Database Outsourcing with Hierarchical Authenticated Data Structures

Mohammad Etemad and Alptekin Küpçü

Koç University, İstanbul, Turkey
{metemad, akupcu}@ku.edu.tr

Abstract. In an outsourced database scheme, the data owner delegates the data management tasks to a remote service provider. At a later time, the remote service is supposed to answer any query on the database. The essential requirements are ensuring the data integrity and authenticity with efficient mechanisms. Current approaches employ authenticated data structures to store security information, generated by the client and used by the server, to compute proofs that show the answers to the queries are authentic. The existing solutions have shortcomings with multi-clause queries and duplicate values in a column.
We propose a hierarchical authenticated data structure for storing security information, which alleviates the mentioned problems. We provide a unified formal definition of a secure outsourced database scheme, and prove that our proposed scheme is secure according to this definition, which captures previously separate properties such as correctness, completeness, and freshness. The performance evaluation based on our prototype implementation confirms the efficiency of the proposed outsourced database scheme, showing more than 50% decrease in proof size and proof generation time compared to previous work, and about 1-20% communication overhead compared to the query result size.

1 Introduction

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

An important problem is that by data outsourcing, the owner loses the direct control over her data and should rely on answers coming from the remote service provider (who is not fully trusted). Therefore, there should be mechanisms giving the data owner the ability for checking the integrity of the outsourced data. To make sure that the server operates correctly, the client should verify the query answers coming from the servers [8]. The server sends to the client a *verification object* (*vo*) along with the query answer (the *result set*). The *vo* enables the client to verify the authenticity of the answer. Since the client may be a portable device with limited processing power, the *vo* should be small, and efficiently verifiable. The client uses the *vo* to verify that the response is [25, 9, 24, 8, 15]:

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

We want to perform authentic queries on all *searchable* columns (the columns that can be used to build clauses) of a table. All existing methods have a common problem with duplicate values in non-PK searchable columns [18, 10, 14]. The general method is to sort a table by each searchable column, and build an Authenticated Data Structure (ADS) on the result. In other words, a total ordering on the values of searchable columns is required, which together with the fact that the duplicate values belong to different records, make building the ADS using those values problematic.

We introduce the *hierarchical ADS* (HADS) for solving this problem. HADS is also advantageous in proof generation for multi-dimensional (multi-clause) queries. The HADS can be stored in the same database [2], or separately. Storing it separately breaks the tie to a specific database and brings more flexibility. This way, the outsourced data can be changed without affecting the proof system.

The rationale behind this work is to relate everything to the PKs. Since PKs are unique identifiers of records in a database, we can 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). We also support dynamic databases where the data owner may apply modification queries (Insert, Delete, Update), in a provable manner.

1.1 Related Work

Elementary approaches A simple way for verifying the authenticity of an answer to an outsourced database query is to sign each table and store the signature locally [25, 4]. This method requires sending the whole table to the client for verification, and hence, does not scale up. Another method is to compute and store with each record, a signature that verifies the contents of the record [4]. The problems are that computing a signature (for each record) is an expensive operation, and this method does not provide completeness.

Approaches based on Verifiable B-tree Pang and Tan [19] propose using one or more *verifiable* B-trees (VB-tree) for each table. The VB-tree is an extension of B-tree using the Merkle hash tree [12]. The records are sorted by a column before constructing the VB-tree, and for each table we need VB-trees in the number of searchable columns. It does not support completeness, and found insecure for the insecurity of the function used to compute the signatures [14].

A variant of this method, named MB-tree, is used intensively in the literature [3, 13, 15, 25]. MB-tree is similar to VB-tree except that a light hash function is used instead of expensive signatures. The client stores locally the root's digest, or signs and stores it on the server.

Approaches based on authenticated skip list Another line of work is using an authenticated skip list to store the required information for the verification [16, 24]. Skip list is suitable and efficient enough for this purpose, especially when we consider dynamic scenarios [24]. Wang and Du [24] prove that skip-list-based ADS provides soundness and completeness for one-dimensional range queries, and multiple ADSs are required for multi-dimensional range queries.

Palazzi [16, 17] constructs one skip list for each searchable column in each table. An important problem with this scheme is that for multi-dimensional queries, only the result of one skip list (determined by the First Column Returned approach [16], or the fastest one [17]) is used. The result set is sent to the client who will apply the other clauses. Therefore, the result set contains a larger set than the real result set of the query, and hence, is not efficient.

Authenticated range query is the method used to prove completeness (i.e., no extra records and no missing ones) [3, 9, 25, 14]. The server 1) finds the *contiguous* records as the result set, and the *boundary* records (one immediately before the first record, the *before* record, and one immediately after the last record, the *after* record), 2) selects the values needed for the ordered ADS membership proof of the boundary records, and 3) puts all these values into the verification object and sends it to the client. The set {*before record, result set, after record*} is guaranteed by the ordered ADS to be a sorted and contiguous set of records, with no extra or missing record between them [24, 11].

