correct pairs of tuple ids from two tables are in the result. While β-correctness does not check the tuple contents, α-correctness ties the tuple id to its contents, and ensures that they are not tampered.

R	
A	B
a_1	b_1
a_2	b_2

S		
A	C	D
a_1	c_1	d_1
a_1	c_3	d_2
a_2	c_2	d_1

Figure 2: Example tables for Query Correctness.

Consider two relations, $R(A, B)$ and $S(A, C, D)$, and the query $\pi_{B,C}(\sigma_{S.D=d_1}(R \bowtie_{R.A=S.A} S))$. Example instances of the two relations are shown in Figure 2. The correct result of the query should be the tuples: $\{< b_1, c_1 >, < b_2, c_2 >\}$. α-correctness requires the database to prove beyond any doubt that the tuples in the result are indeed committed values. For this specific example, this amounts to proving that b_1 and b_2 are part of some frozen tuples in table R (and similarly for c_1 and c_2). β-correctness requires that the selections and joins are correctly performed and all the resulting tuples are returned. For example, if the database only returns: $\{< b_1, c_1 >\}$ (an incomplete result), or $\{< b_1, c_1 >, < b_2, c_2 >, < b_1, c_3 >\}$ (incorrect selection) or $\{< b_1, c_1 >, < b_2, c_2 >, < b_1, c_2 >\}$ (incorrect join) then Alice should be able to discover this inconsistency. Note that all these results are α-correct as the values returned do belong to some tuple in the table. The β-correctness proof will verify that the tuples containing the values returned (as certified by α-correctness) are in fact the result of the query.

These two definitions are independent of each other and together imply the correctness of query results. As described later, for some specific cases we may not need the β-correctness requirement in order to verify the correctness of query results.

3.4 Query

The solutions proposed in this paper can guarantee α-correctness for any arbitrary query over the database. However, for the case of the more challenging β-correctness, we need to limit the types of queries to the form:

$$\pi_{p_1,p_2,\ldots,p_m}(\sigma_{s_1=a_1,s_2=a_2,\ldots,s_n=a_n}(T_1 \bowtie T_2 \bowtie \ldots T_q))$$

where $p_1, p_2 \ldots p_m, s_1, s_2, \ldots s_n$ are the attributes of the tables $T_1, T_2, \ldots, T_q$ of the database. The joins between the tables are assumed to be *equality* joins.

Our approach can also prove correctness for queries whose results are essentially derived from queries of the type shown above (e.g. aggregate queries). By proving the correctness of the underlying query, we can show that the derived query was also correctly evaluated. For example, we can prove correctness for a query that computes an aggregate over a set of tuples generated by a query of the type shown above. However, in order to prove its authenticity, we would have to expose the values of the underlying tuples (i.e. we can not ensure the privacy of underlying query and expose only the aggregate).

4. PRELIMINARIES

This paper employs two standard data security tools: *strong one-way hash functions* [13] and *Merkle Trees* [10]. We provide a brief description of these tools before discussing the proposed solutions.

4.1 One-Way Hashing

A one-way hash is a function, h, that takes as input a data item, x and produces as output the hash of the data item, $y = h(x)$. Important requirements for a one-way hash function are:

1. Given a hash value, y, and the details of the hashing function h, it is very difficult to find x such that $h(x) = y$. In other words, given the hash of a data item it is hard to work back and determine the data value that generated this hash value.

2. The probability that $h(x) = h(y)$ for $x \neq y$ is very low. That is, it is very unlikely that two different input values will yield the same hash.

Therefore, given a hashed value, y, it is virtually impossible to discover any data value that yields y as its hash value. If a value x is known, then it is virtually impossible to generate a second value z such that $h(z) = h(x)$. Thus, given a hash value y, it is possible to determine x such that $y = h(x)$ only if x is already known (or one gets extremely lucky). There are many well-known and commonly used strong one-way hash functions, such as SHA-256.

Another important class of hash functions are *cryptographically secure keyed hash* functions, denoted by h_k. These work in much the same way as a one-way hash function, but they take as an additional input, a key, k. With such a function, given x it is not possible to determine $h_k(x)$ even if we know the hash function unless the key k is also known.

4.2 Merkle Trees

A Merkle tree [10] is a binary tree (not necessarily complete) with labeled nodes. The labels are binary strings of length k. Let $\Phi(n)$ represent the label of node n, thus $\Phi(n) \in \{0,1\}^k$. The label for each internal node of the tree, n_{parent}, with children, n_{left} and n_{right}, is derived from the labels of its children using a hash function, h as:

$$\Phi(n_{parent}) = h(\Phi(n_{left}) \parallel \Phi(n_{right}))$$

The function h is a candidate one-way function such as SHA-256. The above equation gives assignment of Φ for internal nodes. For leaf nodes, Φ is usually chosen depending upon the application of merkle trees. For example, Φ for a leaf can be the hash of a small part of a document whose integrity we want to establish.

