iBigTable: Practical Data Integrity for BigTable in Public Cloud

Wei Wei
North Carolina State University
890 Oval Drive
Raleigh, North Carolina, United States
wwei5@ncsu.edu

Ting Yu
North Carolina State University
890 Oval Drive
Raleigh, North Carolina, United States
yu@csc.ncsu.edu

Rui Xue
State Key Lab. of Information Security
Institute of Information Engineering
Chinese Academy of Sciences, China
xuerui@iie.ac.cn

ABSTRACT

BigTable is a distributed storage system that is designed to manage large-scale structured data. Deploying BigTable in a public cloud is an economic storage solution to small businesses and researchers who need to deal with data processing tasks over large amount of data but often lack capabilities to obtain their own powerful clusters. As one may not always trust the public cloud provider, one important security issue is to ensure the integrity of data managed by BigTable running at the cloud. In this paper, we present iBigTable, an enhancement of BigTable that provides scalable data integrity assurance. We explore the practicality of different authenticated data structure designs for BigTable, and design a set of security protocols to efficiently and flexibly verify the integrity of data returned by BigTable. More importantly, iBigtable preserves the simplicity, applicability and scalability of BigTable, so that existing applications over BigTable can interact with iBigTable seamlessly with minimum or no change of code (depending on the mode of iBigTable). We implement a prototype of iBigTable based on HBase, an open source BigTable implementation. Our experimental results show that iBigTable imposes reasonable performance overhead while providing integrity assurance.

Categories and Subject Descriptors

H.2 [**Database Management**]: Security, integrity, and protection

General Terms

Security, Design, Algorithms

Keywords

Data Integrity; Cloud Storage; Big Data

1. INTRODUCTION

BigTable [10] is a distributed data storage system designed to scale into the petabyte range across hundreds or even thousands of commodity machines. It has been widely used in several products at Google such as Google Maps, Google Analytics and Gmail. Moreover, in recent years many organizations have adopted the data model of BigTable, and developed their own implementations of BigTable such as HBase [4] and Cassandra [2]. HBase is used to power the messages infrastructure at Facebook [7], and also used as a data storage for Hadoop [3] and MapReduce [11] to facilitate large-scale data processing. Cassandra is used in companies such as Twitter, Cisco and Netflix as a reliable and scalable storage infrastructure.

Running BigTable in a cloud managed by a third party is an economic storage solution to small businesses and researchers who need to deal with data processing tasks over large amount of data but often lack capabilities to obtain their own powerful clusters. However, it introduces several security issues. In particular, if we do not fully trust the cloud provider, we have to protect the integrity of one's data. Specifically, when we retrieve data from BigTable, there should be a way to verify that the returned data from the cloud are indeed what we want, i.e., no data are improperly modified by the cloud, and it has returned exactly the data we request, nothing less, nothing more.

This problem shares a lot of similarities with integrity protection in outsourced databases. Indeed, many techniques have been proposed in the literature to address data integrity issues, including correctness, completeness and freshness. Many of these techniques are based on cryptographic authenticated data structures, which require a database system to be modified [12, 16, 18]. Some others are probabilistic approaches, which do not require to modify existing systems but may inject some fake data into outsourced databases [20, 24, 25]. It seems that we can directly apply existing techniques for database outsourcing to BigTable in the cloud. However, though the two systems share many similarities (e.g., they both host data at an untrusted third party, and support data retrieval), and the principle ideas of integrity verification can be applied, the actual design and deployment of authentication schemes are significantly different, due to several fundamental differences between DBMSs and BigTable. In fact, such differences bring both challenges and opportunities to assure the integrity of BigTable.

For instance, on the one hand, BigTable by design distributes data among large number of nodes. As BigTable horizontally partitions data into tablets across multiple nodes, it is common to merge or split the data of multiple nodes from time to time for load balancing or to accommodate new data. How to handle authenticated data structures during data merging or splitting is not considered in past work on database outsourcing, as it is commonly assumed that data are hosted by a single database. Also, because of the distributed nature of BigTable, it is impractical to store authenticated structures for data residing in different machines into a single node, due to the limited storage capacity of a single node. It also brings scalability issues if we adopt a centralized integrity verification scheme at a single point (e.g., at a trusted third-party). On the other hand, the data model of BigTable is significantly simpler than that of traditional DMBSs. In particular, its query model (or the interface to retrieve data) is extremely simple. For example, it does not support join and other complex query operators as in DBMSs. This may allow us to design much simpler and efficient authenticated structures and protocols to verify data integrity.

Besides integrity verification and efficiency, another important consideration is to preserve the interface of BigTable as much as possible so that existing applications running over BigTable do not need to be re-implemented or modified significantly. Ideally, it should only involve minor change (or no change at all) at the application to enjoy integrity guarantee from BigTable.

In this paper, we present iBigTable, an enhancement to BigTable with the addition of scalable data integrity assurance while preserving its simplicity and query execution efficiency in the cloud. To be scalable, iBigTable decentralizes integrity verification processes among different distributed nodes that participate in data retrieval. It also includes efficient schemes to merge and split authenticated data structures among multiple nodes, which is a must to preserve the scalability and efficiency of BigTable. iBigTable tries to utilize the unique properties of BigTable to reduce the cost of integrity verification and preserve its interface to applications as much as possible. Such properties include its column oriented data model, parallel data processing, and its cache mechanism. Our major contributions are summarized as follows:

- We explore different authenticated data structure designs, and propose a *Two-Level Merkle B+ Tree*, which utilizes the column-oriented data model and achieves efficient integrity verification for projected range queries.
- We design efficient mechanisms to handle authenticated data structure changes for efficient batch update, and tablet split and merge by introducing a *Partial Tree Verification Object*.
- We build a prototype of iBigTable based on HBase [4], an open source implementation of BigTable. The prototype shows that the security components in iBigTable can be easily integrated into existing BigTable implementations.
- We analyze the security and practicability of iBigTable, and conduct experimental evaluation. Our analysis and experimental results show that iBigTable can ensure data integrity while imposing reasonable performance overhead.

Though our discussion in the rest of the paper is for BigTable, the proposed authenticated data structures and integrity verification mechanisms can be similarly applied to distributed storage systems modelled after BigTable.

The rest of the paper is organized as follows. We introduce BigTable in Section 2. In Section 3, we describe the data outsourcing model we target, state assumptions and attack models. Section 4 explains the major design choices of iBigTable, and section 5 illustrate how the data integrity is guaranteed for different data operations. Section 6 discusses the security and practicability of iBigTable and provides the experimental evaluation results. Section 7 compares our work with related work. Finally, the paper concludes in Section 8.

2. BACKGROUND

Bigtable is a distributed storage system for managing structured data. A table in BigTable is a sparse distributed, persistent, multidimensional sorted map [10]. Columns in BigTable is grouped together to form a column family. Each value in BigTable is associated with a row key, a column family, a column and a timestamp, which are combined to uniquely identify the value. The row key, column family name, column name and value can be arbitrary strings. A key-value pair is called a cell in BigTable. A row consists of a group of cells with the same row key. A tablet is a group of rows within a row range specified by a start row key and an end row key and is the basic unit for load balancing in BigTable. In BigTable, clients can insert or delete rows, retrieve a row based on a row key, iterate a set of rows similar to range queries, or only retrieve specific column families or columns over a set of rows similar to projected range queries in databases.

