Authentic Third-party Data Publication

Prem Devanbu, Michael Gertz, Chip Martel
Department of Computer Science
University of California
Davis, California CA 95616 USA
{devanbu|gertz|martel}@cs.ucdavis.edu

Stuart G. Stubblebine*
CertCo
55 Broad Street – Suite 22
New York, NY 10004
stubblebine@cs.columbia.edu

Abstract

Integrity critical databases, such as financial information, which are used in high-value decisions, are frequently published over the internet. Publishers of such data must satisfy the integrity, authenticity, and non-repudiation requirements of end clients. Providing this protection over public data networks is an expensive proposition. This is, in part, due to the difficulty of building and running secure systems. In practice, large systems can not be verified to be secure and are frequently penetrated. The negative consequences of a system intrusion at the publisher can be severe. The problem is further complicated by data and server replication to satisfy availability and scalability requirements.

To our knowledge this work is the first of its kind to give general approaches for reduce the trust required of the *publisher* of large, infrequently updated databases. To do this, we separate the roles of owner and publisher. With a few digital signatures on the part of the owner and no trust required of the publisher, we give techniques based on Merkle hash trees, that publishers can use to provide authenticity and non-repudiation of the answer to a database query. This is done *without* requiring a key to be held in an on-line system, thereby reducing the impact due to the likely system penetration. By reducing the trust required of the publisher, our solution is a step towards the publication of large databases in a scalable manner.

1 Introduction

Consider a financial markets database, used by businesses to make high-value decisions. Examples include financial data about a range of investment vehicles such as stocks, bonds and mortgage-backed securities. These databases might be resold to "financial portals", which republish the information over insecure public networks. This data will be queried at high rates, for example by *client* investment tools. We focus our attention on data which changes infrequently[1]. We assume extremely high Query/Update ratios, with millions of queries executed daily. The data needs to be delivered promptly, reliably and accurately. One approach to this problem is to digitally sign the results to a query. Here, $\sigma_{K_O^{-1}}()$ represents the signature using the private signature key, K_O^{-1}.

1. *Client* $\longrightarrow$ *Owner :* $Query, Nonce$
2. *Owner* $\longrightarrow$ *Client:* $Data,\ Timestamp,\ \sigma_{K_O^{-1}}(Data, Timestamp, Nonce)$

The *client* may thus be assured of the correctness of the answer, since it is in response to the query, the data is adequately recent, and it is signed by the owner (and thus satisfies the non-repudiation requirements). However, there are several issues here: first the *owner* may not be willing or able to provide a reliable, efficient, database service to query over this data at the scale and the rates required by the clients and republishers of the information. Second, even if the owner is willing and able to

*This paper is under continuous revision; rather than forwarding a copy, please get the latest from `http://seclab.cs.ucdavis.edu/~devanbu/authdbpub.pdf`

[1]Examples of relatively static information include financial earnings reports, market history such as individual trades, offers and bids, intra- and inter- day highs and volume, and sometimes financial ratings.

provide a service, the owner needs to maintain a high-level of physical and computer security required to defend against attacks. This has be done to protect the signing key, which must be resident in the database server at all times to sign outgoing data. However, in practice, most large software systems have vulnerabilities. Some these can be "patched", (albeit usually only partially) with a great deal of skill, effort, time and expense. Using cryptographic techniques like threshold cryptography or using hardware devices to protect the private key help but do not solve the systems vulnerability problem and are generally too expensive to implement in our application domain.

A more scalable approach to this problem is to use trusted third-party publishers of the data, in conjunction with a key management mechanism which allows certification of the signing keys of the publishers to speak for the author of the data. The database (or updates), DB, is provided securely to the publishers.

$Owner \longrightarrow Publishers\ P_i,\ i = 1 \ldots n :$ $\qquad DB, Timestamp,\ \sigma_{K_O^{-1}}(DB, Timestamp)$

This is followed by the clients accessing the publishers in a query response style similar to the previous protocol description, with publishers signing answers using $K_{P_i}^{-1}$, $i = 1 \ldots n$.

Presumably, the market for the useful data in DB will motivate other *publishers* to provide this service, unburdening *owner* of the need to do so. The *owner* simply needs to sign the data once and distribute it to the *publishers*. As the demand increases, more such *publishers* will emerge, thus making this approach inherently more scalable. However, this approach also suffers from the problem and expense of maintaining a secure system accessible from the internet. Furthermore, the *client* might worry that a *publisher* engages in deception. The *client* has to find a *publisher* that she can trust, such as a trusted brand-name belonging to a large corporation. The *client* would also have to trust the key-management infra-structure that allowed the keys of the *publishers* to be certified. In addition, she would have to believe that her *publisher* was both competent and careful with site administration and physical access to the database. Particularly, she might well worry that the private signing-key of the *publisher* would have to be resident at the database server, and is therefore vulnerable to attack. To get a *client* to trust him to provide really valuable data, the *publisher* would have to adopt careful and stringent administrative policies, which be more expensive for him (and thus also for the *client*). The abovementioned need for trusted publishers would increase the reliance on brand-names, which would also limit the tendency of market competition to reduce costs. Tygar lists this as important open problem in e-commerce. He asks [18]: "How can we protect re-sold or re-distributed information...?". We present an approach to this problem.

Our Approach We propose an approach to certifying responses to queries, by which a completely *untrusted publisher* can provide a *verification object* $\mathcal{VO}$ to a *client*, along with the answer to the query; this $\mathcal{VO}$ is generated by the *publisher*, and provides an independent means of verifying that the answer to the query is correct. The verification object is based on a small number of *summary signatures* (Σ's) that are distributed periodically to the clients by the data *owner*. The summary signatures are bottom-up hashes computed recursively over B-tree type indexes for the entire set of tuples in each relation of the *owner*'s database, signed with K_O^{-1}. Answers to queries are various combinations of subsets of these relations. Given a query, the *publisher* computes the answer. To show that an answer is correct the *publisher* constructs a $\mathcal{VO}$ using the same B-tree structures that were used by *owner* to compute the summary signatures. This $\mathcal{VO}$ validates an answer set by providing an unforgeable[2] "proof" which links the answer to the appropriate Σ, which was already signed by *owner*. Our approach has several features:

1. With the exception of the security at the *client* host, a *client* needs only trust the key of the *owner*; in addition, *owner* only needs to distribute Σ during update cycles. Hence, the private signing key need not be resident in a "online" machine, and can be better protected; *e.g.,* it can be ensconced in a hardware token that is used only to sign hash digests during updates.

[2]Our techniques are founded on cryptographic assumptions regarding the security of hash functions and public-key cryptosystems.

2. *clients* need not trust the *publishers*, nor their keys.

3. For all the techniques we describe, the $\mathcal{VO}$ is always linear in the size of the answer to a query.

4. The $\mathcal{VO}$ guarantees that the answer is exact, without any superfluous or missing tuples.

5. If the *publisher* evaluates queries using the same data structures used by *owner* to compute Σ, *publisher*'s overhead to compute $\mathcal{VO}$ is relatively minor.

An incorrect answer and $\mathcal{VO}$ will almost always be rejected by the *client*; it is infeasible for *publisher* to forge a correct $\mathcal{VO}$ for a wrong answer.

We offer solutions for authenticity in a specific context. We do not directly address the orthogonal issue of access control policies [9], which might restrict queries and updates, nor issues of confidentiality and privacy. Our techniques are restricted to relational databases. In addition, we handle only some relational queries. Our techniques do involve the construction of some complex data structures, although the cost of this can be amortized over more efficient query processing. Some of the approaches are similar to view materialization, which can enable data warehouses to provide efficient query processing.

The outline of this paper is as follows. In Section 2, we begin with a discussion of the problem space, as we have framed it, and briefly set the context for the preliminary results presented in this paper, and open issues that still remain. Next, after a brief background on relational databases, we describe the essence of our extension of the work of Naor & Nissim [13] to secure third-party data publication. In Section 4, we give our basic approach. In Section 7, we give our conclusions.