Merkle trees are used to establish the authenticity of the leaf node labels. This is achieved by simply publishing the value of the root's label. Publishing in this context, refers to recording the value in a manner that cannot be modified later. This can be achieved by printing this value in a newspaper, or using an authentication service. With this published and unmodifiable value, one can now establish that the value for any of the leaf labels has not been modified after the publication of the root's label. Consider for example, the merkle tree shown in Figure 3. In this example, the labels for the leaf nodes correspond to the hash values of data items whose authenticity we would like to establish. In order to prove that the value of the data item x (shown as shaded box in the figure) has not been modified, we simply need to provide the value of x (the hash function for the tree is well known), and the labels of the sibling nodes on the path from x to the root. These nodes are shown as black circles in the figure. This path is called the *authentication*

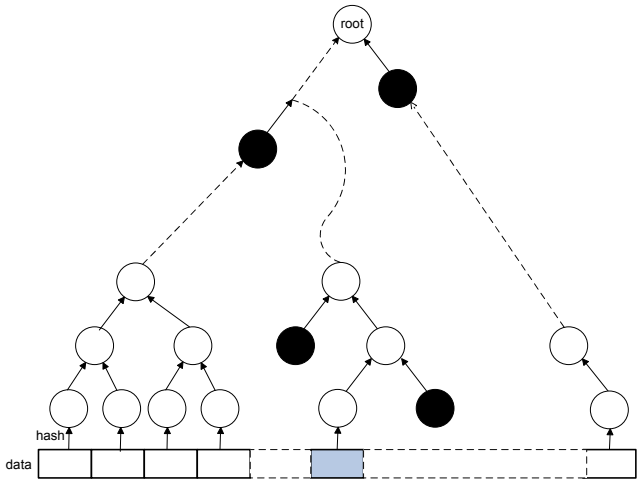

Figure 3: An example Merkle tree. The black nodes form the authentication path of shaded data block.

path for x. To verify the authenticity claim, one computes the labels of the nodes along the path using the definition of Φ for internal nodes. If the computed label for the root matches the previously published value then the value of x is authenticated.

To see why this is so, consider what happens if the value of x has indeed been modified. Given the new value of x, one would have to create labels for each sibling such that after repeated computations of the parent labels, the final hash value is the same as the one that was generated earlier. By the very nature of one-way hash functions, this is extremely difficult to do, thus the labels cannot have been modified.

5. PROPOSED SOLUTIONS

In this section, we discuss several solutions with various degrees of privacy, computational complexity and data bandwidth. For the moment, we assume that the query contains only selection operators and we are only verifying the α-correctness of the query results in each of these solutions. We will discuss the general case later. For each solution, we discuss the **Proof of integrity** that must be shipped by Bob whenever he freezes the data, and the **Verification** steps that Alice must carry out in order to establish the authenticity of query results in case she suspects foul play. We begin with a very simple solution to illustrate the nature of the problem. In the rest of the paper, the term database refers to the part of Bob's private database that is frozen.

In all of these solutions, the hash function h is assumed to be public. The hash function implementations normally take a sequence of bytes as input and returns an output of fixed length. Using this, we can define the function h on any chunk of data in the database. For an attribute value, hash is simply the hash of bytes representing that value. For bigger units like a tuple or (ordered) set of attributes, h is defined as hash of concatenation of all the attributes making up that tuple or set respectively.

In some solutions we assume that each tuple is identified uniquely. This can be achieved by using RIDs, or for simplicity, generating an explicit ID within the database.

Solution 1.
Proof of integrity: Bob computes the hash of the entire database and sends it to Alice.
Verification: If challenged by Alice to prove the authenticity of the results of a query, Bob ships the entire contents of the database to her. Alice can easily i) verify that the result values are indeed part of this database; and ii) compute the hash of this database and verify if the overall hash match the earlier proof. If these two values match, then Bob must have sent the same database that was used to generate the earlier proof (given the difficulty of finding two numbers that hash to the same value) and has not modified it since.

This approach has the advantage that the database owner needs to send only one number (the hash value) to Alice. However, this is obviously a very bad solution since it violates the privacy of Bob's database as he has to reveal the entire database when challenged. The bandwidth required for verification is also huge as the entire database needs to be shipped for verification.

Solution 2.
Proof of integrity: We consider a strong hash function h such as SHA-256. For each tuple r_i in the database, Bob generates $h_i = h(r_i)$ and ships (i, h_i) $\forall i$ to Alice.
Verification: As the result of a query, Alice gets back (i, r_i) for i in the result set. She can easily hash the result tuples and verify their integrity. In other words, no extra data is needed for Alice to verify the query results.

The i used in this solution can be something which uniquely identifies a tuple in a database table such as RID. We can use other approaches which do not use RIDs such as sending the individual hashes in sorted order of hash value, but using RIDs makes the exposition easier. This approach respects the privacy of Bob's database. There is no communication between Bob and Alice during the verification phase as she has all the information needed to verify the results. This approach can very easily be implemented by maintaining a separate field for the hash along with each tuple. The hash is updated whenever a tuple is inserted or modified. Although this increases privacy, it is not very practical as the size of the proof is proportional to the size of database. The size of the proof, rather than the size of the verification, is the major concern as verification may be rare (only when challenged by the querier) as compared to sending the proof of integrity (which may be periodic).

Solution 3.
Proof of integrity: Let the tuples to be frozen be $r_1, r_2, \ldots, r_n$. Bob computes the hash of individual tuples $r_1 \ldots r_n$ with hash function h to generate $a_1 \ldots a_n$, where $a_i = h(r_i)$. Next, he computes the hash of $a_1 \ldots a_n$ to generate the final proof which he ships to the querier. Thus the proof is a hash-tree (of height 1) over the hashes of individual tuples.
Verification: Let the result set S be set of all tuple numbers returned by the query. i.e., the result of the query is a set $\{(j, s_j)\}$, $j \in S$. In order to verify this result, Alice asks Bob for (i, a_i), $\forall i \notin S$. She computes the hash of each result $b_j = h(s_j)$, $j \in S$. Finally, she computes the hash over b_j and a_j hash values received from Bob. Hence, Alice computes: $h(c_1||c_2||\ldots||c_n)$ where $c_i = b_i$ if $i \in S$, and $c_i = a_i$, otherwise. If this overall hash value equals the proof sent earlier by Bob, then Alice is convinced that the result values were indeed part of the frozen database.