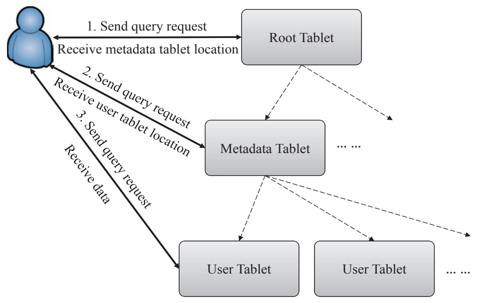


Figure 1: BigTable: Tablet Location Hierarchy.

BigTable consists of a master and multiple tablet servers. It horizontally partitions data into tablets across tablet servers, which achieves scalability. The master is mainly responsible for assigning tablets to tablet servers. Each tablet server manages a set of tablets. Tablet servers handle read and write requests to the tablets that they serve. There are three types of tablets: root tablet, metadata tablet and user tablet. All three types of tablets share the same data structure. There is only one root tablet. The root tablet contains the locations of all metadata tablets. Each metadata tablet contains the locations of a set of user tablets. All user data are stored in user tablets. The root tablet is never split to ensure that the tablet location hierarchy has no more than three levels. Figure 1 shows the tablet location hierarchy and how a query is executed by traversing the tablet location hierarchy, which usually requires three network round-trips (find metadata tablet through the root table, find user tablet through a metadata tablet, and retrieve data from a user tablet) if tablet locations are not found in client-side cache.

3. SYSTEM MODEL

3.1 BigTable in Cloud

BigTable can be deployed in either a private cloud (e.g., a large private cluster), or a public cloud, for example Amazon EC2 [1]. In a private cloud, all entities belong to a single trusted domain, and all data processing steps are executed within this domain. There is no interaction with other domains at all. Thus, security is not taken into consideration for BigTable in a private cloud. However, in a public cloud, there are three types of entities from different domains: cloud providers, data owners, and clients. Cloud providers provide public cloud services. Data owners store and manage their data in BigTable deployed in public cloud. Clients retrieve data owners' data for analysis or further processing. This data processing model presents two significant differences:

- Cloud providers are not completely trusted by the public - data owners and clients. Furthermore, could providers may be malicious or compromised by attackers due to different vulnerabilities such as software bugs, and careless administration.

- The communications and data transmitted between the public and cloud providers are through public networks. It is possible that the communications are eavesdropped, or even tampered to launch different attacks.

Therefore, before BigTable can be safely deployed and operated in a public cloud, several security issues need to be addressed, including confidentiality, integrity, and availability. In this paper, we focus on protecting data integrity of BigTable deployed in a public cloud, which includes three aspects: correctness, completeness and freshness.

Correctness: it verifies if all rows in a query result are generated from the original data set without being tampered. It is generally achieved by verifying signatures or hashes that authenticate the authenticity of the query result.

Completeness: it verifies if all rows in a query result are generated from the original data set without being tampered. It is generally achieved by verifying signatures or hashes that authenticate the authenticity of the query result.

Freshness: it verifies if queries are executed over the up-to-date data. It is challenging to provide freshness guarantees because old data is still valid data at some past time point.

3.2 Assumptions and Attack Models

First, we assume that cloud providers are not necessarily trusted by data owners and clients. Second, we assume that a data owner has a public/private key pair, its public key is known to all, and it is the only party who can manage its data, including data updates and tablet split and merge. Third, we assume that all communications go through a secure channel (e.g., through SSL) between the cloud and clients. Any tampered communication can be detected by both the cloud and clients at each end immediately.

Based on the above assumptions, we concentrate on the analysis of malicious behavior from the public cloud. We do not limit the types of malicious actions a cloud provider may take. Instead, they may behave arbitrarily to compromise data integrity at its side. For example, the cloud can maliciously modify the data or return an incorrect result to users by removing or tampering some data in the result. Moreover, it could just report that certain data does not exist to save its computation and minimize the cost even if the data does exist in the cloud. Additionally, it may initiate replay attacks by returning some old data instead of using the latest data updated by the data owner.

4. SYSTEM DESIGN

In this section, we illustrate the major design of iBigTable, and explain the design choices we make to provide scalable integrity assurance for BigTable. One straightforward way to provide integrity assurance is to build a centralized authenticated data structure. However, data in BigTable is stored across multiple nodes, and may go up to the scale of petabytes. The authentication data could also go up to a very large size. Thus, it is impractical to store authentication data in a single node. Moreover, the single node will become a bottleneck for data integrity verification. To ensure performance and scalability, we propose to build a Merkle Hash Tree (MHT) based authenticated data structure for each tablet in BigTable, and implement a decentralized integrity verification scheme across multiple tablet servers to ensure data integrity of BigTable. Note that we assume that readers have certain knowledge of MHT. If readers are not familiar with MHT, please refer to Appendix A for details.

4.1 Distributed Authenticated Data Structure

BigTable horizontally partitions data into tablets across tablet servers. A natural solution is to utilize BigTable's distributed nature to distribute authenticated data across tablets. Figure 2(a) shows a distributed authenticated data structure design. First, it builds a MHT-based authenticated data structure for each tablet in BigTable, including the root tablet, metadata tablets, and user tablets. Second, it stores the root hash of the authenticated data structure of a tablet along with the tablet location record in its corresponding higher level tablet (either the root tablet or a metadata tablet), as shown in Figure 2(a). Third, the root hash of the root tablet is stored at the data owner so that clients can always retrieve the latest root hash from the data owner for integrity verification. To improve performance, clients may not only cache the location data of tablets, but also their root hashes for efficient integrity verification.

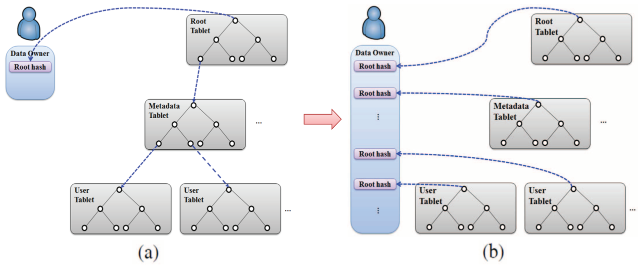


Figure 2: Distributed Authenticated Data Structure Design.

This design distributes authenticated data across tablets, which are served by different tablet servers. To guarantee integrity, it only requires the data owner to store a single hash for the whole data set in BigTable. However, any data update requires authenticated data structure update to be propagated from a user tablet to a metadata tablet and from the metadata tablet to the root tablet. The update propagation process requires either the data owner or tablet servers get involved, either of which complicates the existing data update process in BigTable and downgrades the update performance. Moreover, as the root hash of the root tablet is updated, the root hashes of other tablets cached in clients to improve performance are not valid any more. Thus, clients have to retrieve the latest root hash of the root tablet, and contact tablet servers to retrieve the latest root hashes of other tablets for their requests even if the location data of the tablets stored in metadata tablets or the root tablet is not changed, which hurts read performance.

To address the above issues, we propose a different distributed MHT-based design, which is shown in Figure 2(b). This design also distributes authenticated data across tablets like what the previous design does. But it makes two major design changes. First, it removes the dependency between the authenticated data structures of tablets so that there is no need to propagate an authenticated data update across multiple tablets. In this way, the authenticated data update process is greatly simplified since it does not require either the data owner or tablet servers to propagate any update, which preserve the existing data update protocols in BigTable and minimize communication cost. Second, instead of storing one hash in the data owner, it stores the root hash of each tablet in the data owner, which requires more storage compared with the previous design. However, note that the root hash that the data owner stores for each tablet is only of a few bytes (e.g., 15 bytes for MD5 and 20 bytes for SHA1), while the data stored in a tablet is usually from several hundred megabytes to a few gigabytes [10]. Therefore, even for BigTable with data of petabyte scale, the root hashes of all tablets can be easily maintained by the data owner with moderate storage. Our discussion in the rest of the paper is based on this design.