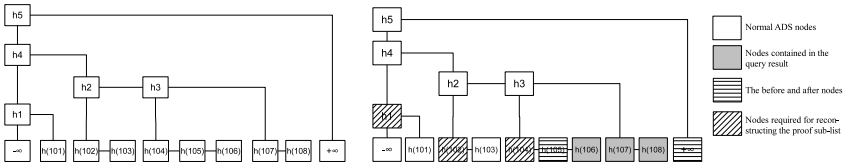
A common **problem** with all these methods [19, 3, 13, 15, 25, 16, 9, 18, 14, 8, 17] is the duplicate values in non-PK columns that make building the ADSs problematic since distinct values are required. Pang *et al.* [18] and Li *et al.* [10] propose applying the standard geometric perturbation method, guaranteeing a total ordering. Although this solution works for static data, it is not suitable for a dynamic case, since perturbing a duplicate value may result again in duplicate values. Narasimha and Tsudik [14] propose appending a value, i.e., the record ID, to solve the problem; but then searching on the column will be problematic, i.e., we cannot search by only the real values of the column. Palazzi *et al.* [17] appends to each value, hash of the record the value belongs to, and builds the ADSs using these values. In our performance results, we show that these solutions that keep all duplicate values in the same ADS result in significantly slower systems.

2 Preliminaries

Notation: We use N to denote the number of records of a table, $|C_i|$ to denote the number of distinct values in column i, and t to denote the number of records in the result set. The '|' denotes the concatenation, and PPT denotes probabilistic polynomial time. 'PK' denotes 'primary key' in a database table, and 'pk' stands for 'public key'. A function $\nu(k) : Z^+ \rightarrow [0, 1]$ is called *negligible* if $\forall$ *polynomials* p, $\exists$ *constant* k_0 s.t. $\forall\ k > k_0,\ \nu(k) < |1/p(k)|$. Overwhelming probability is greater than or equal to $1 - \nu(k)$ for some negligible function $\nu(k)$.

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

An **authenticated data structure (ADS)** is a scheme for data authentication, where untrusted responders answer queries on the data structure and provide extra information used to produce a cryptographic proof that the answers are valid [22, 23, 20]. The client uses the proof to reconstruct the ADS locally and verify correctness of the answer [20]. Each node of the ADS is assigned a value that is computed as a function of some neighboring nodes in the structure. We provide a formal definition in Appendix A.



(a) ADS for a PK column. (b) A membership proof.

Fig. 1: (a) A regular (non-hierarchical) ADS storing the PK column of the `Student` table, and (b) the membership proof for a query whose result set is {106, 107, 108}.

Authenticated skip list is an ADS constructed using a collision resistant hash function. Labels from the queried node to the root make a proof of membership [6]. Figures 1a and 1b present an authenticated skip list storing a PK column, and the membership proof for the result set $\{106, 107, 108\}$, respectively.

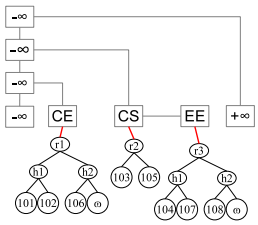
Papamanthou *et al.* [21] introduce the **authenticated hash table**, that is a hierarchy of one-way accumulators in a way that provides constant proof size and verification time. It also keeps either query or update time constant while providing the other with sub-linear complexity.

3 Hierarchical Authenticated Data Structures

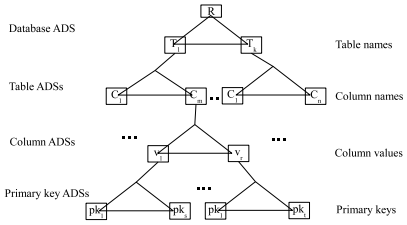
Hierarchical ADS (HADS) is an ADS consisting of multiple levels of ADSs. Each ADS at level i is constructed on top of a number of ADSs at level $i + 1$. Each element of an ADS at level i stores the digest of and a link to an ADS at level $i+1$. Therefore, multiple ADSs with different underlying structure can be linked together to form a hierarchical ADS with multiple levels. The only restriction is that all ADSs at level i must be of the same underlying structure to have consistent proofs.[1] The elements of last level ADSs contain data items (without links to other ADSs). The client stores the digest of the first level ADS as metadata. Figure 2a presents a two-level HADS instantiation using authenticated skip list at the first, and Merkle hash tree at the second level, and Figure 2b shows a general four-level HADS architecture (the ADSs are represented like trees for simplicity of presentation, but they can be of any type).

An **HADS scheme** is an ADS scheme defined with three PPT algorithms *(HKeyGen, HCertify, HVerify)* for the sake of distinguishing them from non-hierarchical versions. All definitions for the ADS (the Appendix A), using HADS algorithm names, directly provide a formal framework for HADS schemes.

[1] We can handle the heterogeneous case as well, but it complicates the presentation.



(a) A two-level HADS.



(b) A general four-level HADS.

