Data Integrity Verification in Column-Oriented NoSQL Databases

Grisha Weintraub, Ehud Gudes

Data Integrity Verification in Column-Oriented NoSQL Databases

Grisha Weintraub[1] and Ehud Gudes[2]

[1] The Open University, Department of Mathematics and Computer Science, Raanana, Israel
[2] Ben-Gurion University of the Negev, Department of Computer Science, Beer-Sheva, Israel

Abstract. Data integrity in cloud databases is a topic that has received a much of attention from the research community. However, existing solutions mainly focus on the cloud providers that store data in relational databases, whereas nowadays many cloud providers store data in non-relational databases as well. In this paper, we focus on the particular family of non-relational databases — column-oriented stores, and present a protocol that will allow cloud users to verify the integrity of their data that resides on cloud databases of this type. We like our solution to be easily integrated with the existing real-world systems and therefore assume that we cannot modify the cloud; our protocol is implemented solely on the client side. We have implemented a prototype of our solution, that uses Cloud BigTable as a cloud database, and have evaluated its performance and correctness.

Keywords: data integrity, database outsourcing, NoSQL

1 Introduction

For a long time, relational database management systems (RDBMS) have been the only solution for persistent data storage. However, with the phenomenal growth of data, this conventional way of storing has become problematic. To manage the exponentially growing data volumes, the largest information technology companies, such as Google and Amazon, have developed alternative solutions that store data in what have become to be known as NoSQL databases [1, 2]. Some of the NoSQL features are flexible schema, horizontal scaling, and relaxed consistency. Rather than store data in heavily structured tables, NoSQL systems prefer simpler data schema such as key-value pairs or collections of documents. They store and replicate data in distributed systems, commonly across datacenters, thereby achieving scalability and high availability. NoSQL databases are usually classified into three groups, according to their data model: key-value stores, document-based stores, and column-oriented stores. The latter group was inspired by BigTable [3] - a distributed storage system developed by Google that is designed to manage very large amounts of structured data.

In column-oriented stores data is organized in tables, which consist of row keys and column keys. Column keys are grouped into sets called column families.

Column-oriented stores can be used either as internal database systems (just like BigTable inside Google) or as cloud databases - cloud services that provide users with access to data without the need for managing hardware or software. However, storing data in a cloud introduces several security concerns. In particular, since cloud users do not physically possess their data, data integrity may be at risk. Cloud providers (or some malicious entity) can change users' data, omit some of the data from query results or return a version of the data which is not the latest. In other words, data *correctness*, *completeness* and *freshness* might be compromised.

Data integrity in outsourced relational databases has been studied for several years [7-15]. Nevertheless, existing solutions are inappropriate for column-oriented NoSQL databases for the following reasons:

1. Data volumes in NoSQL are expected to be much higher than in RDBMS.
2. Data model of column-oriented systems significantly differs from relational model (e.g. a single row in column-oriented database may contain millions of columns).
3. The query model in NoSQL is much simpler than in RDBMS (e.g. joins are usually not supported).

These differences between RDBMS and NoSQL introduce both challenges and opportunities. On the one hand, data integrity assurance in NoSQL systems requires more sophisticated solutions due to its unusual data model. On the other hand, extremely simple query model of NoSQL may allow us to design much simpler and efficient protocols for data integrity verification. The goal of this paper is to demonstrate that data integrity of column-oriented NoSQL databases in the cloud can be verified better (in terms of efficiency and applicability) than it was proposed in previous work. Our main contributions are as follows:

- Development of a novel probabilistic method that allows users to verify data integrity of the data that resides in cloud column-oriented stores and its analysis.
- A demonstration of the feasibility of our method by a prototype implementation and its experimental evaluation.

The rest of the paper is structured as follows: Section 2 provides background information and overviews related work. Section 3 presents our method for data integrity verification in column-oriented stores. Security analysis of our approach is presented in section 4. Section 5 introduces our proof-of-concept implementation and provides an experimental evaluation thereof. Section 6 presents our conclusions.

2 Background and Related Work

As mentioned earlier, the main goal of this paper is to propose a novel method that provides *data integrity* assurance for *column-oriented* NoSQL databases in the *cloud*. In this section, the relevant background is provided. In particular,

an overview of column-oriented NoSQL systems is presented in Section 2.1, the cloud model is described in Section 2.2 and possible data integrity attacks are discussed in Section 2.3. Section 2.4 reviews related work.