2 The Design Space

To begin with, we clarify the problem setting outlined above, and map out the potential design space of solutions. Our general setting can be described thus:

1. The *owner* populates the relations in the database, does some pre-computation on the database to construct data structures that support query-processing (such as B-trees), and computes summary signatures (i.e., Σs) of these data structures, signed with K_O^{-1}

2. The *owner* distributes these Σs to *clients* and the database to the *publishers*.

3. A *client* sends a query to a *publisher*. The *publisher* computes the answer q, and a verification object $\mathcal{VO}$ and sends both back to *client*.

4. The *client* verifies the correctness and completeness of q by recomputing Σ using q, $\mathcal{VO}$ and K_O.

Given the general problem of publishing data securely using third parties, different approaches are possible, which lead to different computing and storage costs for the *clients*, the *publishers*, and the *owners*. One extreme approach, for example, would be for the *owner* to simply deliver the entire database to all the *clients* and let them do their own query processing. This approach entails huge data transmission costs and requires a great deal of storage and processing by the *clients*. The other extreme is for the *owner* to pre-compute and sign a whole range of possible queries. The *publishers* would simply cache the queries along with the applicable signatures from the *owner*, and return the signatures with the answers to the queries. From the *client*'s perspective, this approach is attractive: each pre-computed answer comes with a *constant*-length verification object (a signature) directly from the *owner*. However, this approach is not practical in general: there are just simply too many possible queries.

Our goal is a compromise design: one that does not require pre-computation of arbitrary query answers, nor shipment of the entire database to *clients*. Given a particular database, the *client* can choose from a potentially infinite set of queries (although constrained by the query patterns implemented in the applications at the client sites). We adopt the position that requires a certain amount of effort from all parties: the *owner*, the *publisher*, and the *client*. In all our suggested techniques, the *owner* has to do work $O(n \log^{d-1} n)$ time and space in the size n of the database (where d is a small constant denoting roughly the number of operations, such as selections and projections, in the query). In particular, simple Θ selections over a particular attribute of a single relation only require $O(n)$ work of the *owner*. However, this work is mostly towards the construction of index structures, which is amortized over repeated, efficient query processing. These index structures can either be transmitted to the *publishers*, or (exactly and precisely) recomputed by them. Our query processing algorithms are similar in performance to those of standard databases. The construction of the $\mathcal{VO}$'s are a small constant overhead over the standard query processing algorithms. Finally, and perhaps most importantly, the size of the $\mathcal{VO}$ grows linearly with the size of the answer set, and poly-logarithmically with the size of the database. The verification step itself takes time linear in the size of the $\mathcal{VO}$.

This view suggests that there may be many other perspectives on this problem of secure third-party data publication. Differing assumptions on the levels of storage and computational effort expected of *owners*, *publishers* and *clients* may lead to different viable solutions to this problem. We believe that (in addition to the techniques presented below) many useful "operating points" await discovery.

3 Preliminaries

In this section we will discuss the basic notions, definitions and concepts necessary for the approach presented in this paper. In Section 3.1 we will present the basic notions underlying relational databases and queries formulated in relational algebra. In Section 3.2 we will discuss the computation and usage Merkle Hash Trees.

3.1 Relational Databases

The data model underlying our approach is the relational data model (see, e.g., [5, 17]). That is, we assume that the data owner and the publisher manage the data using a relational database management system (DBMS). The basic structure underlying the relational data model is the relation. A *relation schema* $R\langle A_1, A_2, \ldots, A_n\rangle$ consists of a relation name R and an ordered set of attribute names $\langle A_1, A_2, \ldots, A_n\rangle$, also denoted by $schema(R)$. Each attribute A_i is defined on a domain D_i. An *extension* of a relation schema with arity n (also called *relation*, for short) is a finite subset of the Cartesian product $D_1 \times \ldots \times D_n$. The extension of a relation schema R is denoted by r. The value of a tuple $t \in r$ for an attribute A_i is denoted by $t.A_i$. We assume that with each relation schema R a set $pk(R) \subseteq \{A_1, \ldots, A_n\}$ is associated which designates the primary key. The number of tuples in a relation r is called the *cardinality* of the relation, denoted by $\mid r \mid$. A *database schema* $\mathcal{S}$ is a collection of relation schemas $\mathcal{R} = \{R_1, \ldots, R_m\}$. For a database schema $\mathcal{S}$, the extension of the relation schemas at a particular point in time is called a *database instance* (or *database*).

Queries against a database are formulated in SQL [12, 6]. Such queries are typically translated by the DBMS query processing engine into expressions of the relational algebra for the purpose of query optimization and execution. In this paper we are mainly concerned with providing verification objects for query results where the queries are formulated as expressions of relational algebra. Those queries are either simple, containing at most one basic operator, or complex, containing a composition of basic operators. The basic operators of the relational algebra are as follows (with R, S being relation schemas):

- Selection (σ): $\sigma_P(r) := \{t \mid t \in r \text{ and } P(t)\}$ where r is a relation (name), P is a condition of the

form $A_i\ \Theta\ c$ with $A_i \in schema(R)$, $c \in D_i$, and $\Theta \in \{=, \neq, <, >, \leq, \geq\}$.

- Projection (π): $\pi_{A_k,\ldots,A_l}(r) := \{t.A_k, \ldots, t.A_l \mid t \in r\}$.
- Cartesian Product ($\times$): $r \times s := \{tq \mid t \in r \textbf{ and } q \in s\}$.
- Union Operator ($\cup$): $r \cup s := \{t \mid t \in s \textbf{ or } t \in s\}$. R and S must be union compatible.
- Set Difference Operator ($-$): $r - s := \{t \mid t \in r \textbf{ and } t \notin s\}$. R and S must be union compatible.

Additional operators of the relational algebra, which are typically used in complex queries, are *natural join or equi-join* $\bowtie$, *condition join or theta-join* $\bowtie_C$ (with C being a condition on join attributes), and *set-intersection* $\cap$. All these operators can be derived from the above five basic operators.

3.2 Merkle Hash Trees

We describe the computation of a Merkle Hash Tree [11] for a given relation r with relation schema $R = \langle A_1, \ldots, A_n\rangle$. For this, assume that $\mathcal{A} = \langle A_i, \ldots, A_k\rangle$ is a list of attributes from $schema(R)$. The Merkle Hash Tree computed is denoted by $MHT(r, \mathcal{A})$.

1. First, compute the *tuple hash* h_t for each tuple $t \in r$ thus:

$$h_t(t) \ = \ h(h(t.A_1) \ || \ \ldots \ || \ h(t.A_n))$$

   The tuple hash (by the collision resistance of the hash function) functions as a "nearly unique" tuple identifier (for a hash-length of 128 bits, probability of collisions approaches 2^{-128}).

2. Next, compute the Merkle hash tree for relation r. For this, assume that r is sorted by the values of $\mathcal{A}$ so that for two distinct tuples $t_{i-1}, t_i \in r$, $t_{i-1}.\mathcal{A} \leq t_i.\mathcal{A}$.

$$\text{Leaf-nodes}: \qquad h^0(i) \ = \ h_t(t_i) \ , \ i = 1 \ldots \mid r \mid, \ 0 \ otherwise$$

$$h^j(i) \ = \ h(h^{j-1}(2i-1) \ || \ h^{j-1}(2i)) \ for \ i \ = 1 \ldots \left\lceil (\frac{\mid r \mid}{2^j}) \right\rceil, j \ = \ 1 \ldots \lceil \log_2(\mid r \mid)\rceil$$

We note that there is only one hash value at the level $\lceil \log_2(\mid r \mid)\rceil$, and this is the "root hash" of the Merkle tree. In the sequel, we denote the two values $h^{j-1}(2i-1)$ and $h^{j-1}(2i))$ used to compute $h^j(i)$ as *hash siblings*; $h^j(i)$ is their *parent*. This construction is illustrated in Figure 1. $h34$ is the parent of $h3$ and $h4$; $h3$ and and $h4$ are hash siblings. We note that this construction easily generalizes to a higher branching factor $K > 2$, such as in a B^+-tree; however, for our presentation here, we primarily use binary trees. Indeed, our approach works best if the *owner* and the *publisher* build an MHT around index structures that are used in query evaluation. In this case, constructing a $\mathcal{VO}$ is a very minor overhead over the query evaluation process itself.

