IntegriDB: Verifiable SQL for Outsourced Databases

Yupeng Zhang
ECE Dept. & UMIACS
University of Maryland
zhangyp@umd.edu

Jonathan Katz
CS Dept. & UMIACS
University of Maryland
jkatz@cs.umd.edu

Charalampos Papamanthou
ECE Dept. & UMIACS
University of Maryland
cpap@umd.edu

ABSTRACT

This paper presents INTEGRIDB, a system allowing a data owner to outsource storage of a database to an untrusted server, and then enable anyone to perform verifiable SQL queries over that database. Our system handles a rich subset of SQL queries, including multi-dimensional range queries, `JOIN`, `SUM`, `MAX/MIN`, `COUNT`, and `AVG`, as well as (limited) nestings of such queries. Even for tables with 10^5 entries, INTEGRIDB has small proofs (a few KB) that depend only *logarithmically* on the size of the database, low verification time (tens of milliseconds), and feasible server computation (under a minute). Efficient updates are also supported.

We prove security of INTEGRIDB based on known cryptographic assumptions, and demonstrate its practicality and expressiveness via performance measurements and verifiable processing of SQL queries from the TPC-H and TPC-C benchmarks.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

Keywords

Verifiable Computation; Authenticated Data Structures

1. INTRODUCTION

With the advent of cloud computing, there has been significant interest in techniques for ensuring correctness of computations performed by an untrusted server on behalf of a client. An *authenticated data structure* [22, 41] (ADS) allows a *data owner* to outsource storage of data to a *server*, who can then verifiably answer queries posed by multiple *clients* on that data. (Also related is work on *verifiable computation*; see Section 1.1 for further discussion.) Of particular importance is the *expressiveness* of an ADS; namely, the class of queries it can support.

In this work we are interested in designing an ADS and an associated system that supports *native SQL queries* over a relational database. Such a system would be suitable for integration into the most prevalent applications running in the cloud today, and could be offered as a software layer on top of any SQL implementation.

Toward this end we design, build, and evaluate INTEGRIDB, a system that efficiently supports verifiability of a rich subset of SQL queries. Specific advantages of INTEGRIDB include:

1. INTEGRIDB is *expressive*. It can support multidimensional range queries, `JOIN`, `SUM`, `MAX/MIN`, `COUNT`, and `AVG`, as well as (limited) nestings of such queries. As an illustration of its expressiveness, we show that INTEGRIDB can support 12 out of the 22 SQL queries in the TPC-H benchmark, and support 94% of the queries in the TPC-C benchmark.

2. INTEGRIDB is *efficient*. Even for tables with hundreds of thousands of entries, INTEGRIDB is able to achieve small proofs (a few KB), low verification time (tens of milliseconds), and feasible server computation (under a minute). In fact, the proof size and verification time in INTEGRIDB are *independent* of the size of the stored tables (up to logarithmic factors). INTEGRIDB also supports efficient updates (i.e., `INSERT` and `DELETE`).

3. INTEGRIDB is *scalable*, and we have executed it on database tables with up to 6 million rows.

4. INTEGRIDB can be proven secure based on known cryptographic assumptions.

INTEGRIDB outperforms state-of-the-art verifiable database systems in terms of its expressiveness, performance, and scalability. We provide a comparison to prior work next.

1.1 Comparison to Prior Work

Relevant prior work which could be used to construct an ADS handling some subset of SQL queries can be classified into two categories: *generic approaches* and *specific approaches*. We discuss these in turn. (See also Table 1.)

Generic approaches. A verifiable database supporting any desired SQL query (of bounded size) could be derived in principle using general-purpose techniques for verifiable computation [15, 5, 6, 16, 7], or one of the systems that have been built to apply these techniques [39, 38, 37, 2, 9, 42, 4, 3, 13]. To apply this approach, the data owner would first compile its database into a program (expressed either as a boolean/arithmetic circuit, or in the RAM-model of computation) that takes as input a query and returns the corresponding result. This, however, will not necessarily yield a practical protocol. Circuit-based techniques will be inefficient since the size of a circuit for the program just described will be at least as large as the database itself; moreover, using a circuit-based approach will not allow efficient updates. RAM-based systems will

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

DOI: http://dx.doi.org/10.1145/2810103.2813711.

Table 1: Comparing the expressiveness of INTEGRIDB with prior work. A ✓ is placed only if a given scheme supports a given query with proofs whose size is independent of the table size (up to logarithmic factors). Note that although generic systems are as expressive as INTEGRIDB, they are less efficient in practice (see Section 7). Below, FUNC can be any of {SUM, MAX/MIN, COUNT, AVG}.

reference	JOIN	multidimensional range queries	JOIN on multidimensional range queries	FUNC on intermediate results	polylog updates
INTEGRIDB	✓	✓	✓	✓	✓
RAM-based [9, 2, 4, 3]	✓	✓	✓	✓	✓
circuit-based [37], [4, libsnark], [13]	✓	✓	✓	✓	✗
tree [43] or signature [31]-based	✗	✗	✗	✗	✗
multirange-based [32]	✗	✓	✗	✗	✗

partly address these issues; however, as we show in Section 7.3, their performance will still be worse than that of INTEGRIDB.

Specific approaches. Prior work has also explored custom-built verifiable database systems. (INTEGRIDB is in this category.) We can roughly categorize existing approaches as being either *tree-based* (using Merkle trees [23] as a main component) or *signature-based* (where the server stores values pre-signed by the data owner).

Most tree-based approaches [14, 30, 19, 40, 34, 43, 44, 20, 21, 45] support single-dimensional range queries, but not multi-dimensional range queries. Signature-based approaches (e.g., [25, 26, 27, 29, 11, 31, 18]) also only support single-dimensional range queries; moreover, they do not support efficient updates. Martel et al. [22] present a scheme supporting multidimensional range queries, but with costs exponential in the number of dimensions.

Papadopoulos et al. [32] recently showed a scheme explicitly designed for multidimensional range queries; however, their scheme does not support efficient updates.

None of the schemes mentioned above support short proofs for JOIN queries. Specifically, schemes supporting JOIN queries [29, 19, 43, 31, 31] rely on the idea of Pang et al. [29], which requires one column to be returned to the client who must then issue a range query (on the other column) for every element in the returned column. Thus, the proof size and verification time are, in the worst case, linear in the size of the smallest column involved in the query.

In addition, and in contrast to INTEGRIDB, none of the above schemes support short proofs for functions (such as SUM, MAX/MIN, or COUNT) applied on intermediate results (such as summing the values of a column that is the output of a JOIN). Instead, these schemes would require returning the intermediate result back to the client who would then compute the function locally.

Trusted hardware. Bajaj et al. [1] propose a scheme for verifiable SQL that relies on trusted hardware. INTEGRIDB does not make any assumption about trusted hardware.

2. PRELIMINARIES

We let λ denote the security parameter, negl be a negligible function, and PPT stand for "probabilistic polynomial time." We use $\tilde{O}(f(n))$ for $O(f(n) \cdot \mathsf{polylog}(f(n)))$, and let $[n] = \{1, \ldots, n\}$.

2.1 Authenticated Data Structures

We define the abstract notion of an authenticated data structure (ADS) [22, 41], which allows a data owner to outsource data to a server that can then be queried by clients with assurance about the correctness of the result. Particular ADSs are distinguished by the class of queries $\mathcal{Q}$ and type of updates U (if any) they support. By way of notation, we denote the (true) result of applying query Q to data D by $R = Q(D)$, and denote the result of applying an update operation upd to data D by $D' = \mathsf{upd}(D)$.

We consider three types of parties: a *data owner*, a *server*, and a *client* (which may be the data owner itself). To outsource storage of data D to the server using an ADS for some class of queries $\mathcal{Q}$, the data owner first runs an initialization algorithm that outputs a secret sk and a public key pk, followed by a setup algorithm that takes sk and D and outputs a digest δ and an authenticated version of the data $\tilde{D}$. It gives $\tilde{D}$ to the server and publishes δ and pk. Any client can issue a query $Q \in \mathcal{Q}$ to the server; in return, the server uses $\tilde{D}$ and pk to compute a response R along with a proof π. The client can verify correctness of the response R using δ, π and pk. Security ensures that if the client accepts, then R is equal to $Q(D)$ (except with negligible probability).

We will be interested in *dynamic* ADSs that also support updates. When the data owner wishes to update the data D stored at the server, it interacts with the server; assuming the server behaves correctly, the result is a new value $\tilde{D}'$ stored by the server as well as a new digest δ'. (If the server behaves dishonestly, the data owner will realize this and reject.) We assume clients are always able to obtain a fresh copy of the most recent value of the digest; if not, freshness of the results cannot be guaranteed.

Formally, a dynamic ADS for query class $\mathcal{Q}$ and updates U consists of efficient algorithms Init, Setup, Prove, Verify, $\mathsf{UpdateO}$, and $\mathsf{UpdateS}$ that work as follows:

1. Algorithm Init takes as input 1^λ, and outputs a secret key sk and a public key pk. The public key is implicitly provided to all algorithms below.

2. Algorithm Setup takes as input data D and the secret key sk, and outputs a digest δ and authentication information $\tilde{D}$.