2.1 Column-Oriented Stores

Column-oriented stores, also called wide column stores, extensible record stores, and column-oriented NoSQL databases are storage systems that were built after Google's BigTable [3]. A thorough review of BigTable is given in [4], below is a brief summary.

BigTable is a distributed storage system. Data in BigTable is organized in tables consisting of rows. Rows are composed of cells identified by a combination of a row key and a column key. Column key consists of a mandatory column family and an optional column qualifier. Table 1 provides an example of a typical table in BigTable:

- User ID – is a row key.
- Personal Data and Financial Data – are column families.
- Name, Phone, Email, City and Card – are column qualifiers.
- Bob – is a value located in a cell ("1457","Personal Data: Name")

Table 1: Sample "Users" Table in Column-Orientes Store

User ID	**Personal Data**	**Financial Data**
1457	Phone = "781455", Name = "Bob"	Card = "9875"
1885	Email = "john@g.com", Name = "John"	
2501	Phone="781526", City="NY", Email="k@zzz.com", Name="Alice"	Card = "6652"
3456	Name="Carol", City="Paris"	Card = "6663"

Supported data operations are:

- *get* – returns values from individual rows.
- *scan* – iterates over multiple rows.
- *put* – inserts a value into a specified table's cell.
- *delete* – deletes a whole row or a specified cell inside a particular row.

2.2 System Model and Assumptions

Database-as-a-service paradigm (DBaaS), firstly introduced more than a decade ago [5], has become a prevalent service of today's cloud providers. Microsoft's Azure SQL Database, Amazon's DynamoDB and Google's Cloud BigTable are just a few examples. We assume that there are 3 entities in the DBaaS model (Fig. 1):

- *Data owner (DO)* – uploads the data to the cloud.

– *Cloud provider (CP)* – stores and provides access to the data.
– *Clients* – retrieve the data from the cloud.

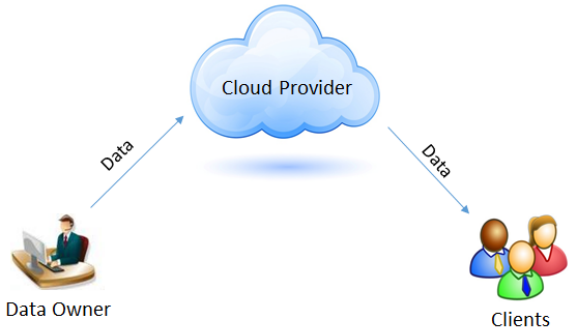

Fig. 1: DBaaS Model

There is only one instance of DO and CP in our model, whereas the number of clients is not limited. The data uploaded to the cloud is stored in a column-oriented NoSQL database and all data operations (the DO's writes and clients' reads) are performed according to the column-oriented data model, described in the previous section. Our system model is both write and read intensive. The DO uploads data to the cloud by bulk loading (the rows in a bulk are not necessarily consecutive). We assume that the DO writes only append new rows to the database; existing rows are not updated. We are interested in a highly applicable solution and therefore we assume that no server changes can be performed on the cloud side.
Some of the use cases that may suit our system model are:

– *Historical financial data* — a stock exchange uploads historical stock prices to the cloud for public use.
– *Open science data* — a scientific organization uploads results of the scientific activities to the cloud for anyone to analyze and reuse.
– *Server metrics* — various server metrics (e.g. CPU, memory and network usage) are periodically uploaded to the cloud for further monitoring and analysis.

2.3 Integrity and Attack Model

We assume that the CP is trusted neither by the DO nor by the clients and that it can behave maliciously in any possible way to break data integrity. For example, the CP (or somebody that had penetrated the CP machine) may modify values of particular table cells, add or delete new rows or column families, or return partial (or empty) results to the clients' queries. We focus on data integrity protection in the following two dimensions:

1. *Correctness* – Data received by the clients was originally uploaded to the cloud by the DO and has not been modified maliciously or mistakenly in the cloud side.
2. *Completeness* – the CP returns to the clients *all* the data that matches the query. In other words, no data is omitted from the result.

Freshness is another important dimension of data integrity, meaning that the clients get the most current version of the data that was uploaded to the cloud. However, since in our system model there are no updates, freshness is not an issue.

2.4 Related Work

Existing solutions can be mainly categorized into three types. The first type is based on the Merkle Hash Tree (MHT), the second is based on digital signatures (DS), and the third uses a probabilistic approach. In the following sections, we review each of these approaches and describe their limitations pertaining to our system model.

