Database Outsourcing with Hierarchical Authenticated Data Structures

Mohammad Etemad   Alptekin Küpçü
Crypto Group, Koç University, Istanbul, Turkey

Abstract

In an outsourced database scheme, the data owner delegates the data management tasks to a remote service provider. At a later time, the remote service is supposed to answer any query on the database. The essential requirements are ensuring the data integrity and authenticity with efficient mechanisms. Current approaches employ authenticated data structures to store security information, generated by the client and used by the server, to compute proofs that show the answers to the queries are authentic. The existing solutions have shortcomings with multi-clause queries and duplicate values in a column.

We propose a hierarchical authenticated data structure for storing security information, which alleviates the mentioned problems. Our solution handles many different types of queries, including *multi-clause selection* and *join* queries, in a *dynamic* database. We provide a unified formal definition of a secure outsourced database scheme, and prove that our proposed scheme is secure according to this definition, which captures previously separate properties such as correctness, completeness, and freshness. The performance evaluation based on our prototype implementation confirms the efficiency of our proposed scheme, showing about 3x to 5x enhancement in proof size and proof generation time in comparison to previous work, and about only 4% communication overhead compared to the actual query result in a real university database.

Keywords: Databases outsourcing, Hierarchical authenticated data structures, Authenticated join processing.

1   Introduction

Huge amount of data is being produced everyday due to the widespread use of computer systems in organizations and companies. Data needs protection, and most of companies lack enough resources to provide it. By outsourcing data storage and management, they free themselves from data protection difficulties, and concentrate on their own proficiency.

Consider a university who stores all data about students, faculty, and courses in a relational database, with limited resources and equipment for hosting a large amount of data and handling a large volume of queries, especially at the beginning and end of each semester. The university wishes to outsource data management to a remote database service provider who offers mechanisms to access and update the database online.

An important problem is that by data outsourcing, the owner loses the direct control over her data and should rely on answers coming from the remote service provider (who is *not* fully trusted). Therefore, there should exist mechanisms giving the data owner (the client) the ability for checking the integrity of the outsourced data. To make sure that the remote server operates correctly, the client should verify the answers coming from the server in response to her queries [15]. The remote server sends to the client a *verification object* (*vo*) along with the answer to the query (the *result set*). The *vo* gives the client the ability to verify that the server's answer is authentic. Since the client may be a portable device with limited processing power, the *vo* should be small, and efficiently verifiable. The client uses the *vo* to verify that the query answer is [42, 18, 41, 15, 29]:

- *complete*: the result set sent to the client is exactly the set of records that are the output of executing the query, i.e., no record is added or removed.
- *correct*: the result set sent to the client is provided by the client already, i.e., no unauthorized modification.
- *fresh*: the result set sent to the client is provided using the most recent data on the server, and does not belong to old versions, i.e., no replay attacks.

Assume that the university database is outsourced, and the client wants to execute the query: `SELECT * FROM Student WHERE stdID>105`. A small part of the database together with the result of this query is shown in Figure 1. We want the completeness, correctness, and freshness properties hold in the returned answer, guaranteeing that the answer is genuine.

Student

StdID	StdName	Major	BCity
101	Ali	CE	Istanbul
102	Emir	CE	Istanbul
103	Hande	CS	Istanbul
104	Ates	EE	Istanbul
105	John	CS	Ankara
106	Tommy	CE	Ankara
107	Katty	EE	Tebriz
108	Matt	EE	Tebriz

S2C

StdID	CrsID	Mark
101	501	A
101	502	B
101	504	C
103	503	B
103	504	C
106	500	B
106	502	A
106	504	B
108	501	C
108	503	A

Course

CrsID	CrsName	Credit	CrsType
500	Soft. Eng.	3	A
501	Prog. Lang.	3	A
502	DB Design	3	A
503	Alg. Design	3	E
504	DB Lab.	1	X
505	OS Lab.	1	X

StdID	StdName	Major	BCity
106	Tommy	CE	Ankara
107	Katty	EE	Tebriz
108	Matt	EE	Tebriz

(a) Our sample database.

(b) The result set of the query `SELECT * FROM Student WHERE StdID > 105`.

Figure 1: Our sample database (a), and the result of a query on it (b).

The *authenticated range query* is a way of providing completeness in the outsourced database context. Multiple implementations have been proposed by researchers using different data structures [33, 6, 25, 29, 42, 30, 18, 32, 28, 15]. In all these methods, the records are linked together in a way that we can prove there is no extra or missing record in between. It requires two records surrounding the (sorted) result set: one immediately before the first record (the *left boundary record*) and one immediately after the last record (the *right boundary record*). They together are referred to as the *boundary* records. We call such data structures with the ability to prove the predecessor (left boundary) and successor (right boundary) *ordered*.

We know that the primary key (PK) column in a table, as the stdID in the Student table in our example, contains unique values, while non-PK columns may contain duplicate values, as the major and stdName columns in our example. We want to perform authentic queries on all *searchable* columns (the columns that can be used to build clauses) of a table. The general method is to sort a table by each searchable column, and build an authenticated data structure (ADS) on the result, that will be used to generate cryptographic proofs for queries having a clause using the column. There is a problem with duplicate values in non-PK searchable columns [32, 19]: a total order on the values of searchable columns is required to build the ADS, which together with the fact that the duplicate values belong to different records, make building the ADSs complicated. As clarified later, the existing solutions are not *efficient*.

We introduce a *hierarchical ADS* scheme (HADS) for solving this problem. HADS is also advantageous in proof generation for multi-clause (multi-dimensional) queries. The HADS can be stored in the same database [4], or separately. Storing the HADS separately breaks the tie to a specific database and brings more flexibility. This way, the DBMS used for data storage can be changed without affecting the proof system.

The rationale behind this work is to relate everything to the PKs. Since the PKs are unique identifiers of records in a database, they enable us to compare and combine the results of different queries and check the correctness and completeness at the same time (freshness is provided by storing a constant-size metadata locally at the client). This is an important distinction between our HADS and similar (multi-level) ADSs, as their proofs cannot be combined and compared together. We also support dynamic databases where the data owner issues modification queries (Insert, Delete, Update), in a provable manner. We believe that our HADS may also be of independent interest, applicable to other scenarios.

Our contributions can be summarized as follows:

- We provide a *unified security definition* for an outsourced database scheme (ODB) that captures *completeness*, *correctness*, and *freshness* simultaneously.
- We formalize the *hierarchical ADS* scheme and prove its security, for the first time.
- We build a provably-secure ODB using HADS that supports efficient proof generation for not only single-clause but also *multi-clause* queries.
- We handle proofs on columns containing duplicate values with around 3x to 5x better efficiency, regarding both proof generation time and size, compared to previous work.
- Our scheme supports the tables with *composite keys*, for the first time.
- Our ODB construction efficiently handles proofs for *join* queries, even for multi-table joins, non-equijoins, and queries containing both join and selection.
- Our ODB provides efficient proofs for almost all query types. We achieve only 4% communication overhead compared to the actual result size, using our Koç University database.

1.1 Related Work

Inefficient approaches. An elementary way to verify the authenticity of an answer to an outsourced database query is to sign each table and store the signature locally. This method requires sending the whole table to the client for verification, and hence, does not scale up. Another method is to compute and store, with each record, a signature that verifies the contents of the record. The problems are that computing a signature (for each record) is an expensive operation, and this method does not provide completeness.

ADS-based approaches. A more suitable approach towards answer verification is to use ADSs [6, 21, 33, 9, 41, 42] to store authentication information, and send the relevant parts of these ADSs to the client to prove authenticity of the answer.

Devanbu *et al.* [6] proposed one of the first schemes using ADS for checking integrity of the remote data. They used a Merkle hash tree to store the security information about an outsourced static data (which changes infrequently). The scheme supports the projection and simple join operations inefficiently.

Pang and Tan [33] used one or more *verifiable* B-trees (VB-tree) for each table. The VB-tree is an extension of B-tree using the Merkle hash tree. A VB-tree is generated (using the table sorted on that column) for each searchable column of the table. This method does not support completeness [32], and found insecure for the insecurity of the function used to compute the signatures [28].

A variant of this method, named MB-tree, is also used in the literature [6, 25, 29, 42]. MB-tree is similar to VB-tree except that a light hash function is used instead of expensive signatures. The client stores locally the root's digest, or signs and stores it on the server.

Another line of work is using an **authenticated skip list** to store the required information for the verification [30, 41]. It is suitable and efficient enough for this purpose, especially when we consider dynamic scenarios. Wang and Du [41] proved that such ADSs provide soundness and completeness for one-dimensional range queries, and multiple ADSs are required for multi-dimensional range queries.

Palazzi [30, 31] built one authenticated skip list for each searchable column in each table. For a query with one clause, a proof is computed using the corresponding skip list and sent back to the client along with the result set. For multi-clause queries, the result set of one clause that is finished earlier is considered and separated into a '*YesSet*' and a '*NoSet*' by applying the other clauses on top. The result sent to the client is a larger set than the real result set of the query, and hence, is not efficient. The problem is that each proof authenticates a set, and these sets cannot be compared against each other.

Authenticated range query is an important method used to prove the completeness (i.e., no extra records and no missing ones), which works as follows [6, 18, 42, 28]:

- Find the *contiguous* nodes storing the values corresponding to the result set of the query, as well as the *left boundary record* and the *right boundary record*. Note that, to be able to work with such proofs, the underlying ADS needs to be *ordered*.
- Compute the ADS membership proofs of the boundary records.
- Put all these values into the verification object and send it to the client.
- The client uses the values in the result set together with the membership proofs to reconstruct the corresponding part of the ADS, and computes the digest.
- She compares the computed digest with the locally stored metadata. If they are the same, then the query result is accepted, and rejected otherwise.

If the proof is accepted, the set {*left boundary record, result set, right boundary record*} is guaranteed by the *ordered* ADS to be a sorted and contiguous set of values, with no extra or missing value between them [41, 21].

Hierarchical ADSs. Due to their widespread use, work has been done to improve the efficiency of authenticated range queries. Nuckolls [29] proposed a flexible structure called Hybrid Authentication Tree, which uses the one-way accumulators in upper levels to break the dependence on tree height of the MB-tree.

Goodrich *et al.* [15] gave a super-efficient answer verification method by decoupling the authentication structure from the search data structure. They divided a tree with n leaves (and height $\log n$) into sub-trees with $\log n$ leaves (and height $O(\log \log n)$), and stored their roots in another structure. The sub-trees are divided further into sub-trees with $O(\log \log n)$ leaves. This process is repeated recursively up to an optimal level. Note that none of the previous work formalizes or generalizes such hierarchical ADSs.

Hash chaining is another method for providing authentic query results where the records are linked together to show that there is no extra or missing records between them. The client sorts the table by a searchable columns (and repeats this process for all searchable columns), and link all two (or three in some approaches) consecutive records

$[r_{i-1},]$ r_i, r_{i+1} together, i.e., compute $h([h(r_{i-1})|]h(r_i)|h(r_{i+1}))$, where h is a collision resistant hash function, and '|' denotes the concatenation. The first and last records are linked to special records indicating the beginning and end of the records. [18, 32, 28, 15]. To provide authentic proofs, the client either computes signatures for the links, or relates them to each other in a tree structure.

Upon receipt of a query, the server (1) executes the query, (2) finds the result set and the boundary records, (3) computes the proof as either the set of links' signatures, or the corresponding part of the tree, and (4) sends them all to the client. Using signatures, the verification object contains a linear (in the size of the result set) number of signatures, and hence, computation and communication costs are high. The *aggregated hash chaining* [26, 28] tries to reduce the proof size by combining multiple proofs into one, using the aggregation capability of the underlying scheme. The main problems with such schemes are the cost of updates and the lack of join possibility.

Provable join. Devanbu *et al.* [7] pre-computed all possible joins and constructed the corresponding ADS to enable proof generation by the server. They further suggested constructing the ADS on the *differences* between the values of the matching columns in the result set. This way, they can support queries with equi-join (difference = 0), >-clause (difference > 0), and <-clause (difference < 0).

Li *et al.* [18] proposed the Embedded Merkle B-tree (EMB-tree) whose nodes consist of regular B^+-tree entries augmented with an embedded MB-tree, and used it to support authentic join queries. To join two tables R and S, $R \bowtie_{C_i=C_j} S$, where $C_i \in R$ and $C_j \in S$, they: (1) find the smaller table, say R, (2) insert it as a whole into the *vo*, along with its proof, and (3) for each $v_k \in C_i$, construct a range query proof for the query 'SELECT * FROM S WHERE $C_j = v_k$', and append it to the *vo*. It requires $|C_i|$-many range queries, hence, is not efficient regarding the client and server computation, and communication.

Pang *et al.* [34] used signature aggregation to propose a scalable query result authentication mechanism for dynamic databases. Their first attempt is similar to the schemes of Li *et al.* [18], and results in a huge verification object. Their second attempt uses a certified Bloom filter [2] to show that some of records of the first table has no matching records on the second table.

Join algorithms that use the ADSs for both tables and generate reasonable proofs are proposed by Yang *et al.* [42]. The first algorithm, *Authenticated Indexed Sort-Merge* join, is an efficient form of previous join algorithms with one ADS [18, 34], and eliminates the repeated range queries and redundant proofs. The second algorithm, *Authenticated Indexed Merge* join, improves the previous algorithm using two ADSs, one for each table. It traverses each ADS once, and each required node is inserted only once into the *vo*. Although it is efficient regarding both computation and communication, for any (mis)match, two boundary records are inserted into the *vo*, which is unnecessary as we show in our join algorithms. The third algorithm, *Authenticated Sort-Merge* join, is used to perform the join on a column for which no ADS is generated. The server inserts the whole first table into the *vo*, together with the matching records of the second table, and the *rank lists* used to prove the matching. The client verifies all of them and generates matching pairs (the expected result) locally.

Recently, an integrity-checking mechanism is given for join queries performed by an *untrusted* computational server working together with some *trusted* storage servers [5]. The client gives the storage servers a query, an encryption key, and information on how to inject some fake records (*marker*s and *twin*s) into the result. The storage servers execute the query, inject the fake records, encrypt and send the result to the computational server who performs the join and sends the final result to the client.

Private query processing. Carbunar and Sion [3] suggested a private join on the outsourced databases that supports equi-join, and can be extended to support range join queries, assuming an honest-but-curious server. For each value in a column, the client finds all matching values in all other tables, encrypts them, and stores them all in a Bloom filter. For a join between two tables, the client computes a trapdoor and sends it to the server, so that the server can find all matching pairs of the requested tables, and send the corresponding records. The scheme is not computation- and storage-efficient, especially for range queries and dynamic data. Another privacy-preserving join scheme proposed by Ma *et al.* [20] only supports equi-join, but uses randomized trapdoors. It is not computation-efficient, since for a join between two columns A and B, each value $a_i \in A$ should be checked against all values $b_j \in B$. In both schemes, there are no (correctness, completeness, and freshness) proofs accompanying the server answers.

1.2 Overview of Our Solution

To be able to provide proof for different kinds of queries in a database, one ADS per searchable column in a given table is built. We also follow a similar approach, and build a hierarchical ADS (HADS) for each searchable column. Figure 4b visualizes the idea for a database. At the topmost ADS, the *database ADS*, the table names are stored. For

each table, we have a *table ADS*, which stores the names of the columns in that table. For each column, we have a *column ADS* that stores the unique values in that column. Finally, the bottommost ADSs are *primary key ADSs*, associated with each *unique* value v_i in a column C_j, storing the primary key (PK) values of the records having v_i in column C_j. For example, in our sample database in Figure 1, a column-level ADS for `major` will contain only three leaves, with labels `CE`, `CS`, `EE`. The lower-level ADS connected to the `CE` will contain the primary key values 101, 102, and 106. Similarly, the lower-level ADS connected to `CS` will contain 103 and 105. Note that, our HADS definition is flexible, and hence such a four-level hierarchy is not a requirement, but a sample deployment that makes sense.