3. Algorithm Prove takes as input $\tilde{D}$, δ, and $Q \in \mathcal{Q}$. It returns a result R and a proof π.

4. Algorithm Verify takes as input digest δ, query $Q \in \mathcal{Q}$, result R, and proof π. It outputs 0 or 1.

5. $\mathsf{UpdateO}$ and $\mathsf{UpdateS}$ are interactive algorithms run by the data owner and server, respectively. $\mathsf{UpdateO}$ takes as input the secret key sk, a digest δ, and an update $\mathsf{upd} \in U$, while $\mathsf{UpdateS}$ takes as input $\tilde{D}$. After completing their interaction, $\mathsf{UpdateO}$ outputs a digest δ' along with a bit indicating acceptance or rejection. (Rejection implies $\delta' = \delta$.) $\mathsf{UpdateS}$ outputs $\tilde{D}'$.

For an ADS to be non-trivial, the sizes of δ and π should be much smaller than the size of D.

Correctness of an ADS is defined in the natural way and is omitted. The definition of security is also intuitive, though the formalism is a bit cumbersome. Consider the following experiment based on an ADS specified by the algorithms above and an attacker $\mathcal{A}$, and parameterized by security parameter λ:

Step 1: Run $(sk, pk) \leftarrow \mathsf{Init}(1^\lambda)$ and give pk to $\mathcal{A}$, who outputs a database D. Then $(\delta, \tilde{D}) \leftarrow \mathsf{Setup}(D, sk)$ is computed, and $\mathcal{A}$ is given $\delta, \tilde{D}$. Values sk, δ, D are stored as state of the experiment.

Step 2: $\mathcal{A}$ can run either of the following two procedures polynomially many times:

- Query:
  - $\mathcal{A}$ outputs (Q, R, π) with $Q \in \mathcal{Q}$, after which $b = \mathsf{Verify}(\delta, Q, R, \pi)$ is computed.
  - Event attack occurs if $b = 1$ but $R \neq Q(D)$.
- Update:
  - $\mathcal{A}$ outputs $\mathsf{upd} \in U$, and then interacts (playing the role of the server) with $\mathsf{UpdateO}(sk, \delta, \mathsf{upd})$ until that algorithm halts with output (δ', b). The digest δ' is given to $\mathcal{A}$.
  - Set $\delta := \delta'$. Also, if $b = 1$ set $D := \mathsf{upd}(D)$. (Recall that δ, D are stored as part of the state of the experiment.)

DEFINITION 1. *An ADS is* **secure** *if for all* PPT *adversaries $\mathcal{A}$, the probability that* attack *occurs in the above is negligible.*

2.2 SQL queries Supported by IntegriDB

We briefly describe the SQL queries that INTEGRIDB supports. Every SQL database consists of a collection of SQL *tables*, which are two-dimensional matrices. As we explain each query, we give an example using a database containing two tables, Table A:

row_ID	student_ID	age	GPA	First_name
1	10747	22	3.5	Bob
2	10715	24	3.3	Alice
3	10721	23	3.7	David
4	10781	21	3.0	Cathy

and Table B:

row_ID	student_ID	Year_enrolled
1	10715	2010
2	10791	2012
3	10747	2011
4	10771	2013

1. `JOIN`: A join query is used to combine rows from two or more tables, based on common values in specified columns. E.g., the query "`SELECT` A.student_ID, A.age, A.GPA, B.Year_Enrolled `FROM` A `JOIN` B `ON` A.student_ID = B.student_ID" returns:

student_ID	A.age	A.GPA	B.Year_Enrolled
10747	22	3.5	2011
10715	24	3.3	2010

2. Multidimensional range: A range query selects rows whose values in one or more specified columns lie in a certain range. The *dimension* of such a query is the number of columns involved. E.g., the two-dimensional query "`SELECT` * `FROM` A `WHERE` (age `BETWEEN` 22 `AND` 24) `AND` (student_ID > 10730)" returns:

row_ID	student_ID	age	GPA	First_name
1	10747	22	3.5	Bob

3. `SUM`, `MAX`, `MIN`, and `COUNT`: These queries return the sum, maximum, minimum, or number of the entries in a specified column. E.g., the query "`SELECT SUM`(age) `FROM` A" returns 90.

4. `LIKE`: This query finds rows with strings in a specified column matching a certain pattern. E.g., the query "`SELECT` * `FROM` A `WHERE` First_name `LIKE` 'Ali%' " returns:

row_ID	student_ID	age	GPA	First_name
2	10715	24	3.3	Alice

5. Nested queries: Since the answer to any SQL query is itself a table, SQL queries can be nested so that queries are applied to intermediate results. For example, the query "`SELECT SUM`(A.age) `FROM` A `JOIN` B `ON` A.student_ID = B.student_ID" returns 46, which is the summation of the column "A.age" in the table generated by the `JOIN` query earlier. As a more complicated example, a `JOIN` query can be applied to the result of a multi-dimensional range query. E.g., the query "`SELECT` A.student_ID, A.age, A.GPA, B.Year_Enrolled `FROM` {`SELECT` * `FROM` A `WHERE` (age `BETWEEN` 22 `AND` 24) `AND` (student_ID > 10730)} `JOIN` B `ON` A.student_ID = B.student_ID" returns:

student_ID	A.age	A.GPA	B.Year_Enrolled
10747	22	3.5	2011

3. BUILDING BLOCKS

In this section we describe two authenticated data structures that we use as building blocks in INTEGRIDB: one for set operations (that we call $\mathcal{ASO}$), and one for interval trees (that we call $\mathcal{AIT}$).

3.1 ADS for Set Operations with Summation

INTEGRIDB uses an ADS for verifying set operations that we introduce here. Our construction is based on prior work of Papamanthou et al. [36] and Canetti et al. [10], with the main difference being that we additionally support summation queries over sets.

For our purposes, a data structure for set operations considers data as a collection $S_1, \ldots, S_\ell \subset \mathbb{Z}_p$ of sets over the universe[1] $\mathbb{Z}_p$. The class $\mathcal{Q}$ of supported queries consists of nestings (see below) of the following basic queries (we discuss updates in Section 4.5):

Union: Given identifiers of sets $S'_1, \ldots, S'_k$, it returns $\bigcup_i S'_i$.

Intersection: Given identifiers of sets $S'_1, \ldots, S'_k$, it returns $\bigcap_i S'_i$.

Sum: Given an identifier of set S', it returns $\sum_{x \in S'} x \bmod p$.[2]

The identifier of one of the original sets outsourced by the data owner is simply its index. (Thus, for example, a user might make a query $\mathsf{Union}(I)$ with $I \subseteq [\ell]$; this would return $\bigcup_{i \in I} S_i$.) Sets computed as the intermediate result of various queries also have (implicit) identifiers, allowing *nested* queries. E.g., a user could query $\mathsf{Sum}(\mathsf{Intersection}(\mathsf{Union}(I), \mathsf{Union}(J)))$ with $I, J \subseteq [\ell]$; this would return $\sum_{x \in S'} x \bmod p$ where

$$S' \stackrel{\text{def}}{=} \left(\bigcup_{i \in I} S_i\right) \bigcap \left(\bigcup_{i \in J} S_j\right).$$

We construct an ADS supporting the above queries using the *bilinear accumulator primitive* introduced in [28]. Fix groups $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$ of order p, and a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. For simplicity we assume $\mathbb{G}_1 = \mathbb{G}_2 = \mathbb{G}$ but this is not essential. Let g

[1]Formally, the universe is $\{0, \ldots, B\}$, with p chosen by Setup such that $p \gg B$.

[2]Note this gives the true result $\mathsf{sum} = \sum_{x \in S'} x$ if $\mathsf{sum} < p$ and so there is no overflow.

be a generator of $\mathbb{G}$. The Init algorithm begins by choosing uniform the *set trapdoor* $s \in \mathbb{Z}_p$ and letting $sk = s$ be the secret key and $pk = (g^s, \ldots, g^{s^q})$ be the public key, where q is an upper-bound on the cardinality of any set. For a set $S = \{x_1, \ldots, x_n\} \subset \mathbb{Z}_p$, define the *accumulation value* of S as:

$$\mathsf{acc}(S) \stackrel{\text{def}}{=} g^{\prod_{i=1}^{n}(x_i^{-1}+s)}.$$

The digest δ output by Setup for a particular collection of sets $S_1, \ldots, S_\ell$ is simply $\mathsf{acc}(S_1), \ldots, \mathsf{acc}(S_\ell)$.