Fig. 2: (a) A two-level HADS using authenticated skip list and Merkle hash tree, and (b) a general construction of a four-level HADS for storing the whole database.

3.1 HADS Construction

We construct HADS using (possibly different) ADSs at multiple levels in a hierarchical structure. The modification or proof generation is a recursive operation that needs to traverse the ADSs in a top-down manner. We provide the input as a series of $(key, value)$ pairs in such a way that the pairs needed for the upper levels appear first. The command execution will begin on the first-level ADS, and be directed by the data provided in the form of $(key, value)$ pairs, which are parsed to proper sub-ADSs at each level. For a query command, only the keys will be used, but for the modifications, both key and value will be used.

The $HKeyGen$ algorithm generates a public and private key pair for each level, combines all public keys into pk, and all private keys into sk, and outputs the result as the private and public key pair of the HADS (Algorithm 3.1).

Algorithm 3.1: HKeyGen, run by the client.

Input: security parameter k, number of levels n, and type of each level.
Output: the private and public keys of the HADS

$sk_{HADS} = \{\}$; `//private key of the HADS.`
$pk_{HADS} = \{\}$; `//public key of the HADS.`
for $i = 1$ *to* n **do**
    $(sk, pk) = ADS_i.KeyGen(1^k)$; `//Ask the level i ADS to produce its keys.`
    $sk_{HADS} = sk_{HADS} \cup sk$;
    $pk_{HADS} = pk_{HADS} \cup pk$;
return (sk_{HADS}, pk_{HADS});

The modification and proof generation on HADS are recursive operations, starting at the root ADS, and repeating on all affected ADSs in the hierarchy. Each level's ADS generates its own proof. The ADSs are tied to each other in a way that each leaf node of an ADS at level i stores the digest of and a link to an ADS at level $i + 1$, thus the HADS proof will combine all required ADS proofs (Algorithm 3.2). To simplify this operation, we use another PPT algorithm as a helper method to find the sub-ADSs of a given ADS:

Find$(\mathbf{key}, \mathbf{value}) \rightarrow (\{(\mathbf{ADS'}, \{(\mathbf{key'}, \mathbf{value'})\})\})$: This algorithm is used (inside the HCertify) to interpret the input data and find the corresponding part for each level, as we traverse the levels. It first finds the ADS at the

next level by traversing the current ADS with the provided *key*. Then, it extracts the data inside the *value* as a set of $\{(key', value')\}$ pairs. Finally, it outputs the set of next ADSs and their associated $\{(key', value')\}$ pairs. An example showing this process is given in Section 4.1.

Algorithm 3.2: HCertify, run by the server.

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

```
1  P_own = {}; // Proof of the current ADS.
2  P_child = {}; // Proof of all children combined together.
3  {(ADS', {(key', value')})} = Find(key, value); // Null for the last level.
4  for each element in {(ADS', (key', value')}) do
5      P = element.ADS'.HCertify(pk, cmd, element.(key', value')); //Ask each
       child to compute its proof.
6      P_child = P_child|P; // Combine the proofs.
7  P_own = Certify(pk, OP, (key, value)); // Current ADS proof (not hierarchical).
8  return P_child|P_own;
```

The verification is also a recursive process. The client reconstructs the required parts of the HADS in a bottom-up manner, i.e., she verifies the last level, then uses its digest and the next level proof to verify the above level, and so forth. Finally, when the client reaches the upper-most level and obtains a single digest, she compares it with the local metadata for verification.

4 Outsourced Database Scheme

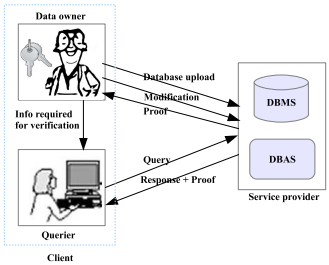


Fig. 3: The ODB model.

Model The outsourced database (ODB) model, as depicted in Figure 3, is composed of three parties: the *data owner*, the *querier*, and the *service provider*. The data owner performs required pre-computations, uploads the database, and gives the querier(s) the security information she needs for verification. The data owner then may perform modifications (insertion, deletion, or update) on the outsourced database. The service provider (or simply, the *server*) has the required equipment (software, hardware, and network resources) for storing and maintaining the database in a provable manner. The internal structure of the server is transparent to the outside, i.e., the server may use some levels of replication and distribution to increase the performance and availability. The *querier* (or the *user*) issues a query (either select or modification) to the server, who executes the query, collects the result set, computes the corresponding proof, and sends all back to the querier. The querier then verifies the answer using the security information given by the data owner. It is possible to have multiple queriers or data owners, and data owners can also act as queriers. For simplicity, we will refer to them simply as the *client*. This paper considers only single-client case.