2.5 MHT-based approach

MHT [6] is a binary tree, where each leaf is a hash of a data block, and each internal node is a hash of the concatenation of its two children. Devanbu et al. in [7] introduce a method that uses MHT as Authenticated Data Structure (ADS) to provide data integrity assurance in the DBaaS model. The general idea is to build an MHT for every database table such that MHT leaves are hashes of table's records ordered by a search key. To reduce the I/O operations cost in both client and server sides, instead of using binary trees, trees of higher fanout (MB-Trees) can be used [8]. Different MHT-based techniques to provide efficient integrity assurance for join and aggregate queries are presented in [9] and [10] respectively. One of the difficulties in adopting the MHT-based approaches in practice is that RDBMS modification is required for managing ADS on the cloud side. Wei et al. in [11] propose to serialize ADS into RDBMS thereby providing integrity assurance without RDBMS modification.

All of the MHT-based approaches discussed so far assume that the outsourced database is RDBMS. Among other things, it implies that the data is organized in accordance with the relational model and the CP stores the database and the ADS's on a single node. These assumptions are not valid for column-oriented stores and hence another solution is required.

iBigTable [16] is a BigTable [3] enhancement that utilizes an MHT to provide scalable data integrity assurance in column-oriented NoSQL databases. It uses Merkle B+ tree – the combination of MHT and B+ tree as an ADS. iBigTable overcomes the limitations of the MHT-based approaches for RDBMS mentioned above. However, it requires modifications of core components of BigTable and hence cannot be applied directly to existing BigTable implementations. Moreover, since iBigTable relies on BigTable internal architecture it cannot be applied to column-oriented stores that have different infrastructure (e.g. Cassandra).

2.6 DS-based approach

A natural and intuitive approach to provide data integrity in RDBMS is to use the digital signatures scheme (e.g. RSA [17]) in the following way:

- An additional column containing a hash of concatenated record values signed by the DO's private key is added to every table.
- Clients verify record integrity by using the DO's public key.

To reduce the communication cost between client and server and the computation cost on the client side, *signature aggregation* technique [12] can be used to combine multiple record signatures into a single one. To guarantee completeness, rather than sign individual records, the DO signs consecutive pairs of records [13].

Signature based technique for RDBMS uses row-level granularity of integrity – every single row is signed by the DO and hence the smallest unit of the data retrieval is a whole row. Whereas row-level granularity is a natural decision for relational databases, it does not fit column-oriented stores design, where rows may contain millions of column values.

2.7 Probabilistic approach

Probabilistic approaches provide only *probabilistic* integrity assurance, but do not require DBMS modifications and have better performance than MHT-based and DS-based approaches. In this approach, a number of additional records is uploaded to the cloud along with the original records. The more additional records are being uploaded, the higher is the probability to detect data integrity attack. All data is encrypted on a client side so the CP cannot distinguish between the original and the additional records. These additional records may be completely *fake* as was proposed in [14] or original records encrypted with a different (secondary) secret key as was proposed in the *dual encryption* scheme of [15].

Applying probabilistic approach to column-oriented stores raises difficulties similar to those of the signature-based approach. In addition to the granularity of a signature, granularity of additional data should be chosen according to the column-oriented data model. Another limitation is that in probabilistic approach the whole database must be encrypted.

2.8 Summary

In this section we have shown that as of today there is no practical solution for data integrity assurance in column-oriented NoSQL databases; RDBMS approaches cannot be applied directly and the only column stores approach (iBigTable) proposes a customized solution. In the next section, we will introduce our novel approach for data integrity protection in column-oriented NoSQL databases that overcomes the limitations of existing approaches presented above. Some initial ideas of our approach were outlined in [27], but were limited to key-value stores only.

3 Our approach

Inspired by the existing probabilistic approaches for RDBMS [14,15], we propose a novel method that provides probabilistic data integrity assurance in column-oriented NoSQL databases without DBMS modification. In our solution, we also eliminate the main limitation of the existing probabilistic approaches – a requirement that the whole database must be encrypted. Below we describe our protection techniques for correctness and completeness verification.

3.1 Preliminaries

In this section we present the basic notions and concepts necessary for the implementation and analysis of our approach.

Hash function: We use collision-resistant hash function that has a property that it is computationally hard to find two inputs that hash to the same output. SHA-256 and SHA-512 [20] are examples of such functions. Hash operation on value x is denoted by $H(x)$.