Union and intersection queries can be handled as in [36, 10]. (We use the inverse of set elements rather than the elements themselves, but this is handled in a straightforward manner.) We now show how to release the *sum* sum of the elements in a set $S = \{x_1, \ldots, x_n\}$ in a way that can be verified by any client in possession of $\mathsf{acc}(S)$. To do this, the server computes

$$\prod_{i=1}^{n}(x_i^{-1} + s) = s^n + a_{n-1}s^{n-1} + \cdots + a_1 s + a_0$$

as a formal polynomial in the variable s. It then releases $a_1 = \sum_{i=1}^{n}(\prod_{j\neq i} x_j^{-1})$ and $a_0 = \prod_{i=1}^{n} x_i^{-1}$, along with

$$w_1 = g^{s^{n-1}+\cdots+a_2 s+a_1} \text{ and } w_2 = g^{s^{n-2}+\cdots+a_3 s+a_2}.$$

(Note that the server can compute w_1, w_2 using pk.) To verify, the client checks that $\mathsf{sum} = a_1 a_0^{-1} \bmod p$ and that:

1. $e(g^s, w_1) \stackrel{?}{=} e(\mathsf{acc}(S)/g^{a_0}, g)$ and
2. $e(g^s, w_2) \stackrel{?}{=} e(w_1/g^{a_1}, g)$.

Security of the above relies on the q-SBDH assumption [8] in bilinear groups. We sketch a proof. Given an adversary $\mathcal{A}$ violating correctness, we construct an algorithm $\mathcal{A}'$ breaking the q-SBDH assumption [8]. $\mathcal{A}'$ takes $(g^s, \ldots, g^{s^q})$ as input and, given a set S (of size $n \leq q$) output by $\mathcal{A}$, computes the accumulator value $\mathsf{acc}(S)$. Say $\mathcal{A}$ next outputs $\mathsf{sum}^*, a_0^*, a_1^*, w_0^*, w_1^*$ such that verification succeeds yet $\mathsf{sum}^* \neq \sum_{x\in S} x$. Let

$$\prod_{i=1}^{n}(x_i^{-1} + s) = s^n + a_{n-1}s^{n-1} + \cdots + a_1 s + a_0.$$

Note that we must have $a_0^* \neq a_0$ or $a_1^* \neq a_1$. If $a_0^* \neq a_0$ then, since $e(g^s, w_1^*) = e(\mathsf{acc}(S)/g^{a_0^*}, g)$, we must have $g^{1/s} = (w_1^*/(g^{s^{n-1}+\cdots+a_2 s+a_1}))^{(a_0-a_0^*)^{-1}}$. As the right-hand side of this equation can be computed by $\mathcal{A}'$, this violates the q-SBDH assumption. If $a_0^* = a_0$ and $a_1^* \neq a_1$, since $e(g^s, w_1^*) = e(\mathsf{acc}(S)/g^{a_0^*}, g)$ we have $w_1^* = g^{s^{n-1}+\cdots+a_2 s+a_1}$. Since $e(g^s, w_2) = e(w_1/g^{a_1^*}, g)$, we must have that $g^{1/s} = (w_2^*/(g^{s^{n-2}+\cdots+a_2}))^{(a_1-a_1^*)^{-1}}$. Since the right-hand side of this equation can be computed by $\mathcal{A}'$, this violates the q-SBDH assumption.

Security of our $\mathcal{ASO}$ also relies on an extractability assumption [5], which is inherited from [10].

Support for multisets. Our $\mathcal{ASO}$ can be naturally generalized to support operations on *multisets*, in which a given element can have multiplicity greater than 1. The intersection (resp., union) of multisets S, S' yields a multiset in which each element appears with multiplicity equal to its minimum multiplicity (resp., the sum of the multiplicities) in S and S'. The sum of a multiset is defined in the natural way, taking multiplicity into account.

Complexity. The complexity of handling union and intersection is as in [36, 10], and in particular proofs are constant size. A sum query also has constant-size proofs. In a query involving d set operations (e.g., $d = 3$ in the query $S_1 \cap (S_2 \cup S_3)$) with result R, the proof size is $O(d)$ and the verification time is $O(d + |R|)$.

3.2 ADS for Interval Trees

Fix a function f (possibly randomized). For our purposes, an *interval tree* is a binary tree $\mathcal{T}$ associated with a set $S = \{(k, v)\}$ of key/value pairs, where the keys lie in a totally ordered set. Each leaf node of $\mathcal{T}$ stores one element of S, with the leaves sorted by key. Each internal node u also stores a key/value pair (k, v), where v is computed by applying f to the values stored in the left and right children of u. The key k at internal node u is equal to the maximum key stored at any node in the left subtree of u, and is strictly less than the key stored at any node in the right subtree of u.[3]

For a node u, let T_u denote the set of leaf nodes in the subtree rooted at u. For a set N of leaf nodes, we say a set of nodes U is a *covering set of* N if $\cup_{u\in U} T_u = N$. For our application, we need support for two types of queries:

Search: Given a key k, this returns the value stored at the leaf node (i.e., in the original set S) with key k.[4]

RangeCover: Given k_L, k_R with $k_L \leq k_R$, let N be the set of leaf nodes whose key k satisfies $k_L \leq k \leq k_R$. This query returns the key/value pairs stored at all the nodes in the minimal covering set of N.

We also need to support two types of updates to S:

Insert: Given a key/value pair, this inserts a leaf node containing that key/value pair into the interval tree, and updates internal nodes of the tree accordingly.

Delete: Given a key, this deletes all leaf nodes storing that key, and updates internal nodes of the tree accordingly.

An ADS supporting the above can be constructed generically using a Merkle tree, as in [24]. This gives a construction in which each leaf node is augmented with a hash computed over its stored key/value pair, and each internal node is augmented with a hash computed over the key/value/hash tuple stored by its children. The key, value, and hash at the root is the digest of the tree. Insertions/deletions can be performed using rotations as in standard red-black tree algorithms [12], with hash values being updated when rotations are performed.

Setup algorithm. We note here AITSetup algorithm is parameterized by function f used to compute the values of the internal nodes and is written as $(\tilde{D}, \delta) \leftarrow \mathsf{AITSetup}_f(D, sk_D)$—see Figure 2.

Complexity. Let n be the number of leaves. The size of the minimal covering set output by **RangeCover** is $O(\log n)$. The size of the proof and the complexity of verification for **Search**, **RangeCover**, **Insert**, and **Delete** is $O(\log n)$.

4. OUR CONSTRUCTION

We now describe the ADS used in INTEGRIDB.

[3]Duplicate keys are accumulated at leaves, i.e., if there are multiple key/value pairs $(k, v_1), (k, v_2), \ldots$ with the same key k, then a single leaf is created with key k and with value computed by applying f recursively to all the values stored at that leaf. A counter is also stored in the leaf to indicate the number of duplicates.

[4]The counter is also returned in the case of duplicates.

4.1 Setup

The high-level idea is that for each table, and each pair of columns in that table, we create an authenticated interval tree. (Although this requires storage quadratic in the number of columns, this is fine for practical purposes since the number of columns is typically much smaller than the number of rows.) The key/value pairs stored at the leaves of the tree corresponding to columns i, j are the entries in those columns that lie in each of the rows (see below). The value stored at an internal node u will (essentially) correspond to the accumulation (cf. Section 3.1) of the values stored at the leaves in the subtree rooted at u. Details follow.

For a table T, let x_{ij} denote the element in row i, column j of that table. Let

$$S_{i\times j} \stackrel{\text{def}}{=} \{(x_{1i}, x_{1j}), \ldots, (x_{ni}, x_{nj})\},$$

and view this as a set of key/value pairs. Construct an interval tree for $S_{i\times j}$. Leaf nodes hold key/value pairs from $S_{i\times j}$. Each internal node u stores a key equal to the minimum key stored at the leaves of the left subtree rooted at u, and stores the value

$$f_{s,sk}(u) = \mathsf{Enc}_{sk}\left(\prod_{v\in T_u}(v^{-1}+s)\right) \| \, g^{\prod_{v\in T_u}(v^{-1}+s)}, \quad (1)$$

where T_u now denotes the values stored at the leaves in the subtree rooted at u, and Enc is a CPA-secure encryption scheme.

c_1	c_2	c_3	c_4
$x_{11}=24$	x_{12}	x_{13}	x_{14}
$x_{21}=59$	x_{22}	x_{23}	x_{24}
$x_{31}=47$	x_{32}	x_{33}	x_{34}
$x_{41}=11$	x_{42}	x_{43}	x_{44}
$x_{51}=13$	x_{52}	x_{53}	x_{54}
$x_{61}=36$	x_{62}	x_{63}	x_{64}
$x_{71}=19$	x_{72}	x_{73}	x_{74}
$x_{81}=27$	x_{82}	x_{83}	x_{84}

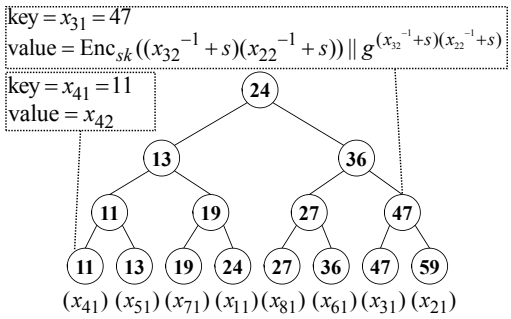


Figure 1: A database table (top) and its interval tree for columns 1 and 2 (bottom). Leaves are sorted based on the key x_{i1}.