Note that (by the cryptographic assumption of a collision-resistant hash function) if the correct value of the *parent* is known to the client, the publisher cannot forge the value of the hash siblings. Our entire approach flows from the signed, correct value of the root of a Merkle tree, just as in the work of Naor & Nissim [13].

Definition 1 (Hash Path) *Let $h^0(i)$ be a leaf node in $MHT(r, \mathcal{A})$ corresponding to a tuple $t_i \in r$. The nodes necessary to compute the hash path up to the root hash is denoted as $path(t_i)$. Such a hash path always has the length $\lceil \log_2(\mid r \mid)\rceil$ and comprises $2 * \lceil \log_2(\mid r \mid)\rceil - 1$ nodes where exactly two nodes are leaf nodes. Of these, only $\lceil \log_2(\mid r \mid)\rceil + 1$ need be provided to recompute the value at the root. Hash paths can also be provided for non-leaf nodes.*

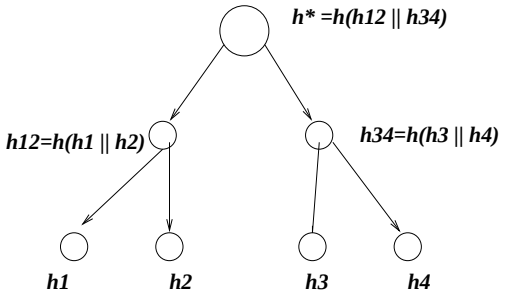

Figure 1: Computation of a Merkle hash tree

The $\lceil \log_2(\mid r \mid) \rceil + 1$ nodes in $path(t_i)$ constitute the $\mathcal{VO}$ showing that t_i is actually in the relation rooted by the hash value at the root node; the *owner*'s signature on the root node certifies its authenticity. Indeed any interior node within the hash tree can be authenticated by giving a path to the root.

Definition 2 (Boundaries) *For a given non-empty contiguous sequence $q = \langle t_i, \ldots, t_j \rangle$ of leaf nodes in a Merkle Hash Tree $MHT(r, \mathcal{A})$, there are two special leaf nodes $LUB(q)$ and $GLB(q)$ that describe the lowest upper and greatest lower bound values, respectively, of q and are defined as follows:*

(1) $GLB(q) := \{t \mid t \in r \wedge t.\mathcal{A} < t_i.\mathcal{A} \wedge (\neg \exists s \in r : s.\mathcal{A} > t.\mathcal{A} \wedge s.\mathcal{A} < t_i.\mathcal{A})\}$

(2) $LUB(q) := \{t \mid t \in r \wedge t.\mathcal{A} > t_j.\mathcal{A} \wedge (\neg \exists s \in r : s.\mathcal{A} < t.\mathcal{A} \wedge s.\mathcal{A} > t_j.\mathcal{A})\}$

We assume that both $GLB(q)$ and $LUB(q)$ are singletons. This can easily be accomplished by adding $pk(R)$ to the list $\mathcal{A}$ of attributes by which the leaves in $MHT(r, \mathcal{A})$ are ordered.

Definition 3 ***(Lowest Common Ancestor)*** *For a given non-empty contiguous sequence $q = \langle t_i, \ldots, t_j \rangle$ of leaf nodes in a Merkle Hash Tree $MHT(r, \mathcal{A})$, the lowest common ancestor $LCA(q)$ for q in $MHT(r, \mathcal{A})$ is defined as the root of the minimal subtree in $MHT(r, \mathcal{A})$ that has all tuples in q as leaf nodes.*

This situation is illustrated in Figure 2. Given $LCA(q)$, one can show a hash path $path(LCA(q))$ to the authenticated root hash value. After this is done, (shorter) hash paths from each tuple to $LCA(q)$ can provide evidence of membership of q in the entire tree. This is also useful to build a $\mathcal{VO}$ showing that two nodes occur consecutively in the tree.

Definition 4 ***(Proximity Subtree)*** *Consider a consecutive pair of tuples (leaf nodes) s, t in $MHT(r, \mathcal{A})$, and their lowest common ancestor, $LCA(\langle s, t \rangle)$. This node, along with the two paths showing that s (respectively, t) is the rightmost (leftmost) element in the left (right) subtree of $LCA(\langle s, t \rangle)$ constitute the "proximity subtree" of s and t, denoted by $ptree(s, t)$.*

Proximity subtrees are used in boundary cases, with $GLBs$ and $LUBs$ *i.e.,* to show a "near-miss" tuple that occurs just outside the answer set lies next to the extremal tuple in the answer set. In this case, it is important to note that by construction, we just need to reveal the relevant attribute value in the "near-miss" to show that it is indeed a near miss; with just the hash of the other attributes, the tuple hash, and the rest of the proximity tree can be exhibited.

We finally define important properties of the answer set q returned by *publisher*. For this, we assume that *owner* can use a database system to process queries from the *client* in the same fashion as done by the *publisher*

Definition 5 *Assume a query Q issued by a client. Let q_{pub} and q_{owner} denote the query result computed at the data publisher and data owner site, respectively.*

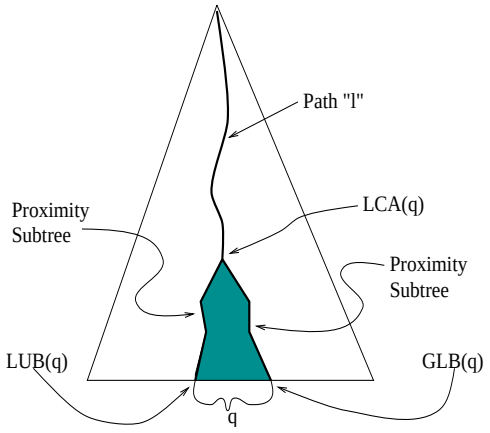

Figure 2: A Merkle tree, with a contiguous subrange q, with a least common ancestor LCA(q), and upper and lower bounds. Note verifiable hash path "l" from LCA(q) to the root, and the proximity subtrees (thick lines) for the "near miss" tuples for LUB(q) and GLB(q) which show that q is complete.

q_{pub} is said to be a inclusive *answer to Q iff* $\forall t\colon t \in q_{pub} \Rightarrow t \in q_{owner}$ *holds.*

q_{pub} is said to be a complete *answer to Q iff* $\forall t\colon t \in q_{owner} \Rightarrow t \in q_{pub}$ *holds.*

4 Base level Relational Queries

In this section we outline the computation of $\mathcal{VO}$ for answers to queries. We illustrate the basic idea behind our approach for selection and projection queries in Section 4.1 and 4.2, respectively. Slightly more complex types of queries (join queries) and set operators are discussed in Sections 4.3 and 4.4.

4.1 Selections

Assume a selection query of the form $\sigma_{A_i \Theta c}(r), c \in D_i$ which determines a result set $q \subseteq r$. Furthermore, assume that the Merkle Hash Tree $MHT(r, A_i)$ has been constructed. For each possible comparison predicate $\Theta \in \{=, \neq, <, >\}$, we show how the *publisher* can construct the $\mathcal{VO}$, with which the *client* can verify the inclusiveness and completeness of the query answer q. Again, we emphasize that in all the following cases, *if* the Merkle hash tree is constructed by *owner* and *publisher* over the same index structures used for querying, the overhead for constructing the $\mathcal{VO}$ is minor. We first consider the cases for the comparison predicate $\Theta \equiv =$.