Secret keys: We assume that the DO and the clients share two secret keys — one for data encryption and another one for data authenticity.

Data authentication: To verify data authenticity we use message authentication codes (MAC's). The DO signs its data according to the MAC scheme (e.g. HMAC [24]) and stores the MAC value in the cloud along with the signed data. Then, based on the MAC value and the received data, clients can verify data authenticity.

Data encryption: Sensitive data that is stored in the cloud is encrypted by the DO and then decrypted by the clients by using symmetric encryption (e.g. AES [23]).

Bloom filter: Bloom filters [18] are randomized data structures that are used to test whether an element is a member of a set. To reduce storage and network overhead Bloom filters can be compressed [19].

The notation we use is presented in Table 2.

Table 2: Notation

Symbol	Description
$\|\|$	String concatenation
D	A database
r	A database row
cf	A column family
N_{cf}^{r}	Number of column families in row r
r_{key}	Row key of the row r
BF_{cf}	Bloom filter containing all the columns of the family cf
p	Number of rows linked to each DB row

3.2 Correctness

We have to provide correctness assurance to the following types of client queries:

1. Get row by row key.
2. Get column family by row key and family name.
3. Get column(s) by row key, column family and column name(s).

In order to support these queries, we use the MAC scheme in the following way: When the DO inserts a new row, it calculates hash values for all of the column families and stores them in a special "I-META" column family. After that it calculates the *row hash* (hash of the concatenation of all of the column family hashes), calculates the MAC of the row hash, and stores it under the special "Row-Mac" column in "I-META" family. Then the correctness of queries is verified as follows.

Verification of queries of type 1 Correctness of the queries of type (1) is simply based on the computation and verification of the row MAC.

Verification of queries of type 2 Queries of type (2) are verified as follows:

1. Client receives the requested column family and computes its hash value.
2. Client computes the row hash from the column family hash (step 1) and all other column families' hashes from "I-META" family.
3. Client computes and verifies the row MAC

Verification of queries of type 3 There is no trivial solution for queries of type (3). One possible approach would be to transform queries of type (3) into queries of type (2) by omitting column names, and then, after correctness verification for all columns, filter out all the columns but the requested ones. Another way would be to sign each column separately. However, since the number of columns in column stores may be very large, both these approaches are inevitably going to produce an enormous overhead.

In our approach, for queries of type (3) we use Bloom filters. For each column family the DO computes a Bloom filter that contains all column values of this family and stores it as a column in "I-META" family. To verify data correctness of the particular column values, clients retrieve the corresponding Bloom filter and use it to check the existence of the received values. If one of the values does not exist in the Bloom filter, the client can be completely sure that data integrity was compromised. If all the values exist, the client has a certain level of confidence (a detailed analysis is provided in Section 4) that the data was not tampered in the cloud side.

A malicious CP may modify the Bloom filters on the cloud side. To avoid that, we compute hash values for all of the Bloom filters and insert them into the row hash along with the column family hashes. We also add to the row hash the row key, so the CP will not be able to return unrelated data values.

Hence, the formal definitions of the row hash is as follows:

Definition 1. *Row hash*
$H(r) = H(r_{key}||H(cf_1)||H(BF_{cf_1})||H(cf_2)||H(BF_{cf_2})...H(cf_{N^r_{cf}})||H(BF_{cf_{N^r_{cf}}}))$

If we would apply the scheme above to the Table 1 it would look as in Table 3. The row hash of the row 1457 would be computed as follows:
H(Row-Key || Personal-Data-Hash || Financial-Data-Hash || Personal-Data-Bloom-Hash || Financial-Data-Bloom-Hash) = H(1457 || 2873992 || 1976503 || 8703341 || 5848258)

Table 3: Correctness scheme example for data from Table 1

User ID	Personal Data	Financial Data	I-META
1457	Phone = "781455", Name = "Bob"	Card = "9875"	Personal-Data-Hash = "2873992" Financial-Data-Hash = "1976503" Personal-Data-Bloom-Value = "0010010" Financial-Data-Bloom-Value = "1000000" Personal-Data-Bloom-Hash = "8703341" Financial-Data-Bloom-Hash = "5848258" Row-Mac = "A73Djd@83393k"

Note that our solution does not require DBMS modification but only slight schema changes (an addition of the I-META column family).

