Published version available at Elsevier: https://doi.org/10.1016/j.jnca.2018.04.006

Verifiable Database Outsourcing Supporting Join

Mohammad Etemad[a], Alptekin Küpçü[a]

[a]*Koç University, İstanbul, Turkey*

Abstract

In an outsourced database scheme, the data owner delegates the data management tasks to a remote service provider who is supposed to answer owner's queries 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 query answers 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: *correctness*, *completeness*, and *freshness*. The performance evaluation based on our prototype implementation confirms the efficiency of our proposed scheme, showing ∼3x smaller proofs and ∼5x improvement in proof generation time compared to previous works (Devanbu *et al.* 2002, Pang *et al.* 2005, Li *et al.* 2010, Palazzi *et al.* 2010).

Keywords: Outsourced databases, Hierarchical authenticated data structures

1. Introduction

Popularity and pervasiveness of computer systems and networks have led to production of huge amount of data in organizations and companies. Data needs protection, and most 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 data about all 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.

The main concern is that the owner loses the direct control over her data and should rely on answers coming from the service provider (who is *not* fully trusted). This necessitates mechanisms giving the data owner (the client) the ability for checking authenticity of the answers to her queries. For this, the answer of a client query (the *result set*) is accompanied with a *verification object* (*vo*). Since the client may be a portable device with limited processing power, the *vo* should be small and efficiently verifiable. It is used to verify whether the query answer is [1, 2, 3, 4, 5, 6]:

- ***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 (sound)***: the result set originates from the data owner, i.e., no unauthorized modification on records.
- ***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.

A small part of our sample database together with the result of the query "`SELECT * FROM Student WHERE stdID > 105`" is shown in Figure 1. We want the completeness, correctness, and freshness properties all holding in the answer, guaranteeing its authenticity. Recall that a primary key (PK) column in a table (e.g., `stdID` in Figure 1a) contains unique values, while non-PK columns (e.g., `Major` and `BCity` in Figure 1a) may contain duplicate values.

Email addresses: metemad@ku.edu.tr (Mohammad Etemad), akupcu@ku.edu.tr (Alptekin Küpçü)

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

(a) Our sample database.

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

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

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

The client issues queries with clauses on some *searchable* columns, and checks authenticity of the answers. The general method is to sort a table by each searchable column and build an authenticated data structure (ADS) on it. Each ADS is later used to generate cryptographic proofs for queries having a clause on the corresponding column.

There is a problem with duplicate values in non-PK searchable columns [7, 8]: a total order on the values of searchable columns is required to build the ADSs, which, together with the fact that duplicate values belong to different records, make building the ADSs complicated. As clarified later in Section 1.2, the existing solutions are not *efficient*. We introduce a *hierarchical ADS* scheme (HADS) for this problem that is also advantageous in proof generation for multi-clause and multi-table (join) queries. The HADS can be stored in the same database [9] or separately. It does not need to be tied to a specific database and can be changed without affecting the proof system.

The rationale behind our work is to relate everything to the PKs. As the unique identifiers of records in a database, the PKs 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, rendering them inefficient for multi-clause and multi-table queries. We also support dynamic databases where the data owner issues modification queries (`Insert,Delete,Update`), in a provable manner. Overall, we believe our HADS may also be of independent interest, applicable to other scenarios.

Our contributions are summarized as follows:

- We give a **unified security definition** for an outsourced database scheme (ODB) covering *completeness*, *correctness*, and *freshness* simultaneously.
- We formalize the **hierarchical ADS** scheme and prove its security based on the security of the underlying ADSs.
- We build a provably-secure ODB using HADS that supports efficient proof generation for not only single-clause but also **multi-clause** queries.
- Our scheme supports tables with **composite keys** as well, for the first time.
- Our ODB construction efficiently handles proofs for **join** queries, multi-table joins, non-equijoins, and queries containing *both join and selection*.
- We handle proofs on columns containing duplicate values more efficiently. While being generated about **5x** faster, our proofs are about **3x** smaller in size, compared to the previous works.
- Our ODB provides efficient proofs for almost all query types. We achieve **4% communication overhead** compared to the actual result size, using our Koç University database.

1.1. Related Work

ADS-based Approaches. Authenticated data structures are used to store authentication information, which is later used to generate authenticity proofs for the query results.

Devanbu *et al.* [10] proposed one of the first schemes using ADS for checking integrity of the outsourced data. Using a Merkle hash tree to store the security information, the scheme supports the projection and simple join operations on static data. Pang and Tan [11] used one or more *verifiable* B-trees (VB-tree) for each table that is an extension of B-tree using the Merkle hash tree. A VB-tree is generated for each searchable column after sorting the table on that column. This method does not support completeness [7], and found insecure for the insecurity of the function used

to compute the signatures [12]. A variant of this method, named MB-tree, is also used in the literature [1, 5, 10, 13]. MB-tree is similar to VB-tree except that a light hash function is used instead of expensive signatures.

Another line of work is using an authenticated skip list to store the required information for verification [3, 14]. It is suitable and efficient enough for this purpose, especially regarding dynamic scenarios. Wang and Du [3] proved that such ADSs provide completeness and soundness for one-dimensional range queries, and multi-dimensional range queries require multiple ADSs. Palazzi [14, 15] built one authenticated skip list for each searchable column in each table. For multi-clause queries, the result of one clause that is finished earlier is returned. This is not efficient since a larger set than the query result is transferred. The problem is that the results of these clauses cannot be compared and combined together. Wang *et al.* [16] used a tuple-based hash tree to support correctness and completeness for static outsourced databases. Chen *et al.* [17] formalized the notion of verifiable databases with incremental updates for the cases that a large number of small modifications occur. Xu *et al.* [18] defined the *improved authenticated skip list* and used it to propose a query authentication mechanism for outsourced databases.

Authenticated range query is a method for supporting the completeness, i.e., there are no extra or missing records in the answer [1, 2, 10, 12]. The server finds the *contiguous* nodes storing the result set of the query, as well as the left and right *boundary records* immediately surrounding the result set. (The underlying ADS needs to be *ordered*.) It then computes the ADS proofs of the boundary records and sends it together with the result set to the client for verification. 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 values [3, 19].

Hierarchical ADSs. Nuckolls [5] 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.* [4] gave a verification method by dividing a tree with n leaves into sub-trees with $\log n$ leaves. 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. None of the previous work formalizes or generalizes such hierarchical ADSs.

Provable join. Li *et al.* [2] proposed the Embedded Merkle B-tree to support authentic join queries. To join two tables `R` and `S`, $\texttt{R} \bowtie_{C_i=C_j} \texttt{S}$, where $C_i \in \texttt{R}$ and $C_j \in \texttt{S}$, they find the smaller table, say `R`, and insert it as a whole into the *vo* along with its proof. For each $v_k \in C_i$, they construct a range query proof (using the other table) 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.* [20] used signature aggregation to propose a scalable query result authentication mechanism for dynamic databases. Their first attempt is similar to [2], with a huge verification object. Their second attempt uses a certified Bloom filter [21] to show that some records has no matching records on the other table.

Join algorithms that use the ADSs for both tables and generate reasonable proofs are proposed by Yang *et al.* [1]. The *Authenticated Indexed Sort-Merge* is an efficient form of previous join algorithms with one ADS [2, 20]. The *Authenticated Indexed Merge* improves the previous one 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 every (mis)match, two boundary records are inserted into the *vo* that is unnecessary.

An integrity-checking mechanism for join queries performed by an *untrusted* computational server working together with some *trusted* storage servers is given in [22]. The client gives the storage servers a query and information on how to inject some fake records 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.

Verifiable computation supports a general set of functionalities over the outsourced data. IntegriDB [23] supports SQL queries such as max, min, count, sum, and avg on an outsourced database. Pinocchio [24] enables verifying general computations outsourced to the cloud. ADSNARK [25] proves the results of computations over authenticated data to the third parties in a privacy-preserving manner. These works also can be utilized in database settings.

Encrypted databases store the data in encrypted format. However, range queries require order-preserving encryption (OPE) [6, 26, 27] to find all matching records. Xiang *et al.* [28] proposed a cloud database model where the computation service providers undertake most of the post-processing and reconstruction burden for database query. Then, they employed secret sharing and tree-based OPE to give a database outsourcing scheme. Liu *et al.* [6] used programmable OPE for outsourced databases. They discussed the ciphertext-only attacks and statistical attacks for such schemes, and how to mitigate them.

1.2. Overview of Our Solution

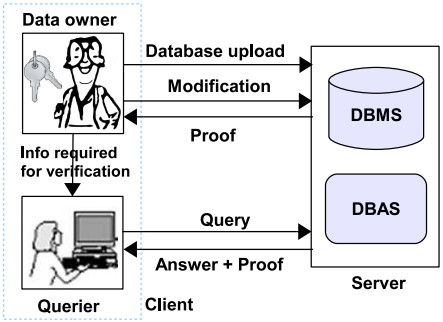


Figure 2: The ODB model.

Model. The outsourced database (ODB) model, as depicted in Figure 2, is composed of three parties: the *data owner*, the *querier*, and the *service provider*. The data owner prepares and uploads the database, and gives the querier(s) the security information for verification. He may perform modifications (`insert`,`delete`,`update`) on the outsourced database.