(I) If $A_i = pk(R)$ and $q \neq \{\}$, then the $\mathcal{VO}$ is just $path(t)$ where $t = q$ is the only tuple that satisfies the selection condition. In this case, the size of the $\mathcal{VO}$ is $O(\log_2 \mid r \mid)$. (II) If $A_i = pk(R)$ and $q = \{\}$, then we have to show that no tuple exists that satisfies the selection. For this, we have to provide paths of the two tuples that would "surround" the non-existing tuple. The two tuples are determined by $GLB(q')$ and $LUB(q')$ with $q' = c$. Determining $path(GLB(q'))$ and $path(LUB(q'))$ requires searching the two associated tuples in the leaf nodes of $MHT(r, A_i)$. The proximity subtree $ptree(GLB(q'), LUB(q'))$ provides required evidence that the answer set is empty. The size of the $\mathcal{VO}$ again is $O(\log_2 \mid r \mid)$.

(III) A_i is not a primary key and $q \neq \{\}$. The result is a set of tuples which build a contiguous sequence of leaf nodes in $MHT(r, A_i)$. In order to provide a $\mathcal{VO}$ for q, the following approach is taken. First, identify $l := LCA(q \cup GLB(q) \cup LUB(q))$ in $MHT(r, A_i)$, and show a verifiable path from l to the root. Next, identify proximity subtrees showing that $GLB(q)$ $(LUB(q))$ occur consecutively to

the smallest (largest) element of q. Now, the entire sub-tree from the elements of the set q to l can be constructed, using the hash values of the tuples in q. This verifies that the entire set occurs in the leaf nodes of the tree. To construct this subtree and to verify the root hash on the $LCA(q)$ of this subtree, the length of the $\mathcal{VO}$ is $O(\mid q \mid * \log_2(\mid r \mid))$. The proximity subtrees establish that no tuples are left out.

(IV) If A_i is not a primary key and $q = \{\}$, we can apply the same approach as for (II).

With these fundamental techniques, proving that the answers to selections over a relation r are inclusive and complete is simple. First, using normal query evaluation, the answer set q is determined. Since we only consider simple relational queries here, the answer set q is a contiguous subset of r based on $MHT(r, A_i)$. We also retrieve two additional tuples, $GLB(q)$ (respectively, $LUB(q)$) which is immediately smaller (larger) than the smallest (largest) tuple in r with regard to the answer set q (using the ordering based on A_i). It should be noted that if the answer set is empty, these two will occur consecutively in the Merkle tree. The cost of this readily seen to be $O(\mid q \mid * \log_2(\mid r \mid)$.

For $\Theta \equiv \neq$, we can make the following observation: The answer set to a query of the pattern $\sigma_{A_i \neq c}(r)$ determines at most two contiguous sets of leaf nodes in $MHT(r, A_i)$. For each of these sets, we have to follow an approach similar to (III) shown above.

For $\Theta \in \{<, >\}$, the scenario is as follows. The answer set to a query of the pattern $\sigma_{A_i<c}(r)$ or $\sigma_{A_i>c}(r)$ determines at most one contiguous sets of leaf nodes in $MHT(r, A_i)$. For this set, we have to follow an approach similar to (III) discussed above. If q is empty, we just have to give the $\mathcal{VO}$ for the tuple $t = \{t \mid t \in r \wedge t.\mathcal{A} = \min\{s.\mathcal{A} \mid s \in r\}\}$ (analogous for $\Theta \equiv >$). Cases for $\Theta \in \{\leq, \geq\}$ can be handled in a very similar fashion by just shifting the boundaries.

4.2 Projections

For queries of the pattern $\pi_{\mathcal{A}}(R), \mathcal{A} \subset schema(R)$, the projection operator eliminates some attributes of the tuples in the relation r, and then eliminates duplicates from the set of shortened tuples, yielding the final answer q. There may be many different possible projections on a relation R. If *client* wishes to choose among these dynamically, it may be best to let the *client* perform the projection. The *client* will then also have to eliminate duplicates because these are not automatically eliminated in SQL, unlike the relational algebra. So in this case, the *client* is provided with the whole relation r (or some subset thereof after intermediate selections etc) and the $\mathcal{VO}$ for r *before* the projection; so the $\mathcal{VO}$ will be linear in size $\mid r \mid$, rather than the smaller size $\mid q \mid$ of the final result. Note also that the projection may actually mask some attributes that the *client* is not allowed to see; if so, with just the hash of those attributes in each tuple, the *client* can compute the tuple hash, and the $\mathcal{VO}$ for r will still work.

Consider the case where a particular projection $\pi_{\mathcal{A}}(r)$ (which is used often) projects onto attributes and where the values for the projected attributes are poorly distributed *i.e.,* many tuples have the same values for the attribute(s) $\mathcal{A}$. In this case, duplicate elimination will remove numerous tuples, leaving behind a small final answer q. Just given the pre-projection tuple set, the *client* would have to do all this work. Now, suppose we have a Merkle tree $MHT(r, \mathcal{A})$, *i.e.,* we assume that the sets of retained attribute values can be mapped to single values (which corresponds to building equivalence classes) with an applicable total order. In this case, we can provide a $\mathcal{VO}$ for the projection step that is linear in the size of the projected result q.

Each tuple t in the result set q potentially results from a set of tuples $S(t) \subseteq r$. Each tuple in $S(t)$ has identical values for the projected attribute(s) $\mathcal{A}$. We need to establish that the set q is inclusive (i.e., each t is indeed in the projection) and complete (i.e., no tuple that should be there is missing). This is accomplished as follows.

1. To show that $t \in q$, we find *any* witness tuple $y \in S(t) \subseteq r$, with the same attribute value for $\mathcal{A}$, and show the hash path from this tuple to the Merkle Root. This establishes that the tuple

t belongs to the result set q. However, the witness tuple is preferably chosen as a "boundary" value, as we describe next.

2. We must show that there are no tuples missing, say between t and $t', (t, t' \in q)$. To do this, it is enough to show that the sets $S(t), S(t')$ are both in r and occur immediately next to each other in the sorted order. This is done by showing hash paths which prove that two "boundary" tuples $y \in S(t)$ and $x \in S(t')$ occur next to each other in the Merkle tree $MHT(r, \mathcal{A})$ for r.

We observe that both the above bits of evidence are provided by displaying at most $2 \mid q \mid$ hash paths, each of length $\lceil \log_2 r \rceil$. This meets our constraint that the size of the authentication evidence be bounded by $O(\mid q \mid \log_2 \mid r \mid)$.

But how can we assume that a Merkle tree would exist on precisely the intermediate result just prior to the projection? This can be accomplished by so-called multi-dimensional range trees, which are discussed in section 5.

4.3 Joins

Joins between two or more relations are the most common type of operators in relational algebra used to formulate complex queries. This holds in particular for equi-joins where relations are combined based on primary key – foreign key dependencies. There are many alternative realizations for Merkle Tree structures that can be used to provide $\mathcal{VO}$s for query results computed from joins. In this paper we focus on pairwise joins of the pattern $R \bowtie_C S$ where C is a condition on join attributes of the pattern $A_R \Theta A_S$, $A_R \in schema(R), A_S \in schema(S), \Theta \in \{=, <, >\}$. We assume that the data types underlying A_S and A_R are compatible with respect to Θ. For Θ being the equality predicate, we obtain the so-called equi-join.