4.2 Decentralized Integrity Verification

As the authenticated data is distributed into tablets across tablet servers, the integrity verification process is naturally distributed across tablet servers, shown in Figure 3. Like a query execution in BigTable, the query execution with integrity assurance in iBigTable also requires three roundtrip communications between a client and tablet servers in order to locate the right metadata and user tablet, and retrieve data. However, for each round-trip, the client needs a way to verify the data sent by a tablet server. To achieve that, first a tablet server generates a Verification Object (VO) for the data sent to the client, which usually contains a set of hashes. Since the authenticated data for a tablet is completely stored in the tablet server, the tablet server is able to generate the VO without communicating with anyone else, which greatly simplifies the VO generation process and adds no communication cost. Second, the tablet server sends the VO along with the data to the client. Third, when the client receives the data and the corresponding VO, the client runs a verification algorithm to verify the integrity of the received data. One step that is not shown in Figure 3 is that in order to guarantee the freshness of the data, the client needs to retrieve the root hash of the tablet from the data owner on demand or update the cached root hash of the tablet from time to time. How often the client makes such updates depends on the freshness requirement of specific applications, which is a tradeoff between freshness and performance. With the cached root hashes and locations of tablets, the query execution may only require one round-trip between a client and a tablet server, which is exactly the same as that in the original BigTable. This is important as iBigTable strives to preserve the original BigTable communication protocol so that its adoption requires minimum modification to existing BigTable deployment.

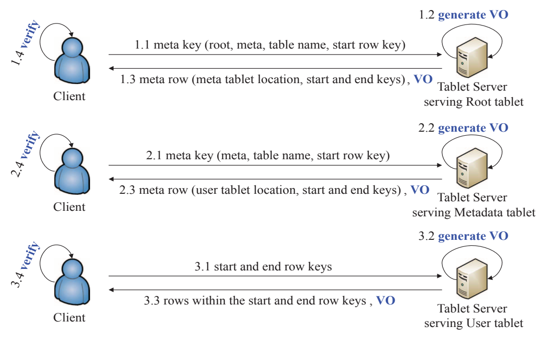

1.4 verify
Client
1.1 meta key (root, meta, table name, start row key)
1.3 meta row (meta tablet location, start and end keys) , VO
1.2 generate VO
Tablet Server serving Root tablet
2.4 verify
Client
2.1 meta key (meta, table name, start row key)
2.3 meta row (user tablet location, start and end keys) , VO
2.2 generate VO
Tablet Server serving Metadata tablet
3.4 verify
Client
3.1 start and end row keys
3.3 rows within the start and end row keys , VO
3.2 generate VO
Tablet Server serving User tablet

Figure 3: Decentralize Integrity Verification.

As can be seen from Figure 3, the major performance overhead in iBigTable comes from three parts: the computation cost at tablet servers for VO generation, the communication cost between clients and tablet servers for VO transmission, and the computation cost at clients for VO verification. We will evaluate and analyze the performance overhead added by the three parts in section 6.3.

Note that although in our design we assume that the data owner as a trusted party stores the root hashes and handles the root hash retrieval requests to guarantee that clients can always get the latest root hashes for freshness verification, many approaches that have been studied extensively in the field of certificate validation and revocation for ensuring the freshness of signed messages can be directly applied to our design, which do not requires that the data owner be online to handle the root hash retrieval requests [16, 17, 25]. For example, the data owner can sign the root hashes with an expiration time and publish those signatures at a place that can be accessed by clients, and reissues the signatures after they are expired. In the rest of the paper, for simplicity, we still assume that the data owner stores the root hashes and handles the root hash retrieval requests.

4.3 Tablet-based Authenticated Data Structure

In BigTable, since all three types of tablets share the same data structure, we propose to design an authenticated data structure based on the tablet structure, and use it for all tablets in BigTable. We compare different authenticated data structures by analyzing how well they can support the major operations provided in BigTable. Authenticated data structure based approaches are mainly divided into two categories, signature aggregation based approaches [16, 18] and Merkle Hash Tree(MHT) based approaches [12, 16]. Although both of them can guarantee correctness and completeness, it is unknown how to efficiently guarantee freshness using signature aggregation based approaches [16]. Moreover, techniques based on signature aggregation incur significant computation cost in client side and much larger storage cost in server side compared with MHT-based approaches [16]. Thus, we focus on exploring MHT-based authenticated data structures in the following.

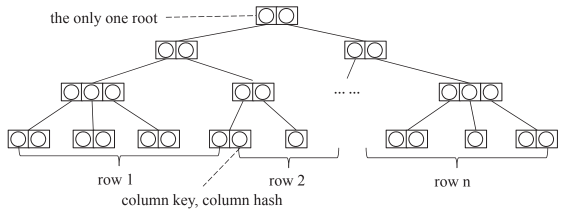


Figure 4: SL-MBT: Single-Level Merkle B+ Tree.

SL-MBT: A single-level Merkle B+ tree. BigTable is column-oriented data storage. Each column value is stored with a key as a key value pair $(key_c, value_c)$, where key_c includes row key and column key specified by column family name and column name. It is straightforward to build a Merkle B+ tree based on all key value pairs in a tablet, which is called SL-MBT shown in Figure 4. In a SL-MBT, all leaves are the hashes of key value pairs. In this way, it is simple to generate a VO for a single column value. Now that all column values are strictly sorted based on row key and column key, the hashes of the key value pairs belonging to a row are adjacent to each other among the leaves of the tree. Thus, it is also straightforward and efficient to generate a VO for a single row query. The same logic can be applied to row-based range queries.

Suppose the fan-out of SL-MBT is f, there are n_r rows in a tablet, each row r_i has n_{cf} column families, and each column family has n_c columns. Then, the height of SL-MBT in the tablet is equal to $h_t = \log_f(n_r \cdot n_{cf} \cdot n_c)$. Say we run a range query with n_q rows returned, where n_q is greater than 0. The height of the partial tree built based on all key value pairs returned equals to $h_p = \log_f(n_q \cdot n_{cf} \cdot n_c)$. The number of hashes H_r returned in the VO is:

$$\begin{aligned} H_r &= (f-1) \cdot (h_t - h_p) \\ &= (f-1) \cdot \log_f(n_r/n_q) \end{aligned} \tag{1}$$

The number of hashes H_c that need to be computed at the client side includes: 1) the number of hashes in the partial tree built based on all received key value pairs; 2) the number of hashes for computing the root hash using hashes in the VO, computed as follows:

$$H_c = \sum_{i=0}^{\log_f(n_q \cdot n_{cf} \cdot n_c)} f^i + \log_f(n_r/n_q) \tag{2}$$

If the range query is projected only to one column, it means that the server only needs to return the column values for n_q rows. To verify those column values, one way we can do is to verify each column value separately. In this case, both H_r and H_c are linear to n_q, which are computed as follows:

$$H_r = n_q \cdot (f-1) \cdot h_t \tag{3}$$
$$H_c = n_q \cdot (1 + h_t) \tag{4}$$

Based on SL-MBT, it is expensive to generate and verify VOs for projected range queries.

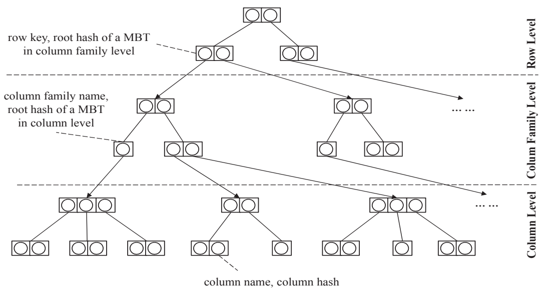

row key, root hash of a MBT in column family level
Row Level
column family name, root hash of a MBT in column level
Column Family Level
Column Level
column name, column hash

Figure 5: ML-MBT: Multi-Level Merkle B+ Tree.