Efficient duplicate handling. The reason for the necessity of such a hierarchical structure comes from the shortcomings of previous ADS-based solutions. Note that columns, such as `major`, contain duplicate values. Obviously, such duplicates can be made unique, for example, by appending a random perturbation [19], hash of the record [31], or the replica number [32]. Yet, the server should traverse the whole resulting (big) ADS to search for a value. Since the HADS stores the unique values in an upper level, which is a much smaller ADS, the server first finds a value in this ADS, and accesses the whole related values in the lower level, without further computation. As an example, consider a column containing 1000 unique values, each of which is repeated 100 times. A regular (single-level) ADS would need to integrate 100,000 values, whereas our HADS will have one upper-level ADS with 1000 values, and 1000 lower-level ADSs with 100 values each. Hence, instead of searching for 100 values in an ADS with 100,000 values, the server looks for **only one value** in an ADS with **only 1000 values** (and access the whole lower-level ADS storing 100 values). This results in great performance improvements regarding both communication and computation.

We use multi-proof supporting ADSs (e.g., the FlexList [11]) to construct the HADSs, which in turn, makes efficient authenticated range queries possible. A multi-proof supporting ADS generates an efficient (non-)membership proof for a set of values, instead of separate proofs for each value in the set. The proof for the clause $a < col_i < b$, indeed, consists of membership proofs of a and b, and the values matching the clause.
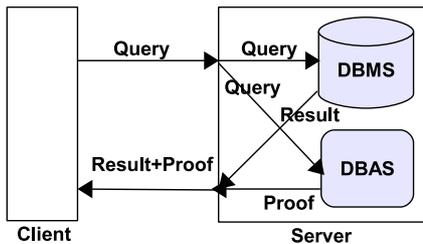
Figure 2: Server architecture.

Server architecture. There are two parts on the server side: the DBMS (database management system) who stores the client data and responds to the SQL queries coming from the client, and the DBAS (database authentication system) who stores the security information in the form of the ADSs and HADSs, and generates cryptographic proofs to the queries. The DBMS choice is independent of our work, and any available DBMS can be employed. But, we design and implement our own DBAS. Since our *DBAS works independently of the underlying DBMS* (on the same query), our proofs do not include any extra record, making it an efficient scheme. Figure 2 shows this architecture.

Join. Another advantage of the HADS is an *improved join algorithm*. Since we use similar ordered HADSs, the items contained in them are comparable, and hence proving mutual memberships (i.e., for 'AND' connector and join queries) is easy. To join two tables on two columns, we start at the leftmost leaf nodes of both ADSs and compare them together. If they store the same value, it is reflected in the proof. Otherwise, we jump over the nodes of the ADS containing the smaller value, to a node containing the smallest value that is less than or equal to the bigger value. This process goes on until the end of either ADS is met. The proof size and proof generation time is reduced due to the lack of duplicates.

Combining proofs. Another important advantage of our scheme is that since the HADS ties all values to their related PKs, all proofs prove to the client the authenticity of a set of PKs. This makes possible the results of the proofs to be compared and combined together, which was a common problem among most of the existing solutions [7, 32, 28, 31]. Stated differently, for queries with more than two clauses, the server starts by generating proof for the first two clauses on their ADSs, and uses the result (that is not in the form of an ADS) with the next clause, who has an ADS, to generate a new proof. This is repeated (with proper ordering on the clauses, detailed in Section 5.4) until all clauses are processed. **Thus, more than two clauses or joins on more than two tables can be handled as well.**

2 Preliminaries

Notation. We use N to denote the number of records of a table, and $|C_i|$ to denote the number of *distinct* values in a column. The symbol '|' denotes the concatenation, h denotes a collision-resistant hash function, and *PPT* stands for probabilistic polynomial time. 'PK' denotes 'primary key' in a database table, and 'pk' stands for 'public key'.

A function $v(k) : Z^+ \to [0, 1]$ is called *negligible* if $\forall$ *polynomials* p, $\exists$ *constant* k_0 s.t. $\forall k > k_0$, $v(k) < |1/p(k)|$. *Overwhelming* probability is greater than or equal to $1 - v(k)$ for some negligible function $v(k)$.

(a) ADS of `Student` table's PK column.   (b) A membership proof.
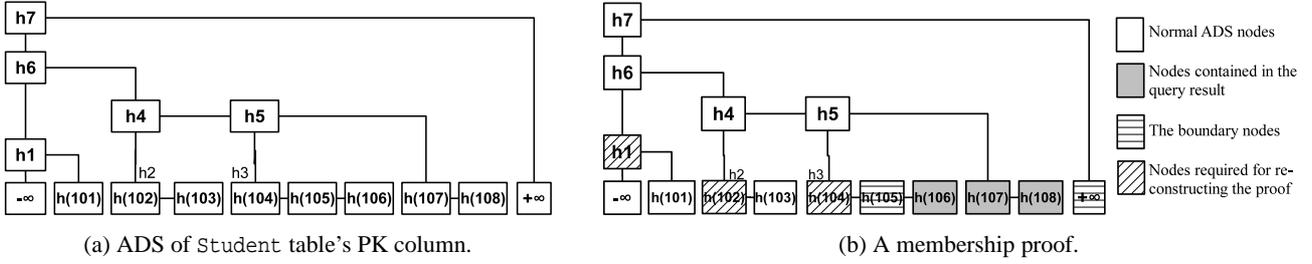
Figure 3: (a) An ADS storing the PK column of the `Student` table, and (b) the membership proof for the query `SELECT StdID FROM Student WHERE StdID > 105`.

Hash functions are functions that take arbitrary-length strings, and output strings of some fixed length. Let $h : \mathcal{K} * \mathcal{M} \rightarrow \mathcal{C}$ be a family of hash functions, whose members are identified by $k \in \mathcal{K}$. A hash function family is collision resistant if $\forall\, PPT$ adversaries $\mathcal{A}, \exists$ a negligible function $v(\ell)$ such that: $Pr[k \leftarrow \mathcal{K}; (x, x') \leftarrow \mathcal{A}(h, k) : (x' \neq x) \wedge (h_k(x) = h_k(x'))] \leq v(\ell)$, where ℓ is the security parameter of the hash function family (e.g., related to $|k|$).

An **authenticated data structure (ADS)** is a scheme for data authentication, where untrusted responders answer client queries and provide cryptographic proofs that the answers are valid [39, 40, 35, 16]. The client constructs the ADS and uploads it to a server who answers later queries. On receipt of membership queries, the server sends back a proof, using which the client can verify the answer against some local metadata. There are different types of ADSs: accumulators, authenticated skip lists, authenticated hash tables, Merkle hash trees, 2-3 trees. We provide a formal definition in Appendix A.

A **one-way accumulator** [1] is defined as a family of *one-way*, *quasi-commutative* hash functions. A function $f : X * Y \rightarrow X$ is *quasi-commutative* if $\forall x \in X, y_1, y_2 \in Y : f(f(x, y_1), y_2) = f(f(x, y_2), y_1)$. Benaloh and de Mare [1] proposed a one-way accumulator based on an RSA modulus.

The **authenticated skip list** is an extension of a skip list [37]. It is constructed using a commutative hash function h, which in turn can be constructed from a collision resistant hash function f as: $h(x, y) = f(min(x, y), max(x, y))$. The leaves store hashes of data items, and each intermediate node stores hash of a function of values of its children. The values on the path from a leaf node up to the root constitute a proof of membership. **Merkle hash tree** [23] is another widely used ADS for *static* data. Both ADSs have *linear* space complexity, and *logarithmic* proof size and verification time, in the number of the items stored [16]. Figure 3a presents an authenticated skip list storing the PK column of the `Student` table, and Figure 3b illustrates the membership proof for the query `SELECT StdID FROM Student WHERE StdID>105`.

Papamanthou *et al.* [36] introduced the **authenticated hash table**, which constitutes a hierarchy of one-way accumulators. It keeps either the query or update time constant while providing the other with sub-linear complexity.

An **ordered ADS** can be used to show some elements are consecutive (essential for authenticated range queries). A total order on the elements to be stored in an ordered ADS is required. Assume that x, y and z are *consecutive* elements of a *total order* $(A, <)$ such that $x < y < z$, and A is stored at ADS_A. Informally, we say ADS_A is *ordered* if it can prove that $x = predecessor(y)$ and $z = successor(y)$ for all consecutive $x, y, z \in A$. The Merkle hash tree and authenticated skip list are ordered ADSs, while the accumulator is not. An ordered ADS is perfectly suited for authenticated range queries.

A **multi-proof ADS** can prove (non-)membership of multiple elements in one proof. To prove (non-)membership of a set of elements, it does not need to do the job for each element one-by-one, and instead, generates a proof showing (non-)membership of all elements in only one traversal of the ADS. This will reduce the server computation, the communication, and the client verification, though not asymptotically. These ADSs suit the authenticated range queries well. FlexList [11] is an ADS with multi-proof capabilities.

3 Hierarchical Authenticated Data Structures

The Hierarchical ADS (HADS) is an ADS consisting of multiple levels of ADSs. Each ADS at level i is constructed on top of a number of ADSs at level $i + 1$. Each element of an ADS at level i stores the digest of, and a link to an ADS at level $i + 1$. Therefore, multiple ADSs with different underlying structures can be linked together to form a hierarchical ADS with multiple levels. The only restriction is that all ADSs at level i must be of the same underlying structure to have consistent proofs. (We can handle the heterogeneous case as well, but it complicates the presentation). At the bottommost level, the hash of the data is stored as well (e.g., the hash of records in the database). The client stores the digest of the topmost ADS as metadata. Figure 4a presents a two-level HADS instantiation (based on our sample
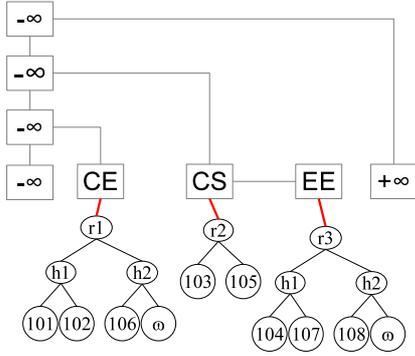
(a) A two-level HADS.
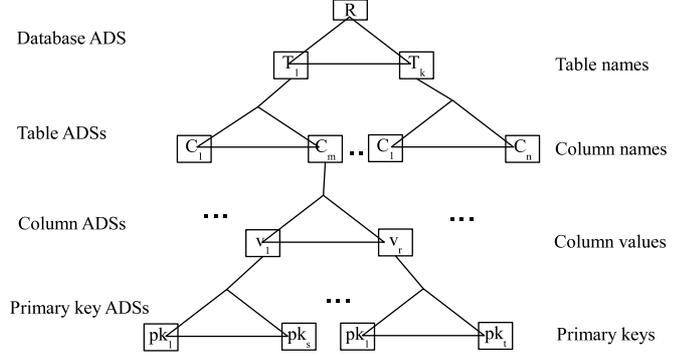
(b) A general four-level HADS to store a database.

Figure 4: HADS constructions with different levels to store security information for a database.

database in Figure 1) using authenticated skip list and Merkle hash tree at the first and second levels, respectively. Similarly, Figure 4b shows a general four-level HADS architecture to store a database (the ADSs are represented as tree for simplicity, but they can be of any type as long as they can store digest of the corresponding lower level ADSs).

An **HADS scheme** is an ADS scheme defined with three PPT algorithms (HKeyGen, HCertify, HVerify) to distinguish them from non-hierarchical ADSs. Definitions A.1, A.2, A.3, A.4 (using HADS algorithm names) provide a formal framework for HADS schemes.

3.1 HADS Construction

We construct an HADS using (possibly different) ADSs at multiple levels in a hierarchical structure. First, all lowest-level ADSs are constructed using the data (in the form of different groups). Then, these ADSs are divided into groups according to some relation, and their digests together with information about where they are stored and the data of the upper level, are used to build the upper-level ADSs. This process is followed until a single ADS is built whose root will be stored as metadata by the client.

To generate a membership proof, the client should provide the server with the required information directing the traversal on the HADS at all levels. In other words, the client tells the server which element(s) at each level should be looked for. The server follows down the HADS until the last level, generates and combines the proofs for all levels, and sends the resultant proof to the client. If ADSs with modification capabilities are used, a similar recursive strategy is employed for provable modification operations as well.

We provide the input as a set of $(key, value)$ pairs in a way that the pairs needed for the upper levels appear first. The command execution will begin on the topmost ADS, and be directed by the input data customized to proper sub-ADSs at each level. A query command uses the keys, while a modification requires both the keys and values.

3.2 HADS Operations

The HKeyGen algorithm generates a public and private key pairs for each level, combines all public keys into pk, and all private keys into sk, and outputs the result as the private and public key pair of the HADS (Algorithm 3.1). Again, even though conceptually one may employ different ADS structures or use the same structure with different keys within the same level, to keep the presentation simple, we present as all ADSs at level i being of the same type and with the same key pair.

Algorithm 3.1: HKeyGen, run by the client.

Input: the security parameter k, no. of levels n, and the underlying structure of each level.
Output: the private and public keys of the HADS

1 $sk_{HADS} = \{\}$ //private key of the HADS.
2 $pk_{HADS} = \{\}$ //public key of the HADS.
3 **for** $i = 1$ to n **do**
4      $(sk, pk) = ADS_i.\text{KeyGen}(1^k)$ //Ask level i ADS to produce its security keys.
5      $sk_{HADS} = sk_{HADS} \cup sk$
6      $pk_{HADS} = pk_{HADS} \cup pk$
7 **return** (sk_{HADS}, pk_{HADS})

The HCertify performs the proof generation and modification on HADS. The recursive operation starts at the topmost ADS, and is repeated on all affected ADSs in the hierarchy. The ADS in each level generates its own proof.

Since the ADSs are tied together such that each leaf node of an ADS at level i stores a link to an ADS at level $i+1$, their proofs will be combined together according to their order in the hierarchy, as presented in Algorithm 3.2. The HADS proof contains all required ADS proofs. To simplify this operation, we use another PPT algorithm as a helper method to find the sub-ADSs of a given ADS:

$\text{Find}(key, value) \rightarrow (\{(ADS', \{(key', value')\})\})$ This is used (inside HCertify) to interpret the input data and find the next level ADS(s) together with the related input value(s). It traverses the current ADS with the provided key(s) and finds the leaf node(s) storing address(es) of the ADS(s) at the next level to continue with. Finally, it outputs the set of next-level ADSs and their $(key', value')$ pairs. Examples are given in Section 4.3.1.

Algorithm 3.2: HCertify, run by the server.

Input: the public key pk, the command cmd, the data given as a *(key, value)* pairs.
Output: the generated proof

1   $P_{own} = \{\}$ // Proof of the current ADS.
2   $P_{child} = \{\}$ // Proof of all children combined together.
3   $\{(ADS', \{(key', value')\})\} = Find(key, value)$
     // Output is null if already at the bottommost level.
4   **for** *each element* $e \in \{(ADS', (key', value'))\}$ **do**
5       $P = e.ADS'.\text{HCertify}(pk, cmd, e.(key', value'))$ //Ask each child compute proof.
6       $P_{child} = P_{child}|P$ // Combine the proofs.
7   $P_{own} = \text{Certify}(pk, OP, (key, value))$ //Compute this ADS proof (not hierarchical).
8   **return** $P_{child}|P_{own}$

The HVerify is also a recursive process that is run by the client to verify each level's proof in a bottom-up manner. It first verifies the bottommost ADSs. If they are all accepted, then it uses their digests together with the proofs of the above-level ADSs to verify the level above, and so forth. Finally, when the upper-most level is reached and a single digest is obtained, which is verified against the local metadata.