Given a query of the pattern $R \bowtie_C S$, a Merkle Hash Tree structure that supports the efficient computation of a $\mathcal{VO}$ for the query result is based on the materialization (i.e., the physical storage) of the Cartesian Product $R \times S$. Note that such a structure has to be constructed before the *publishers* accept any query from *clients*. The reason for choosing a materialization of the Cartesian Product is that this structure supports all three types of joins mentioned above. This is due to the fact that these joins can be formulated in terms of basic relational algebra operators, i.e., $R \bowtie_{A_R \Theta A_S} S := \sigma_{A_R \Theta A_S}(R \times S)$. The important issue in constructing the Merkle Hash Tree for queries of the pattern $R \bowtie_C S$ is that first the Cartesian Product is determined and then the resulting tuples are sorted on the *difference* between the values for A_R and A_S, assuming such an operation can be defined. We thus obtain three "groups" of leaf nodes in the Merkle Tree: (1) nodes for which the difference $t.A_R - s.A_S$ for two tuples $t \in R, s \in S$ is 0, thus supporting equi-joins, (2) nodes where the difference is positive, thus supporting the predicate $>$, and (3) nodes where the difference is negative, thus supporting the predicate $\Theta \equiv <$.

For each of the three cases above, assume a query result set q. Our burden again is provide inclusiveness and completeness evidence for the query result.

1. For each tuple $t \in q$, we provide a hash path in the (sorted) Merkle tree for $R \times S$ showing that tuple is in the relation.
2. To show that no tuples are missing, we show two pairs of boundary hash paths (within and without) verifying the boundaries of the answer set.

If it is known that the querys will only result in equi-joins against the database, an optimized Merkle Tree structure can be used. We will only sketch the basic concept for using this structure here. Instead of a space-consuming materialization of the Cartesian Product $R \times S$, we materialize the *Full Outer Join* $R \sqsupset\bowtie\sqsubset S$ which pads tuples for which no matching tuples in the other relation exist with null

values (see, e.g., [5, 17]). The result tuples obtained by the full outer-join operator again can be grouped into three classes: (1) those tuples $ts, t \in R, s \in S$, for which the join condition holds, (2) tuples from r for which no matching tuples in s exist, and (3) tuples from s for which no matching tuples in r exist. Constructing a $\mathcal{VO}$ for the result of query of the pattern $R \bowtie_{A_R \Theta A_S} S$ then can be done in the same fashion as outlined above.

4.4 Set Operations

All set operations involve two relations u and v. We may assume that u and v are intermediate results of a query evaluation, and are subsets of some relations r and s respectively, and that r and s are each sorted (possibly on different attributes) and have its own Merkle tree $MHT(r, \mathcal{A})$ and $MHT(s, \mathcal{A}')$, the root of which is signed as usual. We consider the set operations union and set intersection.

Union In this case, the answer set is $q = u \cup v$. Evidence that $u \cup v$ is inclusive and complete is straightforward; it is sufficient to provide verification paths for each element of $u \cup v$ showing that it belongs to one of the sets. Additionally a single pass over the union can show that no elements of u or v are omitted. This can be done with a $\mathcal{VO}$ of size $O(\mid q \mid \log_2(\max\{\mid r \mid, \mid s \mid\})$. If u and v are presented as contiguous subsets of r and s, inclusiveness can be done even faster; all that is necessary to check the union is the order in which the elements of $u \cup v$ occur in r and/or s, along with the authenticating information for the sets u and v within r and s respectively. Given this order, inclusiveness and completeness can be evidenced by a $\mathcal{VO}$ of linear size.

Intersection The approach for union, however, does not produce compact $\mathcal{VO}$s for set intersection. Suppose $q = u \cap v$ where u and v are as before. Inclusiveness is easy, with the $\mathcal{VO}$ providing $O(\mid q \mid)$ verification paths, as before, showing elements of q belong to both u and v; but completeness is harder. *User* needs assurance that all elements of u or v that belong in q are present. One can pick the smaller set (say u) and for each element in $u - q$, construct a $\mathcal{VO}$ show that it is $\notin \ v$. In general, if u and v are intermediate results not occurring contiguously in the same Merkle tree, such a $\mathcal{VO}$ is linear in the size of the smaller set (say u). Consider for example a set of tuples $\langle \texttt{name}, \texttt{age}, \texttt{salary} \rangle$, where one wishes to select tuples in a specific salary and age range. Assume then that u has been obtained by performing a selection based on `salary`, and v based on `age`. u and v would be verified by $\mathcal{VO}$'s resulting from different Merkle hash trees: one sorted by `salary`, and one sorted by `age`. Computing the intersection $u \cap v$ would result in a $\mathcal{VO}$ with size linear in $\mid u \mid$: this $\mathcal{VO}$ would provide inclusiveness evidence (in u and v) for each element of $u \cap v$, and shows completeness by showing that each remaining element in $(u - (u \cap v))$ is not in v. This again leaves us with the unsatisfactory situation of a $\mathcal{VO}$ being linear in the size of a potentially much larger intermediate result (if $\mid u \mid \gg \mid u \cap v \mid$). A similar problem occurs with set differences $u - v$.

We have not solved the general problem of constructing $\mathcal{VO}$'s linear in the size of the result for intersections and set differences. Indeed, the question remains as to whether (in general) linear-size $\mathcal{VO}$s *can* even be constructed for these objects. However, in the following section, we provide an approach to constructing linear-size $\mathcal{VO}$s for a specific type of intersection, *range query*. This is accomplished using a data structure drawn from computational geometry called a *multi-dimensional range tree.* This approach also work set differences over range querys on different attributes.

5 Multi-dimensional verification objects

In d-dimensional computational geometry, when one is dealing with sets of points in d-space, one could ask a d-space range query. Consider a spatial interval $(< x_1^1, x_2^1 > \ldots < x_1^d, x_2^d >)$: this represents a axis-aligned rectilinear solid in d-space. A query could ask for the points that occur within this solid.

Such problems are solved efficiently in computational geometry using so-called *Range Trees* (See [8], Chapter 5). We draw an analogy between this problem and a database query of the form

$$\sigma_{c_1^1<A_1<c_1^2}(r) \bigcap \cdots \bigcap \sigma_{c_d^1<A_d<c_d^2}(r)$$

where $\{A_1, \ldots A_d\} \subseteq schema(R)$ for a relation R. We use the multi-dimensional range tree ($mdrt$) data structure to solve such queries and provide compact verification objects.

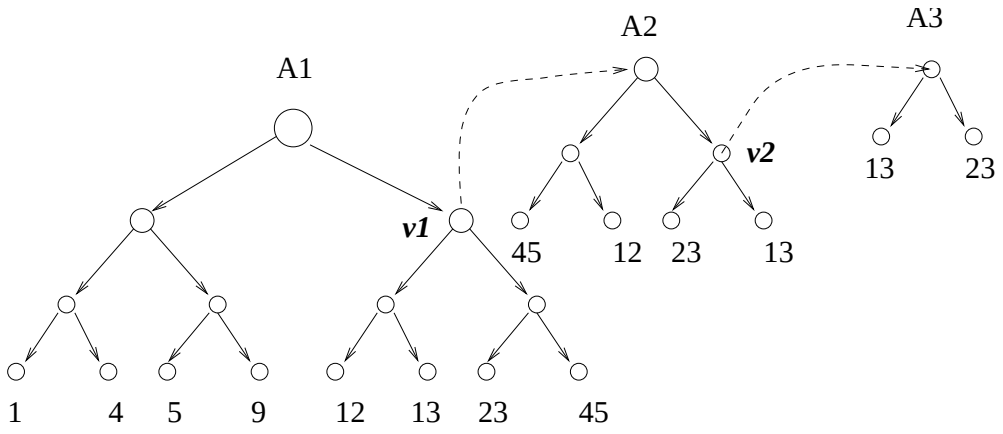

Figure 3: Excerpt of a 3-dimensional range tree, sorted by attributes A_1, A_2 and A_3

Consider the example $mdrt$ shown in Figure 3. Let us assume a relation schema R, with 3 attributes A_1, A_2, A_3 on which we want to perform combined selections, and provide $\mathcal{VO}$'s for the final answer. Consider the first, 3-dimensional $mdrt$ (labeled A1). This is simply a tree which sorts the tuples according to attribute A_1. The numbers at the leaves denote unique primary keys for the tuples. Each interior node in tree A1 is the ancestor of a set of tuples. Consider such an interior node $v1$, which is the ancestor of tuples 12, 13, 23, and 45. An $mdrt$ contains a link now from $v1$ to the root of an *associated tree* A2, (which we can also denote as $T_{assoc}(v1, A_2)$). This 2-dimensional $mdrt$ A2 contains the same set of tuples 45, 11, 23, and 13; however, in this tree, they are sorted according to attribute A_2. Likewise each interior node v_i in A1 is the ancestor to a set of tuples, and contains a pointer to an associated 2-dimensional $mdrt$ $T_{assoc}(v_i, A_2)$ which sorts the tuples in the subtree below v_i by attribute A_2. In general, each node v_i^j of a $\{d-j+1\}$-dimensional $mdrt$ contains a pointer to a $\{d-j\}$-dimensional $mdrt$. The nodes of the final 1-dimensional tree, corresponding to attribute A_d, do not have such pointers.