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

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

4.1 Generic ODB Construction

A generic way to construct an ODB is to employ a regular DBMS, together with a DBAS based on an ADS. The HADS can be used to construct the DBAS, solving the problem of duplicate values. If the query has a clause on a non-PK column, say col_i, containing duplicate values, the result set of the query includes all records with the specified value(s) in col_i. The way we can identify these records and compare them with the result set of the other clauses is to relate each record to its corresponding (unique) PK. For each *distinct* value in a non-PK column, we define a *PK set* as:

Definition 1. ***PK Set****: For each value v_i in a non-PK column, the set of all PK values corresponding to v_i in all records of a table T is called the PK set of v_i, and represented as $PK(v_i)$, i.e., $PK(v_i) = \{k_j \in PK(T) : \exists \text{ record } R \in T \text{ s.t. } k_j \in R \wedge v_i \in R\}$.*

Note that the PK set includes only the PK values, not the whole records. Any membership scheme can be used for assigning the PK set to a non-PK value, regarding the client and server processing power, and communication requirements of the application under construction. The only difference is the type of corresponding proof that is generated by the server and verified by the client. This brings the flexibility to support multiple membership schemes, and select one based on the state of the system at that time. We will discuss applicable membership schemes in detail, and compare their efficiency in Appendix B.

We use a two-level HADS for implementing an OBD scheme. To store a non-PK column, the distinct values are located in the first level (i.e., each duplicate value will be stored once), and the PK set of each value is located in the second level. The (membership) proof for this scheme consists of two parts: one for proving the existence (or non-existence) of the value in the non-PK column, and one for proving the association of the corresponding PK set with that value.

In our scheme, the client constructs a separate HADS for each searchable column. A one-level HADS (a regular ADS) will be used to store the PK column,

similar to the ones presented in the previous work [16, 24]. An example ADS for storing a PK column using an authenticated skip list is presented in Figure 1a. For non-PK columns, a two-level HADS will be used. A sample HADS for storing the column `major` is illustrated in Figure 2a. It uses an authenticated skip list at the first level, and a Merkle hash tree at the second level. Each modification on the column leads to an update in both ADSs in the hierarchy.

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

Using the authenticated range query for proof generation ensures completeness. To provide correctness, we store along with each record, the hash of that record, $h(record)$. 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 leaves. (The computation of the values of the intermediate nodes, if there exists any, depends on the underlying structure of the ADS in use.) For a PK column, we store $h(record)$, and for non-PK searchable column, we store $h(h(v_i)|h(record)|h(\textit{digest of the corresponding PK set}))$, where v_i values are the distinct values of that column. Storing these hashes together with the elements, binds each PK set to the corresponding value in the column, and to the record.

Upon receipt of a result set (and proof), the client verifies it using the information provided in the proof and hashes of received records. If all records are used and the computed digests are equal to the stored ones, then the client accepts the answer.

Therefore, our construction provides the three properties required for a secure ODB scheme: freshness (by storing digests of the HADSs locally as metadata), correctness, and completeness (guaranteed by the HADSs, as discussed). We prove this formally in Appendix A.

Proof Generation This section provides 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 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.` This case presents a simple range query. The server asks the PK HADS of the `Student` table to compute and return his proof, and sends it back to the client. The proof includes the *before* and *after* records, and all intermediate nodes' values required for reconstructing the proof sub-list by the client. (Note that we need to employ an ADS which supports efficient range query.) Figure 1b 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 clause is on a non-PK column**: A sample query is `SELECT * FROM Student WHERE major = 'CE'`. The server finds the PK set of the value `'CE'` using the HADS storing the *major* column, and adds it to the verification object followed by the proof of membership of the `'CE'` itself (in the first level ADS of the *major* column). The client verifies the HADS using the values in the verification object and the answer (the query result set).

Multi-clause queries: For each clause, the server asks the corresponding HADS to give its proof, collects them into the verification object *vo*, and sends the resulting *vo* to the client. Upon receipt, the client verifies all proofs one-by-one, and accepts if all are verified. If the clauses were connected by 'OR', then each proof verifies a subset of the received records, and the answer should be the union of all these verified records. For 'AND' connectors, each proof verifies a superset of records in the result set. To prevent leakage of records not in the result set, the server sends their hashes to enable the client to verify the proofs. Possible scenarios for the two-clause case are:

– **One clause on the PK, the other on a non-PK column**: For example, the query is `SELECT * FROM Student WHERE StdID > 105 AND major = 'CE'.` Since the order in which the clauses are applied is not important for the proof, we can consider the non-PK clause first, then apply the PK clause on the results of the first step. Therefore, the server first applies the non-PK clause on the first level ADS of the non-PK column's HADS. Then, he applies the PK clause on second level ADSs of the results of the first step. Finally, he adds them both to the *vo*, and sends it to the client.
– **Both clauses on non-PK columns**: A sample query is `SELECT * FROM Student WHERE BCity = 'Istanbul' AND major = 'CE'.` In this case, the server generates two proofs (one for each clause), each containing the first level ADS proof for the value itself and the corresponding PK set, puts them into the *vo*, and sends the resulting *vo* to the client. If the clauses were connected by 'AND', he also puts the hash of missing records into the *vo*. Missing records in this context are the ones that contain either `BCity = 'Istanbul'` or `major = 'CE'` but not both. Note that, we only add hashes of the missing records to the *vo*, but the answer sent by the DBMS does not contain those records and is thus optimal.