4 Outsourced Database Scheme

4.1 Model

The outsourced database (ODB) model, as depicted in Figure 5, is composed of three parties: the *data owner*, the *querier*, and the *service provider*. The data owner performs the required pre-computations, uploads the database, and gives the querier(s) the security information she needs for verification. The data owner then may perform modifications (insertion, deletion, or update) on the outsourced database.
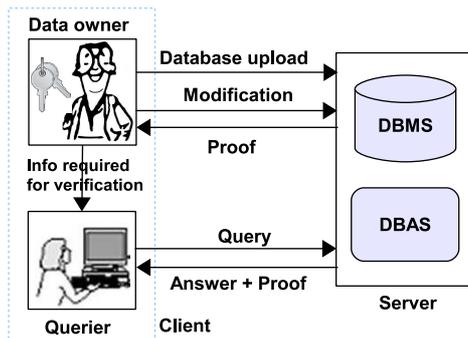
Figure 5: The ODB model.

The service provider (or simply, the *server*) has the required equipment (software, hardware, and network resources) for storing and maintaining the database in a provable manner. We do not know or care about the internal structure of the server, i.e., the server may use some levels of replication and distribution to increase the performance and availability. The *querier* (or the *user*) issues a query to the server, who executes the query, computes the result set, generates the proof, and sends all back to the querier. The querier then verifies the answer using the security information given by the data owner. For the sake of a simpler presentation, we refer to them as the *client*. It is possible to have multiple queriers or data owners, and data owners can also act as queriers. In this paper, we focus on the single-client case.

We decouple the real data from the authentication information on the service provider. The *DBMS* is a regular database management system responsible for storing and updating the data, and executing the queries on the data and giving the answer back. The *DBAS* (*database authentication system*) stores the authentication information about the data, and generates the proofs to be sent to the client. Thus, a DBAS can be used together with any DBMS, and the focus of this work is to construct an efficient and secure DBAS. The DBMS and DBAS together constitute an ODB.

Adversarial Model. The remote server is not fully trusted: he can either act maliciously, or be subverted by attackers to do so, or may suffer failures. He may cheat by attacking the integrity of the outsourced data (modifying the records) and giving fake responses to the client queries (executing the query processing algorithm incorrectly, or modifying the results), or by performing unauthorized modifications on data, while trying to be undetected.

4.2 Definitions

An outsourced database requires certification and verification algorithms, similar to an ADS. Thus, the following definitions follow the same ideas. A corollary to this is that an ADS scheme can be employed to construct an ODB system (or vice versa).

Definition 4.1 *An **outsourced database scheme** consists of three probabilistic polynomial-time algorithms* `(OKeyGen, OCertify, OVerify)` *where:*

- `OKeyGen`$(1^k) \to (sk, pk)$*: is a probabilistic algorithm run by the client to generate a pair of secret and public keys (sk, pk) given the security parameter k. She keeps both keys, and shares only the public key with the server.*
- `OCertify`$(pk, cmd) \to (ans, \pi)$*: is run by the server to respond to a command cmd issued by the client. It produces an answer ans and a proof π that proves authenticity of the answer. If the command is a modification command, the answer is empty, and the proof proves that the modification is done properly.*
- `OVerify`$(pk, sk, cmd, ans, \pi, st) \to (\{$`accept`, `reject`$\}, st')$*: is run by the client upon receipt of the answer ans and proof π, to be verified using the public and private key pair. It outputs an '`accept`' or '`reject`' notification. If the command was a modification command and the verification result is '`accept`', then, the client updates her local metadata as st', according to the proof.*

Definition 4.2 *ODB security game. There are two parties playing this game: the challenger who acts as the client, and the adversary who plays the role of the server.*

Key generation *The challenger generates the private and public key pair (sk, pk) using* `OKeyGen`*. She keeps both keys locally, and sends the public key to the adversary.*

Setup *The adversary specifies a command cmd (either a query or a modification) together with an answer ans and a proof π, and sends them to the challenger. The challenger runs the algorithm* `OVerify`*, and notifies the adversary about the result. If the command was a modification command, and the proof is accepted, then the challenger applies the changes on her local metadata. The adversary can repeat this interaction polynomially-many times. Let D be the database resulting from verified commands.*

Challenge *The adversary specifies a command cmd', an answer ans', and a proof π', and sends them to the challenger. He wins if the answer ans' is different from the result set of running cmd' on D, and cmd', ans', and π' are verified as accepted by the challenger.*

Definition 4.3 *ODB Security. We say that an ODB scheme is secure if no PPT adversary can win the ODB security game with non-negligible probability.*

Note that *the ODB security game covers all previously separate guarantees: correctness, completeness, and freshness*. This is simply due to the fact that the game requires that no adversary can return a query answer together with a valid proof such that the returned answer is different from the answer that would have been produced by the actual database. If any one of the freshness, completeness, or correctness guarantees were to be invaded, the adversary would have won the game. Looking ahead, in our proofs, the challenger keeps a local copy of the database, and can detect whether or not the adversary succeeded. If he succeeds, our reduction shows that we break some underlying security assumption.

4.3 Generic ODB Construction

A generic way to construct an ODB is to employ a regular DBMS, together with a DBAS built using a number of ADSs. A common problem among all previous ODB schemes is the existence of duplicate values in non-PK columns, since making an ordered ADS (which is necessary for range queries) requires a total order on the data items. The existing solutions [6, 32, 19, 31] are not efficient (see Section 6.2). Our HADS solves the problem efficiently, and easily generates proofs for the answers to multi-dimensional queries.

If the query has a clause on a non-PK column, say col_i, containing duplicate values, the result set of the query includes all records with the specified value(s) in col_i. The way we can identify these records and compare them with the result set of the other clauses is to relate each record to its corresponding (unique) PK.

Definition 4.4 *PK-set. For each distinct value v_i in a non-PK column of a table T, the set of all PK values corresponding to v_i in all records of T is called the PK-set of v_i, represented as $PK(v_i)$, i.e., $PK(v_i) = \{k_j \in PK(T) : \exists \text{ record } R \in T \text{ s.t. } k_j \in R \land v_i \in R\}$.*

Note that the PK-set includes only the PK values, not the whole records. Any membership scheme can be used for assigning the PK-set to a non-PK value, regarding the client and server processing power, and communication requirements of the application under construction. The only difference is the type of corresponding proof that is generated by the server and verified by the client. This brings the flexibility to support multiple membership schemes, and select one based on the state of the system at that time (further discussed in Section 4.4). The PK-sets of distinct elements of column `major` are shown in Figure 4a.

We construct the DBAS in the following way. Since all values in the PK column(s) are distinct, we use a regular (single-level) ordered ADS to store the corresponding security information, similar to the ones presented in the previous work [30, 41]. An example ADS for storing the PK column of the `Student` table, using an authenticated skip list is presented in Figure 3a. For a non-PK column, for simplicity, a two-level HADS stores the security information: the distinct values are located at the first (upper) level (i.e., each duplicate value is stored exactly once), and the corresponding PK-sets of these values are located at the second (lower) level. A sample HADS for storing the `major` column of the `Student` table is illustrated in Figure 4a. It uses an authenticated skip list at the first level, whose leaves are tied to Merkle hash tree digests at the second level.

The client locally stores the digests of the HADSs of each searchable column as metadata. Later, she checks the authenticity of server's answers against these digests. This method requires the client to store digests in the number of searchable columns in the database. As an alternative design, the client can put the digests of searchable column of each table in another ADS (the table ADS), and on top of them make another ADS (the database ADS) just as in Figure 4b. Then, she needs to store only the digest of this new (four-level) HADS as metadata. One may further extend this idea to multiple databases a user owns, and then multiple users in a group, and so forth. By increasing the number of levels of the HADS, it is possible to always make sure the client stores a single digest. This presents a nice trade-off between the client storage and the proof-verification performance. For the sake of simple presentation, we will employ two-level HADS constructions.

Using the authenticated range query for proof generation ensures completeness. Freshness is provided through storing the digest(s) at the client side. To provide correctness (i.e., the horizontal proof [30]), we store the hash of the corresponding record, $h(record)$, with each PK. In flat ADSs like the accumulator, the hash values are tied to the elements, while in tree-structured ADSs, the hash values are stored at the leaves. (The computation of values of the intermediate nodes, if there exists any, depends on the underlying structure of the ADS in use.) The ADS of the PK column of a table T is built using the set of all PK values and hashes of their records $\{(pk_i, h(record_i))\}_{i=1}^{|T|}$ as (key, value) pairs. For a non-PK searchable column col_j of a table T with d distinct values $\{v_i\}_{i=1}^{d}$, the corresponding HADS is constructed as follows: For each distinct $v_i \in col_j$, a second-level ADS is built using the (key, value) pairs $\{(pk_s, h(record_s))\}$, where $pk_s \in PK(v_i)$. Then, a first-level ADS storing pairs $\{(v_i, h(h(v_i)|h(\textit{digest of the corresponding second-level ADS})))\}$ is constructed.

The client outsources these (H)ADSs together with the database, while keeping their digests locally as metadata. Later, upon receipt of a proof and answer (result set), she performs the verification using the information provided in the proof and hashes of the records in the result set. If all records are used (to be discussed in Section 4.3.2) and the proofs verify according to the local digests, then the client accepts the proof and the answer.

We decouple the security information from the real data as Goodrich *et al.* [15] did. The DBAS stores the security information and generates proofs to be sent to the client. The DBMS stores the client's data. They can reside both on the same machine, or on different machines. By using techniques in [30, 41, 4], it is possible to implement authenticated skip list or Merkle tree proofs of the DBAS using a DBMS as well. In such a case, the DBAS can share the same DBMS with the data, or use a separate DBMS. When the server receives a command, he relays it to both the DBMS and the DBAS, collects their responses, and forwards them to the client. We focus on the DBAS, since the DBMS has nothing to do with proof generation and authentication.

HADS proofs. The membership proofs of HADSs for non-PK columns consist of two parts: the first part proves the (non-)existence of the *unique* value(s) in the column, and the second part ties each value to the respective PK-set. A key difference with a regular ADS is that after showing the existence of a value in the first-level ADS, all values in the related second-level ADS (storing the related PK-set) should be included without further computation, since they all share the same values in the queried column. This reduces both the proof size (communication) and proof generation time (server computation). However, the client verification cost for HADS is very close to ADS, since she needs to reconstruct the whole second-level ADS along with the membership path in the first-level ADS. For the ADS, the client reconstructs the whole sub-tree consisting of the values in the proof.

Consider a table with d distinct values in column C_j, each repeated r times, on average, leading to rd records in

(a) Proof generation in an ADS after duplicate elimination.   (b) Proof generation using HADS
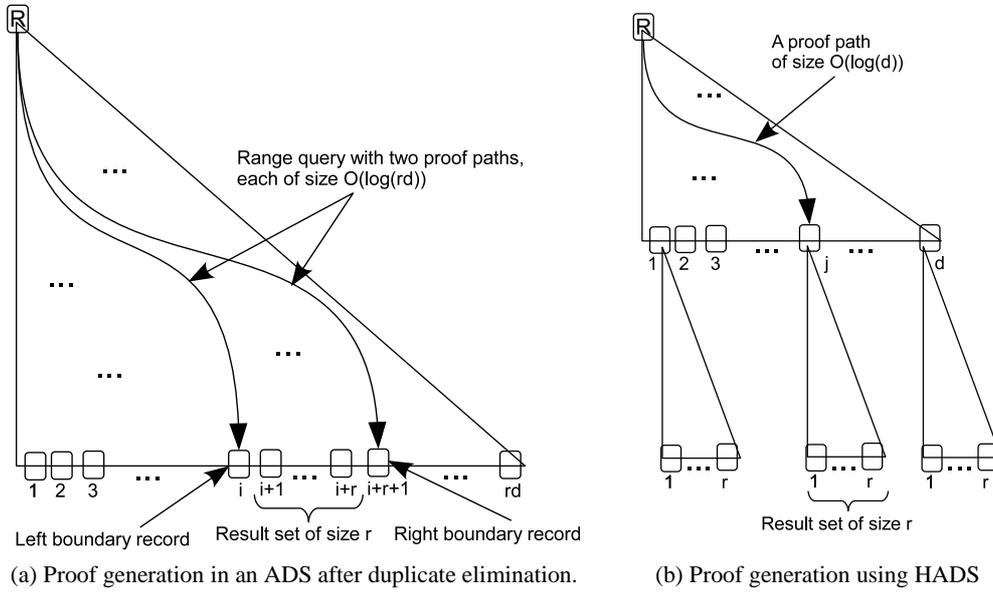
Figure 6: A comparison of proof generation and proof sizes in an ADS and an HADS.

total. Using a duplicate elimination mechanism [6, 32, 19, 31], we can store such a table inside a regular ADS. The HADS builds a first-level ADS of size d, whose leaves are each connected to a second-level ADS of size r, leading to HADS size rd. Therefore, the server storage remains the same. However, for a query about a value v_i in C_j, the ADS proof size and proof generation time both are $O(2\log rd + r) = O(\log r + \log d + r)$, while those of the HADS are both $O(\log d + r)$. The ADS uses a range query with $2O(\log rd)$ cost, and processes the r values as the result set. However, the HADS finds v_i at the first-level ADS with cost $O(\log d)$ and accesses all r values in the second-level ADS. This is presented in Figure 6 and further detailed in Section 4.3.2.

4.3.1 Illustrative Examples

We give some examples to better understand our construction.

Selection in a four-level HADS (Figure 4b). The DBAS first converts the query SELECT * FROM Student WHERE major in(`CE`,`CS`) and BCity=`Istanbul` to (key, value) pairs: (Student,{(major,{CE,CS}),(BCity,{Istanbul})}). Then, it asks the HADS to generate and return the corresponding proof. The HADS runs HCertify. With the help of the Find algorithm that decomposes the converted query into the proper parts and finds the next-level ADSs, HCertify works as follows: It asks the database ADS to give its proof, given the converted query. The database ADS, in turn, executes the Find algorithm, which interprets the converted query and uses the key Student to find the next-level ADS. Then, the database ADS asks the Student ADS to recursively give its proof, supplying it with the input {(major, {CE,CS}), (BCity, {Istanbul})}. Now, the Student ADS, via the Find algorithm, finds two next-level ADSs: the major ADS and the BCity ADS, and asks them to give their proofs by providing the required inputs {CE,CS} and {Istanbul}, respectively. These two ADSs, working in parallel, repeat the same steps and find the last-level ADSs storing the PK-sets of values CE, CS, and Istanbul, and ask them to give their proofs. After receiving proofs from the last-level ADSs, the major ADS and the BCity ADS generate and add their own proofs, and relay the result back to the Student ADS who will do the same job and send the result to the database ADS. The database ADS generates and adds its own proof and sends the resultant full proof to DBAS to hand on to the client.

Selection in a two-level HADS. Figure 7 presents another example showing the proof generation with a two-level HADS, for the query SELECT * FROM Student WHERE major=`CS` and stdId=103, which is translated by the DBAS into '(Student,{(major, {CS}),(stdId, {103})})'. The first level is an authenticated skip list containing unique values of the major column, and the second level has three Merkle hash trees containing stdId values matching each major value (i.e., their PK-sets). The first-level ADS needs to prove membership of CS. This can be done by returning 'h$'$1,**CS**,h(EE),h(+∞)'; essentially the result, together with the hashes of the nodes required to obtain the corresponding digest. At the second level, the Merkle tree needs to prove membership of 103. This is done by returning '**103**,h(105)'. The generated verification object will look like: vo='h$'$1,**CS**(**103**,h(105)),h(EE),h(+∞)'. The client can verify both levels using this vo together with the hash of the records in the returned result.