Note that the right-most component in (1) is simply $\mathsf{acc}(T_u)$, and the left-most value is an encryption of the corresponding exponent. Roughly, the right-most component is used by the server to process all queries on a static database; the left-most component is used by the server and data owner jointly when the data owner wishes to update the database. In that case, we will rely on the fact that, given the secret key sk, the value of an internal node can be computed as a (randomized) function f of the values of its two children.

Figure 1 shows an example of an interval tree computed for columns 1 and 2 of the depicted table.

For each table T in the outsourced database, and each pair of columns i, j (where we may have $i = j$), Setup computes an $\mathcal{AIT}$ for $S_{i\times j}$ as described above. (Note that $\mathcal{AIT}$ does not use any Init algorithm.) In addition, Setup computes a hash H of each row in each table.[5] Figure 2 gives pseudocode for Init and Setup of the overall ADS. (Note that although the $\mathsf{ASOSetup}$ algorithm does not appear explicitly, it is called by $\mathsf{AITSetup}$ when computing the function $f_{s,sk}$ from Equation 1.) We assume the attributes of D are public, or else are included as part of the digest.

Algorithm $(sk_D, pk_D) \leftarrow \mathsf{Init}(1^\lambda)$

- $(s, pk_{\mathcal{ASO}}) \leftarrow \mathsf{ASOInit}(1^\lambda)$, where s is as in Section 3.1.
- Choose secret key sk for encryption scheme Enc.
- Output $sk_D = (s, sk)$ and $pk_D = pk_{\mathcal{ASO}}$.

Algorithm $(\tilde{D}, \delta) \leftarrow \mathsf{Setup}(D, sk_D)$

- For each table T in D, with n rows:
  for every pair of columns i, j:
  (a) Let $S^T_{i\times j} = \{(x_{1i}, x_{1j}), \ldots, (x_{ni}, x_{nj})\}$.
  (b) $(\delta^T_{i\times j}, \mathsf{ait}^T_{i\times j}) \leftarrow \mathsf{AITSetup}_{f_{s,sk}}(S^T_{i\times j}, \perp)$, where function $f_{s,sk}$ is defined in Equation 1.
- For each table T in D, with n rows and m columns:
  for every row i:
  set $h^T_i = H(x_{i1}, \ldots, x_{im})$.
- The authenticated database $\tilde{D}$ includes all the $\mathsf{ait}^T_{i\times j}$. The digest δ includes all the digests $\delta^T_{i\times j}$ and all the hashes h^T_i.

Figure 2: The Init and Setup algorithms of INTEGRIDB.

Intuitively, the accumulation value stored in each node will be used to handle `JOIN` and multidimensional range queries using the underlying $\mathcal{ASO}$ scheme, as shown in Sections 4.2 and 4.3. Section 4.4 shows how SQL functions can also be supported. The encryption of the exponent in each node helps to perform updates efficiently, as will be described in Section 4.5.

Digest size and setup complexity. Let m_i and n_i be the number of columns and rows in table i. The size of the secret key is $O(1)$. The size of the digest δ is $O(\sum m_i^2 + \sum n_i)$, but this can be reduced to $O(1)$ by outsourcing the digest itself using a Merkle tree (as we do in our implementation). The setup complexity is $O(\sum_i m_i^2 n_i)$.

4.2 Join Queries

We first describe how to handle a `JOIN` query involving two columns, neither of which contains any duplicate values; we discuss how to deal with duplicates below. (Generalizing to more than two columns is straightforward.) Consider a `JOIN` query involving column i of table T and column j of table T'. Let C_i and C'_j be the set of values contained in each of the respective columns. Observe that a **RangeCover** query on $S^T_{i\times i}$ using $k_L = -\infty$ and $k_R = \infty$ will return, in particular, the value stored at the root of the $\mathcal{AIT}$ associated with $S^T_{i\times i}$; this value contains $\mathsf{acc}(C_i)$. An analogous query on $S^{T'}_{j\times j}$ yields $\mathsf{acc}(C'_j)$. Making an intersection query using the $\mathcal{ASO}$ scheme results in the set $C^* = C_i \cap C'_j$ of all values in

[5] A Merkle tree over the elements of each row could also be used here, but in practice (assuming the number of columns is small) doing so will not have much effect.

common. For each such value $x \in C^*$, the server returns the entire row in T (resp., T') containing x in the appropriate column; the client verifies the result by checking that x is in the correct position of the claimed row k, and then hashing the row and ensuring that it matches the corresponding hash value h_k^T in the digest.

A point to stress here is that the client does not need C_i, C'_j in order to verify the intersection C^*; instead, it only needs $\mathsf{acc}(C_i)$, $\mathsf{acc}(C'_j)$. This ensures that the entire proof sent back by the server is proportional to the size of the final result, which is important when C^* is small even though C_i, C'_j are large.

Handling duplicates. The above approach can be modified easily to deal with the (possible) presence of duplicates in one or both of the columns. We view the columns as multisets when computing their intersection. Then, for each unique $x \in C^*$ we use a **Search** query on the $\mathcal{AIT}$ associated with $S^T_{i\times i}$ (resp., $S^{T'}_{j\times j}$); the result indicates exactly how many entries in column i of T (resp., column j of table T') have value x. The client then simply needs to verify that the server returns the correct number of rows from each table.

Pseudocode for Prove and Verify algorithms for handling JOIN queries, taking duplicates into account, is given in Figure 3. (To make them easier to read, the algorithms are presented somewhat informally and we omit small implementation optimizations. The algorithms are presented assuming a JOIN query involving two columns/tables, but can be extended for the general case.) We use **RC** in place of **RangeCover**, and let R_x denote the set of rows in T containing x in column i. (Define R'_x similarly.) Algorithms in Figure 3 refer to proving and verifying R_x and R'_x for $x \in C^*$; the result of the JOIN query can be easily computed from $\{R_x, R'_x\}_{x\in C^*}$ and this is not much larger than the result itself.

Assume a JOIN query on columns $T.i$ and $T'.j$.

Algorithm Prove

- $(\mathsf{acc}(C_i), \pi_i) := \mathsf{AITProve}(S^T_{i\times i}, \mathsf{ait}^T_{i\times i}, (\mathbf{RC}, -\infty, \infty))$.
- $(\mathsf{acc}(C'_j), \pi'_j) := \mathsf{AITProve}(S^{T'}_{j\times j}, \mathsf{ait}^{T'}_{j\times j}, (\mathbf{RC}, -\infty, \infty))$.
- $(C^*, \pi_\cap) := \mathsf{ASOProve}(C_i, C_j, \mathsf{acc}(C_i), \mathsf{acc}(C'_j), \text{“}\cap\text{”})$.
- For each unique element $x \in C^*$:
  - $(|R_x|, \pi_x) = \mathsf{AITProve}(S^T_{i\times i}, \mathsf{ait}^T_{i\times i}, (\mathbf{Search}, x))$.
  - $(|R'_x|, \pi'_x) := \mathsf{AITProve}(S^{T'}_{j\times j}, \mathsf{ait}^{T'}_{j\times j}, (\mathbf{Search}, x))$.
- $R := \{R_x, R'_x\}_{x\in C^*}$.
- $\pi := (C^*, \mathsf{acc}(C_i), \mathsf{acc}(C'_j), \pi_i, \pi'_j, \pi_\cap, \{\pi_x, \pi'_x\}_{x\in C^*})$.

Algorithm Verify

- Verify $\pi_i, \pi'_j, \pi_\cap$ and $\{\pi_x, \pi'_x\}_{x\in C^*}$.
- For all $x \in C^*$ and $r \in R_x$ (resp., $r' \in R'_x$), verify that:
  - r (resp., r') contains x in $T.i$ of T (resp., in $T'.j$ of T').
  - $H(r)$ (resp., $H(r')$) is equal to a (distinct) δ^T_k (resp., $\delta^{T'}_k$).

Figure 3: Handling JOIN queries on two tables.

Security. We briefly sketch why this is secure. The security of $\mathcal{AIT}$ ensures that $\mathsf{acc}(C_i)$ and $\mathsf{acc}(C'_j)$ reflect the correct (multi)sets C_i, C'_j. Security of $\mathcal{ASO}$ then guarantees that $C^* = C_i \cap C'_j$. Security of $\mathcal{AIT}$ then ensures that $|R_x|$ (resp., $|R'_x|$) is a correct count of the number of rows in T (resp., T') that contain x in column i (resp., column j). The client then verifiers that each row in R_x, R'_x contains x in the appropriate column, and is unaltered.

Complexity. The proof size and verification time are $\tilde{O}(|R|)$, where R is the final result. This is independent (up to logarithmic factors) of the sizes of any intermediate results or the original tables.

4.3 Multidimensional Range Queries

Fix a table T, and let x_{ij} refer to the value in row i and column j of T. Consider a two-dimensional range query on columns w and z of T, with bounds $[w^-, w^+]$ and $[z^-, z^+]$, which should return every row i in which $w^- \le x_{iw} \le w^+$ and $z^- \le x_{iz} \le z^+$. (Extending our approach to larger dimensions is straightforward. We also mention below how to handle OR instead of AND.)

We require a (known) column in T in which every element is guaranteed to be distinct; we refer to this as the *reference column*. Such a column is often present, anyway (e.g., serving as a row key); if not, then such a column can be added before Setup is run. We assume for simplicity in what follows that column 1 is the reference column, and moreover that it simply contains the row number.