3.3 Completeness

Rows linking The intuition behind the *rows linking* is that every row *knows* some information about the data stored in some other rows. For example, if we would apply rows linking to the sample data from Table 1, it might look as in Table 4. Row 1457 knows that row 1885 has columns "Email" and "Name" and row 3456 has columns "Name", "City" and "Card". Row 1885 knows what columns exist in rows 2501 and 3456, etc. The formal definition of rows linking is as follows:

Definition 2. *Rows Linking*
$\forall$ *x,y* $\in$ *D, x is linked to y* $\iff$ *x contains y's row key and all its column names*

The DO is responsible for the linking between the rows when uploading the data to the cloud. This *linking data* then encrypted and stored under "I-META" column family. Afterwards, the clients can rely on linking data to verify the result completeness. For example, consider a query "get all users with id between 1000 and 2000" on Table 1 with linking data from Table 4 and the result that contains only row with id 1457. By checking linking data of the row 1457, the client knows that row 1885 should be a part of the result and thus detects the attack. Here too, the addition of linking data does not require any DBMS modification, only an addition of the new column.

To increase the probability of assurance that all inserted rows are present we use the crowdsourced verification technique.

Table 4: Rows linking example for data from Table 1

Row Key	Linking Data
1457	1885:email,name; 3456:name,city,card
1885	2501:phone,email,city,name,card; 3456:name,city,card
2501	1457:phone,name,card; 1885:email,name
3456	1457:phone,name,card; 2501:phone,email,city,name,card

Crowdsourced verification In crowdsourcing (CS) systems [21] users collaborate to achieve a goal that is beneficial to the whole community. The collaboration may be explicit (e.g. Wikipedia and Linux projects) or implicit as in ESP game [22] where users label images as a side effect of playing the game. In our approach we build CS system where users implicitly collaborate to achieve a mutual goal – database integrity assurance. It works as illustrated in Fig. 2:

1. A client sends a query to the CP.
2. The CP sends the query result along with the linking data back to the client.
3. The client builds *verification queries* based on the received linking data and sends them to the CP.
4. The CP sends the result of the verification queries back to the client.
5. The client verifies that the result of the verification queries matches the linking data.

Verification queries (step 3) are built such that the CP cannot distinguish between them and the regular client queries (step 1). Thanks to that, CP's malicious behavior with the client queries will inevitably cause malicious behavior with the verification queries as well and thus will be detected. Note that there is no dependency between the client query (step 1) and the verification queries (step 3) and hence steps 3-5 can be executed asynchronously (i.e. without hurting reads latency).

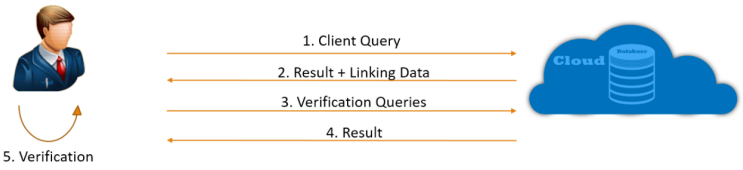

Fig. 2: Crowdsourced Verification

To ensure that all types of queries are covered (without performing all types of queries each time) we build verification queries according to the original query type. So if the original query was of type 1, in order to build verification query we use only row keys from the linking data. For queries of type 2 and 3 we use column families and column names respectively.

4 Security Analysis

4.1 Correctness

In section 3.2 above we presented our technique for data correctness verification. We considered three different types of queries that might be executed by the clients. Verification of the queries of type (1) and (2) is based on provably secure primitives — MAC scheme and collision resistant hash function, and hence provides 100% data correctness assurance.

Correctness verification of the queries of type (3) is based on Bloom filters and hence provides probabilistic correctness assurance. The DO calculates a Bloom filter for each column family cf as follows:

$\forall c \in cf$, $c.name||c.value$ is inserted as a value to BF_{cf}

Clients verify the correctness of the retrieved columns by testing their existence in the corresponding Bloom filter. If a particular column is not present in the Bloom filter, they know for sure that its value was changed. In the opposite direction, since the probability of a false positive for an element not in the Bloom filter is:

$$(1 - (1 - \frac{1}{m})^{kn})^k$$

where k is the number of the used hash functions, m is a Bloom filter size and n is a number of elements in a Bloom filter [19], if it does exist they know that the value is correct with the following probability:

$$1 - (1 - (1 - \frac{1}{m})^{kN_{col}^{cf}})^k$$

where N_{col}^{cf} is a number of columns in a column family.