Modification. As an example targeting modification, consider adding a new record into the Student ta-

The query is converted to: key=Student, value={(major, CS), (stdId, 103)}
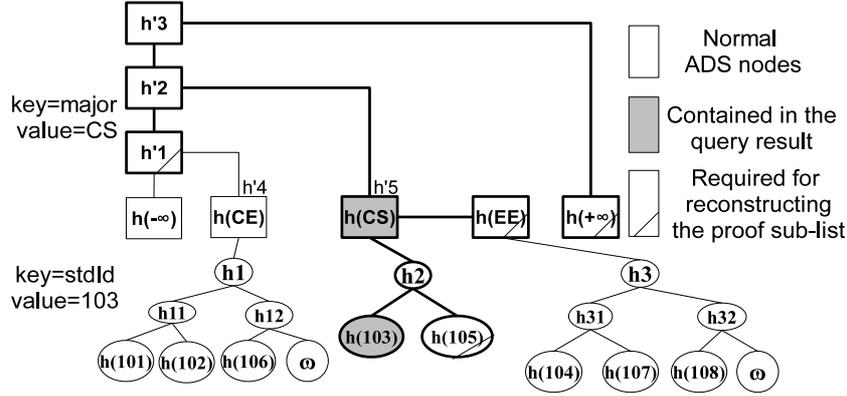
Figure 7: Proof generation for `SELECT * FROM Student WHERE major='CS' and stdId=103`.

ble: `INSERT INTO Student VALUE(109,'Cem','CE','Izmir')`. This adds the pair (109,h(record)), where h(record)=h(h(109)|h('Cem')|h('CE')|h('Izmir')), into the ADS of the PK column. We further need to add (109,h(record)) to the second-level ADS associated with CE. Once this is done, since the digest of the CE ADS would be modified, we need to reflect this in the major ADS as well. Similarly, we need to construct a new Izmir ADS, containing only (109,h(record)), and add its digest to the BCity ADS. Therefore, using two-level HADS constructions, there will be three parts in the translated command: (109,h(record)) to be executed by the ADS of the PK column, (CE,(109,h(record))) for the major HADS, and (Izmir,(109,h(record))) for the BCity HADS. In a four-level HADS construction, the translated command looks like: (Student,{⟨stdId, (109,h(record))⟩,⟨major,(CE,(109,h(record)))⟩,⟨BCity,(Izmir,(109,h(record)))⟩}).

Verification. Verification is fulfilled similarly in a bottom-up manner. The client first verifies the PK-sets' proofs. If all are verified, it goes on to use them for verifying the column ADSs' proofs. If this step also was successful, its results are used to verify proofs of the table ADSs (the Student table, in our example). Finally, the database ADS proof is verified in a similar manner. If all proofs are verified employing *all* and *only* the records in the answer, then the client accepts the answer as authentic.
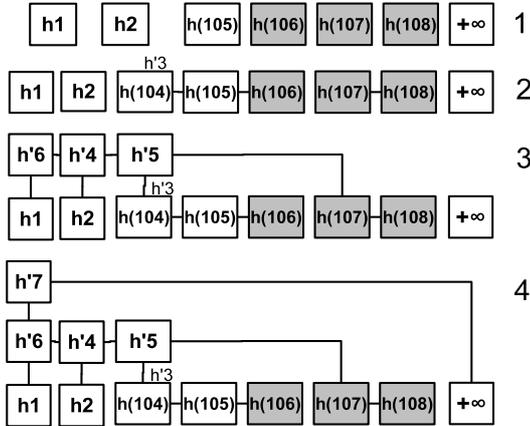
Figure 8: Proof verification for *vo*='h1, h2,h(104),105,**106,107,108**,h(+∞)'.

Since the verification is accomplished similarly at all levels, we give an example showing verification in the ADS of Figure 3b, where the proof *vo*='h1,h2,h(104),105,**106,107,108**,h(+∞)' is given for the query `SELECT * FROM Student WHERE StdID>105`. The verification algorithm extracts the result set {106,107,108} and the boundary records {105,+∞}, and checks whether 105<106 <107<108< +∞ (step 1). If the check is passed, it uses h(104) to compute h'3 (step 2). In the step 3, it uses h'5 and h(107) to compute h'5, which is used together with h2 to compute h'4, which in turn, is used along with h1 to compute h'6. Finally, it uses h'6 and h(+∞) to computes h'7, the digest of the computed ADS. Now, it compares h'7 against the digest stored locally (h7). This process is illustrated visually in Figure 8. Note that, a full proof would also contain information about the levels of these nodes in a skip list, but those parts are hidden for the sake of a simpler presentation. Thus, assume that the server also tells the client where to connect these nodes at in the proof.

4.3.2 Proof Generation

To provide details on how the DBAS generates proofs, we consider different cases where the query has only one clause, or multiple clauses. For each case we discuss how the proof is generated, and what is included in the proof.

One-dimensional queries: contain only one clause. There are two possible cases:

- **The clause is on the PK column**: For example, the query is `SELECT * FROM Student WHERE stdID > 105`. The server asks the HADS of the PK column of the Student table to compute and return its range proof, and sends it back to the client. The proof includes the *boundary* records, and all intermediate nodes' values required for verification at the client. (Note that we employ ADSs supporting multi-proofs.) Figure 3b depicts

an example, using authenticated skip list as the underlying ADS, where the result set is (106, 107, 108), and the boundary records are 105 and $+\infty$. The proof looks like: *vo*=‘h1,h2,h(104),105,**106**,**107**,**108**,$+\infty$’.

- **The clause is on a non-PK column**: A sample query is SELECT * FROM Student WHERE major=‘CE’. The server uses the HADS of the major column to find CE at the first level. If not found, he puts the non-membership proof in *vo*. Otherwise, he puts the CE's membership proof and all values in its PK-set (in the second-level ADS) in the *vo*. In contrast to storing duplicate-eliminated data in regular ADSs, the first-level ADS is very small, and all values in the second-level ADS are used without further computation. The proof will look like: *vo*=‘h($-\infty$),**CE**(**101**,**102**,**106**),h$'$5,h($+\infty$)’, using Figure 7.

Multi-dimensional queries: For each clause, the server asks the corresponding HADS to give its proof, collects them into the verification object *vo*, and sends it to the client. Upon receipt, the client verifies all proofs one-by-one, and accepts if all are verified. If the clauses were connected by 'OR', then each proof verifies a subset of the received records, and the result set should be the union of all these verified records. For 'AND', each proof verifies a superset of records in the result set, and hence the answer is the intersection of results of the individual clauses. Therefore, each proof must verify all records in the result set. *An important distinction between our HADS and previous schemes [7, 32, 28, 31] is that our proofs can be compared and combined together.* Possible scenarios for two-clause case are:

- **One clause on the PK, the other on a non-PK column**: For example, the query is SELECT * FROM Student WHERE StdID > 105 AND major = ‘CE’. Since the order in which the clauses are applied is not important for the proof, we can consider the non-PK clause first, then apply the PK clause on the results of the first step. Therefore, the server first applies the non-PK clause on the corresponding first-level ADS, and then, applies the PK clause on the resultant second-level ADSs. Finally, he adds them both to the *vo*, and sends it to the client. On Figure 7, this method produces the proof *vo*=‘h(-∞),**CE**(h(101),102,**106**),h$'$5,h($+\infty$)’.

- **Both clauses on non-PK columns**: A sample query is SELECT * FROM Student WHERE BCity=‘Istanbul’ AND major=‘CE’. The server generates one proof for each clause, each containing the first-level ADS proof for the value itself (e.g., Istanbul and CE) and the corresponding PK-set, puts them into the *vo*, and sends it to the client. Each proof proves authenticity of a set of PK values (of the same table) that can be combined and compared together. If the clauses were connected by 'AND', the client only takes their intersection and checks whether the result set contains only records with these PKs. For 'OR', union of these authentic sets is used.

The above process can be generalized to more than two clauses and supports any combination of 'AND', 'OR', and 'NOT' operators. The client verifies the proofs, performs a number of set operations on the resulting authentic sets of PKs, and compares them with the result set. Note that in all our proofs, **we do not require any additional records to be sent to the client on top of the result set of the original query**.

4.3.3 Tables with Composite Keys

The foreign keys are used to relate the tables to each other, and hence some tables may employ composite keys (i.e., a PK includes multiple columns). This, in turn, makes the construction problematic: we cannot relate a non-PK column to any of the foreign key columns due to the existence of duplicate values (each foreign key column alone may contain duplicate values). Previous schemes [31, 42] that use regular ADSs cannot handle this case efficiently, as they need to construct and use multiple ADSs for each column.
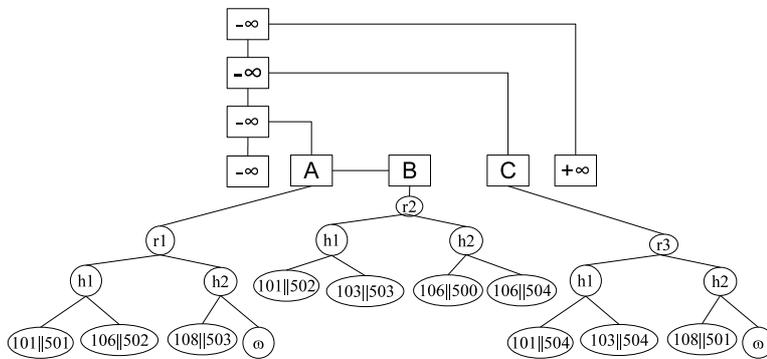
Figure 9: Storing the column Mark from table S2C with composite PK (stdId and crsId).

HADS solves this problem efficiently. Note that generally the concatenation of multiple foreign keys forms the composite key. Thus, we use this composite key as the PK of the table, and use it to construct the HADSs. One HADS is constructed for each searchable column(including foreign key columns), relating the column's values (containing duplicates) stored at the first-level ADS (remember that the ADS contains only one copy of each replicated value) to the unique PK values (constructed as the composite key) stored at the second-level ADSs. These HADSs can be used in connection with other HADSs to generate the proofs. An example is depicted in Figure 9 where the composite key for table S2C is stdId$\|$crsId.

4.4 Efficient ODB Construction

Different ADSs can be chosen for HADS levels subject to their requirements and the application. We employed two-level HADSs, with special role and considerations for each level. We compare the existing ADSs and investigate their eligibility to be used in each level. We consider three classes of ADSs: *linear* (e.g., one-way accumulator [1]), *sublinear* (e.g., authenticated hash tables [36]), and *logarithmic* (e.g., authenticated skip list [13, 10]).

For each level in an HADS, an ADS can be chosen subject to the requirements of that level and the application. We employed two-level HADSs, each level having a special role and posing special considerations. We compare the existing ADSs and investigate their eligibility to be used in each level. We consider three classes of ADSs: *linear* (e.g., one-way accumulator [1]), *sublinear* (e.g., authenticated hash tables [36]), and *logarithmic* (e.g., authenticated skip list [13, 10]).

First level. This level stores the distinct values of a column, and generates the first part of the proof to be sent to the client. Proof generation is based on the authenticated range queries, which implies that this level should use an *ordered* ADS. One-way accumulator and hash tables do not support this property efficiently, and hence cannot be used for this level.

Therefore, we choose the authenticated skip list (alternatively, the Merkle hash tree) to be used in the first level. The proof time/space is $O(\log(|C_i|))$ for an update, and $O(\log(|C_i|) + t)$ for a query with $O(t)$ records in the result set. There are $|C_i|$ distinct values, on average, stored in the first-level ADS, therefore, the storage complexity is $2|C_i|$, which is $O(|C_i|)$.

Second level. This level stores the PK-sets of values in the first level. For one-dimensional queries, and multi-dimensional queries connected with 'OR', the order of values in the PK-set is not a matter of importance, thus, any ADS can be used with time/space trade-offs discussed below. The second-level ADSs of multi-dimensional queries connected with 'AND' should be compared to generate efficient proofs, hence, an ordered ADS should be employed.

Accumulator. For each distinct value in a column, an accumulated value is computed using all values in its PK-set. For each PK value, a witness is computed which proves that it belongs to the specified PK-set. If we need to select all PK values, the second-level proof is essentially empty, but to select a subset of the PK values (mostly required for 'AND'), the witnesses of the selected PK values are required to be sent to the client.

For each distinct value in the first-level ADS, $N/|C_i|$ PK values and witnesses should be computed and stored, on average, where N is the total number of records in the table. In total, $2|C_i| + |C_i| * N/|C_i| = 2|C_i| + N$ (which is $O(|C_i| + N)$) storage is required (including the $2|C_i|$ space for the first-level ADS).

A proof for each value is made up of two parts, one for the first-level ADS (e.g., for authenticated skip list, a path from the leaf up to the root, which is $O(\log |C_i|)$), and the other is the accumulated value along with all values in the PK-set, which is $N/|C_i|$ (the accumulated value is already included in the hash value stored at the corresponding leaf of the first-level ADS). The client herself can check validity of the PK-set against the accumulated value. Therefore, for a result set of size t, the asymptotic size of *vo* will be $O(\log |C_i|) + 2t \simeq O(\log |C_i| + t)$.

The main problem with the accumulator is the cost of update: with each update, all witnesses should be updated using costly operations (e.g., modular exponentiation).

Authenticated hash table. This is a sublinear membership scheme with constant query and verification time, making it an interesting scheme for clients with resource-constrained devices. It is a good choice if the data is static. For a leaf node storing v_i, we put the PK-set of v_i in an authenticated hash table, and store its digest at the level above.

On average, $N/|C_i|$ PK values are linked to each leaf node, hence, we require $O(|C_i| + (1 + \varepsilon)N/|C_i| * |C_i|) = O(|C_i| + (1 + \varepsilon)N)) \approx O(|C_i| + N)$ storage (including the $O(|C_i|)$ space for the first-level ADS). Here, ε is a constant.

The first-level ADS proof is the same, but the authenticated hash table requires only constant proof size ε [35], reaching $(O(\log |C_i|) + t)$ for t records in the result set. Moreover, hash operations are much faster than modular exponentiations of the accumulator.

Merkle tree or authenticated 2-3 tree or authenticated skip list. These are logarithmic membership schemes with logarithmic height and proof size. The way the second-level schemes are modified, or the proofs are generated, are the same as for the first-level.

Each node requires $\approx 2(N/|C_i|)$ storage to store the PK-set, therefore, $2|C_i| + 2|C_i| * N/|C_i| = 2(|C_i| + N) = O(|C_i| + N)$ storage is required to store a column. The proof size and time for one record are both $O(\log |C_i| + \log(N/|C_i|)) = O(\log N)$, and for $r = tN/|C_i|$ records are both $O(\log |C_i| + r)$.

A comparison of ODB construction via various ADS schemes is given in Table 1, where the first level is a logarithmic ordered ADS and the second levels are shown in the table. Note, however, that the unit operations in the accumulator are more costly than those in the others. It shows that using a logarithmic ADS such as an authenticated

Table 1: A comparison of schemes for the second level where the first level is a logarithmic ADS, for storing a single table. Proof size and verification time is given for one-dimensional queries. s and t denote the number of searchable columns, and the number of records in the first level, respectively.