The service provider (the *server*) has the required equipment (software, hardware, and network resources) for storing and maintaining the database. We do not assume anything about the internal structure of the server, i.e., it may use replication and distribution to increase the performance and availability (e.g., [29, 30]).

The *querier* (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 together as the *client*. We focus on the single-client case.

We decouple the security information from the real data and split the server into two parts: the DBMS (database management system) who stores the client data and responds to the client queries, and the DBAS (database authentication system) who stores the security information as a set of ADSs and HADSs, and generates proofs for the queries. The server relays the received queries to both the DBMS and DBAS, collects and forwards their responses to the client. They can reside both on the same machine, or on different machines. It is also possible to store the DBAS inside a DBMS, employing techniques from [3, 9, 14]. Then, both parts can share the same DBMS or employ separate DBMSs. The DBAS works independently of the underlying DBMS, and any available DBMS can be employed. The focus of this work is to construct an efficient and secure DBAS. The DBMS and DBAS together constitute an ODB.

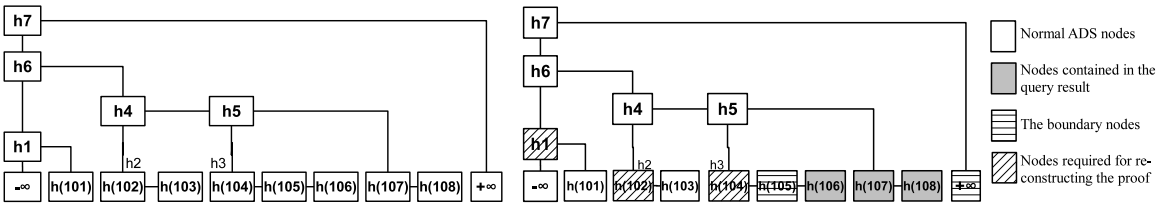
Adversarial model. We assume a malicious adversary who may cheat by attacking the integrity of the outsourced data (doing unauthorized modifications) and giving fake responses to the client queries (running the query processing algorithm incorrectly, or modifying the results), while trying to be undetected.

ADS-based solution. We build a *hierarchical* ADS (HADS) for each searchable column of each table to be able to generate proofs for different queries. Figure 4b visualizes the idea for a database. The topmost ADS, the *database ADS*, stores the table names. For each table, we have a *table ADS*, which stores 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 bottom-most 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 1a, 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 (both in terms of the number of levels, and the types of underlying ADSs used), and hence such a four-level hierarchy is not a requirement, but a sample deployment that makes sense.

Efficient duplicate handling. Columns, such as `Major`, contain duplicate values. Obviously, such duplicates can be made unique, for example, by appending a random perturbation [8], hash of the record [15], or the replica number [7]. 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 without further computation). This results in great performance improvements regarding both communication and computation.

We use multi-proof supporting ADSs (e.g., the FlexList [31]) 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. The proof for the range query clause $a < col_i < b$, indeed, only consists of membership proofs of a and b, and the values matching the clause.

Join. Another advantage of the HADS is an *improved join algorithm*. Since we use ordered HADSs, the items



(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`.

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 containing the smaller value, to a node containing the smallest value 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 in HADSs.

Comparing proofs. Since the HADS ties all values to their respective PKs, a proof shows the authenticity of a set of PKs to the client. Hence, it is possible to compare the results of two or more proofs after verification, which was a common problem among most of the existing solutions [7, 12, 15, 32]. Stated differently, for selection queries containing more than one clause, the server generates and sends back one proof per clause, but only the actual result set. Then, the client verifies each proof separately to compute the authenticated PK-sets and performs set operations among them. The result of this steps is another authenticated PK-set that can be compared against the result of the next proof. The same happens for join queries with more than one clauses, with a proper ordering detailed in Section 5.4, until all clauses are processed. The client verification is done as in selection queries. **Thus, more than two clauses or joins on more than two tables can be handled efficiently, without increasing the result set size.**

2. Preliminaries

Notations used throughout the paper are as follows:

Abbreviation	Description	Abbreviation	Description
N	Number of records in a table	$\|C_i\|$	Number of *distinct* values in a column C_i
vo	Verification object	ODB	Outsourced database
PPT	Probabilistic polynomial time	(H)ADS	(Hierarchical) authenticated data structure
pk	Public key	PK	Primary key (in a database table)
DBMS	Database management system	DBAS	database authentication system

A function $\nu(k) : Z^+ \rightarrow [0,1]$ is called ***negligible*** if $\forall$ *polynomials* p, $\exists$ *constant* k_0 *s.t.* $\forall\, k > k_0,\ \nu(k) < |1/p(k)|$.

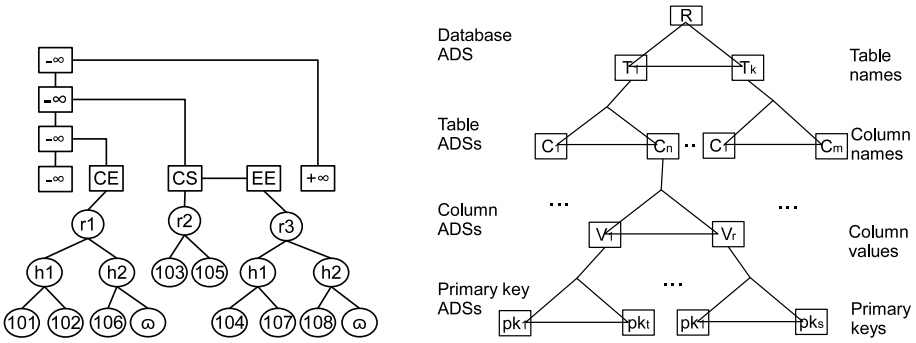
Hash functions take arbitrary-length strings, and output strings of some fixed length. Let $h : \mathcal{K} * \mathcal{M} \rightarrow \mathcal{C}$ be a family of hash functions, whose members are identified by $k \in \mathcal{K}$. A hash function family is collision resistant if $\forall$ *PPT* adversaries $\mathcal{A}, \exists$ a negligible function $\nu()$ *s.t.* $Pr[k \leftarrow \mathcal{K}; (x,x') \leftarrow \mathcal{A}(h,k) : (x' \neq x) \wedge (h_k(x) = h_k(x'))] \leq \nu(k)$.

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 [33, 34, 35]. The client constructs the ADS and uploads it to a server. On receipt of a query, the server sends back a proof, using which the client can verify the answer. 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 and a performance comparison in Appendix C.

Authenticated skip list is an extension of the skip list [36]. The leaves store hashes of data items, and the internal nodes store hash of a function of values of their children. Values on the path from a leaf node up to the root constitute a membership proof. Figure 3a presents an authenticated skip list storing the PK column of the `Student` table, and Figure 3b shows the membership proof for the query `SELECT StdID FROM Student WHERE StdID>105`.

Merkle hash tree [37] 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 [35].

Ordered ADS shows some elements are consecutive (essential for range queries). A total order on the elements to be stored in an ordered ADS is required. Assume x, y and z are *consecutive* elements of a *total order* $(A, <)$ as $x < y < z$, and A is stored at ADS_A. Informally, we say ADS_A is *ordered* if it can prove that $x = pred(y)$ and $z = succ(y)$ for all



(a) A two-level HADS.

(b) A general four-level HADS for a database.

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

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 perfectly suits authenticated range queries.

Multi-proof ADS proves (non-)membership of multiple items in one proof. It does not need to do the job for each item one-by-one, and instead, it generates a proof showing (non-)membership of all items in only one traversal of the ADS. This reduces the server computation, the communication, and the client verification, though not asymptotically. These ADSs suit the authenticated range queries well. FlexList [31] 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 stores the digest of an ADS at a lower level. Therefore, multiple ADSs with different underlying structures can be linked together to form an HADS with multiple levels. The data stored at the bottom-most level are linked to the digest of the topmost ADS through the (data stored at) the internal ADSs. The client stores the digest of the topmost ADS as metadata through which she can verify authenticity of all data stored in the HADS. Figure 4a presents a two-level HADS instantiation (based on Figure 1) using authenticated skip list a the first level and Merkle hash tree at the second level. The node storing a value, e.g., `CS`, ties the value to the digest of its respective second-level ADS, while being tied to the digest of its own ADS. 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 in Appendix A (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. Then, these ADSs are grouped according to some relation, and their digests together with information about their location 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 is 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. 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 (insertion, deletion, and update) as well.

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

3.2. HADS Operations

The `HKeyGen` algorithm generates public and private key pairs for each level and combines all public keys into pk_{HADS} and all private keys into sk_{HADS} (Figure 5).

1: $sk_{HADS} = pk_{HADS} = \{\}$; ▷ Private and public key of the HADS.
2: **for** $i = 1$ *to* n **do** ▷ n is the number of levels of the ADS.
3: $(sk, pk) = ADS_i$.`KeyGen`(1^k); ▷ Ask level i ADS to produce its security keys.
4: $sk_{HADS} = sk_{HADS} \cup sk$;
5: $pk_{HADS} = pk_{HADS} \cup pk$;
6: Return (sk_{HADS}, pk_{HADS}).