Similar to the previous solution, this approach also respects the privacy of the database. But now, the proof size is reduced to just one number (the final hash) at the cost of a greatly increased verification size. This overhead associated with hashing is comparable to that in the previous approach. Assuming the result set is small, the size of the verification is proportional to the size of the database. Unless verification phases are rare and bandwidth for verification is not a concern, this approach is not practical. However, this approach may be useful in situations where the result set is large.

Solution 4.
Proof of integrity: This approach uses merkle tree to reduce the size of the verification from $O(N)$ to $O(\log N)$, where N is the size of the database. Bob computes a merkle tree as described in Section 4. The definition of merkle trees gives the assignment of Φ for internal nodes. For leaf node l_i, $\Phi(l_i) = h(r_i)$, where r_i is the i^{th} tuple in the database. Bob sends $\Phi(root)$ as proof of integrity to Alice.
Verification: For verification, Bob sends authentication path for the result tuples that need to be verified to Alice. The length of such a path is equal to the height of the tree, which is proportional to $O(\log N)$. Alice computes the hash over the result that she has received and the hash values along the authentication path supplied by Bob. If the overall hash generated by Alice using the result values and the authentication path hashes matches the proof sent earlier, all result tuples are authenticated.

This approach preserves database privacy as it reveals only the hashes of tuples that are not part of the result. Thus Alice learns nothing about the values of the other data items. While the size of the proof is the same as in Solution 3 (one single hash), the size of the verification is greatly reduced from $O(N)$ to $O(\log N)$. The computational complexity is proportional to the number of nodes (both leaf and internal) of the merkle tree. For a binary tree with N leaves, the total number of nodes is $2N - 1$. Therefore, the computational complexity of this approach is only twice as much as the previous solution.

Table 1 summarizes the above solutions and their properties. These solutions were developed for proving only the α-correctness of the results with the assumption that no projections are performed and the result set size S is small compared to the database size N. However, as explained below, they can be extended to more general queries.

5.1 α-Correctness for General Queries

As discussed above, Solution 4 is superior to the others for verifying the α-correctness. We next describe this approach in detail for verifying the α-correctness for general queries.

5.1.1 Granularity of Hashing

In all of the above solutions, we have assumed that the granularity of hashing is at the tuple level. The granularity directly determines how much information must be revealed during verification. Consider a table that has attributes $A = \{a_1, a_2, \ldots, a_n\}$ and the query returns an attribute set $B \subset A$. During verification, for each result tuple, the values of all $A - B$ attributes must be revealed for the querier to verify the α-correctness of the results (as the hash of a complete tuple is needed). This increases the data bandwidth required for verification and reduces the privacy of the database.

To overcome this problem, the granularity of hashing can be adjusted so that no additional information is revealed during verification. For the above example query, Bob would hash B and $A - B$ separately, i.e. $\Phi(\text{tuple}) = hash(\Phi(B) \parallel \Phi(A - B))$, where $\Phi(S) = h(S)$ for any $S \subset A$. Now, in order to verify a result tuple which has only B attributes, we do not need to reveal the other $A - B$ attributes – only the hash of the $A - B$ attributes for that tuple is sent to Alice. This enables the database to mask private attributes that are not part of the result.

With a finer granularity of hashing, we can avoid violating the privacy of the database when the query result does not include all the attributes of a table. However, it has a price associated with it. A finer granularity of hashing will increase the authentication data needed for verifying a data item. At the same time, switching to a much finer granularity will increase the time needed to generate the proof and verify the results (as described in Section 7).

Consequently, a judicious choice of the granularity for hashing needs to be made in order to balance the cost of generating a proof and the degree of exposure of private data during verification. Given a set of queries that are allowed to be executed over a relation, let $R_i \subseteq T$ be the set of attributes queried by query i. The granularity of hashing for a relation T, $G_T = \{A_1, A_2, \ldots A_m\}, A_j \subseteq T$, is decided such that

1. $A_i \bigcap A_j = \phi$, $\forall i, j \in \{1, 2, \ldots m\}, i \neq j$; and
2. $\forall i, \exists J_i \subseteq \{1, 2, \ldots m\}$, such that $R_i = \bigcup_{j \in J_i} A_j$

The first condition ensures that we do not include an attribute in more than one hash, which would incur costs in terms of efficiency and space. The second condition implies that we always have a subset of hashed attributes that will cover all the attributes projected by any query. In absence of knowledge about which subsets of attributes are likely to be queried, we can treat each attribute as one of the A_i sets. This allows us to provide maximal privacy for all the attributes.

Solution 5α.
We now present the final solution for proving α-correctness for arbitrary SPJ queries, and also address an important attack for domains with small cardinality. We redefine $\Phi(l_i)$ for a leaf node l_i of the merkle tree as

$$\Phi(l_i) = hash(tuple\ id \| \Phi(A_1^i) \| \Phi(A_2^i) \| \ldots \| \Phi(A_m^i))$$

where A_j^i refers to the value of attribute(s) A_j for i^{th} tuple and $\Phi(A_j^i) = hash(A_j^i)$. With this definition, the leaf nodes of the merkle tree are no longer labeled $h(r_i)$ corresponding to tuple r_i. Instead, the hash values of the various sets of attributes values $h(A_j^i)$ form the new leaf level. The next level up contains one node for each tuple r_i of the database. Its label is the hash of the concatenation of labels for the m leaves of tuple r_i: $h(A_j^i), j = 1..m$ and the tuple_id for r_i. This grouping together of the hash of attributes of one tuple simplifies the implementation and allows Alice to easily verify whether two attributes reported (by Bob) as part of one tuple in fact belong to the same tuple and not to two different tuples.