Queries with more than two clauses can be handled using a similar logic, depending on whether one of the columns is a PK column or none of them are.

Note that in all our proofs, *we do not require any additional records to be sent to the client on top of the result set of the original query.*

An Illustrative Example To better understand our construction, we provide a simple example. Assume the we use a four-level HADS using the authenticated skip lists to store a database. The first level ADS is the *database ADS* that stores the table names. Each leaf node of this ADS is connected to a *table ADS* at the second level. A table ADS stores the names of a table's searchable columns. Each leaf of a table ADS is connected to a *column ADS* at the third level storing distinct values of that column. Each leaf of the column ADS is linked to a *PK ADS* at fourth level, storing the PK set. This is illustrated in Figure 2b.

The SQL query `SELECT * FROM Student WHERE major in ('CE', 'CS') and BCity = 'Istanbul'` is converted into: `(Student, {(major, {CE, CS}), (BCity, {Istanbul})})`, by the DBAS. With the help of the *Find* algorithm of the HADS that decomposes key-value pairs into the proper parts, the *HCertify* algorithm works as follows: First, it looks for the leaf node at the database ADS storing the `Student` table. That node contains a link to the table ADS storing list of its columns. The algorithm, then, investigates this `Student` table ADS with the input `{(major, {CE, CS}), (BCity, {Istanbul})}`. Now, it should find the leaf nodes storing the values `major` and `BCity`. Using the column ADSs for the `major` and `BCity` columns (one by one), it goes forward to search for values `CE` and `CS` in the ADS of `major`, and `Istanbul` in the ADS of `BCity`. Finally, each column ADS asks all his found PK ADSs (the last level ADSs storing the PK set) to give their proofs. In our example, the `major` column ADS asks to retrieve the PK sets of `CE` and `CS`, and the `BCity` column ADS asks to retrieve the PK set of `Istanbul`. The column ADSs then compute their own membership proofs, concatenate them with the PK sets, and return the result to the table (upper level) ADS who performs the same recursive operations. This is repeated until the top level ADS (the database ADS) is reached, resulting in the full proof to be sent to the client. Verification works similarly, in a bottom-up manner, by verifying the PK set proofs first, followed by the column ADS proofs, then table ADS proof, and finally the database ADS proof. If all proofs verify employing all records in the answer, the client accepts the answer as authentic.

Efficient ODB Construction In Appendix B, we compare the existing ADSs and investigate their eligibility to be used in each level for a two-level construction. It shows that using an authenticated skip list in both levels is the efficient choice. Other alternatives can be chosen regarding the requirements of applications, such as the database being static or dynamic.

5 Performance Analysis

Setup: To evaluate our proposed ODB scheme, we implemented a prototype with the efficient HADS construction which uses a two-level HADS with authenticated skip list at both levels. All experiments were performed on a 2.5GHz machine with 4 cores (but running on a single core), with 4GB RAM and Ubuntu 11.10 operating system. The performance numbers are averages of 50 runs.

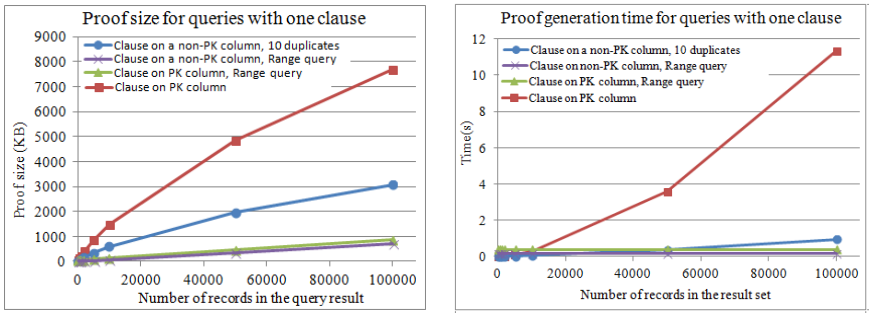
We use a database containing three tables: `Student` and `Course` tables, each with 10^5 randomly-generated records, and `S2C` table storing the courses taken by students, with 10^6 randomly-generated records. There are two scenarios: each registered student has taken 10 courses in the first scenario, and 100 courses in the second scenario, on average. (Not all students are taking courses since we have 10^6 `S2C` records in total.) Each 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) in multiple cases. Since in our scheme proofs are generated using only hashes of values of the column(s) forming the clause (not the whole records), **the proof size is independent of the record size**. Our scheme enhances the efficiency by reducing the computation and proof size, confirmed by experimental results.