ML-MBT: A multi-level Merkle B+ tree. Different from SL-MBT, ML-MBT builds multiple Merkle B+ trees in three different levels shown in Figure 5:

1. Column Level: we build a Merkle B+ tree based on all column key value pairs within a column family for a specific row, called Column Tree. Each leaf is the hash of a column key value pair. We have one column tree per column family within a row.
2. Column Family Level: we build a Merkle B+ tree, based on all column family values within a row, called Column Family Tree. Each leaf is the root hash of the column tree of a column family. We have one column family tree per row.
3. Row Level: we build a Merkle B+ tree based on all row values within a tablet, called Row Tree. Each leaf is the root hash of the column family tree of a row. We only have one row tree in a tablet.

Given the same range query mentioned above, the H_r in ML-MBT is the same as that returned in SL-MBT. However, H_c in ML-MBT is much smaller than that in SL-MBT, computed as follows:

$$H_c = \sum_{i=0}^{\log_f n_q} f^i + \log_f(n_r/n_q) \tag{5}$$

The partial tree built at the client side for ML-MBT is based on all received rows instead of all received key value pairs. Thus, the number of hashes in the partial tree is much smaller than that for SL-MBT. Compared with SL-MBT, another advantage of ML-MBT is that the client is able to cache the root hashes for trees in different levels to improve the performance of some queries. For example, by caching a root hash of a column family tree, for projected queries within the row, we only need to return hashes from trees under the column family level. Although ML-MBT presents some advantages over SL-MBT, it shares the same disadvantage with SL-MBT for projected range queries.

TL-MBT: A two-level Merkle B+ tree. Considering the unique properties of column-oriented data storage, where a column may not have values for many rows, it seems reasonable to build a column tree based on all values of a specific column over all rows. Due to missing column values in rows, the height of different column trees may be different. Based on this observation, we can also build a column family tree based on all values of a specific column family over all rows. To facilitate row-based queries, we can also build a row tree based on all rows in a tablet. In this way, we may build a Merkle B+ tree for rows, for each column family, and for each column respectively. We call them Data trees. Further, we build another Merkle B+ tree based on all root hashes of Data trees in the tablet, which is called an Index tree. Figure 6 shows the structure of such a two-level Merkle B+ tree. The top level is the Index level where the Index tree is, and the bottom level is the Data level where all Data trees are. Each leaf of Index tree points to a Data tree. Its key is a special value for row tree, the column family name for a column family tree, or the column name for a column tree, and its hash is the root hash of its corresponding Data tree.

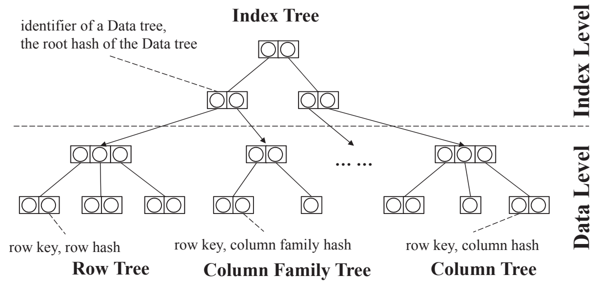


Figure 6: TL-MBT: Two-Level Merkle B+ Tree.

To generate a VO based on TL-MBT, we first need to find all necessary Data trees of a query through the Index tree, which can be done by checking what column families or columns are returned or if the whole row is returned. Second, based on the Index tree and the related Data trees, we use a standard Merkle B+ tree VO generation process to construct a VO for the query. For instance, for row-based range queries, servers only need to find the row tree through the Index tree and use both the Index tree and the row tree to generate a VO, and clients can verify data integrity using the VO efficiently. We argue that although the Index tree increases the total height of the authenticated data structure, its height is relative small since the number of table columns is much less than the number of rows, and the Index tree could be completely cached in both the server side and the client side, which can reduce the communication cost and verification cost. Thus, the performance of TL-MBT is comparable to ML-MBT for row-based queries.

However, it is much more efficient than SL-MBT and ML-MBT for single column projection. Considering the aforementioned range query with single column projection, H_r and H_c in TL-MBT are:

$$H_r = (f-1) \cdot (h_m + \log_f(n_r/n_q)) \tag{6}$$

$$H_c = \sum_{i=0}^{\log_f n_q} f^i + \log_f(n_r/n_q) + h_m \quad (7)$$

In Equation 6 and 7, h_m is the height of the Index tree. Neither of H_r and H_c is linear to n_q. For a projection on multiple columns, we need to verify results for each column separately. In this case, the cost is linear to the number of columns projected in the query. However, compared with SL-MBT and ML-MBT, the update cost may increase by about 3 times since we need to update column tree, column family tree and row tree, which may have the same height, plus the Index tree. We argue that TL-MBT provides a flexible data structure for clients to specify how they want to build such a TL-MBT based on their own needs. For example, if they will never run queries with column-level projection, then it is not necessary to build column trees. In this case, we may only have row tree and column family trees in Data level.

Based on the above analysis, we choose to use TL-MBT as the authenticated data structure for the design of iBigTable.

5. DATA OPERATIONS

Based on TL-MBT, we describe how clients ensure the integrity of the major data operations of BigTable. We address several challenges, including range query across multiple tablets, efficient batch updates, tablet merge and split. In our design, the data owner stores the root hash of each tablet, and any client can request the latest root hash of any tablet from the data owner for integrity verification. Without loss of generality, we assume that clients always have the latest root hashes of tablets for integrity verification.

5.1 Data Access

We start our discussion from range queries[1]. In Section 4.2, we illustrate a general process to run query with integrity protection in iBigTable. The execution of range queries within a tablet follows exactly the same process shown in Figure 3. However, we need to handle range queries across multiple tablets differently. Figure 7 shows a running example for data query and updates. Initially, we have 10 rows with keys from 0 to 90 in a tablet. Figure 7(a) and 7(b) show the initial MB+ tree for the tablet and the result returned for a range query from 15 to 45 including data and VO. We will explain in detail of major operations based on the running example.

Range Queries Across Tablets. To provide integrity assurance for range queries across tablets, it is necessary to retrieve authenticated data from different tablets since the authenticated data structure built for a tablet is only used to verify integrity of data within the tablet. We observe that to handle a range query across tablets, the range query is usually split into multiple sub-queries, each of which retrieves rows from one tablet. More importantly, the query ranges of the sub queries are continuous since the query split is based on the row range that each tablet serves, which is stored along with the tablet location in a meta row as shown in Figure 3. Suppose that there are two tablets, one serves rows from 1 to 10 and the other serves rows from 11 to 20, and a range query is to retrieves rows from 5 to 15 across the two tablets. In this case, the query is splits into two sub queries with query ranges from 5 to 10 and from 11 to 15. In this way, we can apply the same process of range query answering within a tablet to guarantee integrity for each sub query.

However, the completeness of the original range query across tablets may not be guaranteed since a tablet server may return a wrong row range for a user tablet, which results in an incomplete result set returned to clients. Thus, we want to make sure that the row range of the user tablet is correctly returned. During the query verification process, it is guaranteed by the verification of meta row performed by clients because the row range of a user tablet is part of the data of the meta row, which has been authenticated. It is also why we not only need to guarantee integrity of data stored in user tablets, but also data stored in the root and metadata tablets.

[1]A single row query as a special case of range query can be handled in the same way that a range query is executed.