5.1.2 Handling attributes with small domains

The security of hash functions depends on the assumption that the domain of the hash function is large. If the

	Solution 1	Solution 2	Solution 3	Solution 4
Size of Proof	O(1)	O(N)	O(1)	O(1)
Cost of Proof	O(N)	O(N)	O(N)	O(N)
Size of Verification	O(N)	0	O(N)	O($\log N$)
Cost of Verification	O(N)	O(S)	O(N)	O($\log N$)
Exposure of Data	complete exposure	no exposure	no exposure	no exposure

Table 1: Summary of various approaches (N is the database size and S is the size of result set)

domain is small (e.g. age) a simple dictionary attack will allow the querier to deduce the attribute values from their hashes. The querier simply hashes each possible value (e.g. every age from 1 to 120) to produce the corresponding hash. Comparing these hash values with the hash values of private fields allows the querier to determine the value of the field. We solve this problem by generating the hash of the data value concatenated with another value not known to Alice. This secret value is called "salt".

We redefine $\Phi(A_j^i)$ for attributes with small domain cardinality to be:

$$\Phi(A_j^i) = hash(A_j^i \| S_{i,j})$$

$$S_{i,j} = h_k(table\ id \| i \| j)$$

where $S_{i,j}$, is the salt for attribute j of tuple i, and h_k is a cryptographically secure keyed hash function. The key k is kept secret by the database owner.

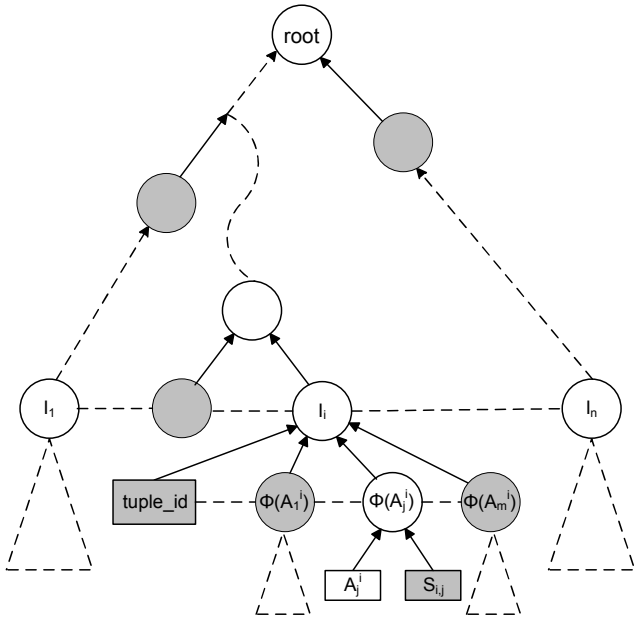

Figure 4: Hash tree for α-correctness. The shaded nodes forms the authentication path of data item A_j^i

The size of the salt $S_{i,j}$ must be large enough to make dictionary attacks computationally very difficult for Alice. $S_{i,j}$ must be revealed to Alice for verification of A_j^i. Figure 4 shows a hash tree and the complete authentication path for a data item A_j^i. In order to verify an attribute value, Alice will be given both the value of the data (A_j^i) and the value of the corresponding salt ($S_{i,j}$) used to generate the proof. Since the salt is released for verification, we cannot use a single value for all data items. Managing different salt values for each data item can be quite cumbersome (and consume storage) for Bob. To avoid both these problems, we define the salt in such a way that it can be easily derived by using a keyed hash function h_k.

As described in Section 3, the α-correctness of results alone does not establish the correctness of query execution. However, in some special cases, verifying only the α-correctness suffices. For example, if the total number of tuples that must be returned by a query is known and all join and selection attributes are retained in the result, then the correctness of query results is completely determined by its α-correctness.

Note that we can verify the α-correctness of the results for any general query. The restriction on queries given in Section 3 is necessary only for verifying β-correctness.

5.2 Verifying β-correctness

In the previous section, we discussed how we can use merkle trees to establish the α-correctness of the results which ensures that all the tuples returned by the query were indeed those frozen by the database. In this section we propose a solution for establishing β-correctness that ensures that all the results for a query were sent to the querier. For this purpose, we need to ensure that the query engine performs all the joins and selections correctly. This is challenging because Bob, the database owner, is not willing to reveal the entire database due to privacy concerns. Merkle trees can be very bandwidth efficient for α-correctness as discussed previously, but they cannot be used for β-correctness. This is due to the fact that we need information about the entire database (as opposed to authenticating a part of the database) for verifying the β-correctness.

As discussed earlier (Section 3), for β-correctness we will assume that the queries only contain equality joins. We use a modified version of the hash tree described in Solution 3. First, we need to define the granularity of hashing to prevent violation of privacy similar to what was proposed above for α-correctness in Solution 5α.