Given a 2-space range query $\sigma_{x_1^1 \le A_1 \le x_2^1}(r) \cap \sigma_{x_1^2 \le A_2 \le x_2^2}(r)$

the structure is used as follows. First, the tuple set $q \subseteq r$ with values for attribute A_1 in the range $< x_1^1, x_2^1 >$ is identified using tree A1. For simplicity, let us assume (we relax this assumption later) that the tuples in q form the leaves of a balanced tree with root $LCA(q)$. With $\mid r \mid = n$, this range can be found in roughly time $O(\log_2 n)$, the time it takes to find the two end-points of the interval in the first tree. We now follow the link over the associated tree for attribute A_2; this tree sorts just the tuple set q according to attribute A_2. So we can find the subset of q satisfying $x_1^2 \le A_2 \le x_2^2$ also in $O(\log_2 n)$ time. This gives the intuition behind the efficient processing of conjunctive range queries using $mdrt$s. We now relax the assumption that the result of the first selection q includes just the leaves of a balanced tree.

Let us call the leaves of the subtree rooted at node v the *canonical subset* of v, denoted as $P(v)$. If v is a leaf, $P(v) = v$. In [8] (pp 103-107) it is shown that *any* subset of leaves which lies in a range can be expressed as a disjoint union of $O(\log_2 n)$ canonical subsets for the given range query. The roots

of these can be found in $O(\log_2\ n)$ time in the process of finding the bounding paths for the interval. Given a range $< x, x' >$, we search for them in the tree until we find node v_{split} where the paths split. Now we search x (x') in the left (right) subtree. At every point in the search for x where the path goes left, the right subtree belongs to the range; the search for x' goes just the opposite way. The result is a quick identification of roots of the canonical subtrees that precisely cover the leaves whose values are in the interval (see Figure 4).

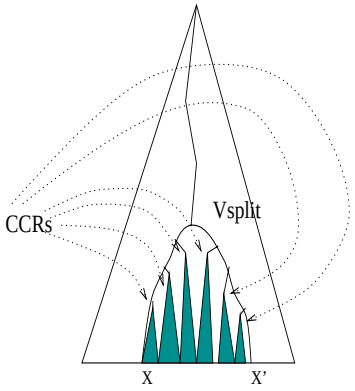

Figure 4: Finding the canonical covering roots

We call these the *covering canonical roots (CCRs)*. Consider, for example, a 2-dimensional range query over attributes A_1 and A_2. First, the CCRs in tree A1 for the given range in attribute A_1 are found; there are $O(\log n)$ of these. Then for each of the CCRs, we go to the associated trees for attribute A_2, and find the CCRs in *that* tree for the given range over A_2. This results in $O(\log^2\ n)$ CCRs in tree A2. The union of all the leaves under these CCRs in A2 constitute the answer. In general, it is shown [8] that d-dimensional range queries can be computed in time $O(\log^{d-1}\ n)$. Range trees require $O(n \log^{d-1} n)$ storage space, and can be constructed in time $O(n \log^{d-1}\ n)$.

We now show how to produce $\mathcal{VO}$ of size $O(\log^{d-1}\ n)$ for an intersection query of the form:

$$\sigma_{c_1^1 < A_1 < c_1^2}(r) \bigcap \cdots \bigcap \sigma_{c_d^1 < A_d < c_d^2}(r)$$

using *mdrt*s. First, we construct a Merkle hash tree over a d-dimensional *mdrt*. Assume that the associated tree for a node i (for the next attribute) in a range tree is given by $A(i)$, and the hash of a node i given by $h(i)$.

Base Case For the base (1-dimensional) *mdrt*s, we build the Merkle hash tree as before.

Induction Given the root hashes for the $\{l-1\}$-dimensional *mdrt*s, we begin with the leaves (height $j = 0$) of the l-dimensional *mdrt*, and compute the tuple hashes, in the usual way. For a node i at height $j > 0$, we compute the hash value thus. First, we append the hashes of all the children of i (say i_1, i_2) together, and also the hash of root of the associated $l-1$-dimensional range tree, and then hash the result.

$$h(i) = h(h(i_1) \mid\mid h(i_2) \mid\mid h(A(i)))$$

This construction can be completed in time $O(n \log^{d-1}\ n)$, and can be overlapped with the construction of the range tree itself.

Now consider the construction of a $\mathcal{VO}$ for a d-way intersecting range selection query. The $\mathcal{VO}$ essentially follows the search algorithm. In any given tree, a range query corresponding to an interval results in a set of CCRs. The $\mathcal{VO}$s for this group of $O(\log\ n)$ CCRs is presented by providing a verification path for the node V_{split} to the root, and verifiable paths (length $O(1)$) from each CCR to

the path to V_{split}. In addition, it can readily be seen that by construction of the path from the CCR's to V_{split}, the canonical subtrees of the CCR's form a contiguous non-overlapping over of the answer set. We also provide proximity trees for the LUB and GLB of the interval and the smallest and largest contained intervals; finally, we must show that each of the $k, k =\mid q \mid$ selected tuples belongs under a CCR. The total size of this $\mathcal{VO}$ is $O(\log\ n + k \log n)$, or $O(k \log n)$. This process needs to repeat, showing verification paths for all the $O(\log^d\ n)$ CCR's found in the process of evaluating the d-way intersecting range selection query, which gives us a $\mathcal{VO}$ size of $O(kd \log n + \log^d\ n)$, which would show that the k tuples belong to the answer, and that the $O(\log^d\ n)$ CCRs together cover the intervals prescribed by the query. In situations where the results of each selection may be large, and the final intersection is small, this approach gives us attractively small $\mathcal{VO}$s.

6 Pragmatic Issues and Related Work

We now examine some pragmatic considerations in using our approach, as well as related work.

6.1 Canonical Join-Select-Project queries

A typical SQL "**select** ... **from** ... **where** ..." can be thought of one or more joins, followed by a (combined) selection, followed by a projection. We describe how an *mdrt* can be used for both efficient evaluation and construction of compact $\mathcal{VO}$s for such queries. Specifically, consider a query that involves the join of two relations R and S, followed by a series of selections and a final projection. Let's assume a Theta-join over attribute A_1 (A_1 occurring in both relations), followed by selections on attributes A_2 and A_3, and a final projection on several attributes, jointly represented by A_4 (as discussed in Section 4.2).