Figure 5: `HKeyGen`, run by the client.

1: $P_{own} = P_{child} = \{\}$; ▷ Proof of the current ADS and the combined proof of all children.
2: $\{(ADS', \{(key', value')\})\}$ = `Find`*(key, value)*; ▷ Output is null at leaves.
3: **for** each element e $\in \{(\text{ADS}', (\text{key}', \text{value}'))\}$ **do**
4: $P = e.ADS'$.`HCertify`$(pk, cmd, e.(key', value'))$; ▷ Ask each child compute proof.
5: $P_{child} = P_{child} || P$; ▷ Combine the proofs.
6: $P_{own} =$ `Certify`$(pk, OP, (key, value))$; ▷ Compute this ADS's proof (not hierarchical).
7: Return $P_{child} || P_{own}$; ▷ The proof.

Figure 6: `HCertify`, run by the server.

`HCertify` performs the modification and proof generation on the HADS. Starting at the topmost ADS, it is repeated on all affected ADSs in the hierarchy. Each ADS generates its own proof, *independent* of other ADSs in the same or other levels. Proofs of these ADSs are combined together according to their order in the hierarchy, as presented in Figure 6. For simplicity, we use another algorithm to find the sub-ADSs of a given ADS:

`Find`$(key, value) \rightarrow (\{(ADS', \{(key', value')\})\})$ This is used to find the next level ADS(s) and 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.

The client uses `HVerify` as a recursive process to verify the proof. 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, it is verified against the metadata.

4. Outsourced Database Scheme

An outsourced database requires certification and verification algorithms, similar to an ADS.

Definition 1. *An **outsourced database scheme** consists of three PPT algorithms (*`OKeyGen`, `OCertify`, `OVerify`*):*

- `OKeyGen`$(1^k) \rightarrow (sk, pk)$*: 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`$(pk, cmd) \rightarrow (ans, \pi)$*: 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 shows that the modification is done properly.*
- `OVerify`$(pk, sk, cmd, ans, \pi, st) \rightarrow (\{$`accept`, `reject`$\}, st')$*: is run by the client upon receipt of the answer ans and proof π for a command cmd. It outputs an '*`accept`*' or a '*`reject`*' notification. If the command intended a modification and the verification result is '*`accept`*', the client updates her local metadata accordingly (to st').*

Definition 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', π') is accepted.*

Definition 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.*

The ODB security game covers all the specified guarantees: correctness, completeness, and freshness. 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.

4.1. 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 [7, 8, 10, 15] are not efficient (see Section 7). Our HADS solves the problem efficiently, and easily generates proofs for the answers to multi-dimensional queries (and join queries in Section 5).

The result set of a query with a clause on a non-PK column containing duplicate values will potentially include some duplicate values in that column. To identify these records and compare them with the result set of the other clauses, we can relate each record to its respective (unique) PK.

Definition 4. ***PK-set***. *For each distinct value v_i in a non-PK column C 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, and represented as $PK_{T,C}(v_i)$, i.e., $PK_{T,C}(v_i) = \{k_j \in PK(T) : \exists \text{ record } R \in T \text{ s.t. } k_j \in R.PK \wedge v_i \in R.C\}$. When T and C are clear from the context, we just write $PK(v_i)$.*

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 intended application. The only difference is the type of proof that is generated by the server and verified by the client. This brings the flexibility to support multiple membership schemes, and select the best fitting one.

We construct the DBAS as follows: Since all values in the PK column(s) are distinct, we use a regular (single-level) ordered ADS to store their security information, similar to the ones presented in [3, 14]. 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 unique value is stored exactly once, without any duplicates), 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 trees at the second level storing the corresponding PK-sets.

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.

Authenticated reange queries ensure completeness of the result. Freshness is provided through storing the HADS digest(s) at the client side. For correctness, we store 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 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 C_j of a table `T` with d distinct values $\{v_i\}_{i=1}^{d}$, the HADS is constructed as follows: For each distinct $v_i \in C_j$, 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 $\{(v_i, h(h(v_i)||h(\textit{digest of the respective second-level ADS}))\}$ is constructed.

The client outsources these (H)ADSs along with the database, while keeping their digests locally. Later, upon receipt of a proof and answer, she performs the verification using the information given in the proof and the records in the result set. If all records are used (discussed in Section 4.1.1) and the proofs verify according to the local digests, the client accepts the proof and the answer.

HADS proofs. The HADS membership proofs 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 value in the queried column. This reduces both the proof size (communication) and proof generation time (server computation). But 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. Both asymptotically and based on our performance measurements, those are equivalent tasks.

Consider a table with d distinct values in column C_j, each repeated r times, on average, leading to rd records in total. Using a duplicate elimination mechanism [7, 8, 10, 15], 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, the ADS proof size and proof generation time of query for $v_i \in C_j$ are both $O(\log rd + r) = O(\log r + \log d + r)$, while those of the HADS are both $O(\log d + r)$. The ADS uses a range query with $O(\log rd)$ cost, while the HADS needs to find v_i at the first-level ADS with cost $O(\log d)$. They both then access r consecutive values as the result set. This is further detailed in Section 4.1.1.

4.1.1. Proof Generation

We now 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 internal nodes' values required for verification at the client. 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 *vo*. Due to storing duplicate-eliminated data, the first-level ADS is very small, and all values in the second-level ADS are used without further computation. The proof looks 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', each proof verifies a subset of the received records, and the result set should be the union of all verified records. For 'AND', each proof verifies a superset of records in the result set, and the answer is the intersection of results of individual clauses. The 'AND', 'OR', and 'NOT' operations among clauses are handled as set intersection, union, and complementation on the authentic PK-sets given by the proof. The resulting set of records must be the same as the result set. An important distinction between our HADS and many previous schemes [7, 12, 15, 32] is that *our proofs can be compared and combined together* via simple set operations, since they authenticate PK-sets.

- **One clause on the PK, the other(s) on a non-PK column**: For example, the query is `SELECT * FROM Student WHERE StdID>105 AND Major='CE'`. Since the order of clauses is not important for the proof, we can consider the non-PK clause first, then apply the PK clause on the results. 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 (all) 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 respective PK-set, and sends it to the client inside the *vo*. Each proof proves authenticity of a set of PK values (of the same table) that can be combined and compared together using proper set operations (intersection, union, and complementation).

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 and accepts the answer if the result matches the result set. Note that in all our proofs, **no additional records are sent to the client on top of the result set of the original query**.

4.1.2. Illustrative Examples

We give some examples to better understand our construction.

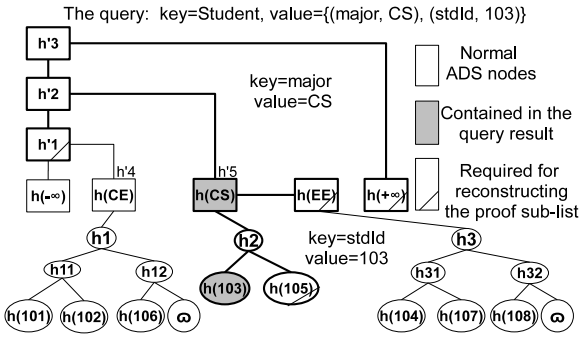


Figure 7: Proof generation for `SELECT * FROM Student WHERE Major='CS' and StdId=103`.

Selection in a two-level HADS. Figure 7 presents an example showing proof generation with a two-level HADS, for the query `SELECT * FROM Student WHERE Major='CS' and StdId=103` translated by 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 searched value 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(+∞)'.

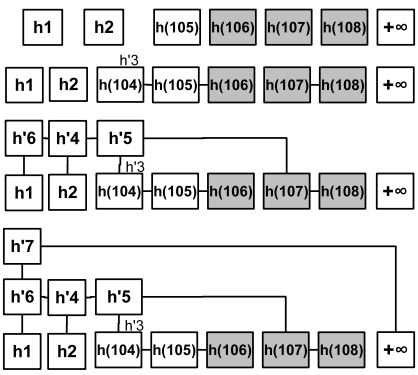


Figure 8: Proof verification for *vo*='h1, h2,h(104),105,**106**,**107**,**108**,h(+∞)'.

Verification is done in a bottom-up manner. The client 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 database ADS proof is verified in a similar manner. If all proofs are verified employing *all* and *only* the records in the answer, the client accepts the answer as authentic.

Since verification is accomplished similarly at all levels, we give an example showing verification in the ADS of Figure 3b, where *vo*='h1,h2,h(104), 105,**106**,**107**,**108**,h(+∞)' is the proof given for '`SELECT * FROM Student WHERE StdID>105`'. The verification algorithm extracts the result set {106,107,108} and boundary records {105,+∞}, checks if 105<106<107<108<+∞, and computes the hashes of records in the result set (step 1). Then, 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 simplicity.

5. Join

In relational database systems, data is organized (divided) into a set of tables. The *join* operations are then used to collect 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 servers effort, the communication, and the clients 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 among the (values of) tables they are created for, we can generate proofs proving correctness of those relations.