Single Row Update. In iBigTable, we support integrity protection for dynamic updates such as insertion, modification and deletion. Due to the space limit, we focus on discussing how to insert a new row into the data storage, which covers most of aspects of how modification and deletion are handled. Insertion shares the same process to find the user tablet where the new row should be inserted based on the key of the new row. Here we do not reiterate this process again. The rest of the steps to insert a row into the user tablet are shown in Figure 8.

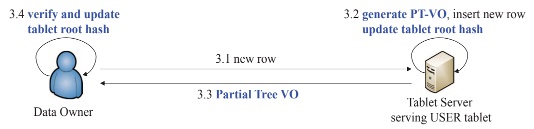

3.4 verify and update tablet root hash
3.2 generate PT-VO, insert new row update tablet root hash
3.1 new row
3.3 Partial Tree VO
Data Owner
Tablet Server serving USER tablet

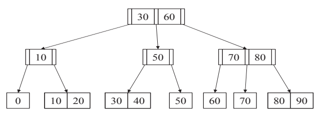
Figure 8: Single Row Insert with Integrity Protection.

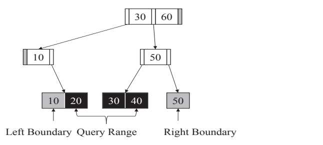
Here, we introduce a new type of VO called *Partial Tree Verification Object* (PT-VO). The difference between a VO and a PT-VO is that a PT-VO contains keys along with hashes, while a VO does not. With those keys, a PT-VO allows us to insert new data within the partial tree range directly. Thus, when the data owner receives a PT-VO from the tablet server, it can directly update the PT-VO locally to compute the new root hash of the original authenticated data structure. Figure 7(c) shows the PT-VO returned for an insertion at key 45. As can be seen from Figure 8, an insertion with integrity protection is completed without additional round-trip, and its integrity is guaranteed since the verification and the root hash update are done directly by the data owner.

Efficient Batch Update. In iBigTable, we can execute a range query to retrieve a set of rows at one time and only run verification once. Motivated by this observation, we think about how we can do a batch update, for example inserting multiple rows without doing verification each time a new row is inserted. We observe the fact from single row update that the data owner is able to compute the new root hash based on a PT-VO. Based on this observation, we propose two simple yet effective schemes to improve the efficiency of insertions for two different cases.

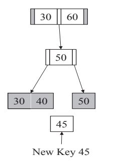
In the first case where we assume that the data owner knows within which range new rows falls, the data owner can require servers to return a PT-VO for the range before any insertion really happens. Any new row falling into the range can be inserted directly without requiring the server to return a VO again because the data owner is able to compute the new root hash of the tablet with the PT-VO and the new row. Thus, even if 1000 rows are inserted within this range, no additional VO needs to be returned for them. But both the data owner and tablet servers have to update the root hash locally, which is inevitable in any case.

In the second case where we assume that a PT-VO for a range is already cached in the data owner side, the data owner does not need to request it. As long as we have the PT-VO for a range, we do not need any VO from servers if we insert rows within the range. For example, Figure 7(b) and 7(d) show such an example. First, the data owner runs a range query from 15 to 45 with a request for a PT-VO instead of a VO without keys. Then, the data owner inserts a row with key 45. In this case, there is no need requiring any VO from the tablet server for the insertion.

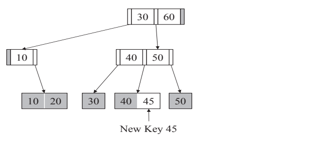

(a) Initial MB+ row tree.



(b) Range query from 15 to 45.

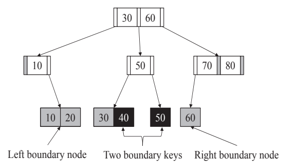


(c) Insert a row with key 45.

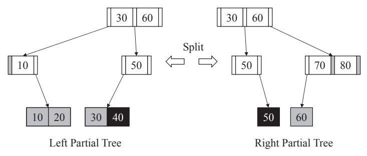


(d) Insert after range query.

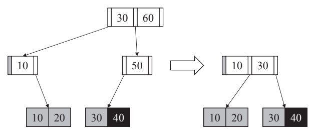
Figure 7: A running example.



(a) VO returned.



(b) Split it into two partial trees.


(c) Adjust left partial tree.

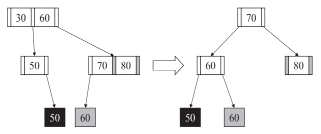

(d) Adjust right partial tree.

Figure 9: Split the tablet at key 45.

5.2 Tablet Changes

As the size of a tablet changes, the data owner may want to split a tablet or merge two tablets for load balancing. For both tablet split and merge, we need to rebuild an authenticated data structure and update the root hashes for newly created tablets. One straightforward way is to retrieve all data hashes in tablets involved and compute new root hashes for newly created tablets in the data owner side. However, this incurs high communication and computation overhead in both the data owner side and tablet servers. In the following, we explain how we can efficiently compute the root hashes for newly created tablets when tablet split or merge happens. For simplicity, we assume that there is only one Data tree and no Index tree in tablets when we discuss tablet split or merge, since the Index tree of TL-MBT in a tablet can be rebuilt based on Data trees. Further, all Data trees are split or merged in the same way.

Tablet Split. Regarding tablet split, a straightforward way is to split the root of each tree and form two new roots using its children. For example, given the current root we can split it in the middle, and use the left part as the root of one tablet and the right part as the root of the other tablet. In this way, to split a Data tree and compute new root hashes for newly created tablets, the data owner only needs to retrieve the hashes of children in the root of the Data tree from an appropriate tablet server.

The main advantage of the above approach is its simplicity. It can be easily implemented. However, splitting at the middle of the root (or any pre-fixed splitting point) prevents us from doing flexible load balancing dynamically based on data access patterns and work loads. Here, we propose a simple yet effective approach to allow the data owner to specify an arbitrary split point for a tablet (instead of always along one child of the root), which can be any key within the range served by the tablet. The approach works as follows: 1) The data owner sends a tablet split request with a key as the split point to the appropriate tablet server. For example, the data owner splits the previous tablet at key 45; 2) The server returns a VO for the split request to the data owner shown in Figure 9(a). The VO for split not only contains all data in a PT-VO, but also includes keys and hashes of the left and right neighbors of each left-most or right-most node in the PT-VO; 3) When the data owner receives the VO, the data owner splits it into two partial trees shown in Figure 9(b). The left tree contains all keys less than the split key, and the right tree contains all keys larger than or equal to the split key; 4)

Algorithm 1 Adjust left partial tree VO

```
Require: $T_l$ {the left partial tree}
Ensure: $T_a$ {the adjusted left partial tree}
 1: $p \leftarrow GetRoot(T_l)$
 2: while $p \neq null$ do
 3:     remove any key without right child in $p$
 4:     $p_{rm} \leftarrow$ the rightmost child of $p$
 5:     if IsValidNode($p$) is false then
 6:         $p_{ls} \leftarrow$ the left sibling of $p$
 7:         if CanMergeNodes($p$, $p_{ls}$) then
 8:             merge $p$ and $p_{ls}$
 9:         else
10:             shift keys from $p_{ls}$ to $p$ through their parent
11:         end if
12:     end if
13:     $p \leftarrow p_{rm}$
14: end while
15: return $T_a \leftarrow T_l$
```

The data owner adjusts both trees using two similar processes and computes the root hashes for the two new tablets. The adjusted trees are shown in both Figure 9(c) and 9(d). Due to the similarity of adjustments for both trees and space limit, we only describe the process for left tree in Algorithm 1.