Let R_w denote the indices of the rows in which the element in column w in within the specified bounds; i.e., $i \in R_w$ if and only if $w^- \le x_{iw} \le w^+$. Define R_z similarly. In our protocol, the client will (verifiably) learn $R^* = R_w \cap R_z$, from which verifiably obtaining the rows themselves is trivial (using $\{h^T_i\}_{i\in R^*}$).

The key observation is that a **RangeCover** query on $S^T_{w\times 1}$, using bounds w^-, w^+, will return a set of nodes $N_w = \{n_1, \ldots\}$ that constitute the minimal covering set for the leaves in $S^T_{w\times 1}$ containing keys (i.e., elements in column w) in the specified range. Each node $n \in N_w$ contains $\mathsf{acc}(C_n)$ such that

$$\bigcup_{n\in N_w} C_n = R_w. \tag{2}$$

Similarly, a **RangeCover** query on $S^T_{z\times 1}$, using bounds z^-, z^+, will return a set of nodes N_z; each $n \in N_z$ contains $\mathsf{acc}(C_n)$ with

$$\bigcup_{n\in N_z} C_n = R_z. \tag{3}$$

We can therefore express the desired answer as

$$R^* = \left(\bigcup_{n\in N_w} C_n\right) \bigcap \left(\bigcup_{n\in N_z} C_n\right);$$

correctness of this answer can then be verified using $\mathcal{ASO}$, given $\{\mathsf{acc}(C_n)\}_{n\in N_w}$ and $\{\mathsf{acc}(C_n)\}_{n\in N_z}$.

Note that by using union instead of intersection we can handle disjunctive queries in addition to conjunctive queries.

Security. The security of $\mathcal{AIT}$ ensures that the client obtains values $\{\mathsf{acc}(C_n)\}_{n\in N_w}$ and $\{\mathsf{acc}(C_n)\}_{n\in N_z}$ for sets $\{C_n\}_{n\in N_w}$ and $\{C_n\}_{n\in N_z}$ such that Equations (2) and (3) hold. Security of $\mathcal{ASO}$ then implies that the claimed value of R^* is correct. Finally, the row hashes guarantee that the returned rows have not been altered.

Complexity. Consider a multidimensional range query of dimension d. The proof size is $O(d \cdot \log n)$ and the verification complexity is $O(d \log n + |R|)$. We stress that the proof size is independent of the sizes of any intermediate results (e.g., the sets R_w, R_z from above), or the size of the original table (up to logarithmic factors).

Although we introduce a logarithmic overhead in proof size and verification time as compared to [32], our scheme supports efficient updates (as we describe later).

4.4 SQL Functions

Here we describe how we can support various SQL functions on entire columns of the original tables in the database. In some sense, this is not very interesting since all the answers could be precomputed for each column at the time the original database is outsourced. These become more interesting, however, when we consider nested queries in Section 4.6.

Summation. As $\mathsf{acc}(C_j)$ is stored in the root of $S^T_{j\times j}$ (where C_j denotes the set of elements in the jth column of T), the client can obtain and verify this value for any desired column j. Since $\mathcal{ASO}$ supports sum queries, the client can then verify a claimed sum over that column. The proof size and verification time are both $O(1)$.

Count and average. We can reduce a `COUNT` query (on a column guaranteed to contain no duplicates) to two `SUM` queries as follows: for each column j of the original table, we include an additional column j' in which each entry is one more than[6] the corresponding entry in column j. (I.e., $x_{ij'} = x_{ij} + 1$ for all i.) Note that if column j contains no duplicates, then neither does column j'. To perform a `COUNT` query on column j, the client issues `SUM` queries on columns j and j' and then takes the difference.

An `AVG` query can be answered by simply dividing the result of a `SUM` query by the result of a `COUNT` query.

Maximum and minimum. We can reduce `MAX`/`MIN` queries to a single-dimensional range query. For example, to answer a query "`SELECT MAX(`j`) FROM` T," the server first computes j_{max}, the maximum value in column j. It then returns a (verifiable) answer to the query "`SELECT * FROM` T `WHERE` $j \geq j_{\mathsf{max}}$." The client verifies the answer and, if multiple rows are returned, also checks that the values in column j are all the same. (Note there may be multiple rows containing j_{max} in column j.)

4.5 Updates

To insert/delete a row, the data owner and server jointly update the corresponding $\mathcal{AIT}$ and $\mathcal{ASO}$ ADSs using their respective interfaces; we provide further details below. In addition, the data owner adds/removes a hash of the inserted/deleted row.

Recall that the value stored at an internal node is an accumulation value along with an encryption of the relevant exponent. The data owner uses the encrypted values to update accumulation values during an insertion or deletion. For example, in Figure 4 there is a rotation (to maintain a balanced tree) after an update in the interval tree. Before the rotation we have g^{f_A}, g^{f_B}, g^{f_C}, $g^{f_B f_C}$, and $g^{f_A f_B f_C}$ (and encryptions of the respective exponents) stored in nodes A, B, C, Y, and X, respectively. To compute $g^{f_A f_B}$ after the rotation, the data owner decrypts to recover f_A, f_B, computes $g^{f_A f_B}$, and sends updated values (including an encryption of the new exponent) to the server. This can be done using $O(1)$ rounds of communication between the data owner and the server.

Complexity. The complexity of updating one $\mathcal{AIT}$ is $O(\log n)$, where n is the number of rows in the table being updated. Since each table corresponds to m^2 interval trees, the overall complexity of an update is $O(m^2 \log n)$. The complexity of updates in [32] is $O(m\sqrt{n})$, which is slower in practice because $m \ll n$. (Typical values might be $m < 20$ and $n \approx 10^6$.)

SQL supports more general updates involving all rows satisfying a given constraint. To perform this type of update, the data owner can first make a query with the same constraints to learn which rows need to be updated, and then make the relevant updates row-by-row. The complexity of doing so is proportional to the number of updated rows.

[6]Non-numeric values are anyway mapped to $\mathbb{Z}_p$ during setup, so can be treated as numeric.

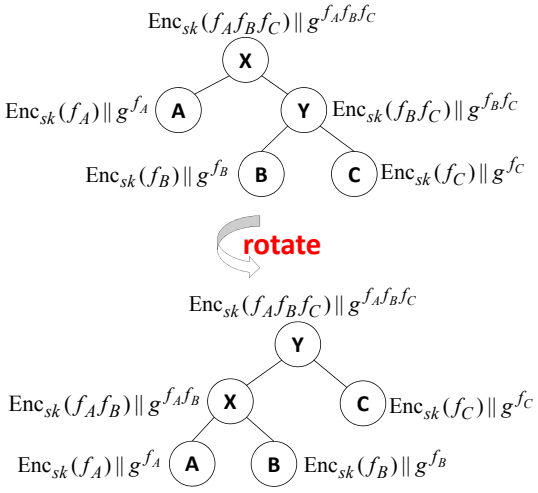


Figure 4: A rotation caused by an update (insertion or a deletion).

4.6 Nested Queries

The result of any SQL query is itself a table, meaning that queries can be nested. A natural way to attempt to answer a nested query of the form $Q_1 \circ Q_2$ verifiably would be to return the result R' of the inner query Q_2 along with a proof of correctness, and then to return the final result R along with an another proof of correctness. (This assumes some structure in the underlying ADS, and not all ADSs will support such nested proofs.) Note, however, that in general the size of the intermediate result R' may be much larger than the size of the final result R, meaning that the total proof will no longer have size proportional to the size of the final result.

Informally, we can avoid this in our case by having the server return a *digest* of the intermediate result rather than the intermediate result itself. Details follow.

Let `FUNC` $= \{$`SUM`, `COUNT`, `AVG`, `MAX`, `MIN`$\}$. IntegriDB supports the following nested queries:

- `JOIN` on the result of a multidimensional range query; (I.e., one of the tables participating in the `JOIN` query is the result of multidimensional range query.)
- `FUNC` on the result of
  - a `JOIN` query;
  - a multidimensional range query;
  - a `JOIN` on the result of a multidimensional range query.

Our system can also handle a range query on the result of a `JOIN` query, since this can be reduced to a `JOIN` on range query. (This is also done in regular SQL databases to improve performance.)

`JOIN` on range. By way of example, consider a `JOIN` query involving column i of a table T_1 in the original database and column j of a table T_2' that is the result of a range query on table T_2 in the original database. We require that there are guaranteed to be no duplicates in column j of table T_2. Under this assumption, we can use column j as the reference column when answering the range query. Instead of returning the result R^* (using the notation of Section 4.3), the server only returns $\mathsf{acc}(R^*)$. This suffices for computing a `JOIN` query over this intermediate result.

We remark that to support this nested query, any column could potentially be chosen as the reference column. This is the reason we need to build an $\mathcal{AIT}$ for each pair of columns during setup.