Without loss of generality, consider the (most widely used) one-to-many relationship: $\texttt{R} \bowtie_{rid=rid} \texttt{S}$, 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.) Since the values are stored sorted, the server traverses each ADS only once.

Efficient proof generation. Compared to [2, 20] that perform a range query on the second table for each value of the first table, our solution is much more efficient as it converts range queries into equalities for matches. In [1] that uses range queries efficiently, for each value in the ADS of one table, the set of matching values in the ADS of the other table is surrounded by *two more* records, for completeness. Since we store and compare unique values in HADSs, a value in the (H)ADS of one table *either matches exactly one* value in the (H)ADS of the other table (as shown by equality in *vo*), or *does not match any* value in the other (H)ADS (shown using range queries). In addition, the first-level ADSs that we use for proof generation are much smaller compared to those of all previous work, reducing the proof size and proof generation time.

Other join types. Our HADS-based solution 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 HADS only once, and is very efficient. Our algorithm for multi-way join queries can be generalized to support queries of the form $\texttt{R} \bowtie_{a=a} \texttt{S} \bowtie_{a=a} \texttt{T} \bowtie_{a=a} \ldots$, between *n* tables.

5.2. Two-way Join

Consider equi-join on two tables R and S represented as $\texttt{R} \bowtie_{C_i=C_j} \texttt{S}$, 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 $HADS_{\texttt{R}}(C_i)$ and $HADS_{\texttt{S}}(C_j)$ for proof generation. He starts by the smallest value (e.g., leftmost leaf node in a tree-based ADS) in the first-level ADS of one of the HADSs, and searches for that 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 a node 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 $\texttt{Student} \bowtie_{\texttt{StdId=StdId}} \texttt{S2C}$, where both tables have an HADS on column StdId: $HADS_{\texttt{Student}}(\texttt{StdId})$ and $HADS_{\texttt{S2C}}(\texttt{StdId})$. The proof generation works as follows: Traverse both HADSs until the leftmost leaf node (at the first level) storing the values v_1 (in $HADS_{\texttt{Student}}(\texttt{StdId})$) and v'_1 (in $HADS_{\texttt{S2C}}(\texttt{StdId})$). Possible cases are:

- $v_1 = v'_1$: 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 $HADS_{\texttt{S2C}}(\texttt{StdId}).\texttt{FindNext}(v_1)$ to find a matching on $HADS_{\texttt{S2C}}(\texttt{StdId})$. If it returns one node, $node_i$, a matching has been found, repeat the process with v_1 and $node_i.val$. If it returns two nodes, $node_j$ and $node_k$, there is no matching, but the value of $node_k$ may match that of the node next to v_1. Hence, add v_1, $node_j.val$, and $node_k.val$ into *vo* and repeat the process with $node_2.val$ and $node_k.val$, where $node_2$=$HADS_{\texttt{Student}}(\texttt{StdId}).\texttt{FindNext}()$ is the node next to v_1.

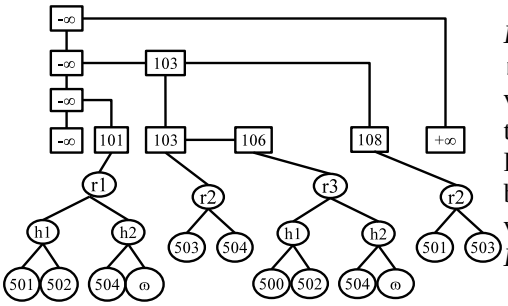


Figure 9: HADS of StdId (table S2C).

Example. Using $HADS_{\texttt{Student}}(\texttt{StdId})$ from Figure 3a and $HADS_{\texttt{S2C}}(\texttt{StdId})$ from Figure 9, we generate proof for Student $\bowtie_{\texttt{StdId=StdId}}$ 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 $HADS_{\texttt{Student}}(\texttt{StdId})$ 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 $HADS_{\texttt{Student}}(\texttt{StdId}).\texttt{FindNext}(103)$ 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 $HADS_{\texttt{Student}}(\texttt{StdId}).\texttt{FindNext}(106)$ 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 $HADS_{\texttt{Student}}(\texttt{StdId}).\texttt{FindNext}(108)$ 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)'.

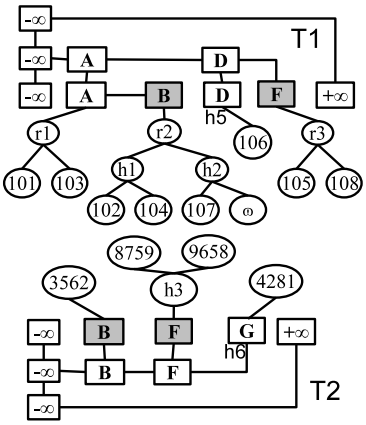


Figure 10: Non-PK join.

Neither C_i nor C_j is a PK column. Each column is stored inside an HADS. If each distinct value of C_i and C_j has an average PK-set of size n and m, respectively, with k matching records, the result set will have knm records. Our proof is of size $O(k(n+m))$, showing again the HADS proofs are efficient.

Imagine two tables $\texttt{T}_1$ and $\texttt{T}_2$, 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 10. The algorithm, starting at the leftmost nodes of both HADSs, finds out that B>A, and executes FindNext('B') on $\texttt{T}_1$, leading to *vo*= 'h(−∞), r1,**B**(102,104, 107);h(−∞),**B**(3562)'. It goes on, putting intermediate value h5 in the *vo*, finds another matching 'F', which is the last node in $\texttt{T}_1$. Later, FindNext('F') on $\texttt{T}_2$ puts h6 in *vo*, and realizes that both columns are fully traversed. These steps yield *vo*='h(−∞), r1,**B**(102,104,107);h(−∞),**B**(3562) : h5,**F**(105,108);**F**(8759,9658) : h(+∞);h6,h(+∞)'.

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 HVerify() 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

The general query optimization rule for queries containing various operations is that the join operation is performed after all selection operations, since the selection operations result in intermediate sub-tables (given as input to the join operations) that are likely to vary substantially in size [38]. Since our proofs are all based on PK-sets, the results of

the selection queries are integrated easily into those of join queries, resulting in small proofs (both communication 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., the 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 from 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 generated independently, 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 algorithm on sorted authentic PK-set resulting from the selection proof, and the other table's HADS. For each PK value in the sorted authentic PK-set, if there is a matching on the corresponding HADS of the other table, we reflect it on the proof. Otherwise, we supply a non-membership proof. Taking 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., $T_1 \bowtie_{C_i=C_j} T_2 \bowtie_{C_k=C_l} T_3 \bowtie$ ..., to combine them back together. Yang *et al.* [1] 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 their AIM algorithm 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.

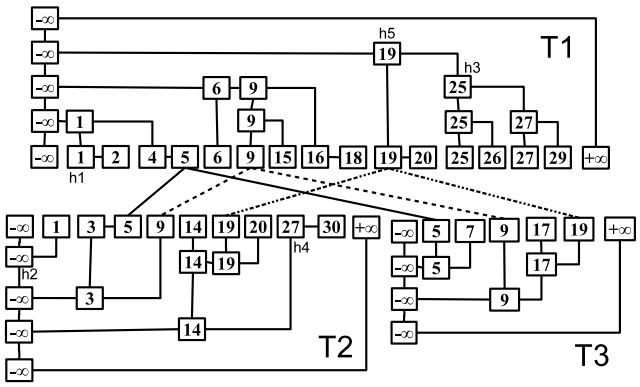


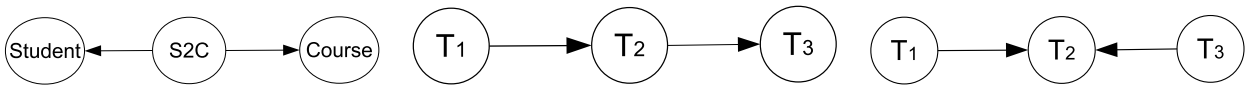
Figure 11: Proof generation for $T_1 \bowtie_{a=a} T_2 \bowtie_{a=a} T_3$.

Multi-way join on the same column. As noted by Raman *et al.* [39] and Yang *et al.* [1], 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: $T_1 \bowtie_{a=a} T_2 \bowtie_{a=a} T_3 \bowtie_{a=a} \ldots$. With slight changes, it can be generalized to support multi-way join among n tables.

We start by the smallest value in all HADSs. If all are the same, this is reflected in the *vo* as a matching. Otherwise, the maximum value among them, v_{max}, is selected and added into the *vo* and all other 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 either HADS is met. The verification object is then 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 skipping the largest possible number of nodes, providing an optimally efficient proof.

An example showing our proof generation for $T_1 \bowtie_{a=a} T_2 \bowtie_{a=a} T_3$ is given in Figure 11. It starts by the leftmost nodes: 1,1,5. Since 5 is the maximum, `FindNext`(5) is run on both T_1 and T_2, leading to *vo*='h($-\infty$),h1,h(4),**5**;h2,h(3),**5**; h($-\infty$),**5**'. Then, it jumps to and processes the next nodes, which are 6, 9, 7, and thus continues by `FindNext`(9) on T_1 and T_3. 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 gave new versions m-AISM, m-ASM, and m-AIM. They require prior information about the third table 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) T_3 refers to T_2 who refers to T_1. (c) Both T_1 and T_3 refer to T_2.