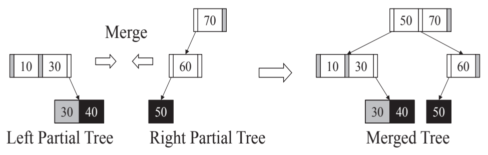


Figure 10: Tablet Merge.

Tablet Merge. Tablet merge is a reverse process of table split. It tries to merge two continuous tablets into a new one. As two tablets merge, we need to merge the authenticated data structures. Motivated by the tablet split approach and the Partial Tree VO, we describe an efficient tablet merge approach as follows: 1) The data owner sends a tablet merge request to the appropriate tablet servers serving two continuous tablets to be merged; 2) The server serving the tablet with smaller keys returns a VO for its largest key, and the server serving the tablet with larger keys returns a VO for its smallest key, which are shown in Figure 10; 3) When the data owner receives two VOs for the two tablets, it directly merges them into one using the process described in Algorithm 2. Then, the data owner computes the root hash for the new tablet based on the merged VO.

Our discussion focus on the tablet-based authenticated data structure split and merge at the data owner side. The same process can be applied at the server side.

Algorithm 2 Merge two partial tree VOs

```
Require: $T_l$ and $T_r$ {represent two partial trees separately}
Ensure: $T_m$ {the merged partial tree}
 1: $k \leftarrow$ the least key in $T_r$
 2: $h_l \leftarrow GetHeight(T_l)$
 3: $h_r \leftarrow GetHeight(T_r)$
 4: $h_{min} \leftarrow GetMin(h_l, h_r)$
 5: if $h_l \leq h_r$ then
 6:     $p_{lm} \leftarrow$ the leftmost node in $T_r$ at $h_{min}$
 7:     add $k$ to $p_{lm}$
 8:     $p_{merged} \leftarrow$ merge the root of $T_l$ and $p_{lm}$
 9:     if IsValidNode($p_{merged}$) is false then
10:         run a node split process for $p_{merged}$
11:     end if
12:     return $T_m \leftarrow T_r$
13: else
14:     $p_{rm} \leftarrow$ the rightmost node in $T_l$ at $h_{min}$
15:     add $k$ to $p_{rm}$
16:     $p_{merged} \leftarrow$ merge the root of $T_r$ and $p_{rm}$
17:     if IsValidNode($p_{merged}$) is false then
18:         run a standard node split process for $p_{merged}$
19:     end if
20:     return $T_m \leftarrow T_l$
21: end if
```

6. ANALYSIS AND EVALUATION

6.1 Security Analysis

We analyze in the following how iBigTable achieves the three aspects of data integrity protection.

Correctness. In iBigTable, the Merkle tree based authenticated data structure is built for each tablet, and the root hash of the authenticated data structure is stored in the data owner. For each client request to a tablet, a tablet server serving the tablet returns a VO along with the data to the client. The client is able to compute a root hash of the tablet using the VO and the data received. To guarantee the integrity of the data received, the client needs to retrieve the root hash of the tablet from the data owner, and then compare the root hash received from the data owner and the computed root hash. The comparison result indicates if the data has been tampered. Thus, the correctness of iBigTable is guaranteed. Any malicious modification can be detected by the verification process.

Completeness. The completeness of range queries within a tablet is guaranteed by the MHT-based authenticated data structure built for each tablet. For range queries across tablets, each of them is divided into several sub-range queries with continuous range based on the original query range and data range served by tablets so that each sub-range query only queries data within a tablet. Thus, the completeness of range queries across tablets is guaranteed by two points: 1) the sub-range queries are continuous without any gap; 2) the completeness of each sub-range query is guaranteed by the authenticated data structure of its corresponding tablet.

Freshness. In iBigTable, the data owner is able to compute the new root hash of the authenticated data structure for a tablet when any data in the tablet is updated. Thus, clients can always get the latest root hash of a tablet from the data owner to verify the authenticity of data in the tablet. Even though there is no data update to any tablet, tablet split or merge may happen since a tablet may become a bottleneck because of too much load or for better tablet organization to improve performance. In this case, iBigTable also enables the data owner to compute the new root hashes for new tablets created during the split or merge process to guarantee the freshness of tablet root hashes, which is the key for freshness verification.

6.2 Practicability Analysis

We argue that iBigTable achieves simplicity, flexibility and efficiency, which makes it practical as a scalable storage with integrity protection.

Simplicity. First, we add new interfaces and change part of existing implementation to achieve integrity protection while keeping existing BigTable interface, which enables existing applications to run on iBigTable without any change. Second, iBigTable preserves BigTable existing communication protocols while providing integrity verification, which minimizes modification to existing BigTable deployment for its adoption.

Flexibility. We provide different ways to specify how and when clients want to enable integrity protection. Existing client applications can enable or disable integrity protection by configuring a few options without any code change, and new client applications may use new interfaces to design a flexible integrity protection scheme based on specific requirements. There is no need to change any configuration of iBigTable servers when integrity protection is enabled or disabled at the client side.

Efficiency. We implement iBigTable without changing existing query execution process, but only attach VOs along with data for integrity verification. We make the best use of cache mechanisms to reduce communication cost. We introduce the *Partial Tree Verification Object* to design a set of mechanisms for efficient batch updates, and for efficient and flexible tablet split and merge.

Note that though iBigTable only allows the data owner to modify data, most applications running on top of BigTable do not involve frequent date updates. So it is unlikely that the data owner becomes a performance bottleneck.

6.3 Experimental Evaluation

System Implementation. We have implemented a prototype of iBigTable based on HBase [4], an open source implementation of BigTable. Although HBase stores data in a certain format and builds indexes to facilitate the data retrieval, we implement the authenticated data structure as a separated component loosely coupled with existing components in HBase, which not only simplifies the implementation but also minimize the influence on the existing mechanisms of HBase. We add new interfaces so that clients can specify integrity options in a flexible way when doing queries or updates. We also enable them to configure such options in a configuration file in the client side without changing their application code. Besides, we add new interfaces to facilitate the integrity protection and efficient data operations. For example, for efficient batch updates a client may want to pre-fetch a PT-VO directly based on a range without returning actual data. Finally, iBigTable automatically reports any violation against data integrity to the client.

Experiment Setup. We deploy HBase with iBigTable extension across multiple hosts deployed in Virtual Computing Lab (VCL), a distributed computing system with hundreds of hosts connected through campus networks [8]. All hosts used have similar hardware and software configuration (Intel(R) Xeon(TM) CPU 3.00GHz, 8G Memory, Red Hat Enterprise Linux Server release 5.1, Sun JDK 6, Hadoop-0.20.2 and HBase-0.90.4). One of the hosts is used for the master of HBase and the NameNode of HDFS. Other hosts are running as tablet servers and data nodes. We consider our university cloud as a public cloud, which provides the HBase service, and run experiments from a client through a home network with 30Mbps download and 4Mbps upload. To evaluate the performance overhead of iBigTable and the efficiency of the proposed mechanisms, we design a set of experiments using synthetic data sets we build based on some typical settings in BigTable [10]. We use MD5 [6] to generate hashes.

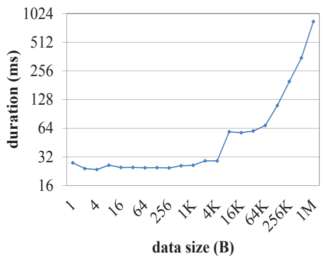


Figure 11: Time to Receive Data from Server.