SUM, COUNT, and AVG on intermediate results. As described in Section 4.4, we only rely on the accumulation value of a column in order to perform a SUM query on that column. Therefore, SUM queries over intermediate results can be handled as long as the client can obtain a verified accumulation value of the desired column of the intermediate result. This holds automatically following a JOIN query (with no duplicates), and the method described above (i.e., letting the desired column serve as the reference column) can be used to ensure that this holds following a range query, whether followed by a JOIN query or not.

We cannot support a SUM query following a JOIN query with duplicates while still having short proofs. This is because to obtain the accumulation value of the column to sum (cf. Section 4.2), the client needs to make a search query for each unique value in the intersection, and the complexity of doing so is proportional to the size of the intermediate result (rather than the final result).

As already described in Section 4.4, COUNT and AVG can be reduced to SUM.

Note that to perform a SUM query on the result of a range query, the column to sum must be used as the reference column. Thus, to support such queries we need to build an $\mathcal{AIT}$ for each pair of columns during setup. However, this can be improved to linear in the number of columns as we show in Section 5.

MAX and MIN on intermediate results. Recall from Section 4.4 that we reduce MAX/MIN queries to a range query. So if MAX/MIN is applied to the result of a range query (whether also followed by a JOIN query or not), we simply incorporate the additional constraint in the range query (possibly increasing the dimension by one). The case of MAX/MIN applied to a JOIN query can be reduced to a JOIN query on a single-dimensional range query.

5. ADDITIONAL DISCUSSION

We now discuss various extensions, optimizations, and limitations of INTEGRIDB.

5.1 Extensions and Optimizations

Improving setup complexity. In case we are interested in supporting only SUM queries over multidimensional range queries, we can improve the setup time from $\tilde{O}(m^2n)$ to $\tilde{O}(mn)$, and the server storage from $\tilde{O}(m^2)$ to $\tilde{O}(m)$. To do this, we construct a *homomorphic* Merkle tree [35] over each row, and add a column c to each table that stores the root of the Merkle tree for the corresponding row. Informally, a homomorphic Merkle tree has the property that the sum of the roots of multiple trees is the root of a new Merkle tree with the leaves being the component-wise sum of the leaves of the original trees.

To answer a SUM query on a desired column i, the server first verifiably answers a SUM on column c. This sum is the root of a Merkle tree in which the sum of the desired column is a leaf. So the server can additionally send the desired sum along with a Merkle proof, and the client can verify correctness.

Using this modification, only column c is ever used as a reference column for handling SUM queries, and so we can reduce complexity as claimed.

Supporting LIKE queries. We can add support for LIKE queries by building on the authenticated pattern matching scheme of Papadopoulos et al. [33]. Their scheme is based on an authenticated suffix tree, and it is not hard to generalize it to return the number and locations of all matches. If we view a column as a string, where each element is delimited by a special character, then we can build an authenticated suffix tree on top of the entire column and support LIKE queries over that column. A limitation of this technique is that a suffix tree does not support efficient updates. Supporting LIKE queries in a dynamic database is an open problem.

Supporting GROUP BY queries. A GROUP BY query is used to combine rows having duplicates. E.g., "SELECT SUM(c_1) GROUP BY c_2" returns one summation for each set of rows with the same value in c_2. We can support this query by first retrieving all unique values in c_2, which is supported by $\mathcal{AIT}$. Then for each unique value x, issue a SUM query with constraint $c_2 = x$, which is supported by our nested query scheme. The number of SUM queries is the same as the number of unique values in c_2, which is proportional to the size of the result.

5.2 Limitations

Comparison between columns. We are unable to support comparisons between columns (e.g., a query involving the constraint $c_1 \geq 2 \cdot c_2 + 3$) with short proofs. However, we can handle this inefficiently by returning the entire column c_1 and then having the client issue range queries to c_2 for each element in c_1. The complexity is proportional to the size of column c_1, which is not efficient; however, this solution has the same complexity as existing tree-based schemes.

Aggregations among columns. We cannot support aggregations among columns (e.g., a query "SELECT c_1+2*c_2 FROM...") with short proofs. Instead, we can have the server return all columns involved in the aggregation query and then let the client compute the result locally.

Join with duplicates in a nested query. As mentioned in Section 4.2, we can support JOIN queries with duplicates. However, when performing a JOIN query on the result of a range query, as described in Section 4.6, we assume no duplicates are present. Removing this assumption is left open.

6. IMPLEMENTATION

A schematic of our implementation is shown in Figure 5. Recall there are three parties in our model: a data owner, a client (which may be the data owner itself), and a server. The data owner runs the setup algorithm and sends the digest δ to the client, and the database itself along with authentication information $\tilde{D}$ to the server. In our implementation, we logically separate the client into an SQL client and an INTEGRIDB client. The SQL client issues a standard SQL query; the INTEGRIDB client determines whether this query is supported by INTEGRIDB and, if so, translates it into an INTEGRIDB query and sends it to the server. We also logically separate the server into two parts: an INTEGRIDB server and a back-end SQL server. The INTEGRIDB server is responsible for handling all cryptographic computations (thus providing verifiability) using the corresponding algorithms described in Section 4. During this process, the INTEGRIDB server may make one or more SQL queries to the SQL server. When done, the INTEGRIDB server sends back the final result along with a proof for the INTEGRIDB client to verify. The INTEGRIDB client returns the result to the SQL client if the proof is correct, and rejects the result otherwise.

Notice that the INTEGRIDB server can be combined with any existing implementation of an SQL server; only the SQL API methods in the INTEGRIDB server need to be changed. The client, data owner, and most of the INTEGRIDB server remain unchanged.

In our implementation, the client, data owner, and INTEGRIDB server are implemented in C++ using approximately 1,000 lines of

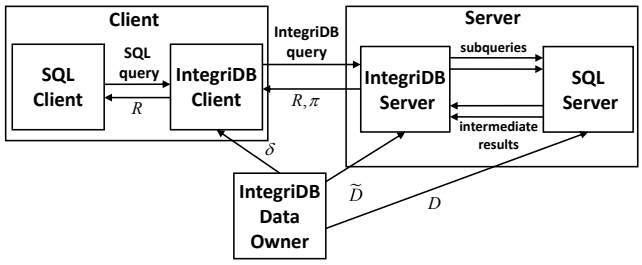


Figure 5: Structure of INTEGRIDB implementation.

code for the INTEGRIDB client, 1,000 lines of code for the data owner, and 2,000 lines of code for the INTEGRIDB server. We use a standard MySQL server as the back-end SQL server. We use the OpenSSL library for encryption and hashing. In particular, we use AES-CBC-128 for encryption[7] and SHA-256 for hashing. For our bilinear pairing we use the Ate-paring[8] on a 254-bit elliptic curve; this is estimated to offer 128-bit security.

We implement the authenticated interval tree in our construction using the authenticated skip list of Goodrich et al. [17]. This provides expected logarithmic complexity (rather than worst-case logarithmic complexity), yet its practical performance is better.

Information about INTEGRIDB and links to its code can be found at `integridb.github.io`.

7. EVALUATION

We executed our experiments on an Amazon EC2 machine with 16GB of RAM running on Linux. We collected 10 runs for each data point and report the average.

7.1 Evaluation Using the TPC Benchmark

TPC-H benchmark. We first evaluate the expressiveness of INTEGRIDB using the TPC-H benchmark.[9] The TPC-H benchmark contains 22 SQL queries and a dataset and is widely used by the database community to compare the performance of new systems. The performance presented in this section is on query #19 of TPC-H, which is shown in Figure 6. This query is a `SUM` query on the result of a `JOIN` applied to two multidimensional range queries on tables lineitem and part.

To support query #19, we encode characters in ASCII (so we can view them as numeric values) and answer the query as follows: inside each segment separated by "`OR`," we have constraints on a relevant table. E.g., in lines 5–11 of Figure 6, lines 6, 7, and 9 are constraints on Table part. In particular, line 6 is an equality check, which is a special case of a single-dimensional range query. Line 7 is parsed into four equality checks: p_container = 'SM CASE' or p_container = 'SM BOX' or p_container = 'SM PACK' or p_container = 'SM PKG.' Therefore, lines 6, 7, and 9 together constitute a multidimensional range query on Table part. Similarly, lines 8, 10, and 11 form a multidimensional range query on Table lineitem. Line 5 is a `JOIN` query on columns p_partkey and l_partkey. We answer these queries and then perform a union on the results obtained. All these queries (also taking nesting into account) are supported by INTEGRIDB.