Figure 12: 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. Note that the result set is just the actual join result, not individual table items.

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 the proof for each join depends on the previous join, which depends, 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 how an efficient ordering can be employed.

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 12a.

Consider the case in Figure 12b: 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 12c, 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 the order of joins does not matter. 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 12a is also dealt with similarly. As both joins output an authentic set of PK values of S2C, the other join can be easily handled using this set and the other table.

Multi-way joins can be divided into a set of two-way joins, and the mentioned categories are 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 [38, 40, 41].

6. Security Analysis

Theorem 1 (Security of HADS). *Our HADS construction is secure according to Definition 8 (employing HADS algorithm names) if the underlying ADSs are secure.*

Proof 1. *We reduce security of the HADS scheme to the security of 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 $\mathcal{E}$ who breaks the security of at least one of the ADS schemes, with non-negligible probability. $\mathcal{E}$ acts as the server in the ADS games with the ADS challengers $\mathcal{C}_1, ..., \mathcal{C}_n$, and simultaneously, plays the role of the challenger in the HADS game with $\mathcal{A}$.*

***Setup**. $\mathcal{E}$ receives the ADS public keys from the respective challengers, and forwards them altogether to $\mathcal{A}$. The adversary $\mathcal{A}$ prepares a dataset D and gives it to $\mathcal{E}$ who employs the ADS challengers to generate the HADS containing the security information, and sends it back to $\mathcal{A}$. $\mathcal{E}$ keeps a local copy of D and the HADS. Note that this is invisible to the adversary $\mathcal{A}$, and thus will not affect his behavior.*

***Query**. $\mathcal{A}$ performs polynomial-many membership and update queries. A membership query is accompanied with the potential answer and proof, while an update query includes information about the update. $\mathcal{E}$ verifies the proofs of membership queries and notifies $\mathcal{A}$ about the results. For update queries, $\mathcal{E}$ asks the ADS challengers to prepare their corresponding updates and sends them to $\mathcal{A}$. $\mathcal{E}$ also updates her local copies of D and the HADS accordingly.*

***Challenge**. $\mathcal{A}$ chooses a membership query q, and gives it together with an answer a_q to $\mathcal{E}$. $\mathcal{A}$ wins if $\mathcal{E}$ accepts the a_q while it differs from the real answer of the query q.*

For $\mathcal{A}$ to win, a_q must be different from the real answer for at least one ADS, ADS_i, with its verifying sub-proof. $\mathcal{E}$ can find it through her local copy. Upon receipt of a_q, $\mathcal{E}$ selects the membership query, answer and proof parts related to ADS_i, and forwards them to $\mathcal{C}_i$. Assume $\mathcal{A}$ passes the HADS verification with non-negligible probability p. (This corresponds to the verification probability of ADS_i as the others would be verified with probability one.) $\mathcal{E}$ also passes ADS_i's verification with probability non-negligible p, breaking the security of ADS_i.

Since we employ secure ADSs, p must be negligible for all ADSs, and the adversary $\mathcal{A}$ has negligible probability of winning the HADS game. Therefore, if the underlying ADSs are secure, the HADS scheme is secure.

Theorem 2 (Security of the ODB scheme). *Our proposed ODB scheme is secure according to Definition 3, provided that the underlying HADS scheme is secure.*

Proof 2. *We reduce security of the ODB scheme to the security of the underlying HADS[2]. If a PPT adversary $\mathcal{A}$ wins the ODB security game with non-negligible probability, we can use it to construct a PPT algorithm $\mathcal{E}$ who breaks the security of HADS scheme with non-negligible probability. $\mathcal{E}$ acts as the server in the HADS game played with the HADS challenger $\mathcal{C}$, and simultaneously, plays the role of the challenger in the ODB game with $\mathcal{A}$.*

Setup. *$\mathcal{E}$ receives the HADS public key from $\mathcal{C}$, and relays them on to $\mathcal{A}$ (note that all HADSs built for each searchable column will use the same key). The adversary $\mathcal{A}$ prepares a database and hands it on to $\mathcal{E}$ who relays them on to the HADS challenger $\mathcal{C}$. $\mathcal{C}$ generates the security information in the form of HADSs and forwards them back to $\mathcal{A}$ through $\mathcal{E}$. $\mathcal{E}$ keeps a local copy of the database and the HADSs.*

Query. *$\mathcal{A}$ performs polynomial-many selection and update queries. A selection query is accompanied with the potential answer and proof, while an update query includes information about the update. $\mathcal{E}$ verifies the proofs of selection queries and notifies $\mathcal{A}$ about the results. For update queries, $\mathcal{E}$ ask $\mathcal{C}$ to prepare the respective HADS updates and sends them to $\mathcal{A}$. $\mathcal{E}$ also updates her local copy accordingly.*

Challenge. *$\mathcal{A}$ chooses a selection query cmd, and gives it together with an answer ans to $\mathcal{E}$. $\mathcal{A}$ wins if $\mathcal{E}$ accepts the answer ans while it differs from the real answer of the query cmd.*

If $\mathcal{A}$ wins the ODB security game with non-negligible probability, we can use it to break the security of HADS scheme with non-negligible probability. For the adversary to win, ans must be different from the real answer on at least one HADS, $HADS_i$, but with a verifying proof. On receipt, $\mathcal{E}$ selects the command, answer and proof parts related to $HADS_i$ from ans (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, $\mathcal{E}$ can pass the HADS verification (i.e., break HADS security) with the same non-negligible probability p. (This is because all other HADS proofs will be verified with probability one, and we only consider the verification probability of $HADS_i$ that is p.)

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

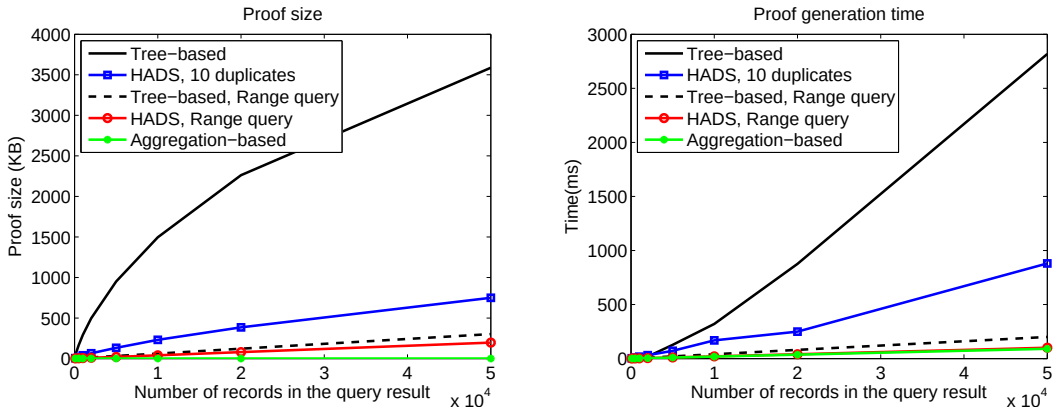
The proof is not specific to our two-level construction. For a four-level construction (Section 4.1.2), $\mathcal{E}$ plays the HADS game with a four-level HADS challenger. In general, for an n-level ODB construction, $\mathcal{E}$ should play the game with an n-level HADS challenger, in a same manner. The proof or the probabilities will not be affected by this change. Moreover, it is possible that different HADS types are used within the same ODB. Identical proofs per HADS type can be employed then, and as long as all underlying HADS schemes are secure, we would obtain a secure ODB scheme.

7. Performance Analysis

Setup. To evaluate our ODB scheme, we implemented a DBAS prototype using the efficient two-level HADS construction, which uses FlexList [31] at both levels, in C++ using Cashlib library [42]. We employ SHA1 with 160-bit digests as our hash function, and 1024-bit RSA as the digital signature scheme. All experiments were performed on a 2.5GHz machine with 4GB RAM and Ubuntu 11.10 operating system. The numbers are averages of 10 runs.

Our DBAS is deployed on the same machine where the DBMS resides, and stores the database security information. Dynamic queries (`Insert,Update,Alter,...`) affect this part as well, after being converted into the

[2]We assume all HADSs are similar, and hence, there is only one challenger. It is straightforward to extend it to the case with different HADS instantiations and multiple challengers.



(a) Proof size.

(b) Proof generation time.

Figure 13: Proof generation time and proof size for one-clause queries.

(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 a converter, but it should not affect the timing as 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 case, not all students are taking courses since we only have 10^6 `S2C` records.) This means 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.