Without loss of generality, let us assume that the query is

$$q_i = \pi_{P_i}(\sigma_{S_i}(T_{i_1} \bowtie_{J_i} T_{i_2}))$$

where $P_i \subseteq T_{i_1} \cup T_{i_2}$ are projected attributes, $S_i \subseteq T_{i_1} \cup T_{i_2}$ are selection attributes and $J_i \subseteq T_{i_1}$ (and $J_i \subseteq T_{i_2}$) are join attributes. Note that the selections and joins are both based on equality (Section 3). To prove the β-correctness of results without revealing any additional information we enforce the following conditions on the granularity $G_T = \{A_1, A_2, \ldots, A_m\}, A_j \subseteq T$,

1. $A_i \bigcap A_j = \phi$, $\forall i, j \in \{1, 2, \ldots m\}, i \neq j$; and

2. $\forall i, \exists K_i \subseteq \{1, 2, \ldots m\}$ and $\exists L_i \subseteq \{1, 2, \ldots m\}$, such that $J_i = \bigcup_{k \in K_i} A_k$ and $S_i = \bigcup_{l \in L_i} A_l$

In other words, we identify maximal, disjoint subsets of attributes of each relation T such that we can obtain each of the sets required for selections and joins over this relation for every query. Note that we can always satisfy these conditions by picking each set A_i consisting of a single attribute.

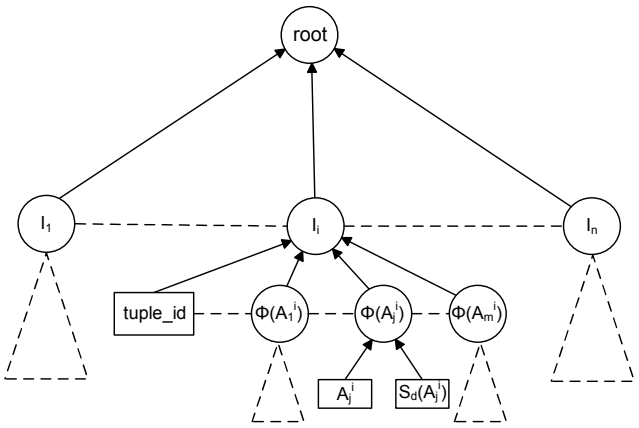

Figure 5: Hash tree for β-correctness

Solution 5β

The hash tree over a database table for proving β-correctness is defined as follows:

$$\Phi(root) = hash(\Phi(l_1) \| \Phi(l_2) \| \ldots \| \Phi(l_n))$$

$$\Phi(l_i) = hash(tuple\ id \| \Phi(A_1^i) \| \Phi(A_2^i) \| \ldots \| \Phi(A_m^i))$$

$$\Phi(A_j^i) = hash(A_j^i \| S_{i,j})$$

$$S_{i,j} = S_d(A_j^i)$$

where, $S_d(A_j^i)$ is the digital signature of Bob on value A_j^i with the private key d. The corresponding public key e is known to Alice. As described later, using a simple keyed hash as the salt (similar to our solution for α-correctness) is not sufficient to ensure the β correctness. Figure 5 shows the hash tree described above. The nodes $l_1, l_2, \ldots, l_n$ correspond to each tuple of the table.

We have included *tuple id* in the definition of $\Phi(l_i)$ similar to the definition of $\Phi(l_i)$ for α-correctness. Keeping this unique identifier along with the tuples in both (α and β) trees helps Alice to verify that the authentication paths provided by Bob for α and β proofs are for the same data item. If we remove *tuple id* from the above definition, it is possible for Bob to fool Alice, by providing an authentication path for one tuple for α-proof verification and path for a different tuple for β-proof verification (possibly having one or more attribute values equal to the first tuple). Without the unique tuple ids, Alice will have no way to verify if the two proofs point to the same tuples. Thus, tuple ids serve to link the two proofs together.

During the proof sending phase, the database owner, Bob, only sends $\Phi(root)$ for each table to Alice. If Alice requests verification after executing a query, Bob sends (for all tables involved in the query) $\Phi(A_j^i)$, $\forall i, j$ and $S_{i,j}$ for all the A_j attributes that are involved in a selection.

Alice checks if the value of $\Phi(root)$ sent for each table matches the value calculated by using hash of all $\Phi(A_j^i)$ sent by Bob. If that succeeds, she only needs to prove that the joins and selections are performed correctly by the database to prove β-correctness.

To verify if selections were performed correctly, Alice checks the following condition for all selections of the form $A_j = a_j$ (where a_j is a constant): $\forall i$, if $\Phi(A_j^i) = hash(a_j \| S_{i,j})$ then tuple i is present in the result. Additionally, she needs to verify that for the tuples returned in response to the query, the signature $S_{i,j}$ matches the signature of Bob. This can be verified by using Bob's public key e.

If the value of salt $S_{i,j}$ is chosen to be $h_k(A_j^i)$ (similar to α-correctness), it is possible for Bob to actually return two (or more) different sets of answers in response to a single selection query and provide a correctness proof for each of them. This is against our assumptions where we want Bob to commit to exactly one state of the database. This attack is possible because Bob may choose to use different keys for the hash function instead of using just one key for all the tuples in the database (as required by the protocol). Defining $S_{i,j}$ as Bob's digital signature on attribute value and later verifying it at Alice's end prevents this problem.

To verify if the join was performed correctly, Alice checks the following for attributes (say A_j) of joined tables: $\forall i_1, i_2 = 1, 2, \ldots, n$ if $\Phi(A_j^{i_1}) = \Phi(A_j^{i_2})$ then tuples i_1 and i_2 are included in the result.

Our solution for α and β correctness requires Bob to send $2|D|$ (root nodes of two trees per table) hashes during the proof sending phase, where $|D|$ is the number of tables in the database. These hashes can be easily combined into one hash to reduce the proof size at the cost of one extra level in all the authentication paths.