One-clause queries: There is only one clause that is on the PK column, or a non-PK column. 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 non-PK column is smaller than the ADS of the PK column. (We do not count the second level ADSs in the one-clause case, since they are included in whole, without any computation to find and select some). The proof generation time and proof size for a non-PK clause is thus smaller compared to the PK clause, as depicted in Figures 4a and 4b. The figures show that the required time and proof size increase very slightly with the result size, for both PK and non-PK columns, if range queries are used.

Two-clause queries: We treat the case with one PK and one non-PK clauses separately from the case with two non-PK clauses. In the first case, we can apply both clauses on the HADS of the non-PK column, by applying the non-PK clause on the first level ADS and the PK clause on the second level ADSs. This is equivalent to applying the non-PK clause first, then applying the PK clause on the results of the first step.

In the second case where both clauses are on non-PK columns, all values of the second level ADSs are included in the result (without further computation), therefore, the dominant factors are the proof generation time and proof size of the first level ADSs. We apply each non-PK clauses on its own HADS and generate two proofs to put in the verification object. Figures 5a and 5b show the proof generation time and proof size for both cases.

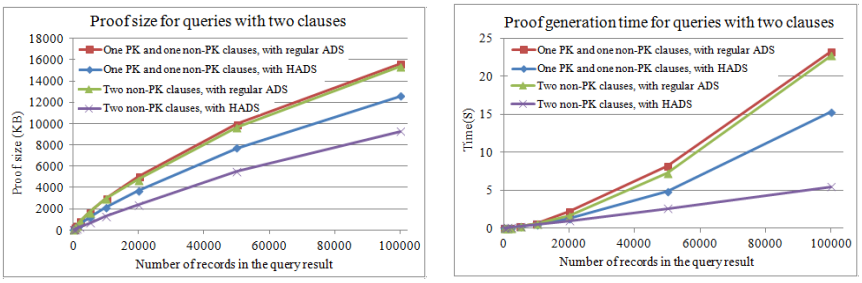


(a) Proof size. (b) Proof generation time.

Fig. 4: Proof generation time and proof size for queries with one clause.

Comparison to previous work Several solutions [18, 10, 14, 17] proposed to make the duplicate values unique, so they can be stored in a regular ADS. Following these solutions, the ADSs of non-PK columns will have the same size, and hence, very close operation costs, as the PK column, since they store the same number of records. Comparing the costs in Figures 4a and 4b, confirms the advantages of the HADS. For one-clause queries, the proof size is reduced about 10%, even with range queries, and the proof generation time is dropped about 50% when HADS is used. Comparing two-clause queries in Figures 5a and 5b for the case with one PK and one non-PK clauses, we observe about 25% reduction in proof size and 40% decrease in proof generation time, and for the case with two non-PK clauses, we observe about 40% reduction in proof size and 65% decrease in proof generation time, when HADS is used.

Multi-clause queries There are more than two clauses in this case, and the two-clause case is a special case of this one. Again, we can separate this case into two cases depending on whether one of the clauses is on the PK column or none of them are. The server asks each HADS sequentially to give its first-level proof. The total proof generation time and proof size of the server is summation of the corresponding values taken by all HADSs.



(a) Proof size.

(b) Proof generation time.

Fig. 5: Proof generation time and proof size for queries with two clauses.

Communication overhead: Another important factor is the overhead of our scheme on the communication, i.e., how much does the proof increase the traffic. As the proof size is independent from the record size, for tables with small record size ($< 1/2KB$) the overhead is close to the query result size. But, for tables with reasonable record size ($\geq 1/2KB$), the proof size falls down (about 1-20%) compared to the result size, and gets smaller as the record size increases.

Boolean combination of clauses Our ODB construction can provably handle selection queries with one or multiple clauses connected by 'OR' or 'AND' connectors. Besides, with reduced use of boundary records, we can easily support clauses formed using the SQL 'IN' operator. This allows us to present proofs for a wide range of database queries.

Acknowledgements

The authors would like to acknowledge the support of TÜBİTAK, the Scientific and Technological Research Council of Turkey, under project numbers 111E019 and 112E115, as well as European Union COST Action IC1206. We also thank Ertem Esiner, Adilet Kachkeev, and Ozan Okumuşoğlu for their contributions during performance evaluation.