We observe the system behavior (proof generation time and proof size) for different queries. Since proofs are generated using only the hashes of values of 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 computation and proof size:

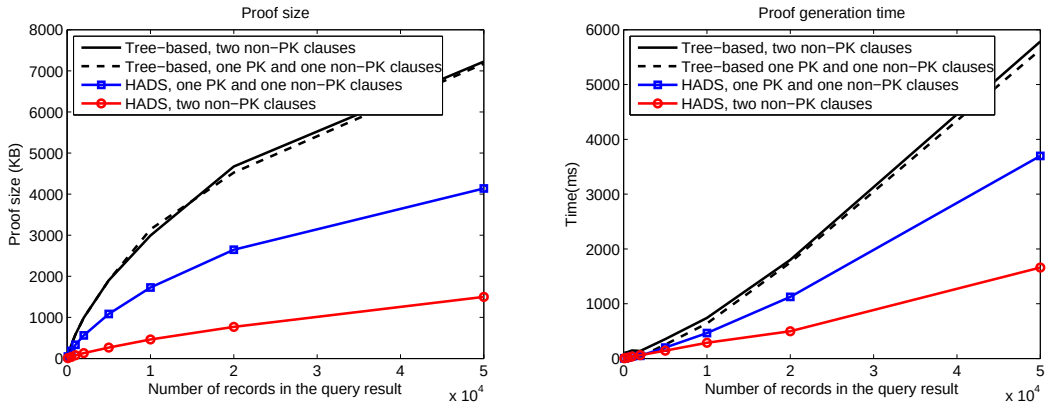
- The proofs are generated using only values of the required columns, and these values already exist in the DBMS answer to the query.
- The concept of PK-sets divides a large ADS into small ADSs in a hierarchy. Hence, the proof size and the computation time decrease as well.
- Using the PK-sets, there 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.

Classification of previous work. We compare our scheme against two types of previous work. The aggregation-based approaches [12, 20, 43] generate efficient proofs and may enhance server performance. They either do not provide completeness [43] or have problems with dynamism and freshness [12, 20]. We evaluate the performance of a prototype implementation of [20], denoted as '*aggregation-based*' in our figures.

The tree-based approaches store the security information in tree-based ADSs, and use different methods for making the duplicate values unique [7, 8, 10, 15] (and support dynamism). 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 concatenated each duplicate value with a replica number as in [7] to implement a regular ADS and compare against our HADS. This is referred to as '*tree-based*' in our figures, and it corresponds to all these works, if they use a similar underlying ADS.

7.1. Selection Queries on One Table

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 ADS storing the same column in tree-based existing schemes. (The second-level ADSs are included in whole, without any computation.) Because some values are repeated on non-PK columns, whereas the PK column contains only unique values. The



(a) Proof size. (b) Proof generation time.

Figure 14: Proof generation time and proof size for queries with two clauses.

proof generation time and proof size for a non-PK clause using HADS are thus expected to be smaller compared to the previous tree-based approaches. Figures 13a and 13b confirm this, showing ∼5x *smaller proofs*, and ∼3x *faster proof generations*. There is a ∼10% efficiency gain even with range queries. The aggregation-based approaches generate efficient (constant-size) proofs, with a server computation cost close to ours.

Two-clause queries. There are two cases: the query has either one PK and one non-PK clauses (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 non-PK columns, all values of second-level ADSs are included in the result (without further computation), hence 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 *vo*. Figures 14a and 14b 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 tree-based approaches, 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 tree-based approaches.

Multi-clause queries. 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 the summation of the respective 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.

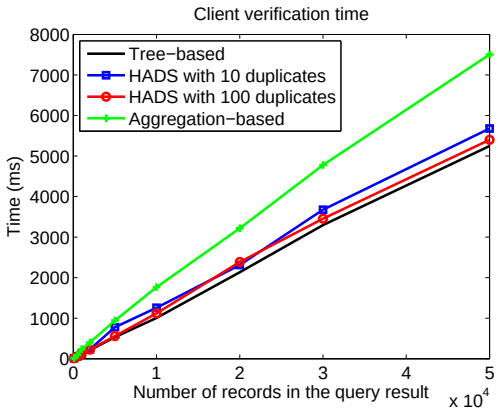
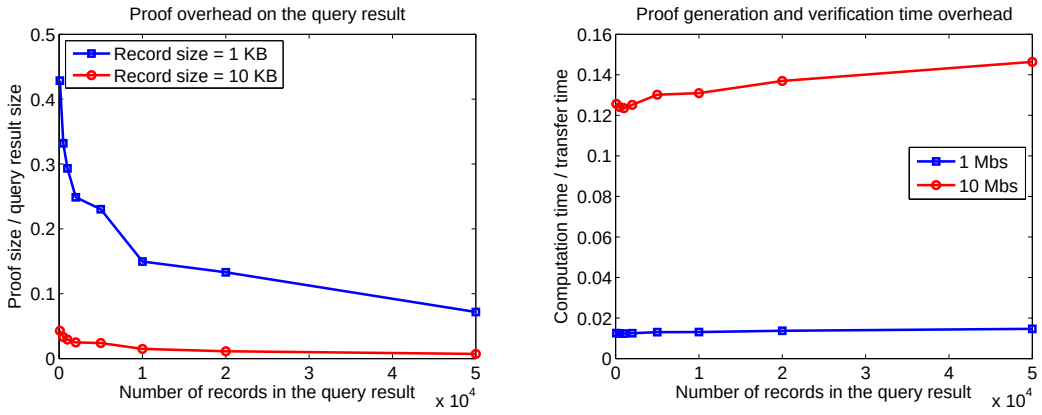


Figure 15: Client verification time.

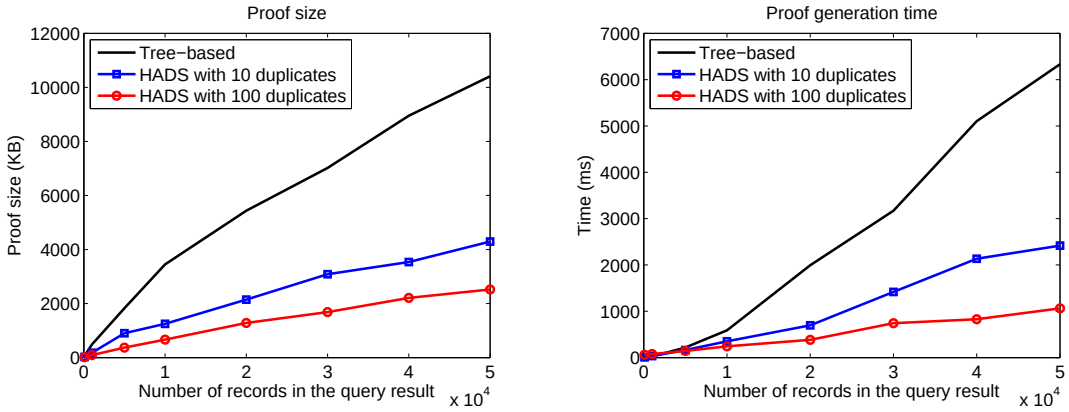
Client computation. We observed the HADS enhances client performance compared to the aggregation-based approaches [12, 20, 43], while posing similar performance as the tree-based existing schemes [7, 8, 10, 15]. The reason is that while the server just puts the whole second-level ADS into the *vo*, the client has to reconstruct the second-level ADS and 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. Hence, the total client computation using our HADS and previous ADSs are very close. In tree-based approaches, the client has to verify all received 'signatures'. These are illustrated in Figure 15 for one-clause queries.

Overhead. Another important factor is the communication overhead, i.e., how much does the proof increase the traffic. As the proof size is independent of the record size, for tables with small record size (∼1 KB), 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 20 KB 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



(a) Proof size overhead. (b) Proof time overhead.

Figure 16: Proof overhead and client verification time.



(a) Proof size. (b) Proof generation time.

Figure 17: Proof generation time and proof size (key-based join).

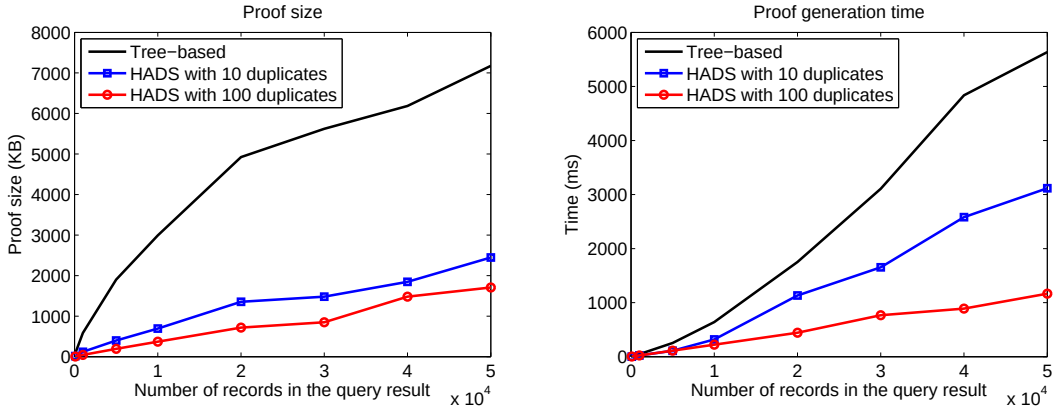
16a. Compared to similar algorithms such as [15] that require $O(\log N + t)$ cost for a query result of size t, using range queries, the cost of our algorithm is $O(\log |C_i| + t)$.

For time overhead, we compared proof generation time plus client verification time to the total time taken for preparing the result set (server) and the result set transfer time. This is, however, an upper bound since the proof generation time normally overlaps with the server computation time. As Figure 16b shows, using 1 Mbs and 10 Mbs bandwidths, the time overheads of our scheme are only ∼1% and ∼14%, respectively.