6. IMPLEMENTATION

We have implemented our proposed solution in PostgreSQL [12]. The algorithms for generating hash trees over the database were implemented in PL/pgSQL. We used the implementation of the hash function SHA-1 from OpenSSL crypto library [14] for our experiments. The database has been extended to allow the owner to freeze the data values by generating proof, and to support authentication of query results. Whenever the querier wants the database to freeze its data, it issues a **send_proof** command to the database. On receiving the **send_proof** command the database sends a single hash value to the querier. If the querier wants to verify the results, it issues a **send_verify** command to the database. On receipt of **send_verify** the database returns the authentication paths for all the tuples in the query result being verified. For simplicity, we have implemented an append-only database. But it is relatively easy to extend it for general databases. In particular, we have to write an update trigger to update the hash tree whenever data in a node is modified. We have implemented Solution 5α and 5β.

The hash trees can be generated over the tuples when the **send_proof** command is received by the database. However, this approach will have a large overhead on receipt of **send_proof**. A better alternative will be to generate the tree as tuples are added to the database. This approach distributes the load evenly during the database updates.

For generating and storing the tree, we add the following

tables to the database. We create a new table – *hash_tree* (*node_id*, *phi*, *parent*) to store the generated hash tree. In this tree representation, the ordering of the child nodes is not explicit. For a given parent node, the child nodes are implicitly ordered by increasing node ids. For example, the child with lowest node id is considered as the leftmost child. A global counter (*count*) is used to obtain new *node_id* values. The schema of each original table in the database is modified to add a new attribute called *node_id*. A temporary table, *height_table(height, node_id)* is also used.

1. Let h=hash(table(tuple_id))
2. INSERT INTO hash_tree (count++, h, NULL)
3. current_node=count -1
4. UPDATE table SET node_id = current_node WHERE key=tuple_id
5. current_height=0
6. Let temp = SELECT node_id FROM height_table WHERE height = current_height
7. **If** temp ≠ NULL
   (a) h=hash(hash_tree(temp)‖ hash_tree(current_node))
   (b) INSERT INTO hash_tree (count++, h, NULL)
   (c) UPDATE hash_tree SET parent_id = (count - 1) WHERE (node_id = temp OR node_id = current_node)
   (d) DELETE FROM height_table WHERE height = current_height
   (e) current_height++; current_node = count -1
   (f) GOTO 6
8. **else** INSERT INTO height_table (current_height, current_node)

Figure 6: Pseudo code for add_node(tuple_id, table). *tuple_id* is the new tuple that is to be added to the tree and *table* is the database table in which tuple_id resides

Upon insertion of a new tuple into the database, the **add_node** algorithm (refer Figure 6) is executed (by means of a trigger). The **add_node** algorithm constantly maintains a forest of partial merkle trees over the rows of the database. The roots of such trees are stored in a temporary table, height_table. On receipt of a **send_proof** (Figure 7) request, the database merges these partial trees into one tree and returns the root of this tree as the proof.

After the **send_proof** algorithm is executed, the database has a complete merkle tree over the database. Given this merkle tree, the **send_verify** algorithm is easy. For each tuple in the result, the database returns an authentication path with the help of the **hash_tree** table. Note that the authentication paths of these tuples will overlap and this can be used to further reduce the size of verification data. The querier on receipt of this authentication path calculates the hashes and traces back the authentication path to the root and checks if the final hash matches the value sent to it earlier by the **send_proof** algorithm.

The implementation of our solution to prove β-correctness is relatively easy because of the simplicity of the hash tree structure defined for it.

1. **For** temp = SELECT node_id FROM height_table ORDER BY height do
   (a) If this is the first iteration then set node=temp and jump to next iteration
   (b) Let h = hash(hash_tree(temp).phi ‖ hash_tree(node).phi)
   (c) INSERT INTO hash_tree (count++,h,NULL)
   (d) UPDATE hash_tree SET parent = count-1 WHERE (node_id = node OR node_id = temp)
   (e) node = count-1
2. **return** hash_tree(node).phi

Figure 7: Pseudo code for send_proof

6.1 Efficiency

A number of optimizations are possible for the algorithm presented above to reduce the space overhead associated with maintaining the merkle trees on the database side. These optimizations come at the cost of additional processing required by the database during the proof verification phase. This may be desirable if we assume that the verification phase is rare (i.e. only when the querier suspects foul play and asks the database to send authentication paths). To achieve this, we note that the entire trees are not required in the **add_node** and **send_proof** algorithms. We only need the root of the partial trees generated. We can easily ignore the *hash_tree* table and store the hashes of trees at height h in the *height_table* (note that there can be only one tree at height h). This reduces the space overhead tremendously, but a ready-made merkle tree over the database will not be available when the **send_verify** command is received. Thus, before executing this algorithm the merkle tree that was used to generate the proof will need to be recreated. This increases the time complexity of the send verification algorithm.

Further, we did not consider the situation where the querier requires a proof periodically. In this case, we can introduce optimizations with various space-time tradeoffs on both the database side and the querier side. (1) Bob can either maintain one *large* merkle tree over the entire database and send the root of this tree whenever he needs to send the proof. In this approach, the size of the merkle tree will gradually increase. This will increase the amount of data that is exchanged between the two parties for proof sending and verification; or (2) The other approach is to maintain many merkle trees – one merkle tree for each proof that Bob sends to Alice. Each merkle tree only covers the data that is added *after* the last proof was sent. This keeps the size of the merkle trees very small. The disadvantage is that Alice has to keep all the previous hashes corresponding to data on which she can possibly run a query at any future time.