Parameters m and k are chosen by the DO according to the desired level of the correctness assurance, and storage and computation time constraints. For example, for column family containing $1,000$ columns, $m = 8,192$ and $k = 5$, the client knows with the probability of 98% that the retrieved column value is correct. With a doubled Bloom filter size (i.e. $m = 16,384$) the probability to detect modified column increases to 99.8%.

4.2 Completeness

Our approach for completeness verification is based on two techniques: rows linking and crowdsourced verification, described in Section 3.3. Simply put, rows linking means that existence of every database row is known to p other rows and crowdsourced verification means that, in addition to their regular queries, clients perform verification queries to verify that rows whose existence is known to other rows do actually exist.

Assuming a uniform distribution of both deleted rows (denoted by d) and range of queries (denoted by q), the probability that after a single query the client will not be able to detect that at least one of the d rows was deleted from the database with $|D|$ rows (or omitted from the result) is:

$$\frac{|D|-pd}{|D|}$$

Hence, the probability of detecting an attack after q queries is:

$$1 - (\frac{|D|-pd}{|D|})^q$$

Fig. 6 shows the probability to detect an attack as a function of a number of queries performed by the clients with $|D| = 1,000,000$, $p = 4$ and $d \in \{1, 5, 10, 20\}$. It can be seen that even with p as small as 4, after a relatively small number of queries (production systems receive tens of thousands of queries per second [25]) and deleted rows, the chance of the CP to escape from being caught is very low. It looks like the addition of verification queries should slow client queries, however as it is shown in the next section, since these queries are run asynchronously, they do not hurt reads latency. Some overhead in terms of CPU, I/O, and other resources on both client and server sides is still expected, but since this overhead does not affect user experience, it seems to be a reasonable price for the achieved data integrity protection.

5 Implementation and Experimental Results

For experimental evaluation, we have implemented a prototype of our solution. As a cloud column-oriented store we use Cloud BigTable – Google's NoSQL Big Data database service based on BigTable [3]. To evaluate our solution, we use Yahoo! Cloud Serving Benchmark (YCSB) framework [26]. YCSB is a framework for benchmarking database systems. The only thing which needs to be done to benchmark a database with YCSB is to implement a database interface layer. This will allow a framework client to perform operations like "read row" or "insert row" without having to understand the specific API of the database.

We use YCSB in the following way (see Fig. 3 below):

- YCSB framework already has BigTable client implementation.
- We implemented our version of BigTabgle client (BigTable-I) based on the techniques presented in Section 3. Our implementation is available online [28].
- For performance analysis, we executed different workloads on both clients (BigTable and BigTable-I) and compared their execution time.
- For correctness analysis, we used our client to upload data to the cloud. Then we deleted random rows from the database and performed random read queries until the deletion of the rows was detected.

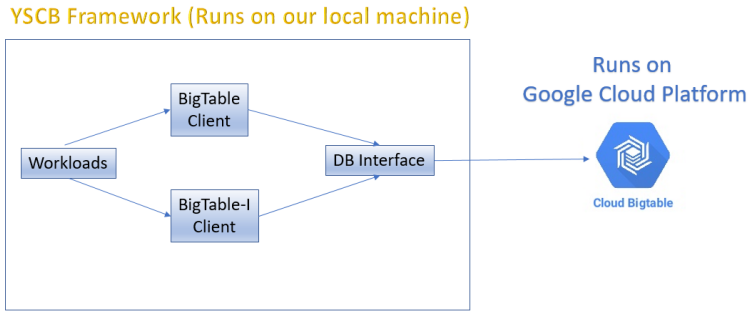

Fig. 3: Experimental evaluation with YCSB framework

5.1 Setup

For our experiments, we created Cloud BigTable cluster with 3 nodes. We configured "zone" to be "europe-west1-b" and storage type to be "HDD". For all workloads, we used random rows of 1KB size (10 columns, 100 bytes each, plus key).

5.2 Performance Analysis

We used the following types of workloads:

- Workload A (Inserts only): Predefined number of rows are inserted into the database.
- Workload B (Reads only): Predefined number of rows are retrieved from the database.

We executed each workload three times for each client with $1,000$, $3,000$ and $5,000$ operations and with parameter p (parameter of the linking factor) set to 4 (runs with the different p values are presented in Section 5.3 below). The results below represent the average value of these three executions.

Workload A (Inserts Only) The cost of Bigtable-I insert operation is dominated by two encryption and one hash operations (MAC calculation and encryption of linking data). Experimental results of workload A (Fig. 4) show that this overhead increases insert execution time by 5% in average for the client.