7.2. Join Queries

We consider two cases. In *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 *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 **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 17a and 17b. 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 tree-based works.



(a) Proof size.

(b) Proof generation time.

Figure 18: Proof generation time and proof size (general join).

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 18a and 18b 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.* [2] for joining two tables $\mathtt{T}_1$ and $\mathtt{T}_2$ of approximate size N is $O(N \log N)$, while that of ours is $O(N+N) = O(N)$. Compared to [1] which 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., our algorithm drops in the worst case to that of [1].

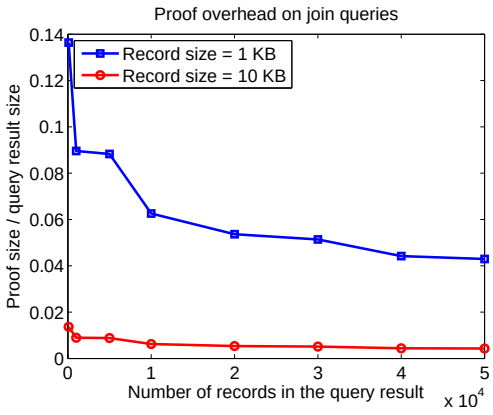


Figure 19: Proof overhead.

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.* [1], where the proof size increases with the record size. Figure 19 shows the overhead of our proofs on the join query result with two record sizes: 1 KB and 10 KB. Our proofs add only ∼1% overhead when record size is 10 KB, and ∼6% overhead when record size is 1 KB.

Comparison to previous work. Join algorithms in [1, 20] always add two boundary *value*s or *record*s for each comparison during a join. Our algorithm adds two boundary *value*s only for non-matching records in a comparison. Table 1 shows some concrete *vo* sizes related to these schemes in a similar setting, for the key-based join. The client and server computation times for the same key-based join are given in Table 2. As our scheme uses HADSs built on column values, the computation times are independent of the record size, helping our scheme outperform those in [1].

Table 1: Proof size comparison for join.

Record size	Yang *et al.* [1]	Pang *et al.* [20]	Our scheme
64 B	∼32 MB	∼13 MB	∼2 MB
512 B	∼202 MB		

Table 2: Client and server computation times for join.

Scheme	Client verification		Server proof generation	
	Record size		Record size	
	64 B	512 B	64 B	512 B
Yang *et al.* [1]	18 *s*	128 *s*	7 *s*	100 *s*
Our scheme	3.6 *s*		1.4 *s*	

8. 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 formally proved these properties using our new unified security definition.

Our outsourced database construction can provably handle selection queries with one or multiple clauses, 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 leave concurrent proof generation 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 $\sim$3x smaller proofs in size and $\sim$5x faster proof generation when HADS is used for queries with one clause. Moreover, for join queries we observed $\sim$4x enhancement in proof size and $\sim$5x enhancement in proof generation time using HADS, when each foreign key is repeated 100 times, on average. With reasonable record sizes, e.g., 5-20 KB in our Koç University database's `Student` table, the communication overhead is $\sim$4% compared to the result size, becoming even smaller with larger record sizes. These all confirm practicality of our outsourced databases scheme.

Acknowledgement

This work is supported by TÜBİTAK, the Scientific and Technological Research Council of Turkey, under project numbers 112E115 and 114E487, and European Union COST Actions IC1206 and IC1306.

References

References

[1] Y. Yang, D. Papadias, S. Papadopoulos, P. Kalnis, Authenticated join processing in outsourced databases, in: ACM SIGMOD International Conference on Management of data, 2009, pp. 5–18.

[2] F. Li, M. Hadjieleftheriou, G. Kollios, L. Reyzin, Dynamic authenticated index structures for outsourced databases, in: ACM SIGMOD, 2006, pp. 121–132.

[3] J. Wang, X. Du, Skip list based authenticated data structure in das paradigm, in: GCC'09, 2009.

[4] M. T. Goodrich, R. Tamassia, N. Triandopoulos, Super-efficient verification of dynamic outsourced databases, in: CT-RSA, Springer, 2008, pp. 407–424.

[5] G. Nuckolls, Verified query results from hybrid authentication trees, in: Data and Applications Security XIX, Springer, 2005, pp. 84–98.

[6] Z. Liu, X. Chen, J. Yang, C. Jia, I. You, New order preserving encryption model for outsourced databases in cloud environments, Journal of Network and Computer Applications 59 (2016) 198–207.

[7] H. Pang, A. Jain, K. Ramamritham, K.-L. Tan, Verifying completeness of relational query results in data publishing, in: ACM SIGMOD, ACM, 2005, pp. 407–418.

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

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

[10] P. Devanbu, M. Gertz, C. Martel, S. G. Stubblebine, Authentic third-party data publication, in: Data and Application Security, Springer, 2002, pp. 101–112.

[11] H. Pang, K.-L. Tan, Authenticating query results in edge computing, in: International Conference on Data Engineering, IEEE, 2004, pp. 560–571.

[12] M. Narasimha, G. Tsudik, Authentication of outsourced databases using signature aggregation and chaining, in: Database Systems for Advanced Applications, Springer, 2006, pp. 420–436.

[13] E. Mykletun, M. Narasimha, G. Tsudik, Providing authentication and integrity in outsourced databases using merkle hash trees, 2003.

[14] B. Palazzi, Outsourced storage services: Authentication and security visualization, Ph.D. thesis, Roma Tre University (2009).

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

[16] J. Wang, X. Chen, X. Huang, I. You, Y. Xiang, Verifiable auditing for outsourced database in cloud computing, IEEE transactions on computers 64 (11) (2015) 3293–3303.

[17] X. Chen, J. Li, J. Weng, J. Ma, W. Lou, Verifiable computation over large database with incremental updates, IEEE transactions on Computers 65 (10) (2016) 3184–3195.

[18] J. Xu, Z. Cao, Q. Xiao, F. Zhou, An improved authenticated skip list for relational query authentication, in: Broadband and Wireless Computing, Communication and Applications (BWCCA), 2014 Ninth International Conference on, IEEE, 2014, pp. 229–232.

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

[20] H. Pang, J. Zhang, K. Mouratidis, Scalable verification for outsourced dynamic databases, VLDB 2 (1) (2009) 802–813.

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

[22] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, P. Samarati, Integrity for join queries in the cloud, IEEE Transactions on Cloud Computing 1 (2) (2013) 187–200.

[23] Y. Zhang, J. Katz, C. Papamanthou, Integridb: Verifiable sql for outsourced databases, in: ACM CCS'15, ACM, 2015, pp. 1480–1491.

[24] B. Parno, J. Howell, C. Gentry, M. Raykova, Pinocchio: Nearly practical verifiable computation, in: Security and Privacy (SP), 2013 IEEE Symposium on, IEEE, 2013, pp. 238–252.

[25] M. Backes, M. Barbosa, D. Fiore, R. M. Reischuk, Adsnark: nearly practical and privacy-preserving proofs on authenticated data, in: 2015 IEEE S & P, IEEE, 2015, pp. 271–286.

[26] A. Boldyreva, N. Chenette, Y. Lee, A. O'neill, et al., Order-preserving symmetric encryption., in: Eurocrypt, Vol. 5479, Springer, 2009, pp. 224–241.

[27] A. Boldyreva, N. Chenette, A. O'Neill, Order-preserving encryption revisited: Improved security analysis and alternative solutions., in: CRYPTO, Vol. 6841, Springer, 2011, pp. 578–595.

[28] T. Xiang, X. Li, F. Chen, S. Guo, Y. Yang, Processing secure, verifiable and efficient sql over outsourced database, Information Sciences 348 (2016) 163–178.

[29] D.-W. Sun, G.-R. Chang, S. Gao, L.-Z. Jin, X.-W. Wang, Modeling a dynamic data replication strategy to increase system availability in cloud computing environments, Journal of computer science and technology 27 (2) (2012) 256–272.

[30] M. Etemad, A. Küpçü, Transparent, distributed, and replicated dynamic provable data possession, in: International Conference on Applied Cryptography and Network Security, Springer, 2013, pp. 1–18.

[31] E. Esiner, A. Küpçü, Ö. Özkasap, Analysis and optimization on flexdpdp: A practical solution for dynamic provable data possession, in: Intelligent Cloud Computing (ICC'14), 2014.

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

[33] R. Tamassia, Authenticated data structures, in: Algorithms-ESA 2003, Springer, 2003, pp. 2–5.

[34] C. Papamanthou, R. Tamassia, Time and space efficient algorithms for two-party authenticated data structures, Springer, 2007, pp. 1–15.

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

[36] W. Pugh, Skip lists: a probabilistic alternative to balanced trees, Comm. of the ACM 33 (1990) 668–676.

[37] R. Merkle, A certified digital signature, in: CRYPTO'89, Springer, 1990, pp. 218–238.

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

[39] V. Raman, L. Qiao, W. Han, I. Narang, Y. Chen, K. Yang, F. Ling, Lazy, adaptive rid-list intersection, and its application to index anding, in: ACM SIGMOD, Vol. 11-14, 2007, pp. 773–784.