7. EXPERIMENTAL RESULTS

The experiments were performed on a SUN SPARC workstation with 1GB of RAM. Data from a Walmart data warehouse was used for all the experiments.

7.1 Overhead of proof generation

In this experiment, the overhead due to proof generation was studied. We generated a hash-tree for α-correctness verification over different table sizes. The schema of the ta-

ble used for this experiment is **sales(item_id, units_sold)**. First, tuples were inserted into the database. Then a hash tree over these tuples was generated. Tuple level granularity was used while generating the hash tree. For comparison, the time taken by the database in inserting the tuples is also plotted along with the cost of α-correctness proof generation. Figure 8 show the results of this experiment for different table sizes. The x-axis gives the number of tuples added to the relation, and the y-axis gives the actual time taken for the insertions in seconds. This time was measured using the UNIX time utility. The graph shows the time required to insert a tuple and also the time required to make necessary changes to the trees. Note that this implementation uses triggers for tree creation which may not be the most efficient implementation. However, the cost of tree maintenance is on the same order of magnitude as the cost of a single tuple insertion.

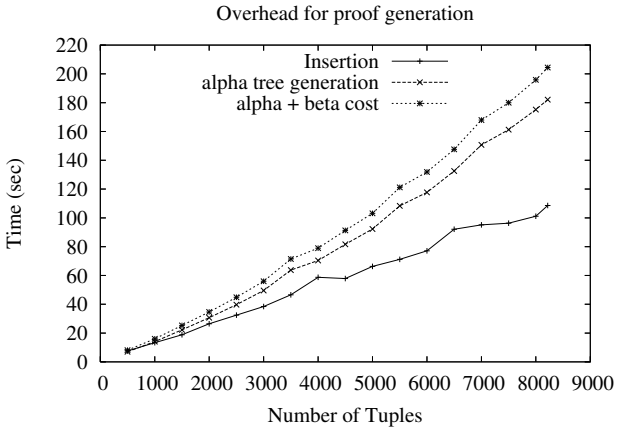

Figure 8: Overhead of proof generation

The results establish that the cost of generating the α-proof hash tree increases linearly with the size of database. For small table sizes, this cost is roughly equal to the cost incurred in inserting the tuples in the database. Both these costs increase linearly and the tree generation cost slope is roughly twice of that of insertion cost.

For β-correctness, the experiment was run on the same table described earlier. We expect the overhead of generating this hash tree to be low. This is because the hash of the data tuples is already available to us (as they are calculated during α-correctness proof generation). Secondly, because of the single level structure of this hash tree, we only need to compute one additional hash (over all leaves) to obtain the root hash. Figure 8 presents the results of this experiment.

Once again, we see an almost linear relationship between the number of tuples in the table and the time required to process the insertions. The overhead for *beta*-correctness proof tree is indeed very small compared to the time required for generating the α-correctness proof tree.

7.2 Amortizing proof generation cost

In the previous experiments, the hash tree generation was done lazily – after inserting a large number of tuples. It resulted in a significant overhead whenever the database was frozen. As mentioned in Section 6, the hash tree generation can also be done eagerly. This results in a cost penalty during insertions to the database, but the freezing of the database is quite fast. In a way, the eager approach amortizes the cost of hash tree generation over insertions.

This experiment compares the two approaches in terms of α-correctness proof generation. The hashing was done at the granularity of tuples. The database was periodically frozen (after insertion of 2000 tuples). Figure 9 shows the time taken by the two approaches as tuples were inserted into the table. In case of lazy computation, the flat lines show the insertion cost. After 2000 insertions, the **send_proof** command is sent to the database. After receiving this command the database starts computing the hash tree. This results in a significant overhead as shown by the vertical lines. On the other hand, in case of eager evaluation, each tuple insertion triggers a function that partially computes the hash tree. This allows the database to quickly freeze itself, whenever a **send_proof** command is received.

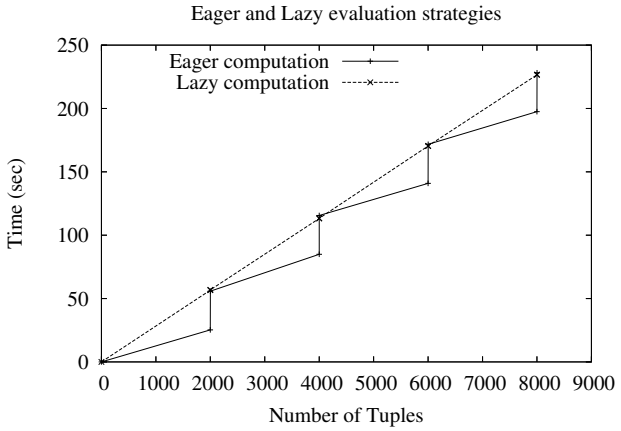

Figure 9: Eager and Lazy evaluation strategies

7.3 Granularity of hashing

As explained earlier, the granularity of hashing determines the data exposure. While keeping a finer granularity seems attractive, this has a performance cost associated with it. This experiment studies the tradeoff between performance and granularity. The table used for this experiment has 16 attributes and 1000 tuples. Different granularities of hashing were tested and the merkle tree (for α-correctness) was generated to measure the performance cost. Figure 10 shows that the proof generation cost increases linearly with the granularity. The x-axis represents the cardinality of G_T for the table.