	Accumulator	Authenticated hash table
Storage	$2N + (s-1)(2|C_i| + 2N)$	$2N + (s-1)(2|C_i| + N)$
Proof size	$2\log|C_i| + t + 2tN/|C_i|$	$2\log|C_i| + t + 2t*N/|C_i|$
Verification time	$t(\log|C_i| + N/|C_i|)$	$t(\log|C_i| + N/|C_i|)$
Update time	$\log N + (s-1)(\log|C_i| + N/|C_i|)$	$\log N + (s-1)(\log|C_i| + N/|C_i|)$
	Authenticated skip list	
Storage	$2N + (s-1)(2|C_i| + 2N)$	
Proof size	$2\log|C_i| + t + tN/|C_i|$	
Verification time	$t(\log|C_i| + 2N/|C_i|)$	
Update time	$\log N + (s-1)(\log|C_i| + \log N/|C_i|) = s\log N$	

skip list at both levels is the efficient choice leading to $O(\log|C_i| + r)$ proof size and time for $r = tN/|C_i|$ records, and $O(\log N)$ update time for one record. Other alternatives can be chosen regarding the requirements of applications, such as the database being static or dynamic.

5 Join

In relational database systems, data is organized (divided) into a set of tables. An important and frequently-used operation is, therefore, the *join* operation, which collects data from two (or more) tables to produce new results. In outsourced databases, the server should perform the join and generate the proof that will be verified by the client. The server can utilize any existing optimal join algorithm, since we put no restriction on the DBMS part. Instead, we design our DBAS proof generation algorithms to produce efficient proofs minimizing the server's effort, the communication, and the client's computation.

5.1 Overview

Our join algorithms use HADSs for both (all) tables that are built on the columns on which the join is formed. Since the HADSs keep the same relationships between the (values of) tables they are created for, we can generate proofs proving correctness of those relations.

Without loss of generality, consider a one-to-many relationship, which is the most widely used relationship: R $\bowtie_{rid=rid}$ S, i.e., the PK column of R, *rid*, is used as a foreign key in S. R contains only distinct values in column *rid*, while S may contain duplicate values. The HADS of S ties each distinct value in *rid* to its respective PK-set in S. Now, we can easily compare the ADS of R built on *rid* with the first-level ADS of the HADS of S (storing unique values) built on *rid*, and generate efficient proofs. (Note that only the first-level ADS of the HADS, which is very small in size, is used for comparison, and in case of any match, all values in the respective second-level ADS are reflected into the *vo* without further computation.) Besides, as the values are stored sorted, the server traverses each ADS only once.

Efficient proof generation. Compared to [18, 34] that for each value of the first table, perform a range query on the second table, and [42] that uses range queries efficiently, ours is more efficient as it converts range queries into equalities for matches. The problem with [42] is that for each value in the first ADS, the set of matching values in the second ADS is surrounded by **two more** records, for completeness. Since we store and compare unique values in HADSs, a value in the first (H)ADS either matches only one value in the second (H)ADS that is shown by equality in *vo*, or does not match any value in the second (H)ADS that is shown using range queries. In addition, the first-level ADSs that we use for proof generation are very small compared to those of all previous work, reducing the proof size and proof generation time.

Other join types. The HADS, in addition to the equi-join, supports non-equi-join and multi-way join as well. Although an inefficient way of doing a non-equi-join between R and S is performing a range query on S for each record in R, our non-equi-join algorithm traverses each ADS only once, and is very efficient. Our algorithm for multi-way join queries can be generalized to support queries of the form $T_1 \bowtie_{a=a} T_2 \bowtie_{a=a} T_3 \bowtie_{a=a} ...$, between n tables.

5.2 Two-way Join

Consider equi-join on two tables R and S represented as $R \bowtie_{C_i=C_j} S$, where C_i and C_j are columns of R and S, respectively. The HADSs of these columns will be used for proof generation. We categorize possible cases and discuss each one separately.

Either C_i or C_j is a PK column that is used as foreign key in the other table. The generated *vo* is a set of PKs that can be used for comparison or combining with other *vo*'s.

The server uses $HADS_R(C_i)$ and $HADS_S(C_j)$ for proof generation. He starts by the smallest item (e.g., leftmost leaf node in a tree or skip list type ADS) in the first-level ADS of one of the HADSs, and searches for its value, say v_i, on the other HADS. If the value is found on the other HADS, both values are inserted into the *vo* showing a matching. Otherwise, the boundary records (the two *consecutive* values on the other HADS that v_i would have been located between them), together with the v_i, are inserted into the *vo*. This shows that v_i has no matching on the other table. Once finished working on it, he jumps to the next *expected* node. By the expected node, we mean the item that either is immediately after the current node or stores the closest value to the current value of the other HADS. If the current and expected nodes are not successive, then the required intermediate information (e.g., for authenticated skip list, the levels and digests corresponding to a part of the ADS not included in the proof) needed for verifying the ADS by the client, will be added to the *vo*. We use the algorithm FindNext to find the expected node:

FindNext(v_i) $\rightarrow$ ($node_j$,$node_k$) If v_i is null, then return the node immediately following the current node as $node_j$ ($node_k$ will be null). Given a value v_i, if a node storing v_i is found, add the required information of the intermediate nodes into the *vo* and return the node storing v_i as $node_j$ ($node_k$ will be null again). Otherwise, add the needed information of the intermediate nodes into the *vo* and return the two *consecutive* boundary nodes $node_j$ and $node_k$ storing v_j and v_k, respectively, such that $v_j < v_i < v_k$.

Consider the join Student $\bowtie_{\text{stdId=stdId}}$ S2C, where both tables have an HADS on column stdId: $HADS_{\text{Student}}(\text{stdId})$ and $HADS_{\text{S2C}}(\text{stdId})$. The proof generation works as follows: Traverse both HADSs until the leftmost leaf node (at the first level) storing the values v_1 (in $HADS_{\text{Student}}(\text{stdId})$) and v_1' (in $HADS_{\text{S2C}}(\text{stdId})$):

- $v_1 = v_1'$: Add them into the *vo* (showing a matching), run the FindNext() on both HADSs to find the next values v_2 and v_2', and repeat the process with v_2 and v_2'.

- $v_1 \neq v_1'$: Add the larger value, say v_1, into the *vo* and run $HADS_{\text{S2C}}(\text{stdId})$.FindNext($v_1$) to find a matching on $HADS_{\text{S2C}}(\text{stdId})$. If it returns one node, a matching has been found, therefore, repeat the process with v_1 and the value of the matched node. On the other hand, if $HADS_{\text{S2C}}(\text{stdId})$.FindNext($v_1$) returns two nodes, say $node_j$ and $node_k$, there is no matching, but the value of $node_k$ may be equal to the value of the next node of v_1. Therefore, add v_1, $node_j$.*val*, and $node_k$.*val* into the *vo*, then find the node immediately after v_1 as $node_2 = HADS_{\text{Student}}(\text{stdId})$.FindNext(), and repeat the process with $node_2$.*val*, and $node_k$.*val*. The Algorithm 5.1 illustrates this process.
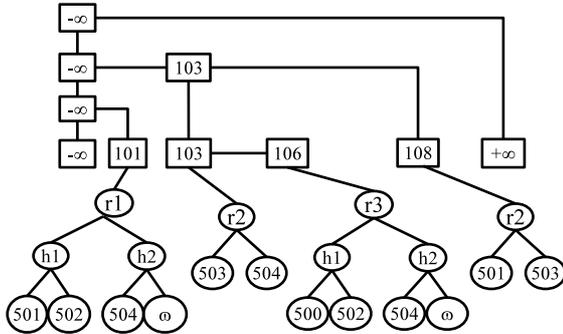
Figure 10: HADS of stdId (table S2C).

Using $HADS_{\text{Student}}(\text{stdId})$ from Figure 3a and $HADS_{\text{S2C}}(\text{stdId})$ from Figure 10, we generate proof for Student $\bowtie_{\text{stdId=stdId}}$ S2C. For simple presentation, we put in the *vo* only the values and hashes stored on nodes, and leave out the other information required for verification (e.g., the level in an authenticated skip list). Furthermore, we separate each round by a column ':', parts belonging to each HADS inside a round by a semi-column ';', and values inside each part by a comma ','. Within a round, values of $HADS_{\text{Student}}(\text{stdId})$ appear first.

We start with the smallest values in the HADSs: $v_1 = 101$ and $v_1' = 101$. Since there is a matching, 101 is added into the *vo* (*vo*='**101**;**101**(501,502,504)'). Then, the FindNext() is run on both HADSs to find the next values: $v_2 = 102$ and $v_2' = 103$. Since $v_2' > v_2$, 103 is inserted into the *vo* and $HADS_{\text{Student}}(\text{stdId})$.FindNext(103) is executed (during which $h(102)$ will be added into the *vo* as an intermediate value, resulting in *vo*='**101**;**101**(501,502,504) : $h(102)$;103(503,504)'), returning the node storing $v_3 = 103$. Due to the matching, 103 is again added into the *vo* (*vo*='**101**;**101**(501,502,504) : $h(102)$,**103**;**103**(503,504)'), and FindNext() is run on both HADSs that will result in: $v_4 = 104$ and $v_3' = 106$. Again, 106 is added into the *vo* and $HADS_{\text{Student}}(\text{stdId})$.FindNext(106) is executed (during which $h(104), h(105)$ will be added into the *vo* as intermediate values, resulting in *vo*='**101**;**101**(501,502,504) : $h(102)$,**103**;**103**(503,504) : $h(104)$,$h(105)$;106(500,502,504)'), returning the node storing $v_6 = 106$, to be added into the *vo* due to the matching. Then, FindNext() is executed on both HADSs, which will give: $v_7 = 107$ and $v_4' = 108$. 108 will be added into *vo* and $HADS_{\text{Student}}(\text{stdId})$.FindNext(108) results in $v_8 = 108$. Finally, *vo* will be *vo*='**101**;**101**(501,502,504) : $h(102)$,**103**;**103**(503,504) : $h(104)$,$h(105)$,**106**;**106**(500,502,504) : $h(107)$,**108**;**108**(501,503)'.

Algorithm 5.1: `JoinCertify`, run by the server.

Input: Two second-level ADSs of the joining tables: ADS_R and ADS_S, and their current nodes: $Node_R$ and $Node_S$, which are initialized by the leftmost nodes of the corresponding ADSs.

Output: the verification object: *vo*

```
1   if NodeR is null OR NodeS is null then
2   |     return vo =the intermediate information of the other ADS
3   if NodeR.val = NodeS.val then
4   |     vo = NodeR.val + ';' + NodeS.val      //A matching is found.
5   |     NextR = ADSR.FindNext()
6   |     NextS = ADSS.FindNext()
7   |     vo = vo + ':' + JoinCertify(NextR,NextS)    //Go to the next round.
8   else
          //Find the matching on the other ADS.
9   |     Find the node holding the bigger value, say NodeR (Next1S,Next2S) = ADSS.FindNext(NodeR.val)
10  |     if Next1S is null then
              //End of ADSS
11  |     |     vo = vo + ';' + NodeR.val, intermediate information of ADSR until the end
12  |     else
13  |     |     if Next2S is null then
14  |     |     |     vo = NodeR.val + ';' + Next1S.val     //A matching is found.
15  |     |     else
16  |     |     |     vo = NodeR.val + ';' + Next1S.val + ',' + Next2S.val   //No matching is found.
17  |     |     NextR = ADSR.FindNext()
18  |     |     NextS = ADSS.FindNext()
19  |     |     vo = vo + ':' + JoinCertify(NextR,NextS)    //Go to the next round.
20  return vo
```

Neither C_i nor C_j is a PK column. Each column has an HADS storing its distinct values and related PK-sets. If each distinct value of C_i and C_j has an average PK-set of size n and m, respectively, and there are k matching records, then the result set will have knm records, on average. Our proof for this query is of size $O(k(n+m))$, showing again the HADS proofs are efficient.
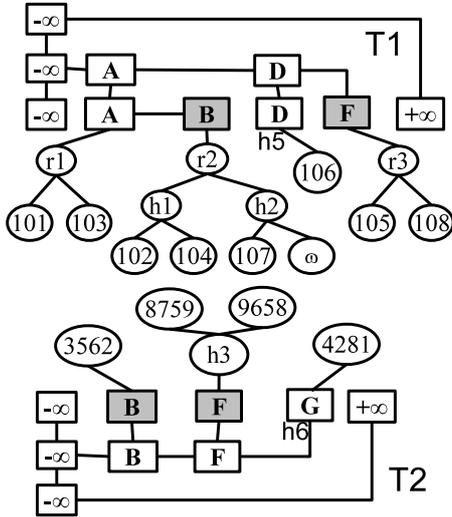
Figure 11: Non-PK join.

Imagine two tables T_1 and T_2, both having an integer PK column and a non-PK column of type character with two matching values 'B' and 'F', whose HADSs are shown in Figure 11. The algorithm, starting at the leftmost nodes of both HADSs, finds out that B>A, and executes `FindNext('B')` on T_1, leading to *vo*= 'h($-\infty$), r1,**B**(102,104, 107);h($-\infty$),**B**(3562)'. It goes on, putting intermediate value h5 in the *vo*, finds another matching 'F', which is the last node in T_1. Later, `FindNext('F')` on T_2 puts h6 in *vo*, and realizes that both columns are fully traversed. These steps yield *vo*='h($-\infty$), r1,**B**(102,104,107);h($-\infty$),**B**(3562) : h5,**F**(105,108);**F**(8759,9658) : h($+\infty$);h6,h($+\infty$)'.

For verification, the client interprets the proof in *vo*, and investigates whether the values in each step are either equal, or one is between the two others. If it is correct, she adds them to the corresponding ADS list, and goes on with the next step proof (any problem leads to rejection). Finally, she uses the `Vertify()` function of the (H)ADS to verify the two ADS lists. If both passed the verification successfully, she accepts the proof, otherwise, rejects.

5.3 Queries with Join and Selection

As denoted by Mishra and Eich [24], the general query optimization rule for queries containing various operations is that the join operation is performed after all selection operations. The reason is that the selection operations result in intermediate sub-tables (to be used as input to the join operations) that are likely to vary substantially in size [24]. Since all proof are based on PKs in our approach, the results of the selection queries are integrated easily into those of join queries, resulting in small proofs (in terms of both space and computation). We distinguish the following cases:

- **The selection uses the same column as the join**. The same HADSs are used to generate proofs for both selection and join, i.e., records in the result set should satisfy the selection constraint in addition to the join constraint. For example, the proof generation for query `SELECT * FROM Student S, S2c C WHERE S.stdId=C.stdId and S.stdId > 105` starts from the node storing the value 104 (the boundary record), and both clauses are applied simultaneously during the join.

- **The selection uses different columns than the join**. The selection proof is generated first that results in an *authenticated set of PKs*. Then, if this is connected to the join clause with 'OR', the proof of the join clause is also generated, and both proofs are sent together to the client. But for 'AND', the join proof-generation algorithm should consider only those records that are in the selection proof, instead of the whole table, leading to smaller join proofs. The server runs the join proof-generation algorithm on sorted authentic PK-set resulting from the selection proof, and the other table. For each PK value in the sorted authentic PK values, if there is a matching on the corresponding HADS of the other table, reflect it on the proof. Otherwise, supply a non-membership proof. For the query `SELECT * FROM Student S, S2c C WHERE S.stdId=C.stdId and S.major ='CS'`, for instance, the selection proof supplies the sorted authentic set of PK values $\{103, 105\}$, used together with table `S2C` by the join proof-generation algorithm to compute the (smaller) join proof.