As we cannot support aggregation, we let the server return the resulting table generated by the constraints in lines 4–27 consisting of columns l_extendedprice and l_discount. The client then computes the aggregation. The size of this table is included in the reported proof size.

```
1. SELECT SUM(l_extendedprice* (1 - l_discount))
2. AS revenue
3. FROM lineitem, part
4. WHERE
5. (    p_partkey = l_partkey
6.    AND p_brand = 'Brand#41'
7.    AND p_container IN ('SM CASE', 'SM BOX', 'SM
          PACK', 'SM PKG')
8.    AND l_quantity >= 7 AND l_quantity <= 7 + 10
9.    AND p_size BETWEEN 1 AND 5
10.   AND l_shipmode IN ('AIR', 'AIR REG')
11.   AND l_shipinstruct = 'DELIVER IN PERSON' )
12. OR
13. (  p_partkey = l_partkey
14.   AND p_brand = 'Brand#14'
15.   AND p_container IN ('MED BAG', 'MED BOX',
          'MED PKG', 'MED PACK')
16.   AND l_quantity >= 14 AND l_quantity <= 14 + 10
17.   AND p_size BETWEEN 1 AND 10
18.   AND l_shipmode IN ('AIR', 'AIR REG')
19.   AND l_shipinstruct = 'DELIVER IN PERSON' )
20. OR
21. (  p_partkey = l_partkey
22.   AND p_brand = 'Brand#23'
23.   AND p_container IN ('LG CASE', 'LG BOX', 'LG
          PACK', 'LG PKG')
24.   AND l_quantity >= 25 AND l_quantity <= 25 + 10
25.   AND p_size BETWEEN 1 AND 15
26.   AND l_shipmode IN ('AIR', 'AIR REG')
27.   AND l_shipinstruct = 'DELIVER IN PERSON' );
```

Figure 6: Query #19 of the TPC-H benchmark.

We executed the query on the TPC-H database. Table lineitem consists of 6 million rows and 16 columns, and Table part contains 200,000 rows and 9 columns. Due to memory limitations, we only preprocess necessary columns as part of setup, i.e., four columns of each table. The performance of INTEGRIDB for handling this query is summarized in Table 2. We also simulated insertion of a row into the lineitem table (reduced to four columns, as just mentioned). We observe that the proof size, verification time, and update time are all very small, although the times for setup and proof computation are large.

Table 2: Performance of TPC-H query #19 on the TPC-H database.

setup time	prover time	verification time	proof size	update time
25272.76s	6422.13s	232ms	184.16KB	150ms

We examined all 22 TPC-H queries, and found that INTEGRIDB supports 12 of them.

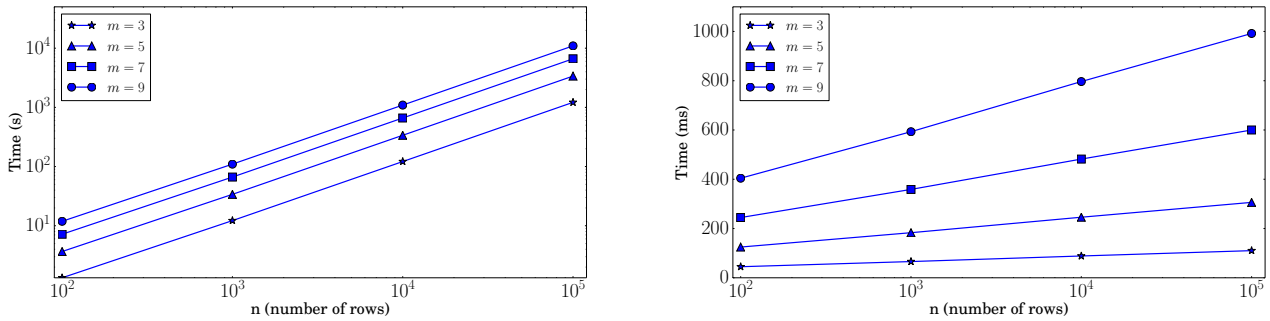
TPC-C benchmark. We also tested our system using the TPC-C benchmark,[10] and found that INTEGRIDB supports 94% of those queries. The only query INTEGRIDB could not support is a `JOIN` query with duplicates following two multidimensional range queries.

[7] Note that authenticated encryption is not needed because ciphertexts are authenticated as part of the overall ADS.

[8] See `https://github.com/herumi/ate-pairing`.

[9] Available at `http://www.tpc.org/tpch`.

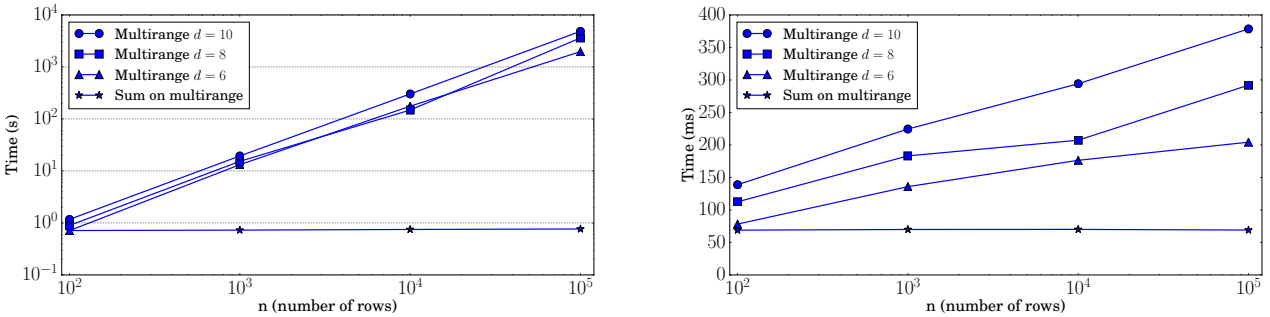
[10] Available at `http://www.tpc.org/tpcc`.



(a) Setup time

(b) Update time

Figure 7: Setup and update time. The database consists of two tables, each having n rows and m columns. Figure (a) is log-log scale; only the x axis in (b) is log scale.



(a) Prover time

(b) Verification time

Figure 8: Prover and verification time for different queries. We run range queries of different dimensions d on a 10-column table and the size of the result is fixed to 100 entries. We also run `SUM` queries applied to the result of a 3-dimensional range query.

INTEGRIDB can support this query if there is no duplicate in the columns to join.

7.2 Performance on Synthetic Data

We now present detailed performance measurements of INTEGRIDB for different queries. All queries were run on synthetic tables. The first column of each table contains a row number. The elements in other columns are randomly generated 32-bit integers.

Server storage. For a table with n rows and m columns, we store m^2 $\mathcal{AIT}$s containing $2n$ nodes on average; the size of each node in our implementation is 99.5 Bytes. Therefore, for a table with 10^5 rows and 10 columns, the server storage is 1.85GB, while the size of the table itself is 30MB (assuming entries are 30-byte integers). However, we believe our overhead is acceptable compared to prior work, e.g., the storage of the signature-based scheme in [27] for the same table is approximately 0.24GB, yet it only supports single-dimensional range queries.

Setup time. Figure 7a shows the setup time of INTEGRIDB on databases of different sizes. As shown in the figure, the setup time grows linearly with the total number of rows in all the tables in the database. In particular, setup takes about 4s for two tables with 100 rows and 5 columns each, and 3,000s for two tables with 100,000 rows and 5 columns each. The setup time also grows quadratically with the number of columns in a table; in practice (as observed with the TPC-H dataset), however, the number of columns tends to be small and the setup time is acceptable in this case. We remark further that setup is only done once, after which arbitrarily many queries can be supported.

Update time. As shown in Figure 7b, updates in INTEGRIDB are very efficient, e.g., it takes only 0.9s to insert a row into a table with 100,000 rows and 9 columns. As indicated in the figure, the update time grows logarithmically with the number of rows in the table.

Join queries. We run a `JOIN` query on two tables and report the performance in Table 3. Both tables are of the same size, with the number of columns set to 10 and the number of rows n varying from 100 to 100,000. There are duplicates in the columns to join. We choose the query such that result always has precisely 100 rows. As shown in Table 3, although the prover time grows linearly with the sizes of the tables to join, the verification time remains unchanged. (Although the asymptotic complexity of verification grows logarithmically with n, in our experiments the running time is dominated by the costs of verifying the set intersection, which has complexity independent of n.) In particular, it only takes 45ms to verify a join query on two tables with 100,000 rows each.

As shown in Table 3, proofs are only a few KBs, and their size grows logarithmically with n as expected. For example, the proof is only 27.97KB for a `JOIN` query on two tables of 100,000 rows by 10 columns, which is 4MB each if elements are 4 byte integers. This is a big improvement compared to prior tree-based and signature-based approaches, where the proof size could be even larger than the original database.

Table 3: A JOIN query on two tables, each with the same number of rows and 10 columns. The result always has 100 rows.

number of rows	prover time	proof size	verification time
100	0.041s	11.97KB	40.7ms
1,000	1.38s	16.77KB	45.2ms
10,000	15.7s	23.17KB	45.3ms
100,000	168s	27.97KB	45.4ms

Multidimensional range queries. Figure 8 shows the prover time and the verification time for range queries of different dimensions on tables of different sizes, where the size of the result is also fixed to 100 matches. As shown in Figure 8a, the prover time grows linearly with the number of rows in the table and the number of dimensions. It takes around 5000s to generate the proof of a 10-dimensional range query on a table of 100,000 rows.

Figure 8b shows that the verification time grows logarithmically with the number of rows in the table and linearly with the number of dimensions in the query. It takes less than 200ms to verify the result of a 6-dimensional range query on a table of 100,000 rows, and around 400ms to verify a 10-dimensional range query on the same table. The corresponding proof sizes are 135KB and 251KB, both significantly smaller than the size of the table.

Our verification times for range queries are worse than those in [32], as we introduce a logarithmic overhead. However, our verification times are still good in practice, while we improve on the time for updates by orders of magnitude.