Workload B (Reads Only) The cost of BigTable-I read operations is similar to the cost of inserts (MAC calculation and linking data decryption) with an additional cost of p verification queries. Since in our implementation we perform verification queries asynchronously, they do not impact read execution time. According to the workload B results (Fig. 5), our protocol increases read execution time by 5% in average for the client.

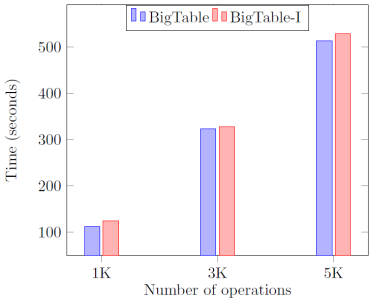

Fig. 4: Workload A results

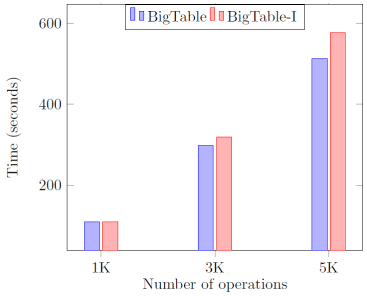

Fig. 5: Workload B results

5.3 Correctness Analysis

To demonstrate that our approach works as expected, we uploaded 10,000 rows to the cloud via BigTable-I client. Then we deleted random rows from the cloud database (up to 20 rows) and ran "random reads" workload until the deletion of rows was detected. We performed this experiment with different values of parameter p - 4, 6 and 8. For each value of p and the number of deleted rows, we ran "random reads" workload 3 times. The results presented in Fig. 7 represent the average value of these three executions. Our experimental results support our security analysis (Section 4.2) – even after a relatively small number of queries and deleted rows the attack detection is inevitable. Only for the rare case of very few deletions, many queries are required.

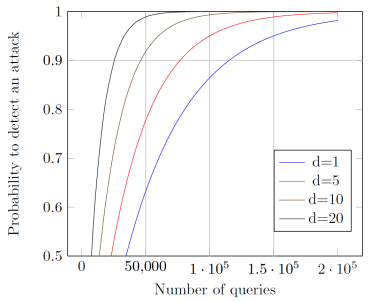

Fig. 6: Completeness verification analysis

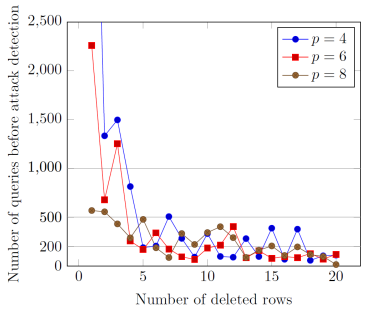

Fig. 7: Correctness analysis results

6 Conclusions and Future Directions

In this paper, we present our novel method for data integrity assurance in cloud column-oriented stores. Our method provides both data correctness and data completeness guarantees. To verify data correctness of rows and column families

we use a MAC scheme similar to the DS-based approach [12], while columns correctness is verified probabilistically by using Bloom Filters [18]. Our method for data completeness verification is inspired by crowdsourcing systems [21] - users collaborate to achieve a mutual goal - database integrity assurance. To the best of our knowledge, our work is the first work that utilizes Bloom Filters and crowdsourcing model for data correctness and data completeness verification, respectively.

The main advantage of our method over existing approaches is its high applicability - it can be applied to existing systems without modifying the server-side. We demonstrate its applicability through a proof-of-concept implementation. While using cloud column-oriented NoSQL database (Cloud BigTable) as a black-box, we have implemented our protocol solely on the client side and conducted experimental evaluation thereof. The experimental results show that our scheme imposes a reasonable overhead (around 5% in average) while it can detect an attack after a relatively small number of client queries (in most cases less than 2% of the database is queried before the attack is detected).

The challenge of data integrity in cloud databases is far from being solved. There are many different types of databases that may be addressed (e.g. document-oriented, graph databases, time-series databases). Different system models may be considered (e.g. multi-data-owner model, multi-cloud model, support for updates and freshness guarantees). For future work, we plan to address some of these challenges by using techniques developed in this work and by investigating other techniques as well.