5.4 Multi-way Join

Since data is distributed over multiple tables, users may issue queries with join on multiple tables, e.g., $T_1 \bowtie_{C_i=C_j} T_2 \bowtie_{C_k=C_l} T_3 \bowtie \dots$, to combine them back together. Yang *et al.* [42] performed the three-table join as $((T_1 \bowtie_{C_i=C_j} T_2) \bowtie_{C_k=C_l} T_3)$ or $(T_1 \bowtie_{C_i=C_j} (T_2 \bowtie_{C_k=C_l} T_3))$. But, the output of the join that is performed first, is not a table having an ADS on the column of the next join. Therefore, their AIM join algorithm is not applicable, and their AISM join algorithm (which uses only one ADS on one table) is used instead. Essentially, they apply AIM for the first join, followed by AISM.[1] We treat the case that all joins are on the same column separately from the case that the columns differ, and present efficient solutions for all such scenarios.

Multi-way join on the same column. As noted by Raman *et al.* [38] and Yang *et al.* [42], these queries are common in data warehousing applications, where a fact table is joined with other tables, on the same column. Our algorithm performs much better for the multi-way join with all join clauses on the same column: $T_1 \bowtie_{a=a} T_2 \bowtie_{a=a} T_3 \bowtie_{a=a} \dots$. Moreover, our algorithm can be generalized to support multi-way joins between n tables, without change.

We start by the smallest items in all HADSs. If all are the same, this is reflected in the *vo*, showing a matching. Otherwise, the maximum value among them, v_{max}, is selected and added into the *vo* and all the remaining HADSs are queried (i.e., FindNext(v_{max})) to either find a matching, or prove non-existence of the value. This is repeated until the last node of one of the HADSs is met. Then, the verification object is finalized with the remaining intermediaries. Each HADS is traversed exactly once, and no item is checked multiple times. Jumping to the maximum value when no matching is found enables us to skip the largest possible number of items, providing an optimally efficient proof.
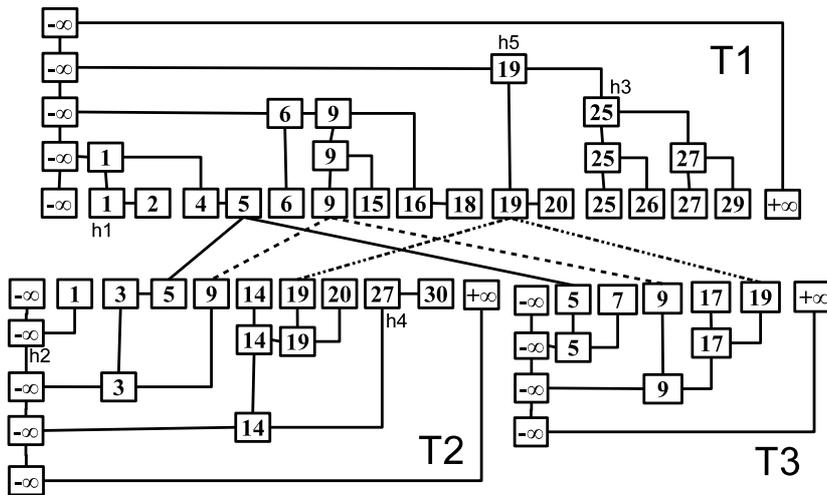
Figure 12: Proof generation for $T_1 \bowtie_{a=a} T_2 \bowtie_{a=a} T_3$.

An example showing our join proof generation algorithm for $T_1 \bowtie_{a=a} T_2 \bowtie_{a=a} T_3$ is given in Figure 12. It first starts by the leftmost nodes that are 1,1,5. Since 5 is their maximum, FindNext(5) is run on both T1 and T2, leading to *vo*='h($-\infty$),h1,h(4),**5**;h2,h(3),**5**;h($-\infty$),**5**'. After this matching, the algorithm then jumps to and processes the next nodes, which are 6,9,7, and thus continues by FindNext(9) on T1 and T3. Following the same logic, it finally outputs *vo*='h($-\infty$),h1,h(4),**5**;h2,h(3),**5**; h($-\infty$),**5**' : h(6),**9**;**9**;h(7),**9** : h(15),16, 18;14,19;17 : **19**;**19**;**19** : h(20),h3, h($+\infty$);h(20),h4,h($+\infty$);h($+\infty$)'.

---

[1] Their algorithms are not directly applicable for multi-join case, so, they provided new versions m-AISM, m-ASM, and m-AIM. They require some prior information about the third table that is used for reducing the proof size of the first join, between the first and second tables, before the second join is performed.

(a) The sample database.　　(b) T3 refers to T2 who refers to T1.　　(c) Both T1 and T3 refer to T2.
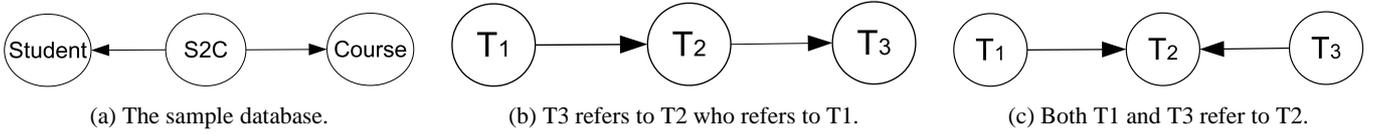
Figure 13: Ordering graphs for different cases.

Multi-way join on different columns. Since our proofs are composed of a set of PKs, we can compare and combine them together. To perform a multi-way join, we separate it into a set of two-way joins (with selections, if there exists any), and apply our two-way join algorithm as described previously. For a query with n joins, we generate and send n proofs to the client who verifies them, and accepts the answer if all proofs are verified.

To perform a multi-way join of the form $T_1 \bowtie_{C_i=C_j} T_2 \bowtie_{C_k=C_l} T_3$, one way is to deal with $T_1 \bowtie_{C_i=C_j} T_2$ independently from $T_2 \bowtie_{C_k=C_l} T_3$, and generate the proofs directly using the HADSs. Another way is to perform one of them first, and use its result, which is an authentic PK-set, to generate the next proof. This means that the proof for each join depends on the previous join, which depends, in turn, on the preceding one. Since a join leaves out some records, using its result for the next join is expected to generate smaller proofs. Thus, we can perform the joins according to an order that generates efficient proofs. We categorize the possible cases and investigate employing an efficient ordering.

Efficient ordering. We define the *ordering graph* as a directed graph to show the relationship between the tables and use it to determine the order of joins. The joined tables constitute the vertices, and an edge from T_i to T_j indicates that table T_i contains a column that refers to a column in T_j (and the join is on these two columns). The ordering graph of our database model (Figure 1a) is represented in Figure 13a.

Consider the case in Figure 13b: We should perform the $T_2 - T_3$ join first, followed by $T_1 - T_2$ join. The reason is that the $T_2 - T_3$ join results in an authentic set of T_2's PKs that can be used in the $T_1 - T_2$ join (that is on T_2's PK), while the result of $T_1 - T_2$ join (authentic sets of PK values of tables T_1 or T_2) cannot be used in $T_2 - T_3$ join that is on T_3's PK. Hence, performing the $T_2 - T_3$ join first, generates efficient proofs.

In Figure 13c, both T_1 and T_3 use the PK of T_2 as foreign key. Therefore, both joins are on T_2's PK, and hence, the order of joins is not a matter of importance. We perform either join first, determine the authentic set of PKs of T_2 contributing to the join, and do the other join between this authentic set and the other table. Figure 13a is also dealt with in a similar manner. Since both joins can output an authentic set of (composite) PK values of S2C, the other join can be easily handled using this set and the other table.

Multi-way joins with more than two joins can be divided into a set of two-way joins, and the above-mentioned categories can be used to determine the order in which these joins should be performed to generate efficient proofs. In cases where the order is not important, the DBAS can use the table sizes and database optimization techniques to estimate the result size, and select the one with small expected size [12, 24, 17].

5.5 Special joins

Equijoin is defined to be the join in which the operator is equality [24, 8]. The *non-equijoin*, which is also called the *band join*, is defined as the join operation that the operator is not equality [24]; i.e., the values of one of the join columns fall within a *band* of values of the other column [8].

Equijoin of the form $\mathbf{T_1.C_i = T_2.C_j \mp n, n \in \mathbb{N}}$. This is a special case of the equijoin. We treat $T_1.C_i = T_2.C_j \mp n$ as matching (instead of $T_1.C_i = T_2.C_j$) and apply the equijoin algorithm. Proof generation for the query $T_1.a + 1 = T_2.a = T3.a - 2$ on Figure 12 works as follows: The algorithm starts with the smallest values 1, 1, 5, respectively. Since the relation $T_1.a + 1 = T_2.a = T3.a - 2$ does not hold, the greatest number according to the relation, which is 5, is used to find the expected node on the two other ADSs. But since we are not looking for 5 in the other tables, we need to adjust our parameter. 5 would be matched with 5-2=3 in T_2, so we run FindNext(3) on T_2. It will also be matched with 5-2-1=2 in T_1, so we run FindNext(2) on T_1. Using our join proof generation algorithm this way generates *vo*= 'h($-\infty$),h(1),**2**;h2,**3**;h($-\infty$),**5** : **4**;**5**;**7** : 6,9;9;9,17 : 15;14,19;17 : h(16),18;19;19,h($+\infty$) : h5,h($+\infty$);h(20),h4,h($+\infty$);', indicating that the query $T_1.a + 1 = T_2.a = T3.a - 2$ executed on Figure 12 has two matchings: (2, 3, 5) and (4, 5, 7).

Non-equijoin. The general form of a non-equijoin query is $|T_1.C_i - T_2.C_j| < n, n \in \mathbb{N}$. A simple proof generation algorithm for this join is to select the HADS of the table with smaller number of records, and for each node of this HADS, perform an authenticated range query on the other HADS. But, this is less efficient regarding computation and communication, due to the many intersections among the sets the authenticated range queries return.

We modify our join algorithm slightly to support the non-equijoin more efficiently, where each HADS is traversed only once. We select the smaller HADS, and for each record in this HADS, compute the matching records on the other HADS. Since one record may correspond to many records, we need to include the boundary records (remember

we are using multi-proof ADSs). To prevent the values to be processed multiple times, we perform as follows:

- If the left boundary of the current record is greater than the right boundary of the previous record, then it is necessary, and hence we add the required intermediate information, the left boundary, the matching records, and the right boundary into the *vo*. Since the left boundary record, and hence, all matching records of the current record reside after the right boundary record of the previous record, the server does not need to go backward after completion of processing of the previous record. He jumps to the left boundary record of the current record, while adding the required intermediate information for reconstructing the HADS.

- If the left boundary of the current record is less than or equal to the right boundary of the previous record, there may be common matching records. Due to the security of the HADS that prevents a malicious server from adding or deleting matching records, no need to go backward. Such a malicious server can try to delete some matching records and put the corresponding intermediate information to pass the client verification. But, such intermediate information can only appear between two sets of matching records (not inside a set of matching records). Therefore, we go on from the current position in the second HADS, and add into the *vo* the remaining matching records until the right boundary record.

Algorithm 5.2: `NEQJoinCertify`, run by the server.

Input: the *band* of the query: n, and two HADSs of the two joining tables.
Output: the verification object: *vo*

1   Select the smaller table. Call it R, and the other S.
2   Traverse both HADSs to reach the leftmost node at the first level.
3   $vo = \{\}$
4   $RNode = R.CurrentNode$
5   $SLeft = SRight = S.CurrentNode$
6   **while** $RNode.val \neq +\infty$ AND $SRight.val \neq +\infty$ **do**
7   **if** $|RNode.val - SRight.val| > n$ **then**
8         $SLeft = $ `FindLeftBoundary`$(RNode.val)$
9         $SRight = $ `FindRightBoundary`$(RNode.val)$
10   Add $SLeft.val$, all records until $SRight.val$, and $SRight.val$ into the *vo*
11   **else**
12        $SRight = $ `FindRightBoundary`$(RNode.val)$
13   Add all records until $SRight.val$, and $SRight.val$ into the *vo*
14     $RNode = R.NextNode$
15 **return** *vo*

Therefore, in both case, the server traverses both HADSs once. The same facts hold for the client during the verification. She only checks the given boundary records and reconstructs the HADS without the need to going backward. This is an important observation that simplifies the client and server computation. This process is shown in Algorithm 5.2. Two helper functions `FindLeftBoundary()` and `FindRightBoundary()` with obvious functionality are used during the algorithm.
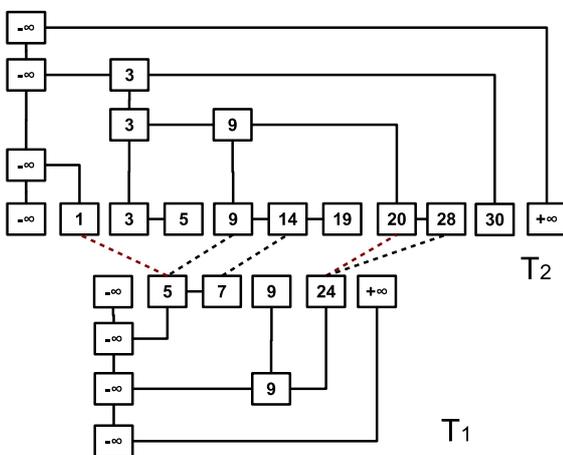
Figure 14: Non-equijoin proof generation for $|T_1.a - T_2.a| < 3$.

Assume that we want to execute the non-equijoin query $|T_1.a - T_2.a| < 3$ on the example given in Figure 14. We start by T_1 (who has fewer records) and for each record, find the set of matching records on T_2. For the first record, 5, `FindLeftBoundary(5)` and `FindRightBoundary(5)` return the boundary records 1 ($|5 - 1| >= 3$) and 9 ($|5 - 9| >= 3$), respectively. These boundary records together with the matching records in between, are added into the *vo*: *vo*=‘**5**;1,**3,5**,9’. The next record is 7 for which $|7 - 9| < 3$, hence, its left boundary record is already in the proof, and we only need to find the right boundary record which is 14. Since all matching records are already in the *vo*, we add only 14, i.e., *vo*=‘**5**;1,**3,5**,9 : **7**;14’. Nothing is inserted for the next record, 9, since $|9 - 14| >= 3$, meaning that even the right boundary is already in the proof, leading to *vo*=‘**5**;1,**3**1**,5**,9 : **7**;14 : 9;-’. Regarding 24, since $|24 - 14| > 3$, we call `FindLeftBoundary(24)` to find the left boundary record, which adds $h(19)$ as the intermediate information into the *vo*, and returns

20. `FindLeftBoundary(24)` returns 28. There are no matching records in between, therefore, only the boundary records are added into the vo=‘5;1,**3**,**5**,9 : **7**;14 : 9;- : 24;h(19),20,28’. Since the end of T_1 is reached, we add $h(30)$ as the intermediate information of T_2. Finally, vo=‘5;1,**3**,**5**,9 : **7**;14 : 9;- : 24;h(19),20,28 : h(+∞);h(30),h(+∞)’ is returned as the proof.

    The proof verification is also accomplished in a similar way as shown in Algorithms 5.3.

Algorithm 5.3: `NEQJoinVeriify`, run by the client.

    Input: the verification object: vo, the difference: n.
    Output: 0 for acceptance, -1 for rejection.