References

1. J. Benaloh and M. De Mare. One-way accumulators: A decentralized alternative to digital signatures. In *EUROCRYPT'93*, pages 274–285. Springer, 1994.
2. J. Celko. *Joe Celko's Trees and hierarchies in SQL for smarties*. Morgan Kaufmann, Washington, USA, 2004.
3. P. Devanbu, M. Gertz, C. Martel, and S. Stubblebine. Authentic third-party data publication. *Data and Application Security*, pages 101–112, 2002.
4. G. Di Battista and B. Palazzi. Authenticated relational tables and authenticated skip lists. *Data and Applications Security XXI*, pages 31–46, 2007.
5. C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *CCS'09*, pages 213–222. ACM, 2009.
6. M. Goodrich and R. Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing. *US Patent App*, 10(416,015), 2000.
7. M. Goodrich, R. Tamassia, and J. Hasić. An efficient dynamic and distributed cryptographic accumulator. *Information Security*, pages 372–388, 2002.
8. M. Goodrich, R. Tamassia, and N. Triandopoulos. Super-efficient verification of dynamic outsourced databases. *CT-RSA*, pages 407–424, 2008.
9. F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *ACM SIGMOD*, pages 121–132, 2006.
10. F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Authenticated index structures for aggregation queries. *TISSEC*, 13(4):32, 2010.
11. C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.
12. R. Merkle. A certified digital signature. In *CRYPTO'89*. Springer, 1990.
13. E. Mykletun, M. Narasimha, and G. Tsudik. Providing authentication and integrity in outsourced databases using merkle hash trees. *UCI-SCONCE Tech. Rep.*, 2003.
14. M. Narasimha and G. Tsudik. Authentication of outsourced databases using signature aggregation and chaining. In *Database Systems for Advanced Applications*, pages 420–436. Springer, 2006.
15. G. Nuckolls. Verified query results from hybrid authentication trees. *Data and Applications Security XIX*, pages 924–924, 2005.
16. B. Palazzi. *Outsourced Storage Services: Authentication and Security Visualization*. PhD thesis, Roma Tre University, 2009.
17. B. Palazzi, M. Pizzonia, and S. Pucacco. Query racing: fast completeness certification of query results. In *Data and Applications Security and Privacy XXIV*, pages 177–192. Springer, 2010.
18. H. Pang, A. Jain, K. Ramamritham, and K. Tan. Verifying completeness of relational query results in data publishing. In *ACM SIGMOD*, pages 407–418, 2005.
19. H. Pang and K. Tan. Authenticating query results in edge computing. In *International Conference on Data Engineering*, pages 560–571. IEEE, 2004.

20. C. Papamanthou and R. Tamassia. Time and space efficient algorithms for two-party authenticated data structures. *Info. and Comm. Security*, pages 1–15, 2007.
21. C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In *CCS'08*, pages 437–448. ACM, 2008.
22. R. Tamassia. Authenticated data structures. *Algorithms-ESA*, pages 2–5, 2003.
23. R. Tamassia and N. Triandopoulos. On the cost of authenticated data structures. Technical report, Center for Geometric Computing, Brown University, 2003.
24. J. Wang and X. Du. Skip list based authenticated data structure in das paradigm. In *GCC'09*, pages 69–75. IEEE, 2009.
25. Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis. Authenticated join processing in outsourced databases. In *ACM SIGMOD*, pages 5–18. ACM, 2009.

A ADS Definitions and Security Analysis

Definition 2. ***ADS scheme*** *consists of three polynomial-time algorithms [20]:*

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

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

Verify$(\mathbf{sk}, \mathbf{pk}, \mathbf{cmd}, \mathbf{ans}, \pi, \mathbf{st}) \rightarrow (\{\mathbf{accept}, \mathbf{reject}\}, \mathbf{st'})$ *: is run by the client upon receipt of a response to verify it. 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 the command was a modification command and the proof is accepted, then the client updates her metadata accordingly (to* st'*).*

Definition 3. ***ADS correctness****: For all valid proofs* π *and answers* ans *returned by the server in response to a command issued by the client, the verify algorithm accepts with overwhelming probability.*

Definition 4. ***The ADS security game****: 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. The adversary 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 5. ***Security of ADS****: We say that the ADS is secure if no PPT adversary can win the ADS security game with non-negligible probability.*

Definition 6. *An* ***outsourced database scheme (ODB)*** *consists of three probabilistic polynomial-time algorithms (OKeyGen, OCertify, OVerify) where:*

$\mathbf{OKeyGen}(\mathbf{1^k}) \rightarrow (\mathbf{sk}, \mathbf{pk})$ *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.*

$\mathbf{OCertify}(\mathbf{pk}, \mathbf{cmd}) \rightarrow (\mathbf{ans}, \pi)$ *is run by the server to respond to a command cmd issued by the client. It produces an answer ans and a proof π proving the authenticity of the answer. If the command is a modification command, the answer is empty, and the proof proves that the modification is done properly.*

$\mathbf{OVerify}(\mathbf{pk}, \mathbf{sk}, \mathbf{cmd}, \mathbf{ans}, \pi, \mathbf{st}) \rightarrow (\{\mathbf{accept}, \mathbf{reject}\}, \mathbf{st'})$ *is run by the client upon receipt of the answer ans and proof π, to be verified using the public and private key pair. It outputs an 'accept' or 'reject' notification. If the command was a modification command and the verification result is 'accept', then, the client updates her local metadata (to st'), according to the proof.*

Definition 7. ***ODB security game****: This game is similar to the ADS game (Definition 4), except that proper algorithm names (from ODB scheme) is used.*

Definition 8. ***ODB Security****: We say that an ODB scheme is secure if no PPT adversary can win the ODB security game with non-negligible probability.*

Since the algorithm *OCertify* is used to execute both query and modification commands, the server utilizes it to generate and update the authentication information. It starts with an empty structure, and updates it according to the received modification commands (e.g., the SQL '`Insert`' command).