(Nested) sum queries. We run a sum query on the result of a 3-dimensional range query. There is no duplicate in the column to sum. We fix the size of the table to 100,000 rows and 10 columns, but vary the size of the result of the range query (i.e., the size of the column to sum) from 10 to 100,000. As shown in Figure 8a, the prover time grows slightly with the size of the intermediate result. This is because the prover time is dominated by the multi-dimensional range query, which does not change as the size of the table is the same. In particular, it only takes 760ms to generate the proof for the sum on a column with 100,000 elements. Moreover, Figure 8b shows that the verification time does not change with the size of the intermediate result, and it only takes around 60ms to verify the sum on a column with 100,000 elements and the proof size is only 45KB. In prior work, the entire intermediate result would be sent to the client, who would then compute the sum by itself.

7.3 Comparison with Generic Schemes

We compare INTEGRIDB to two generic systems for verifiable computation (VC): `libsnark`[11], an efficient circuit-based VC system used in [4], and SNARKs for C [2], an efficient RAM-based VC system. For libsnark, we wrote SQL queries in C, compiled them to a circuit, and ran the VC system on the resulting circuits. Since there is no publicly available code for SNARKs for C, we estimate the performance on a query by first expressing the query in TinyRAM and then using this to determine the three parameters that affect performance: the number of instructions (L), the number of cycles (T), and the size of the input and output (N). We then used those parameters along with [4, Fig.$\tilde{9}$] to estimate the running time. Note that libsnark (and circuit-based VC generally) does not support updates; we were unable to estimate the update time for SNARKs for C.

[11]See `https://github.com/scipr-lab/libsnark`.

For generic VC systems, the database would be hardcoded in a program that takes SQL queries as input and outputs the result. To support different types of SQL queries, there must be a compiler built in the program to interpret the query. Circuit-based VC cannot support such a compiler efficiently, and it is unclear how to implement it using RAM-based VC. Therefore, in our experiments we use a dedicated program for each type of query. This means the performance of generic VC systems reported here is likely better than what they would achieve in practice.

As a representative example, we consider a sum query applied to the result of a 10-dimensional range query, executed on a table with 10 columns and 1,000 rows. As shown in Table 4, the setup time and prover time of INTEGRIDB are $10\times$ faster than that of libsnark and $100\times$ faster than that of SNARKs for C. The proof sizes of the generic VC systems are always of constant size, while the proof size of INTEGRIDB in this case is $200\times$ larger. However, proofs in INTEGRIDB are still only a few KB, which is acceptable in practice. The verification time of INTEGRIDB is $10\times$ slower that in the generic systems; however, even the result of a multidimensional range query can be verified in under 1s in INTEGRIDB.

Table 4: Comparing with generic schemes. Table T has 10 columns and 1,000 rows. Query is "SELECT SUM(c_1) FROM T WHERE (c_1 BETWEEN a_1 AND b_1) AND $\cdots$ AND (c_{10} BETWEEN a_{10} AND b_{10})."

	[4, `libsnark`]	SNARKs for C [2]	INTEGRIDB
setup time	157.163s	2000s*	13.878s
prover time	328.830s	1000s*	10.4201s
verification time	7ms	10ms*	112ms
proof size	288Bytes	288Bytes	84,296Bytes
update time	N/A	??	0.7s

Acknowledgments

This research was sponsored in part by NSF award #1514261 and by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-06-3-0001. The views and conclusions in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence, or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

8. REFERENCES

[1] S. Bajaj and R. Sion. CorrectDB: SQL engine with practical query authentication. *Proceedings of the VLDB Endowment*, 6(7):529–540, 2013.

[2] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Crypto*, pages 90–108, 2013.

[3] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Scalable zero knowledge via cycles of elliptic curves. In *Crypto*, pages 276–294. Springer, 2014.

[4] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a Von Neumann architecture. In *USENIX Security*, 2014.

[5] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. From extractable collision resistance to succinct non-interactive

arguments of knowledge, and back again. In *ITCS*, pages 326–349, 2012.

[6] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. Recursive composition and bootstrapping for SNARKs and proof-carrying data. In *STOC*, pages 111–120, 2013.

[7] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth. Succinct non-interactive arguments via linear interactive proofs. In *TCC*, pages 315–333, 2013.

[8] D. Boneh and X. Boyen. Short signatures without random oracles and the SDH assumption in bilinear groups. *Journal of Cryptology*, 21(2):149–177, 2008.

[9] B. Braun, A. J. Feldman, Z. Ren, S. T. V. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *SOSP*, pages 341–357, 2013.

[10] R. Canetti, O. Paneth, D. Papadopoulos, and N. Triandopoulos. Verifiable set operations over outsourced databases. In *PKC*, pages 113–130, 2014.

[11] W. Cheng, H. Pang, and K.-L. Tan. Authenticating multi-dimensional query results in data publishing. In *DAS*, pages 60–73, 2006.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 3rd edition)*. MIT Press, 2009.

[13] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur. Geppetto: Versatile verifiable computation. ePrint 2014.

[14] P. Devanbu, M. Gertz, C. Martel, and S. Stubblebine. Authentic data publication over the internet. *J. Computer Security*, 11(3):291–314, 2003.

[15] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Crypto*, pages 465–482, 2010.

[16] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Eurocrypt*, pages 626–645, 2013.

[17] M. T. Goodrich, C. Papamanthou, and R. Tamassia. On the cost of persistence and authentication in skip lists. In *Experimental Algorithms*, pages 94–107, 2007.

[18] L. Hu, W.-S. Ku, S. Bakiras, and C. Shahabi. Verifying spatial queries using Voronoi neighbors. In *SIGSPATIAL GIS*, pages 350–359, 2010.

[19] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *SIGMOD*, pages 121–132, 2006.

[20] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Authenticated index structures for aggregation queries. *ACM TISSEC*, 13(4):32, 2010.

[21] X. Lin, J. Xu, and H. Hu. Authentication of location-based skyline queries. In *CIKM*, pages 1583–1588, 2011.

[22] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.

[23] R. C. Merkle. A certified digital signature. In *Crypto*, pages 218–238, 1990.

[24] A. Miller, M. Hicks, J. Katz, and E. Shi. Authenticated data structures, generically. In *POPL*, pages 411–424, 2014.

[25] E. Mykletun, M. Narasimha, and G. Tsudik. Signature bouquets: Immutability for aggregated/condensed signatures. In *ESORICS*, pages 160–176, 2004.

[26] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. *ACM Trans. on Storage*, 2(2):107–138, 2006.

[27] M. Narasimha and G. Tsudik. Dsac: integrity for outsourced databases with signature aggregation and chaining. In *CIKM*, pages 235–236, 2005.

[28] L. Nguyen. Accumulators from bilinear pairings and applications. In *CT-RSA*, pages 275–292. Springer, 2005.

[29] H. Pang, A. Jain, K. Ramamritham, and K.-L. Tan. Verifying completeness of relational query results in data publishing. In *SIGMOD*, pages 407–418, 2005.

[30] H. Pang and K.-L. Tan. Authenticating query results in edge computing. In *ICDE*, pages 560–571, 2004.

[31] H. Pang, J. Zhang, and K. Mouratidis. Scalable verification for outsourced dynamic databases. *Proceedings of the VLDB Endowment*, 2(1):802–813, 2009.

[32] D. Papadopoulos, S. Papadopoulos, and N. Triandopoulos. Taking authenticated range queries to arbitrary dimensions. In *CCS*, pages 819–830, 2014.

[33] D. Papadopoulos, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Practical authenticated pattern matching with optimal proof size. *Proceedings of the VLDB Endowment*, 8(7):750–761, 2015.

[34] S. Papadopoulos, D. Papadias, W. Cheng, and K.-L. Tan. Separating authentication from query execution in outsourced databases. In *ICDE*, pages 1148–1151, 2009.

[35] C. Papamanthou, E. Shi, R. Tamassia, and K. Yi. Streaming authenticated data structures. In *Eurocrypt*, pages 353–370, 2013.

[36] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Optimal verification of operations on dynamic sets. In *Crypto*, pages 91–110, 2011.

[37] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE S & P*, pages 238–252, 2013.

[38] S. T. V. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. In *EuroSys*, pages 71–84, 2013.

[39] S. T. V. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making argument systems for outsourced computation practical (sometimes). In *NDSS*, 2012.

[40] S. Singh and S. Prabhakar. Ensuring correctness over untrusted private database. In *EDBT*, 2008.

[41] R. Tamassia. Authenticated data structures. In *ESA*, pages 2–5, 2003.

[42] V. Vu, S. T. V. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for interactive verifiable computation. In *IEEE S & P*, pages 223–237, 2013.

[43] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis. Authenticated join processing in outsourced databases. In *SIGMOD*, pages 5–18, 2009.

[44] Y. Yang, S. Papadopoulos, D. Papadias, and G. Kollios. Authenticated indexing for outsourced spatial databases. *The VLDB Journal*, 18(3):631–648, 2009.

[45] Z. Yang, S. Gao, J. Xu, and B. Choi. Authentication of range query results in MapReduce environments. In *CloudDB*, 2011.