1    $ADS_R = ADS_S = \{\}$;
2    $RoundProof = vo.\texttt{GetRoundProof}()$; //Get the proof until the next ‘:’.
      //First, interpret the proof and see if non-equijoin condition holds for all records.
3   **while** $RoundProof$ **do**
4       $RNode = RoundProof.\texttt{GetLeftPart}()$; //In each round, only one record of R exists.
5       $ADS_R.\texttt{Add}(RNode)$; //All R records are stored here for later ADS verification.
6       $SProof = RoundProof.\texttt{GetRightPart}()$; //The corresponding proof of S.
7       $SNode = SProof.FirstNode$;
8   **if** $SNode == NULL$ **then**
9   *continue*;//No matching, go to the next round.
        //First, add all intermediate information into the ADS, if there is any.
10   **while** $SNode$ *is intermediate* **do**
11        $ADS_S.\texttt{Add}(SNode)$;
12        $SNode = SProof.NextNode$;
        //Now, check and add to the ADS the left boundary record, if there is any.
13   **if** $SNode.val < RNode.val$ **then**
14        $ADS_S.\texttt{Add}(SNode)$;
15        $SNode = SProof.NextNode$;
        //Add all matching records into the ADS, if there is any.
16   **while** $|SNode.val - RNode.val| < n$ **do**
17        $ADS_S.\texttt{Add}(SNode)$;
18        $SNode = SProof.NextNode$;
        //Check and add to the ADS the right boundary record, if there is any.
19   **if** $SNode.val > RNode.val$ **then**
20        $ADS_S.\texttt{Add}(SNode)$;
21        $SNode = SProof.NextNode$;
        //Error: if there are remaining nodes.
22   **if** $SNode \neq NULL$ **then**
23   **return** -1; //Error occured.
24      $RoundProof = vo.\texttt{GetRoundProof}()$;
      //Now, all verified matching nodes are in ADS_R and ADS_S, verify them.
25   **if** $!ADS_R.Verify()$ *OR* $!ADS_S.Verify()$ **then**
26   **return** -1; //Error occurred.
27   **return** 0; //No error.

---

6 Analysis

6.1 Security

Theorem 6.1 (Security of ADS) *The ADS is secure according to Definition A.4.*

Proof 6.1 *This theorem is proved for different schemes separately by different researchers. Merkle [23] showed the security of Merkle hash tree, Papamanthou et al. [36] did the job for the authenticated hash table, Goodrich et al. [14] proved security of the RSA one-way accumulator [1] based ADS, Noar and Nissim [27] showed security of the 2-3 tree, and Papamanthou and Tamassia [35] proved security of the ADSs constructed using authenticated skip list or red black tree.*

Theorem 6.2 (Security of HADS) *Our HADS construction is secure according to Definition A.4 (employing HADS algorithm names) if the underlying ADSs are secure.*

Proof 6.2 *We reduce security of the HADS scheme to the security of the underlying ADSs. If a PPT adversary $\mathcal{A}$ wins the HADS security game with non-negligible probability, we can use it to construct a PPT algorithm $\mathcal{B}$ who breaks the security of at least one of the ADS schemes used, with non-negligible probability. $\mathcal{B}$ acts as the server in the ADS game played with the ADS challenger $\mathcal{C}$, and simultaneously, $\mathcal{B}$ plays the role of the challenger in the HADS game with the adversary $\mathcal{A}$. He receives the public key of an ADS from $\mathcal{C}$, and himself produces $n-1$ pairs of ADS public and private keys. Then, he puts the received key in random i^{th} position, and sends the n public keys as the public key of an n-level HADS to $\mathcal{A}$. During the setup phase, $\mathcal{B}$ builds a local copy of the HADS for herself. Note that this is invisible to the adversary $\mathcal{A}$, and thus will not affect his behavior. After the setup phase, $\mathcal{A}$ selects a command, generates the answer and proof for the command, and sends them to $\mathcal{B}$. For the adversary to win, the answer must be different from the real answer in at least one location, with its verifying sub-proof. $\mathcal{B}$ can find it since she maintains a local copy. When $\mathcal{B}$ receives them, she selects the related command, answer and proof parts for the ith position, and forwards them to $\mathcal{C}$. If the guess of i was correct, then $\mathcal{B}$ would succeed. If $\mathcal{A}$ passes the verification with non-negligible probability p, then $\mathcal{B}$ passes the ADS verification with probability greater than or equal to p/n (breaking the ADS security with non-negligible probability, since n, the number of HADS levels, is polynomial in the security parameter).*

Since we employ secure ADSs, p/n must be negligible, which implies that p is negligible, and hence, the adversary $\mathcal{A}$ has negligible probability of winning the HADS game. Therefore, if the underlying ADSs are secure, then the HADS scheme is secure.

Theorem 6.3 (Security of the ODB scheme) *Our proposed ODB scheme is secure according to Definition 4.3, provided that the underlying HADS scheme is secure.*

Proof 6.3 *We reduce security of the ODB scheme to the security of underlying HADSs. If a PPT adversary $\mathcal{A}$ wins the ODB security game with non-negligible probability, we can use it to construct a PPT algorithm $\mathcal{B}$ who breaks the security of HADS scheme with non-negligible probability. $\mathcal{B}$ acts as the server in the HADS game played with the HADS challenger $\mathcal{C}$, and simultaneously, $\mathcal{B}$ plays the role of the challenger in the ODB game with the adversary $\mathcal{A}$. He receives the public key of an HADS from $\mathcal{C}$, and relays it to $\mathcal{A}$ (note that all HADSs built for each searchable column will use the same key). During the setup phase, $\mathcal{B}$ builds a local database for herself (which does not change the adversary's view). After the setup phase, $\mathcal{A}$ selects a query, generates the answer and proof for the query, and sends them to $\mathcal{B}$. For the adversary to win, his answer must be different from the real answer on at least one location, but with a verifying proof. On receipt, $\mathcal{B}$ selects the related command, answer and proof parts for the answer that differs from the real answer (she can find it since she maintains a local copy), and forwards them to $\mathcal{C}$. If $\mathcal{A}$ passes the ODB verification with non-negligible probability p, then $\mathcal{B}$ can also pass the HADS verification (i.e., break HADS security) with non-negligible probability p.*

Since we employ a secure HADS, p must be negligible, which implies that the adversary has negligible probability of breaking ODB. Therefore, our ODB scheme is secure (and provides the required properties for an outsourced database: correctness, completeness, and freshness), if the underlying HADS is secure.

Note that this proof is not specific to our two-level construction. If one uses a four-level construction, as we talked in Section 4.3.1, then $\mathcal{B}$ plays the HADS game with a four-level HADS challenger. In general, for an n-level ODB construction, $\mathcal{B}$ should play the game with an n-level HADS challenger, in the same manner as described above. The proof or the probabilities will not be affected by this change.

6.2 Performance

Setup. To evaluate our ODB scheme, we implemented a DBAS prototype using the efficient two-level HADS construction, which uses FlexList [11] at both levels, in C++ using Cashlib library [22]. All experiments were performed on a 2.5GHz machine with 4 cores (but the test running on a single core), with 4GB RAM and Ubuntu 11.10 operating system. The performance numbers are averages of 50 runs.

Our DBAS application is deployed on the same machine where the DBMS resides, and stores the security information of our database. Each dynamic query (`Insert`,`Update`, `Delete`,`Drop`,`Alter`,...) affects this part as well, but the query should be converted into the (key, value)-based format. For example, the query `SELECT * FROM Student WHERE major in('CE','CS') and BCity='Istanbul'` is converted to (Student,{(major, {CE,CS}),(BCity,{Istanbul})}). We did not implement an automatic converter, but it should not affect the timing since its overhead is much smaller than the proofs.

We use a database containing three tables: `Student` and `Course` tables, each with 10^5 randomly-generated records, and `S2C` table storing the courses taken by students, with 10^6 randomly-generated records. There are two

scenarios: each registered student has taken 10 courses in the first scenario, and 100 courses in the second scenario, on average. (In the second scenario, not all students are taking courses since we only have 10^6 S2C records in total.) This means that a distinct StdId is used as a foreign key in S2C 10 times in the first scenario, and 100 times in the second scenario, on average.

Given this database, we observe the system behavior (proof generation time and proof size) for different query types. Since in our scheme proofs are generated using only the hashes of the values of the column(s) forming the clause (not the whole records), **the proof size is independent of the record size**. Our scheme enhances the efficiency by reducing the required computation and proof size, confirmed by experimental results:

- The proofs are generated using only values of the required columns, and these values already exist in the DBMS answer to the query.
- The concept of PK-sets divides a large ADS into small ADSs in a hierarchy. Hence, the proof size and the computation time decrease as well.
- Using the PK-sets, there is a one-to-one correspondence for the matching records, and there is no need for boundary records. This is a very important property for computing boolean operations and join proofs easily.

Comparison to previous work. Different methods were proposed for making the duplicate values unique [6, 32, 19, 31] to store them in a regular ADS. Since they produce the same number of distinct values (= number of records in the table), *their ADS sizes are the same*, leading to *similar performances*. For the sake of comparison, we concatenate each duplicate value with a replica number as in [32], and build a regular ADS to compare our HADS against it. This is referred to as 'previous work' in our figures. (Therefore, the 'previous work' in the figures correspond to all these works, if they used the same ADS as us.)

6.2.1 Selection Queries on One Table

We consider three cases: **One-clause queries**. We investigate the case that the clause is on a non-PK column (e.g., SELECT * FROM Student WHERE major='CE'). Since the number of distinct values in the non-PK column is less than that of the PK column, the first-level ADS of the HADS storing a non-PK column is smaller than the (single-level) ADS storing the same column in the way of the previous work. (We do not count the second-level ADSs in the one-clause case, since they are included in whole, without any computation to find and select some.) This corresponds to the fact that, some values are repeated on non-PK columns, whereas the PK column contains only unique values. The proof generation time and proof size for a non-PK clause using HADS are thus expected to be smaller compared to the previous work. The Figures 15a and 15b show ≈5x smaller proofs, and ≈3x faster proof generations, compared to the previous work. There is a ≈10% efficiency gain even with range queries.

Two-clause queries. There are two cases: the query has either one PK and one non-PK clause (e.g., SELECT * FROM Student WHERE StdID>105 AND major='CE'), or two non-PK clauses (e.g., SELECT * FROM Student WHERE BCity='Istanbul' AND major='CE'). In the HADS of the non-PK columns, all values of the second-level ADSs are included in the result (without further computation), therefore, the dominant factors are the proof generation time and proof size of the first-level ADSs. We apply each clause on its own HADS and generate two proofs to put in the verification object. Figures 16a and 16b show the proof generation time and proof size for two-clause queries. We observe ≈2x smaller proofs and ≈1.5x faster proof generations using HADS, compared to previous work, for the case with one PK and one non-PK clauses. For the case with two non-PK clauses, the proof is ≈5x smaller in size, and ≈3.5x faster in generation time, compared to previous work.

Multi-clause queries. There are more than two clauses in this case, and the two-clause case is a special case of this one. Again, we can separate this case into two cases depending on whether one of the clauses is on the PK column or none of them are. The server asks each HADS sequentially to give its first-level proof. The total proof generation time and proof size of the server is summation of the corresponding values taken by all HADSs. We are not presenting any figures for this, but based on the results presented above, we expect similar gains. Indeed, the gains would be even greater if all clauses are on non-PK columns.

Communication overhead. Another important factor is the overhead of our scheme on the communication, i.e., how much does the proof increase the traffic. As the proof size is independent from the record size, for tables with small record size (≈1KB), the proof size is about 10-40% compared to the result size. As a real example, we used the Student table from Koç University database that stores (student ID, name, address, phone, email, standing, department, advisor, photo) for each student. The records of this table are between 5 and 20KB in size, where the

photo size is dominant. Using the HADS for proof generation imposes only $1-4\%$ communication overhead. The results are shown in Figure 17a. Compared to similar algorithms such as [31] that require $O(\log N + t)$ cost for a query result of size t, using range queries, the cost of our algorithm is $O(\log |C_i| + t)$.

Client computation. We observed that the HADS does *not* increase the client verification time compared to the previous work. The reason is that while the server just puts what a second-level ADS stores into the *vo*, the client has to reconstruct the second-level ADS together with the proof path in the first-level ADS. The computation at the second-level (first-level) ADS of our HADS is very similar to that of the previous schemes at the lower (upper) part of their ADSs. Therefore, the total client computation using our HADS and previous ADSs are very close. This is illustrated in Figure 17b for one-clause queries.

6.2.2 Join Queries

We consider two cases. In the first case, the *key-based join*, the stdID column of the Student table is referred to in the S2C table as a foreign key (e.g., SELECT * FROM Student,S2C WHERE Student.StdID=S2C.StdID), while in the second case, the *general join*, we add two unrelated columns of the same type to Student and Course for this join (e.g., SELECT * FROM Student,Course WHERE Student.TempCol1=Course.TempCol2).

In the **key-based join** scenario, we consider two cases. In the first case, each student has chosen 10 courses, therefore, the first-level ADS stores the students, and for each one, a second-level ADS containing 10 elements stores the selected courses. The first-level ADS contains all 10^4 student IDs. In the second case, each student has taken 100 courses, therefore, a second-level ADS containing 100 courses is linked to each first-level ADS. The first-level ADS in this case is smaller, containing 10^3 records. The experimental results are shown in Figures 18a and 18b. The figures show ≈ 2.5x enhancement for both proof size and proof generation time in 10-course case. There are ≈ 4x smaller proofs and ≈ 6x faster proof generations in 100-course case, compared to the previous work.

We observe a similar behaviour for the **general join** scenario, where each value in temporary columns TempCol1 and TempCol2 is duplicated about 10 or 100 times, similar to our main scenario. Figures 19a and 19b show the experimental results. The proof sizes are reduced ≈ 3x and ≈ 4x in 10-element and 100-element cases, respectively. The proof generation times are decreased ≈ 2x and ≈ 5x in 10-element and 100-element cases, respectively.

Asymptotic complexity. Moreover, the cost of the approach proposed by Li *et al.* [18] for joining two tables T_1 and T_2 of approximate size N is $O(N \log N)$, while that of ours is $O(N+N) = O(N)$. Compared to [42] who has the same asymptotic cost $O(N)$, our HADS generates more efficient proofs as it does not use the boundary records for the matching records, in addition to the fact that it operates on smaller ADSs. Assume that $\alpha |C_i|, 0 \leq \alpha \leq 1$, records of a column have matching on the other table. The cost of our join algorithms using HADS is $\alpha |C_i| * N / |C_i| + (1-\alpha)|C_i| = \alpha N + (1-\alpha)|C_i|$, which means that the cost is close to $O(|C_i|)$ when α is close to zero, and approaches $O(N)$ as α approaches one; i.e., ours drops in the worst case to the algorithm of [42].

Communication overhead. In our scheme, the proof size does not depend on the record size. This is an important difference between ours and the join algorithms proposed by Yang *et al.* [42], where the proof size increases with the record size.