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

[41] G. Graefe, Query evaluation techniques for large databases, ACM Computing Surveys 25 (2) (1993) 73–169.

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

[43] E. Mykletun, M. Narasimha, G. Tsudik, Authentication and integrity in outsourced databases, ACM Transactions on Storage (TOS) 2 (2) (2006) 107–138.

[44] C. Papamanthou, R. Tamassia, N. Triandopolos, Authenticated hash tables, in: ACM CCS'08, 2008, pp. 437–448.

[45] M. T. Goodrich, R. Tamassia, J. Hasić, An efficient dynamic and distributed cryptographic accumulator, in: Info. Security, Springer, 2002, pp. 372–388.

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

[47] D. J. DeWitt, J. F. Naughton, D. A. Schneider, An evaluation of non-equijoin algorithms, in: VLDB, Morgan Kaufmann Publishers Inc., 1991, pp. 443–452.

[48] J. Benaloh, M. De Mare, One-way accumulators: A decentralized alternative to digital signatures, in: EUROCRYPT'93, Springer, 1994, pp. 274–285.

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

[50] C. Erway, A. Küpçü, C. Papamanthou, R. Tamassia, Dynamic provable data possession, in: CCS'09, ACM, 2009, pp. 213–222.

Appendix A. ADS Definitions

Definition 5. *An **ADS scheme** consists of following three polynomial-time algorithms [34]:*

$\texttt{KeyGen}(1^k) \rightarrow (sk, pk)$ *is run by the client to generate a private and public key pair* (sk, pk) *given the security parameter k. She shares the public key pk with the server.*

$\texttt{Certify}(pk, cmd) \rightarrow (ans, \pi)$ *is executed 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.*

$\texttt{Verify}(sk, pk, cmd, ans, \pi, st) \rightarrow (\{\texttt{accept}, \texttt{reject}\}, st')$ *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 6. ***Correctness of ADS**. For all valid proofs* π *and server answers ans in response to client commands, the verify algorithm accepts with overwhelming probability.*

Definition 7. ***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 8. ***Security of ADS**. We say that the ADS is secure if no PPT adversary can win the ADS security game with non-negligible probability.*

Fact 1 (Security of ADS). *The ADS is secure according to Definition 8.*

This is proved for different schemes separately. Merkle [37] showed the security of Merkle hash tree, Papamanthou *et al.* [44] did the job for the authenticated hash table, Goodrich *et al.* [45] proved security of the accumulator based ADS, Noar and Nissim [46] showed security of the 2-3 tree, and Papamanthou and Tamassia [34] proved security of the ADSs based on authenticated skip list or red black tree.

Appendix B. Additional Discussion

Appendix B.1. Modification

As an example targeting modification, consider adding a new record into the `Student` table: `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)))⟩})`.

Appendix B.2. Tables with Composite Keys

Some tables may employ composite keys that makes the construction problematic: We cannot relate a non-PK column to any subset of the PK columns due to the existence of duplicate values. Previous schemes [1, 15] cannot handle this case efficiently, as they need to construct and use multiple ADSs for each column.

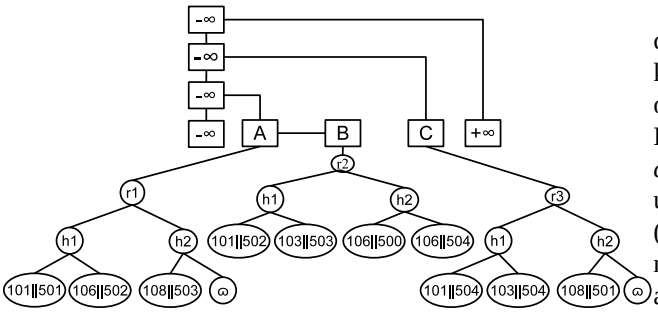


Figure B.20: Storing the column `Mark` from table `S2C` with composite PK (`StdId` and `CrsId`).

HADS solves this problem efficiently. The concatenation of multiple foreign keys forms a composite key, generally. 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 B.20 where the composite key for table `S2C` is `StdId||CrsId`. Since neither `StdId` nor `CrsId` is a PK in table `S2C`, they both have their respective two-level HADSs, where, for example, unique `StdId` values are stored at the first level ADS, and for each unique `StdId`, all associated composite keys `StdId||CrsId` are stored at the second level ADS (a similar HADS is built for `CrsId`).

Appendix B.3. Special Joins

Equijoin is defined to be the join in which the operator is equality [38, 47]. The *non-equijoin*, which is also called the *band join*, is defined as the join operation that the operator is not equality [38]; i.e., the values of one of the join columns fall within a *band* of values of the other column [47].

Equijoin of the form $\mathtt{T_1}.\mathbf{C_i} = \mathtt{T_2}.\mathbf{C_j} \mp \mathbf{n}, \mathbf{n} \in \mathbb{N}$. This is a special case of the equijoin. We treat $\mathtt{T}_1.C_i = \mathtt{T}_2.C_j \mp n$ as matching (instead of $\mathtt{T}_1.C_i = \mathtt{T}_2.C_j$) and apply the equijoin algorithm. Proof generation for the query $\mathtt{T}_1.a+1 = \mathtt{T}_2.a = \mathtt{T}_3.a-2$ on Figure 11 works as follows: The algorithm starts with the smallest values 1, 1, 5, respectively. Since the relation $\mathtt{T}_1.a+1 = \mathtt{T}_2.a = \mathtt{T}_3.a-2$ 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 $\mathtt{T}_2$, so we run `FindNext`(3) on $\mathtt{T}_2$. It will also be matched with 5-2-1=2 in $\mathtt{T}_1$, so we run `FindNext`(2) on $\mathtt{T}_1$. Using our join proof generation algorithm this way generates *vo*= 'h(−∞),h(1),**2**;h2,**3**;h(−∞),**5** : **4**;**5**;**7** : 6,9;9;9,17 : 15;14,19;17 : h(16),18;19;19,h(+∞) : h5,h(+∞);h(20),h4,h(+∞);'. This shows that there are two matchings, (2, 3, 5) and (4, 5, 7), for the query $\mathtt{T}_1.a+1 = \mathtt{T}_2.a = \mathtt{T}_3.a-2$ on Figure 11.

Non-equijoin. The general form of a non-equijoin query is $|\mathtt{T}_1.C_i - \mathtt{T}_2.C_j| < n, n \in \mathbb{N}$. 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*.
- 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 HADS that prevents a malicious server from adding or deleting matching records, there is no need to go backward. Thus, 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.

Therefore, in both cases, 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 go backward. This is an important observation that simplifies the client and server computation. Two helper functions `FindLeftBoundary()` and `FindRightBoundary()` with obvious functionality are used during the algorithm.

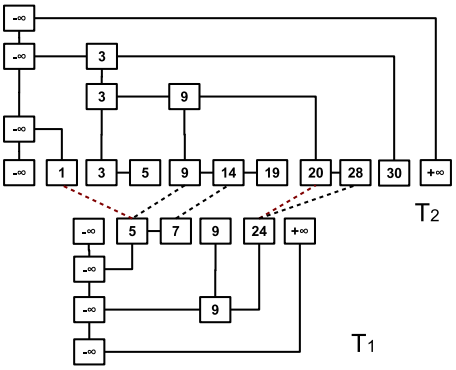


Figure B.21: Non-equijoin proof genera- tion scenario for $|\mathtt{T}_1.a - \mathtt{T}_2.a| < 3$.

Assume that we want to execute the non-equijoin query $|\mathtt{T}_1.a - \mathtt{T}_2.a| < 3$ on the example given in Figure B.21. We start by $\mathtt{T}_1$ (who has fewer records) and for each record, find the set of matching records on $\mathtt{T}_2$. 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,**31**,**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 $\mathtt{T}_1$ is reached, we add $h(30)$ as the intermediate data of $\mathtt{T}_2$. Finally, the proof *vo*='**5**;1,**3**,**5**,9 : **7**;14 : 9;- : 24;h(19),20,28 : h(+∞);h(30),h(+∞)' is returned.

Appendix C. Efficient ODB Construction

Different ADSs can be chosen for HADS levels subject to their requirements and the application. We employ two-level HADSs, with special role and considerations for each level, 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 [48]), *sublinear* (e.g., authenticated hash tables [44]), and *logarithmic* (e.g., authenticated skip list [49, 50]).

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

Table C.3: A comparison of second-level ADSs for storing a single table. Proof size and verification time are given for one-dimensional queries. s denotes the number of searchable columns, and t shows the number of records in the first level.

	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	$\log N+(s-1)(\log\|C_i\|+\log N/\|C_i\|)=s\log N$

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 linked to each leaf node; therefore, we require $O(|C_i|+(1+\varepsilon)N/|C_i|*|C_i|)=O(|C_i|+(1+\varepsilon)N))\approx O(|C_i|+N)$ storage in total (including the $O(|C_i|)$ space for the first-level ADS). Here, ε is a constant. The first-level ADS proof is the same, and the constant proof size ε of the authenticated hash table [34] makes the proof $(O(\log|C_i|)+t)$ for t records in the result set. Hash operations are also 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 C.3, 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 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.