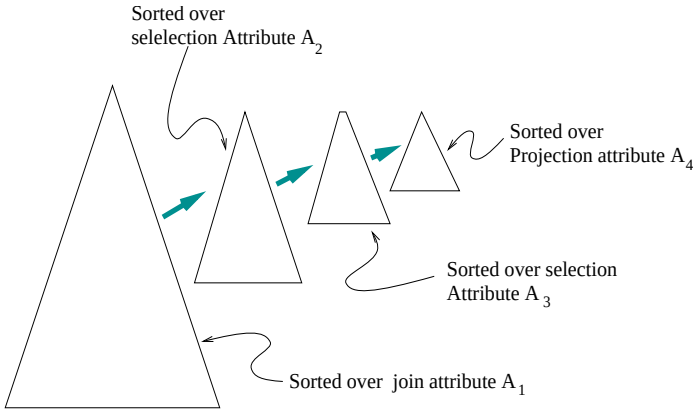

Figure 5: Excerpt of a 3-dimensional range tree, sorted by attributes A_1, A_2 and A_3

We begin this construction (see Figure 5) with the first range tree constructed over the join attribute A_1, as explained in Section 4.3. Then follow the trees sorted by A_2 and A_3, and finally on the projected attributes. Given a query, the evaluation plan (and the construction of the $\mathcal{VO}$) follows this set of range trees as described above, leading to both efficient evaluation of all the steps, and a $\mathcal{VO}$ linear in the size of the final result.

6.2 Query flexibility

For efficient verification of query answering, we make use of different trees over sorted tuple sets. Without such trees, our approach cannot provide small verification objects. This points to a limita-

tion of our approach—only queries for which Merkle trees have been pre-computed can be evaluated with compact verification objects. Our approach cannot support arbitrary interactive querying with compact verification objects. Arbitrary interactive querying, however, is quite rare in the presence of fixed applications at *client* sites.

In practice, however, data-intensive applications make use of a fixed set of queries. Indeed, via mechanisms such as embedded SQL (see, e.g., [17]) database queries are compiled into applications. These queries can still make use of parameters entered by a user and which are typically used in selection conditions. Our approach is compatible with such applications. Essentially, client applications commit *a priori* the queries they wish to execute; the owner and the publisher then pre-compute the required Merkle hash trees to produce short verification objects.

So while our approach cannot provide compact verification objects in the context of arbitrary interactive database querying, it is quite compatible with the widely-used practice of embedding predetermined (and parameterizable) queries within data-intensive applications.

6.3 Conventions

It is important to note that all interested parties: the *owner*, the *publisher* and the *client*, share a consistent schema for the databases being published. In addition there needs to be secure binding between the schema, the queries and the query evaluation process over the constructed Merkle trees. A convention to include this information within the hash trees needs to be established. All parties also need to agree on the data structures used for the $\mathcal{VO}$. It is also important that the *publisher* and the *client* agree upon the format in which the $\mathcal{VO}$ together with the query result is encoded and transmitted. Tagged data streams such as XML provide an attractive option.

6.4 Recent Query Evaluations

Verifiers must verify that query evaluation is due to an "adequately recent" snapshot of the database and not an old version. We assume the technique of recent-secure authentication for solving this problem. Risk takers (e.g. organizations relying on the accuracy of the data) specify freshness policies on how fresh the database must be. The digital signatures over the database include a timestamp of the last update as well as other versioning information. Based on assumptions concerning trusted synchronization paths and synchronization bounds , clients interpret the timestamps and verify the database is adequately recent with respect to their freshness policies.

6.5 Related Work

The use of Merkle hash trees for authentication of data is not new. This work is most closely related to the work of Naor & Nissim [13] for revocation. Haber and Stronetta [7] use similar techniques for timestamping. Similar schemes [16] have also been used for micropayments. All these schemes (including ours) share a common theme of leveraging the trust provided by a few digital signatures from a trusted party over multiple hashes, hash paths or hash trees, with the goal of protecting the integrity of the content, with efficient verification, since hashes are more efficient than digital signatures. However, the use of such trees for authentic data publication is new.

There is quite bit of related work in the general area of database security, particularly on access control, statistical querying etc [4, 10]. Anderson [2] discusses an approach to third-party publication of data in $files$, but without querying over the contents. Again, to our knowledge, this particular problem of authentic database publication has remained unexamined.

Finally, our approach can be viewed as providing "proof-carrying" [14] answers to database queries.

7 Conclusion

We have explored the problem of authentic third party data publication. In particular, we have developed several techniques that allow untrusted third parties to provide evidence of *inclusive* and *complete* query evaluation to clients without using public-key signatures. In addition, the evidence provided is linear in the size of the query answers, and can be checked in linear time. Our techniques do involve the construction of complex data structures, but the cost of this construction is amortized over more efficient query evaluation, as well as the production of compact verification objects. Such pre-computation of views and indexing structures are not uncommon in data warehousing applications [15].

Our techniques suggest the use of a single hash function. In particularly high-integrity applications where tolerance of failure is very low, one can use multiple one-way hash functions to construct each Merkle tree. Clients requiring higher levels of integrity may check more than one hash computation.

However, our techniques are restricted currently to the relational model. Our techniques do not allow interactive querying, but can be used with embedded queries in applications. We cannot currently construct linear-size $\mathcal{VO}$s general SQL queries, such as ones including arbitrary intersections; we also leave open the (lower-bound) question as to whether such $\mathcal{VO}$s are possible. We believe, however, that our techniques are a start on an important problem area, and subsequent work will perhaps remove some of these limitations.

References

[1] N.M. Amato and M.C. Loui. Checking linked data structures. In *Proceedings of the 24th Annual International Symposium on Fault-Tolerant Computing (FTCS)*, 1994.

[2] R. J. Anderson. The Eternity Service. *In Proceedings of Pragocrypt*, 1996.

[3] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Noar. Checking the inclusiveness of memories. *Algorithmica*, 12(2/3):225–244, 1994. Originally appeared in *FOCS* 91.

[4] S. Castano, M. Fugini, G. Martella, P. Samarati Database Security Addison-Wesley, 1995

[5] C.J. Date. An Introduction to Database Systems (7th Ed), Addison-Wesley, 1999.

[6] C.J. Date and H. Darwen. A Guide to the SQL Standard (4th Ed), Addison Wesley, 1997.

[7] S. Haber and W. S. Stornetta. How to timestamp a digital document *J. of Cryptology*, 3(2), 1991.

[8] M. D. Berg , M. V. Kreveld, M. Overmars and O. Schwarzkopf. *Computational Geometry.* Springer-Verlag, New York.

[9] S. Jajodia, P. Samarati, V. S. Subramanian, E. Bertino, A Unified Framework for Enforcing Multiple Access Control Policies *Proceedings ACM SIGMOD*, 1997.

[10] T. Lunt, (Ed.) Research Directions in Database Security Springer-Verlag, 1992

[11] R.C. Merkle. A certified digital signature. In *Advances in Cryptology–Crypto '89*, 1989.

[12] J. Melton, A.R. Simon. Understanding the New SQL. Morgan Kaufmann, 1993.

[13] M. Naor, K. Nissim. Certificate Revocation and Certificate Update. *Proceedings, 7th USENIX Security Symposium.* 1998.

[14] G. Necula. Proof-carrying code. In *Proceedings of POPL 97.* ACM SIGPLAN, 1997.

[15] W.H. Inmon. Building the Data Warehouse John Wiley & Sons, 1996.

[16] S. Charanjit and M. Yung. Paytree: Amortized Signature for flexible micropayments *Second Usenix Workshop on Electronic Commerce Proceedings*, 1996

[17] A. Silberschatz, H. Korth, S. Sudarshan: Database System Concepts, McGraw-Hill, 1997.

[18] J. D. Tygar Open Problems In Electronic Commerce *Proceedings ACM SIGMOD PODS*, 1999.

Appendicitis

We assume that the client knows R_h the correct root hash value for the Merkle tree associated with our relation/attribute. We also assume that the client has a program (including the hash function h) which will correctly check the $\mathcal{VO}$ and the answer. We now show that as long as an adversary cannot forge values which cause collisions in h (more precision on this below), the client will never verify an incorrect answer.

For now, we include here only the proof that simple selections are unforgeable. This proof is an induction on the height of the Merkle tree corresponding to that selection The full proof shows that checks for all other operators are also unforgeable; then we need another induction on the nesting depth of the operators in the query. This proof is also only for binary trees, but is easily extended to multiway trees such as B-trees.