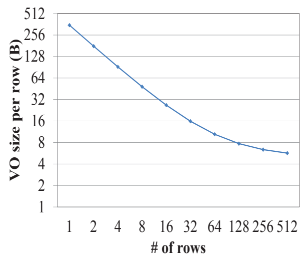


Figure 12: VO Size Per Row vs Number of Rows.

Baseline Experiment. Before we evaluate the performance of write and read operations in iBigTable, we run a simple data transmission through the targeting network because the result is going to be helpful to understand the performance result later. In the experiment, the client sends a request to a server for a certain amount of data. The server generates the amount of random data and sends the data back to the client. Figure 11 shows the time to receive a certain amount of data from a server using logarithmic scale. The result shows that it almost takes the same time to transmit data less than 4KB, where the network connection initialization may dominate the communication time. The time is doubled from 4KB to 8KB till around 64KB. After 64KB, the time linearly increases, which is probably because the network is saturated.

To understand how the VO size changes for range queries, we run an experiment to quantify the VO size for different ranges based on a data set with about 256MB data, which is the base data set for later update and read experiments. Figure 12 shows the VO size per row for different sizes of range queries. For a range with more than 64 rows, the VO size per row is around 10 bytes. Although the total VO size increases as the size of range queries increases, the VO size per row actually decreases, shown in Figure 12 with logarithmic scale.

Write Performance. Regarding different data operations, we first conduct experiments to evaluate the write performance overhead caused by iBigTable, where we sequentially writes 8K rows into an empty table with different write buffer sizes. The data size of each row is about 1KB. Figure 13 shows the number of rows written per second by varying the write buffer size for HBase, iBigTable and iBigTable with Efficient Update (EU). The result shows the performance overhead caused by iBigTable ranges from 10% to 50%, but iBigTable with EU only causes a performance overhead about 1.5%, and the write performance increases as the write buffer size increases. Figure 14 shows the breakdown of performance overhead introduced by iBigTable, which shows the client computation overhead, the server computation overhead and the communication overhead between client and server. As can be seen from the figure, the major performance overhead comes from transmitting the VOs. The computation overhead from both client and server ranges from 2% to 5%.

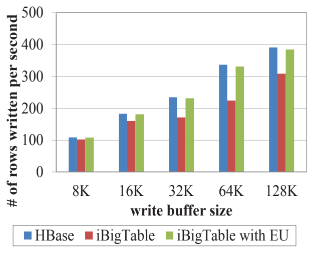


Figure 13: Write Performance Comparison between HBase and iBigTable.

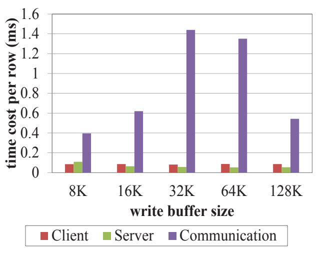


Figure 14: The Breakdown of iBigTable Write Overhead.

Based on our observation, the large variation of performance overhead is caused by the network transmission of VOs generated by iBigTable for data integrity protection. Although the total size of VOs generated for different write buffer sizes is the same, the number of data with VOs transmitted in each remote request is different in different cases. Different sizes of data may cause a different delay of network transmission, but it may not be always a case, which is shown in Figure 11. iBigTable with EU shows a great performance improvement since there is at most one time VO transmission in this case, and the major overhead of iBigTable with EU is the client-side computation overhead of computing the new root hash for newly inserting data, which is very small, compared with iBigTable.

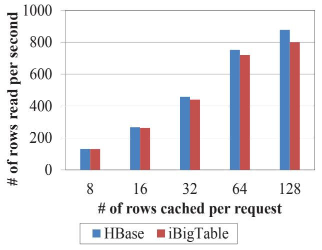


Figure 15: Read Performance Comparison between HBase and iBigTable.

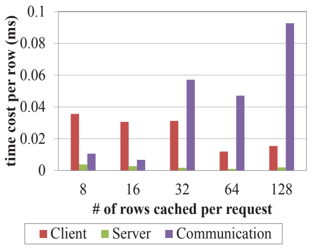


Figure 16: The Breakdown of iBigTable Read Overhead.

Read Performance. We also run experiments to evaluate the read performance overhead of iBigTable. Figure 15 shows how the number of rows read per second changes based on different number of rows cached per request for a scan. The result shows that the read performance overhead ranges from 1% to 8%. Figure 16 shows the breakdown of iBigTable read overhead. The communication overhead can be explained by the result shown in Figure 11 because the total amount of data transmitted for the first two cases ranges from 8KB to 32KB. In the rest of cases, the size of data is about or larger than 64KB, which results in a large network delay for data transmission. In this case, as the VO size increases, the communication overhead becomes more visible. Based on our observation from experiments, the computation overhead of both the client and servers is about 1%. Still, the major part of performance downgrade is caused by the variation of data transmission between the client and servers.

TL-MBT Performance. To illustrate how TL-MBT affects the performance, we execute a single column update of 16K rows on an existing table with about 30GB data across all tablets, each of which has 256MB data or so. The experiments run against different authenticated data structure configurations of TL-MBT: Row Level, Column Family Level and Column Level, which decides what data trees we build for a tablet. For example, regarding Column Level, we build trees for rows, each column family and each column. It means that for a single column update, we need to update four authenticated trees, which are row tree, column family tree, column tree and Index tree. Due to the small size of Index tree, the VO size of Column Level is roughly tripled compared with those of Row Level, and the client-side computation and server-side computation overhead are about triple too. Figure 17 shows the number of rows updated per second versus the write buffer size for three different configurations of TL-MBT. The result indicates that as the number of trees that need to be updated increases, the performance decreases dramatically in some cases. The major reason for the performance downgrade is still caused by the additional data transmitted for data verification.

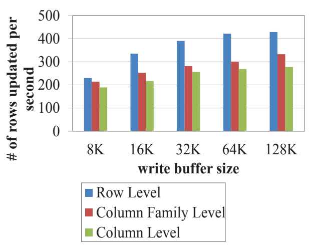


Figure 17: TL-MBT Update Performance in iBigTable.

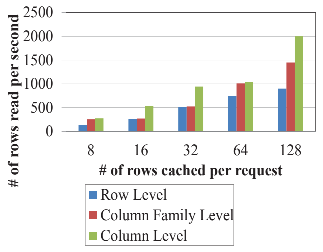


Figure 18: Projection Query with TL-MBT Support in iBigTable.

We also evaluate the read performance for projected queries in iBigTable with TL-MBT by executing a single column scan for 16K rows. For TL-MBT with Row Level, even though we only need a single column value, we still need to retrieve the whole row for data verification in the client side. Similarly, we need to retrieve the whole column family for TL-MBT with Column Family Level. Thus, the TL-MBT with Column Level minimizes the communication overhead since there is no need to transmit additional data for data verification for column projection. Although its computation cost is the lowest one among three different configurations, the major difference is the size of data transmitted between the client and servers. Figure 18 shows how much the TL-MBT can improve the read efficiency for projected queries. Due to the network delay for different data sizes, we see the large performance variation for different cases in the figure.

Overall, the computation overhead in both client and servers for different cases ranges from 1% to 5%. However, the generated VOs for data verification may affect the performance to a large extent for different cases, which depends on how much data is transmitted between the client and servers in a request. Due to the space limit, we do not analyze the performance overhead for tablet split and merge. In general, since the performance overhead for tablet split and merge only involves several hashes transmission and computation, it is negligible compared with the time needed to complete tablet split and merge, which involves a certain amount of data movement across tablet servers.