(a) Proof size. (b) Proof generation time.
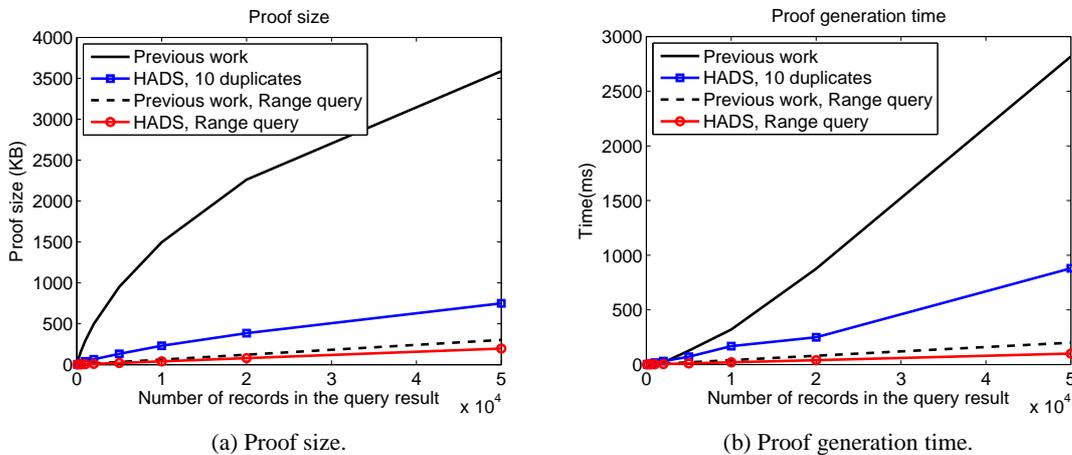
Figure 15: Proof generation time and proof size for one-clause queries.
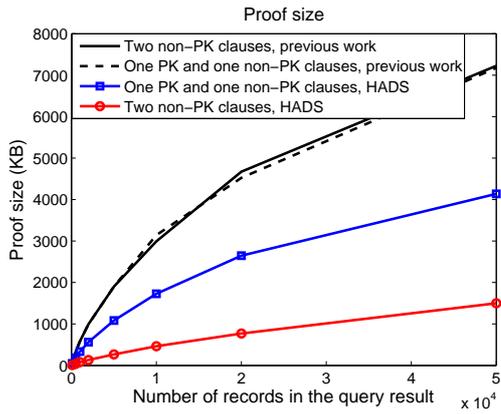
(a) Proof size.
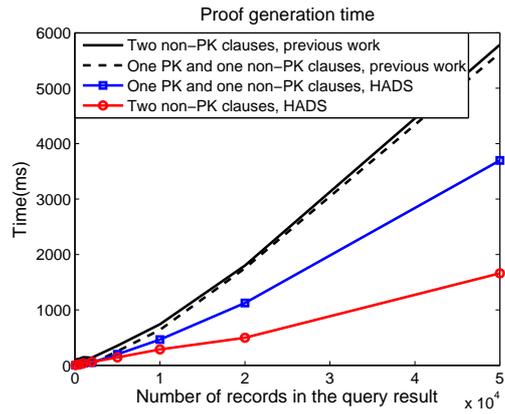
(b) Proof generation time.

Figure 16: Proof generation time and proof size for queries with two clauses. (Note that the values on the x-axis are upper bounds on the query result size, since if the two clauses are connected by 'AND', or if they are connected by 'OR' but have many common PKs in the PK-set, then the actual result size of the query will be less than the values shown in the diagrams.)
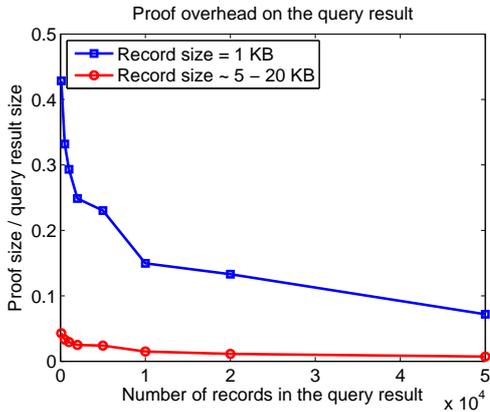
(a) Proof overhead.
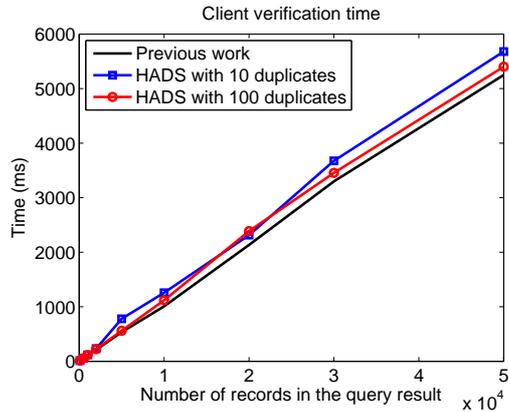
(b) Client verification time.

Figure 17: Proof overhead and client verification time.
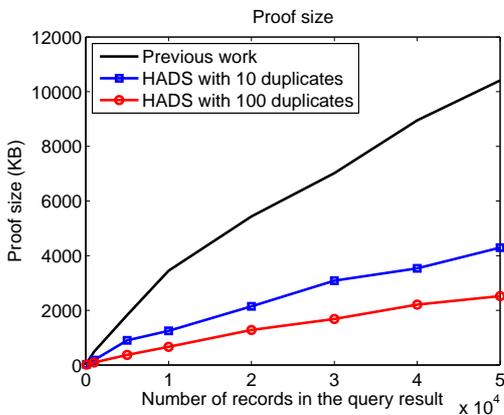
(a) Proof size.
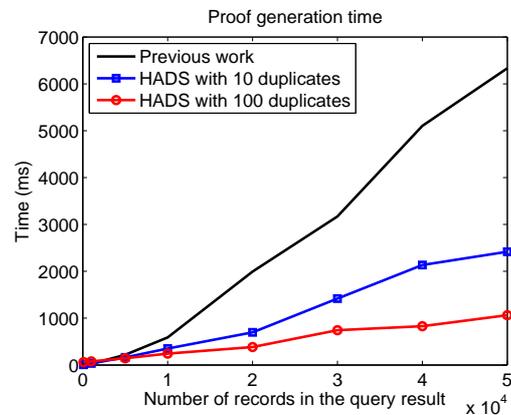
(b) Proof generation time.

Figure 18: Proof generation time and proof size (key-based join).

(a) Proof size.

(b) Proof generation time.
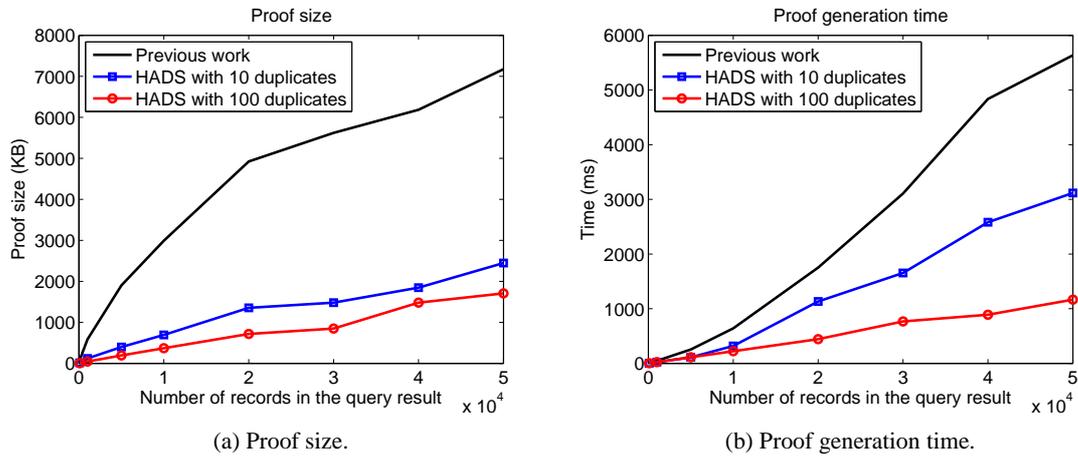
Figure 19: Proof generation time and proof size (general join).

7 Conclusion

In this paper, we presented a hierarchical ADS for storing the security information required for proof generation in outsourced databases. The HADS extends the ADS to support storing duplicate values, and generating comparable and combinable proofs efficiently (useful for boolean operators and joins). We employed the HADS to construct outsourced databases with proofs for query result authenticity, including completeness, correctness, and freshness guarantees. We proved these properties using a new unified security definition we provided.

Our outsourced database construction can provably handle selection queries with one or multiple clauses connected by 'OR' or 'AND' connectors in any manner, join queries including equijoins, non-equijoins, band joins, joins on non-PK columns, joins over more than two tables, and combinations of selection and join queries. Besides, with reduced use of boundary records, we can easily support clauses formed using the SQL 'IN' operator. This allows us to present efficient proofs for a wide range of database queries. We only support the sequential proof generation, and leave the concurrent version as future work.

We have presented performance gains due to our solution over the previous work where regular (one-level) ADSs are used. Our solution achieves $\approx$3x smaller proofs in size and $\approx$5x faster proof generations time when HADS is used for queries with one clause. Moreover, for join queries we observed $\approx$4x enhancement in proof size and $\approx$5x enhancement in proof generation time using HADS, when each foreign key is repeated 100 times, on average. With reasonable record sizes, e.g., $5 - 20KB$ in our Koç University database's Student table, the communication overhead is $\approx$4% compared to the result size, becoming even smaller with larger record sizes. Thus, we believe outsourced databases are finally ready for prime-time.

References

[1] J. Benaloh and M. De Mare. One-way accumulators: A decentralized alternative to digital signatures. In *EUROCRYPT'93*, pages 274–285. Springer, 1994.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[3] B. Carbunar and R. Sion. Toward private joins on outsourced data. *IEEE Transactions on Knowledge and Data Engineering*, 24(9):1699–1710, 2012.

[4] J. Celko. *Joe Celko's Trees and hierarchies in SQL for smarties*. Morgan Kaufmann, Washington, 2004.

[5] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Integrity for join queries in the cloud. 2013.

[6] P. Devanbu, M. Gertz, C. Martel, and S. Stubblebine. Authentic third-party data publication. *Data and Application Security*, pages 101–112, 2002.

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

[8] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equijoin algorithms. In *VLDB*, pages 443–452. Morgan Kaufmann Publishers Inc., 1991.

[9] G. Di Battista and B. Palazzi. Authenticated relational tables and authenticated skip lists. *Data and Applications Security XXI*, pages 31–46, 2007.

[10] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *CCS'09*, pages 213–222. ACM, 2009.

[11] E. Esiner, A. Küpçü, and O. Özkasap. Analysis and optimization on flexdpdp: A practical solution for dynamic provable data possession. 2014.

[12] C. Galindo-Legaria and A. Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Transactions on Database Systems (TODS)*, 22(1):43–74, 1997.

[13] M. Goodrich and R. Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing. *US Patent App*, 10(416,015), 2000.

[14] M. Goodrich, R. Tamassia, and J. Hasić. An efficient dynamic and distributed cryptographic accumulator. *Information Security*, pages 372–388, 2002.

[15] M. T. Goodrich, R. Tamassia, and N. Triandopoulos. Super-efficient verification of dynamic outsourced databases. *CT-RSA*, pages 407–424, 2008.

[16] M. T. Goodrich, R. Tamassia, and N. Triandopoulos. Efficient authenticated data structures for graph connectivity and geometric search problems. *Algorithmica*, 60(3):505–552, 2011.

[17] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, pages 73–169, 1993.

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

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

[20] S. Ma, B. Yang, K. Li, and F. Xia. A privacy-preserving join on outsourced database. *Information Security*, pages 278–292, 2011.

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

[22] S. Meiklejohn, C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya. Zkpdl: A language-based system for efficient zero-knowledge proofs and electronic cash. In *USENIX Security Symposium*, 2010.

[23] R. Merkle. A certified digital signature. In *CRYPTO'89*, pages 218–238. Springer, 1990.

[24] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Computing Surveys (CSUR)*, 24(1):63–113, 1992.

[25] E. Mykletun, M. Narasimha, and G. Tsudik. Providing authentication and integrity in outsourced databases using merkle hash trees. *UCI-SCONCE Technical Report*, 2003.

[26] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. In *NDSS*, 2004.

[27] M. Naor and K. Nissim. Certificate revocation and certificate update. *Selected Areas in Communications, IEEE Journal on*, 18(4):561–570, 2000.

[28] M. Narasimha and G. Tsudik. Authentication of outsourced databases using signature aggregation and chaining. In *Database Systems for Advanced Applications*, pages 420–436. Springer, 2006.

[29] G. Nuckolls. Verified query results from hybrid authentication trees. *Data and App. Sec.*, 2005.

[30] B. Palazzi. *Outsourced Storage Services: Authentication and Security Visualization*. PhD thesis, Roma Tre University, 2009.

[31] B. Palazzi, M. Pizzonia, and S. Pucacco. Query racing: fast completeness certification of query results. In *Data and Applications Security and Privacy XXIV*, pages 177–192. Springer, 2010.

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

[33] H. Pang and K.-L. Tan. Authenticating query results in edge computing. In *International Conference on Data Engineering*, pages 560–571. IEEE, 2004.

[34] H. Pang, J. Zhang, and K. Mouratidis. Scalable verification for outsourced dynamic databases. *VLDB*, 2(1):802–

813, 2009.

[35] C. Papamanthou and R. Tamassia. Time and space efficient algorithms for two-party authenticated data structures. *Information and Communications Security*, pages 1–15, 2007.

[36] C. Papamanthou, R. Tamassia, and N. Triandopolos. Authenticated hash tables. In *ACM CCS'08*, pages 437–448.

[37] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33:668–676, 1990.

[38] V. Raman, L. Qiao, W. Han, I. Narang, Y. Chen, K. Yang, and F. Ling. Lazy, adaptive rid-list intersection, and its application to index anding. In *ACM SIGMOD*, volume 11-14, pages 773–784, 2007.

[39] R. Tamassia. Authenticated data structures. *Algorithms-ESA 2003*, pages 2–5, 2003.

[40] R. Tamassia and N. Triandopoulos. On the cost of authenticated data structures. Technical report, Center for Geometric Computing, Brown University, 2003.

[41] J. Wang and X. Du. Skip list based authenticated data structure in das paradigm. In *GCC'09*. IEEE, 2009.

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

A   ADS Definitions

Definition A.1 *ADS scheme consists of three polynomial-time algorithms (`KeyGen`, `Certify`,`Verify`)[35]:*

KeyGen$(1^k) \rightarrow (\mathbf{sk}, \mathbf{pk})$ *is a probabilistic algorithm executed by the client to generate a private and public key pair (sk, pk) given the security parameter k. The client then shares the public key pk with the server.*

Certify$(\mathbf{pk}, \mathbf{cmd}) \rightarrow (\mathbf{ans}, \pi)$ *is run by the server to respond to a command issued by the client. The public key pk and the command cmd are given as input. If cmd is a query command, it outputs a verification proof π that enables the client to verify the authenticity of the answer ans. If cmd is a modification command, then the ans is null, and π is a consistency proof that enables the client to update her local metadata.*

Verify$(\mathbf{sk}, \mathbf{pk}, \mathbf{cmd}, \mathbf{ans}, \pi, \mathbf{st}) \rightarrow (\{\texttt{accept}, \texttt{reject}\}, \mathbf{st}')$ *is run by the client upon receipt of a response. The public and private keys (pk, sk), the answer ans, the proof π, and the client's current metadata st are given as input. It outputs an `accept` or `reject` based on the result of the verification. Moreover, if cmd was a modification command and the proof is accepted, the client updates her metadata accordingly (to st').*

Definition A.2 *Correctness of ADS. For all valid proofs π and answers ans returned by the server in response to a command issued by the client, the verify algorithm accepts with overwhelming probability.*

Definition A.3 *The ADS security game is played between the challenger who acts as the client and the adversary who plays the role of the server:*

Key generation *The challenger runs KeyGen(1^k) to generate the private and public key pair (sk, pk), and sends the public key pk to the adversary.*

Setup *The adversary specifies a command cmd, and sends it together with an answer ans and proof π to the challenger. The challenger runs the algorithm Verify, and notifies the adversary about the result. If the command was a modification command, and the proof is accepted, then the challenger applies the changes on her local metadata accordingly. The adversary can repeat this interaction polynomially-many times. Call the latest version of the HADS, constructed using all the commands whose proofs verified, D.*

Challenge *The adversary specifies a command cmd, an answer ans', and a proof π', and sends them all to the challenger. He wins if the answer ans' is different from the result set of running cmd on D, and cmd, ans', π' are verified as accepted by the challenger.*

Definition A.4 *Security of ADS. We say that the ADS is secure if no PPT adversary can win the ADS security game with non-negligible probability.*