References

1. Leavitt, N.(2010). Will NoSQL databases live up to their promise?. Computer, 43(2).
2. Cattell, R. (2011). Scalable SQL and NoSQL data stores. ACM SIGMOD Record, 39(4), 12-27.
3. Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., ... & Gruber, R. E.(2008). Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS), 26(2), 4.
4. Weintraub, G.(2014). Dynamo and BigTable — Review and comparison. In Proceedings of the 28th Convention of the Electrical & Electronics Engineers in Israel (pp. 1-5). IEEE.
5. Hacigms, H., Iyer, B., & Mehrotra, S.(2002). Providing database as a service. In Proceedings of the 18th International Conference on Data Engineering (pp. 29-38). IEEE.
6. Merkle, R. C.(1989). A certified digital signature. In Proceedings of the Conference on the Theory and Application of Cryptology (pp. 218-238). Springer, New York, NY.
7. Devanbu, P., Gertz, M., Martel, C., & Stubblebine, S. G.(2003). Authentic data publication over the internet. Journal of Computer Security, 11(3), 291-314.
8. Li, F., Hadjieleftheriou, M., Kollios, G., & Reyzin, L. (2006). Dynamic authenticated index structures for outsourced databases. In Proceedings of the 2006 ACM SIGMOD international conference on Management of data (pp. 121-132). ACM.

9. Yang, Y., Papadias, D., Papadopoulos, S., & Kalnis, P. (2009). Authenticated join processing in outsourced databases. In Proceedings of the 2009 ACM SIGMOD International Conference on Management of data (pp. 5-18). ACM.
10. Li, F., Hadjieleftheriou, M., Kollios, G., & Reyzin, L. (2010). Authenticated index structures for aggregation queries. ACM Transactions on Information and System Security (TISSEC), 13(4), 32.
11. Wei, W., & Yu, T. (2014). Integrity assurance for outsourced databases without DBMS modification. In Proceedings of the 28th conference on Data and Applications Security and Privacy (pp. 1-16). Springer Berlin Heidelberg.
12. Mykletun, E., Narasimha, M., & Tsudik, G. (2006). Authentication and integrity in outsourced databases. ACM Transactions on Storage (TOS), 2(2), 107-138.
13. Narasimha, M., & Tsudik, G. (2005). DSAC: integrity for outsourced databases with signature aggregation and chaining. In Proceedings of the 14th ACM international conference on Information and knowledge management (pp. 235-236). ACM.
14. Xie, M., Wang, H., Yin, J., & Meng, X. (2007). Integrity auditing of outsourced data. In Proceedings of the 33rd international conference on Very large data bases (pp. 782-793). VLDB Endowment.
15. Wang, H., Yin, J., Perng, C. S., & Yu, P. S. (2008). Dual encryption for query integrity assurance. In Proceedings of the 17th ACM conference on Information and knowledge management (pp. 863-872). ACM.
16. Wei, W., Yu, T., & Xue, R. (2013). iBigTable: practical data integrity for bigtable in public cloud. In Proceedings of the third ACM conference on Data and application security and privacy (pp. 341-352). ACM.
17. Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 21(2), 120-126.
18. Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 13(7), 422-426.
19. Mitzenmacher, M. (2002). Compressed bloom filters. IEEE/ACM Transactions on Networking (TON), 10(5), 604-612.
20. Standard, S. H. (2002). FIPS Publication 180-2. National Institute of Standards and Technology (NIST).
21. Doan, A., Ramakrishnan, R., & Halevy, A. Y. (2011). Crowdsourcing systems on the world-wide web. Communications of the ACM, 54(4), 86-96.
22. Von Ahn, L., & Dabbish, L. (2004). Labeling images with a computer game. In Proceedings of the SIGCHI conference on Human factors in computing systems (pp. 319-326). ACM.
23. Pub, N. F. (2001). 197: Advanced encryption standard (AES). Federal Information Processing Standards Publication, 197, 441-0311.
24. Krawczyk, H., Canetti, R., & Bellare, M. (1997). HMAC: Keyed-hashing for message authentication.
25. Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., & Paleczny, M. (2012). Workload analysis of a large-scale key-value store. ACM SIGMETRICS Performance Evaluation Review, 40(1), 53-64.
26. Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., & Sears, R. (2010, June). Benchmarking cloud serving systems with YCSB. In Proceedings of the 1st ACM symposium on Cloud computing (pp. 143-154). ACM.
27. Weintraub, G., & Gudes, E. (2017). Crowdsourced Data Integrity Verification for Key-Value Stores in the Cloud. In Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (pp. 498-503). IEEE Press.
28. BigTable-I-Client. https://github.com/grishaw/ycsb-bigtablei-binding