Before giving the proof we need some preliminaries. Let the correct answer S to a selection query on relation r be the tuples $t_j \leq t_{j+1} \ldots \leq t_r$. If t_j isn't the smallest tuple in r we also return t_{j-1}, and if t_r isn't the largest tuple in r we return t_{r+1}, which are the largest and smallest tuples not in S. We call this larger set S_b (S plus the boundary tuples). We assume that the tree also may have two special tuples which are smaller than anything in r and larger than anything in r (in case the smallest tuple is a right child or the largest a left child). In addition to S_b, the client is provided with the hash values of some internal nodes in the Merkle tree. Specifically, the answer can be computed by the *publisher* as follows by initially calling the function *Answer* with R, the root of the tree. We show below the definition of *Answer*:

Answer(v):

- If all leaves in the subtree rooted at v are in S_b, then return the single tuple if v is a leaf, otherwise the values of all the leaves as a pair [leaves in left subtree], [leaves in right subtree];
- If no leaf in the subtree rooted at v is in S_b, then return the hash value of v in the Merkle tree;
- If some but not all leaves in the subtree rooted at v are in S_b, then return the pair [*Answer*(left-child(v))], [*Answer*(right-child(v))];

Note that this construction algorithm for the answer implies a simple checking mechanism: compute the hash value for the left child u_l of the root using the string in the leftmost group of [ ], then compute the right child value v_r, finally compute R_h using these two values. To check validity the *client* will also compute a status variable for the current subtree rooted at v. This status can be:

- *empty*: no tuples under v were returned;
- *right-terminated (RT)*: a tuple too large to be in S is under v;
- *right-full (RF)*: valid, but not right terminated;
- *left-full (LF) and left-terminated (LT)* are defined similarly;
- *invalid*: the tuples under v are known to be an invalid answer.

We compute the status as follows: a leaf is both left-full and right-full if it is in S; left-terminated and right-full if too small; and left-full and right-terminated if too large.

In general for a node v, if either child is invalid, v will also be invalid. If both children are valid the following table shows how to get the parent's status. L stands for LT or LF and R for RT or RF. If an L or R appears as a child and parent it has the same status both times.

left	*right*	*parent*
empty	empty	invalid
empty	LF,R	invalid
empty	LT,R	LT, R
non-empty	LT,R	invalid
LT,RF	LF,R	LT,R
LF,RF	LF,RF	LF,RF
L,RT	empty	L,RT
L,RT	non-empty	invalid.

This is not an exact security argument. We are not considering the security parameters.

Attack Model We assume that the *owner* has constructed a Merkle hash tree of a relation r over the selection attribute A, and the client knows the correct root hash value R_h.

We assume that *publisher* is the adversary, has access to the relation r, and can construct the same Merkle hash tree; we also assume that the hash function chosen by *owner* cannot be feasibly forged. Specifically, we make the following assumption about the adversary. For any selection query there is a correct answer which the client is supposed to receive, and a correct verification computation. In the course of the correct verification computation the *client* first hashes each tuple given, and then creates a sequence of triples x_j, y_j, z_j where $z_j = h(x_j, y_j)$ is the result of the jth application of the hash function. We assume that the adversary cannot do any of the following:

1. Concoct a tuple t such that t is not in r and $h(t) = h(t_j)$ for t_j in r,
2. Concoct an answer such that in the course of the protocol *client* would compute $z_j = h(a_j, b_j)$ and $a_j \neq x_j$ or $b_j \neq y_j$ (intuitively this would mean the adversary has found an alternate pair of values which hash to the same answer), and
3. Concoct an answer such that in the course of the protocol *client* would compute $h(a_j, b_j) = h(t_j)$ for t_j in r (we don't need this if the *client* knows the height of the tree).

Lemma 6 *Under the above cryptographic assumptions, when* client *executes the verification protocol on an* Answer *ADV provided by* publisher*, if* client *computes the correct root hash value R_h and the root status is valid, ADV is in fact a correct answer and the status of the root is correct.*

Basis We show this by induction. Consider a Merkle hash tree of height 1; this must be a root with just two leaves t_1, t_2. Since the root hash R_h is known, by definition, the final step of an accepting protocol must be to compute $h(v_l, v_r) = R_h$ where $v_l = h(t_1)$ and $v_r = h(t_2)$ are the correct hash values of the left and right subtrees. By our security assumption the only way the protocol could get v_l, v_r is either by being given them directly or by being given a correct value of t_1 or t_2 and hashing them. The protocol cannot be given both v_l and v_r directly since this would mean S_b is empty which the *client* knows is impossible. Thus the only way ADV might be accepted is if it provides two tuples or one tuple and a value.

Case 1 The answer ADV is two tuples t_1 and t_2 which are children of the root (if r has only one tuple the other tuple will be the dummy tuple for maximum/minimum attribute value, if r is empty both tuples will be dummy).Thus t_1 and t_2 are both claimed to be in S_b. The *client* will always be able to look at the tuples to see which are actually part of the correct answer and which are boundary tuples. The user than computes $v_l = h(t_1)$, $v_r = h(t_2)$, and finally $V = h(v_l, v_r)$. By our assumption this is the only hash computation that can occur while processing ADV which can have R_h as its answer. Furthermore, no hash computation other than those listed above can have v_1 or v_2 as its results. Thus

$V = R_h$ if and only if t_1 and t_2 are the correct tuple values and no other values are provided. The protocol also has a completeness proof. For example, if t_1 is in S and t_2 is too large to be in r, since the *client* knows that t_2 is t_1's right sibling, t_1 must be the largest tuple in S. In this case the root status would be left-full, right-terminated.

Case 2 The adversary could also get the correct root value computed by providing one tuple value (say t_1), and the hash value of the other tuple (in essence claiming that $S_b = \{t_1\}$). However, this means the protocol would see the right subtree as empty, so the root is only valid if the left subtree is right terminated. This would only be the case if t_1 is too large to be in S, and since the algorithm has the correct value of t_1 it will know if this is the case.

The setting where ADV is $[v_l], [t_2]$ is analogous.

Step We now assume that the lemma holds for any tree of height less than i, namely that when we apply our protocol to a non-empty tree of height less than i, we get the correct root value and valid status if and only if we are given a correct answer (tuples and hash values) for that tree.

Now consider a Merkle hash tree of height i rooted at R of height $i \geq 2$, with two immediate subtrees rooted at u_l and u_r with hash values v_l and v_r. ADV must be of the form:
$[A_l]$, $[A_r]$
where A_l and A_r are answer strings which our Answer protocol can parse. If the *Answer* doesn't have this high level form it will be immediately rejected since a correct answer cannot be empty (it always has the boundary tuples).

As the final step of an accepting protocol we must compute $R_h = h(v_l, v_r)$. By our assumption, the only way the protocol when run on ADV can yield R_h as its final value is if the protocol evaluates A_l to v_l and A_r to v_r and ends with a valid status for both nodes.

The trees rooted at u_l and u_r are of height $i-1$. Furthermore, the protocol treats non-empty subtrees exactly as it does the entire tree. Thus by the induction hypothesis, if A_l is non-empty, and produces the correct hash value and a valid status, then it must include the correct tuples and end with the correct status for u_l (and similarly for u_r).

If A_l, A_r are both non-empty, evaluate to the correct hash values, and have a valid status, they are the correct trees and thus must fit together in a valid way (e.g. we will never have them both be right terminated). However, if one of the trees is empty we cannot apply our induction hypothesis to it (since this only works for non-empty trees). So suppose that u_l is empty and we have simply been given $A_l = v_l$ as part of ADV (this is easy for the adversary since v_l is known). In order for the root to get a valid status, the status of u_r must be left-terminated (we don't care if its right-terminated or full, either is OK). However, if u_r is left-terminated and correct, we know that u_l should be empty, and we are again correct to conclude that the root status is valid. The case where u_r is empty is analogous.

Thus in all cases we can extend the correctness of our protocol to a tree of height i and the lemma follows.