8. CONCLUSIONS

In this paper we addressed the problem of ensuring the correctness of query results received from a private database. This is a new problem that has not been addressed earlier. We proposed a number of solutions for this problem that differ in degree of exposure, and the cost the generation of a proof and verification of the results. We defined two notions of correctness of results. Our solution is able to prove α-correctness for arbitrary select-project-join queries, and β-correctness for queries with equality joins. This rep-

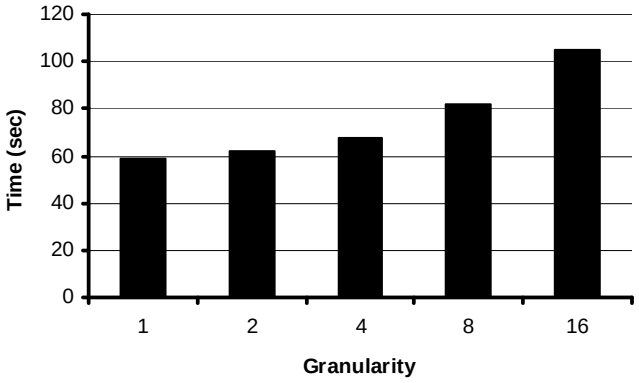

Figure 10: Cost associated with granularity of hashing of attributes

resents a very broad set of queries. The feasibility of our methods was established through an implementation using PostgreSQL, and tested with real data. The results show that the overhead of the proposed approach is on the same order as the cost of inserting data. While this work represents a significant step for solving this important problem, future work will address the more challenging problem of proving β-correctness for general queries.

9. REFERENCES

[1] G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-Molina, K. Kenthapadi, N. Mishra, R. Motwani, U. Srivastava, D. Thomas, J. Widom, and Y. Xu. Enabling privacy for the paranoids. In *Proceedings of the International Conference on Very Large Data Bases*, 2004.

[2] S. Agrawal, V. Krishnan, and J. R. Haritsa. On addressing efficiency concerns in privacy-preserving mining. In *Proceedings of Database Systems for Advanced Applications*, 2004.

[3] P. Devanbu, M. Gertz, A. Kwong, C. Martel, Nuckolls, and S. Stubblebine. Flexible authentication of xml documents. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, 2001.

[4] P. Devanbu, M. Gertz, C. Martel, and S. Stubblebine. Authentic third-party data publication. *Journal of Computer Security*, 11(3), 2003.

[5] A. V. Evfimievski, R. Srikant, R. Agrawal, and J. Gehrke. Privacy preserving mining of association rules. *Information Systems*, 29(4):343–364, 2004.

[6] M. Gertz, A. Kwong, C. Martel, G. Nuckolls, P. Devanbu, and S. Stubblebine. Databases that tell the truth: Authentic data publication. *Bulletin of the Technical Committee on Data Engineering*, 7(1), 2004.

[7] H. Hacigumus, B. R. Iyer, and S. Mehrotra. Providing database as a service. In *Proceedings of the International Conference on Data Engineering*, 2002.

[8] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *Proceedings of the International Conference on Very Large Data Bases*, 2004.

[9] M. Kantarcioglu and C. Clifton. Privacy-preserving distributed mining of association rules on horizontally partitioned data. *Transactions on Knowledge and Data Engineering*, 16(9):1026–1037, 2004.

[10] R. C. Merkle. A certified digital signature. In *Proceedings of Advances in Cryptology (Crypto)*, 1989.

[11] H. Pang and K. Tan. Authenticating query results in edge computing. In *In Proceedings of the IEEE International Conference on Data Engineering*, 2004.

[12] PostgreSQL. http://www.postresql.org.

[13] B. Preneel. *Analysis and Design of Cryptographic Hash Functions*. PhD thesis, Katholieke University Leuven, 1993.

[14] O. Project. http://www.openssl.org.

[15] R. T. Snodgrass, S. Yao, and C. S. Collberg. Tamper detection in audit logs. In *Proceedings of the International Conf. on Very Large Data Bases*, 2004.

APPENDIX

A. AUTHENTIC THIRD PARTY DATABASE PUBLICATION

The crucial difference between the model proposed in this paper and the model of authentic third-party database publication [4] is that we do not trust the database owner to follow the proof generation algorithm honestly. Similar to our α-correctness solution, [4] uses merkle trees to generate proof of correctness. The algorithm presented in [4] assumes that the database is sorted on the attribute on which selection is performed. To ensure β-correctness it simply reveals one tuple before and after the result set. This ensures that no tuples are missing from the result. The solution works because the database owner is trusted with the task of properly sorting the database on the selection key before freezing it. While this is a reasonable assumption for third-party database publication, it does not hold for our problem model. If we remove the assumption of trust on the database owner, then the solutions presented in [4] will not work. This is explained in the following example.

item_id	units sold
1	105
2	97
3	221
1	105
2	0
3	221

Figure 11: Example database

Consider the example presented in Figure 11. Note that the database owner Bob has (maliciously) frozen two values for the number of units sold for item id 2 (along with correct values for items 1 and 3). This scenario is quite possible if Bob does not want tie down his hands completely and would like to change the quantity of item 2 sold based on some future information. Given the query $\Pi_{units\_sold}(\sigma_{item\_id=2}(T))$ If we use the protocol presented in [4], Bob can either report 97 or 0 and provide a proof of correctness for the same! In our approach, this is not possible as Bob would not be able to prove β-correctness for this query. Hence, the previous solutions based on the model of authentic third-party database publication are not applicable to the problem presented in this paper.