7. RELATED WORK

Several open-source, distributed data storages have been implemented modelled after BigTable, for example HBase [4] and Hypertable [5], which are widely used for both academia research and commercial companies. Carstoiu et al. [9] focused on the performance evaluation of HBase. You et al. [27] proposed a solution called HDW, based on Google's BigTable, to build and manage a large scale distributed data warehouse for high performance OLAP analysis. Few work pays attention to the data integrity issue of BigTable in a public cloud. Although Ko et al. [21] mentioned the integrity issues of BigTable in a hybrid cloud, no further discussion on a practical solution was elaborated.

Data integrity issues have been studied for years in the field of outsourcing database [12–14, 18, 19, 24]. Different from traditional database, BigTable is a distributed data storage system involving multi-entity communication and computation, which presents challenges to directly adopt any of existing authenticated data structure. Xie et al. [24] proposed a probabilistic approach by inserting a small amount of fake records into outsourced database so that the integrity of the system can be effectively audited by analyzing the inserted records in the query results. Yang et al. [26] discussed different approaches to handling some types of join queries for outsourced database, which is not relevant to the query model of BigTable. Xie et al. [25] analyzed the different approaches to provide freshness guarantee over different integrity protection schemes, which is complementary to our work for BigTable.

Additionally, Lee et al. [15] proposed algorithms to attach branch to or remove branch from B+ tree for self-tuning data placement in parallel database system, but the branch attaching algorithm is only for two branches that have the same height, and the branch removal algorithm is not flexible because no split point can be specified. Sun et al. [22] designed algorithm to merge B+ tree covering the same key range, which is different from our problem since the row range in each tablet is non-overlapping. Zhou et al. [23] discussed data integrity verification in the cloud, and proposed an approach called partitioned MHT (P-MHT) that may be applied to data partitions. But it may not be scalable since when an update happens to one data partition, the update has to be propagated across all data partitions to update the P-MHT, which renders it as an impractical solution for BigTable. To the best of our knowledge, iBigTable is the first work to propose a practical solution to address the unique challenges and ensure the data integrity for running BigTable in a public cloud.

8. CONCLUSION AND FUTURE WORK

In this paper, we have presented iBigTable, which enhances BigTable with scalable data integrity assurance while preserving its simplicity and query execution efficiency in public cloud. We have explored the practicality of different authenticated data structure designs for BigTable, designed a scalable and distributed integrity verification scheme, implemented a prototype of iBigTable based on HBase [4], evaluated the performance impact resulted from the proposed scheme, and tested it across multiple hosts deployed in our university cloud. Our initial experimental results show that the proposed scheme can ensure data integrity while imposing reasonable performance overhead.

As a storage system, BigTable is often used in conjunction with MapReduce [11] for big data processing. In future, we plan to investigate on the integrity protection of MapReduce with iBigTable for secure big data processing.

9. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their helpful suggestions. This work is partially supported by the U.S. Army Research Office under grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI), by the NSF under grants CNS-0747247 and CCF-0914946, by NSFC under Grants No. 61170280, and SPRPCAS under Grant No. XDA06010701, and by K.C. Wong Education Foundation. The contents of this paper do not necessarily reflect the position or the policies of the U.S. Government.

10. REFERENCES

[1] Amazon Elastic Compute Cloud. *http://aws.amazon.com/ec2/*.

[2] Cassandra. *http://cassandra.apache.org/*.

[3] Hadoop Tutorial. *http://public.yahoo.com/gogate/hadoop-tutorial/start-tutorial.html*.

[4] Hbase. *http://hbase.apache.org/*.

[5] Hypertable. *http://hypertable.org/*.

[6] MD5. *http://en.wikipedia.org/wiki/MD5*.

[7] The Underlying Technology of Messages. http://www.facebook.com/notes/facebook-engineering/the-underlying-technology-of-messages/454991608919.

[8] Virtual Computing Lab. http://vcl.ncsu.edu/.

[9] D. Carstoiu, A. Cernian, and A. Olteanu. Hadoop hbase-0.20.2 performance evaluation. In *New Trends in Information Science and Service Science (NISS), 2010 4th International Conference on*, pages 84 –87, may 2010.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26:4:1–4:26, June 2008.

[11] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[12] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine. Authentic data publication over the internet. *J. Comput. Secur.*, 11:291–314, April 2003.

[13] M. T. Goodrich and R. Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing. Technical report, TECH. REP., JOHNS HOPKINS INFORMATION SECURITY INSTITUTE, 2001.

[14] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 216–227, New York, NY, USA, 2002. ACM.

[15] M. L. Lee, M. Kitsuregawa, B. C. Ooi, K.-L. Tan, and A. Mondal. Towards self-tuning data placement in parallel database systems. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, pages 225–236, New York, NY, USA, 2000. ACM.

[16] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 121–132, New York, NY, USA, 2006. ACM.

[17] S. Micali. Efficient certificate revocation. Technical report, Cambridge, MA, USA, 1996.

[18] H. Pang, A. Jain, K. Ramamritham, and K.-L. Tan. Verifying completeness of relational query results in data publishing. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 407–418, New York, NY, USA, 2005. ACM.

[19] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33:668–676, June 1990.

[20] R. Sion. Query execution assurance for outsourced databases. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 601–612. VLDB Endowment, 2005.

[21] K. J. Steven Y. Ko and R. Morales. The hybrex model for confdentiality and privacy in cloud computing. 2011.

[22] X. Sun, R. Wang, B. Salzberg, and C. Zou. Online b-tree merging. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 335–346, New York, NY, USA, 2005. ACM.

[23] Z. Wenchao, M. William R., T. Tao, Z. Zhuoyao, S. Micah, T. L. Boon, and L. Insup. Towards secure cloud data management. *University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-10-10.*, Feb 2010.

[24] M. Xie, H. Wang, J. Yin, and X. Meng. Integrity auditing of outsourced data. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 782–793. VLDB Endowment, 2007.

[25] M. Xie, H. Wang, J. Yin, and X. Meng. Providing freshness guarantees for outsourced databases. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, EDBT '08, pages 323–332, New York, NY, USA, 2008. ACM.

[26] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis. Authenticated join processing in outsourced databases. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pages 5–18, New York, NY, USA, 2009. ACM.

[27] J. You, J. Xi, C. Zhang, and G. Guo. Hdw: A high performance large scale data warehouse. In *Proceedings of the 2008 International Multi-symposiums on Computer and Computational Sciences*, pages 200–202, Washington, DC, USA, 2008. IEEE Computer Society.

APPENDIX

A. MERKLE HASH TREE

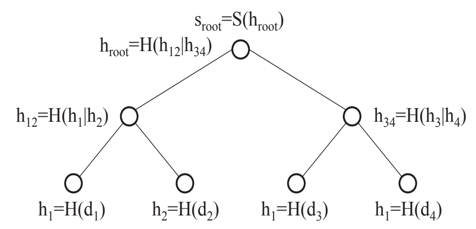


Figure 19: A Merkle Hash Tree Example.

A Merkle Hash Tree (MHT) is a type of data structure, which contains hash values representing a summary information about a large piece of data, and is used to verify the authenticity of the original data. Figure 19 shows an example of a Merkle Hash Tree. Each leaf node is assigned a digest $H(d)$, where H is a one-way hash function. The value of each inner node is derived from its child nodes, e.g. $h_i = H(h_l|h_r)$ where $|$ denotes concatenation. The value of the root node is signed, usually by the data owner. The tree can be used to authenticate any subset of the data by generating a verification object (VO). For example, to authenticate d_1, the VO contains h_2, h_{34} and the root signature s_{root}. The recipient first computes $h_1 = H(d_1)$ and $H(H(h_1|h_2)|h_{34})$, then checks if the latter is the same with the signature s_{root}. If so, d_1 is accepted; otherwise, it indicates that d_1 has been altered.