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

Theorem 1. *The ADS scheme is secure according to Definition 5.*

Proof. It is proved for different schemes separately by different researchers. Papamanthou *et al.* [21] proved the security of the authenticated hash tables,

Goodrich *et al.* [7] proved the security of the RSA one-way accumulator [1] based ADS, and Papamanthou and Tamassia [20] proved the security of the ADSs constructed using authenticated skip list or red black tree.

Theorem 2. *Our HADS construction is secure according to Definition 5 (employing HADS algorithm names) if the underlying ADSs are secure.*

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

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

Theorem 3. *Our ODB scheme is secure according to Definition 8, provided that the underlying HADS scheme is secure.*

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

a local copy), and forwards them to $\mathcal{C}$. If $\mathcal{A}$ passes the ODB verification with non-negligible probability p, then $\mathcal{B}$ can also pass the HADS verification (i.e., break HADS security) with non-negligible probability p.

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

B Efficient ODB Construction

For each level in an HADS, an ADS can be chosen subject to the requirements of that level and the application. Our construction is a two-level HADS, each level having a special role and posing special considerations. We compare the existing ADSs and investigate their eligibility to be used in each level. We consider three classes of ADSs: *logarithmic* (e.g., authenticated skip list [6, 5]), *sublinear* (e.g., authenticated hash tables [21]), and *linear* (e.g., one-way accumulator [1]).

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 ADS who preserves the order of values it stores. One-way accumulator and hash tables does not support this property efficiently, and 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. It requires $O(\log(|C_i|))$ and $O(\log(|C_i|) + |t|)$ time/size for the update and query proofs, respectively. There are $|C_i|$ distinct values, on average, in the first level ADS (stored at leaves), therefore, the storage complexity is $2|C_i|$, which is $O(|C_i|)$.

Second level: This level stores the PK set of values in the first level, where the order of PKs is not a matter of importance (although it can be useful for comparing the PK sets of multiple clauses connected with AND). Thus, any ADS can be used with time/space trade-offs discussed below.

Accumulator: For each distinct value in the first level ADS, an accumulated value is computed using all values in its PK set, and is stored together with the value itself. 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, it suffices to have only the accumulated value (not the witnesses) to check the integrity. But, if want to select a subset of the PK values, then their witnesses are also required.

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 the 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 the validity of the PK set against the accumulated value. Therefore, for a result set of size t, the asymptotic size of the verification object will be $(O(\log |C_i|) + (t|C_i|/N)(1 + N/|C_i|)) \simeq O(\log |C_i| + t)$.

The main problem with the accumulator is the cost of update: with each update, all witnesses should be updated, which is expensive.

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 root at the leaf node itself.

On average, $N/|C_i|$ PK values linked to each leaf node, therefore, we require $O(|C_i|+(1+\epsilon)N/|C_i|*|C_i|) = O(|C_i|+(1+\epsilon)N)) \approx O(|C_i|+N)$ storage in total (including the $O(|C_i|)$ space for the first level). Here $0 < \epsilon < 1$ is a constant.

The first level ADS proof is the same, but the authenticated hash table requires only constant proof size ϵ [20], reaching $(O(\log |C_i|)+1))$ for one record, and $(O(\log |C_i|) + t)$ for t records in the result set. Moreover, hash operations are much faster than accumulator operations using modular exponentiation.

Authenticated Skip list: This is a membership scheme with logarithmic height and proof size. The way the second-level membership schemes are modified, or the proofs are generated, are the same as for the first-level ADS.

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 (including the $2|C_i|$ space for the first level ADS). The proof size and time for one value are both $O(\log |C_i| + \log(N/|C_i|)) = O(\log N)$, and for t values are $O(\log |C_i| + t \log(N/|C_i|))$ and $O(\log |C_i| + t)$, respectively.

A comparison of these schemes is given in Table 1, where the first level is a logarithmic ADS and the second levels are shown in the table. The s, t, t_1, and t_2 denote the number of searchable columns in a table, size of the result set, and number of records in the first and second level ADSs, respectively. Note however that unit operations in the accumulator are more costly than those in the others.

Table 1: A comparison of membership schemes for the second level where the first level is a logarithmic ADS. Proof size and verification time is given for one-dimensional queries. The s, t, t_1, and t_2 denote the number of searchable columns in a table, size of the result set, and number of records in the first and second level ADSs, respectively.

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

	Authenticated skip list
Storage	$2N + (s-1)(2\|C_i\| + 2N)$
Proof size	$2\log \|C_i\| + t_1 + t_1(2\log \|N/C_i\| + t_2)$
Verification	$t_1(\log \|C_i\| + t_2(\log N/\|C_i\|))$
Update	$(\log N + (s-1)(\log \|C_i\| + \log N/\|C_i\|) \approx s \log N$