ABSTRACT

WEI, WEI. Practical Integrity Assurance for Big Data Processing Deployed over Open Cloud. (Under the direction of Ting Yu.)

The amount of data has been exploding in the world. The capability of processing large data sets, so-called big data, is becoming a key basis of competition, underpinning new waves of productivity growth, research innovation, preventing diseases, and combating crime. Big data requires exceptional technologies to efficiently process large amount of data within a reasonable time, which include distributed parallel data processing, distributed file systems, cloud computing platforms, and scalable storage systems. Deploying these technologies over open cloud is a cost-effective and practical 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 parties in open cloud usually comes from different domains, and are not always trusted, several security issues arise, including confidentiality, integrity and availability, for example how to transmit data securely through public network, how to verify the integrity of data received from other parties, how to know if a task is performed correctly, and how to detect malicious behavior during data processing. This thesis focuses on the discussion of providing practical integrity assurance while deploying these techniques over open cloud.

The first work targets at a distributed parallel data processing system - MapReduce. MapReduce has become increasingly popular as a powerful parallel data processing model. To deploy MapReduce as a data processing service over open systems such as service oriented architecture, cloud computing, and volunteer computing, we must provide necessary security mechanisms to protect the integrity of MapReduce data processing services. In this work, we present SecureMR, a practical service integrity assurance framework for MapReduce. SecureMR consists of five security components, which provide a set of practical security mechanisms that not only ensure MapReduce service integrity as well as to prevent replay and Denial of Service (DoS) attacks, but also preserve the simplicity, applicability and scalability of MapReduce. We have implemented a prototype of SecureMR based on Hadoop, an open source MapReduce implementation. Our analytical study and experimental results show that SecureMR can ensure data processing service integrity while imposing low performance overhead.

The second work targets at a scalable data storage system - BigTable. 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 work, we present iBigTable, an enhancement of BigTable that provides scalable data integrity assurance. We explore the practicality of different authenticated data structures 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 high integrity assurance.

The third work targets at the integrity issue of outsourced databases. Database outsourcing has become increasingly popular as a cost-effective solution to provide database services to clients. Previous work proposed different approaches to ensure data integrity, one of the most important security concerns in database outsourcing. However, to the best of our knowledge, existing approaches require modification to existing DBMSs, which greatly hampers the adoption of database outsourcing. In this work, we focus on the design and implementation of an efficient and practical scheme based on Merkle B-tree, which provides integrity assurance including correctness, completeness and freshness *without requiring any modification to existing DBMSs*. We design a novel scheme to serialize a Merkle B-tree (MBT) into a database while enabling highly efficient authentication data retrieval for integrity verification, which makes it attractive and practical. We create appropriate indexes and design efficient algorithms to accelerate query processing with integrity protection. We build a proof-of-concept prototype and conduct extensive experiments to evaluate the performance overhead. The results show that our scheme imposes a low overhead for queries and a reasonable overhead for updates while ensuring integrity of an outsourced database without DBMS modification.

Practical Integrity Assurance for Big Data Processing Deployed over Open Cloud

by
Wei Wei

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2013

APPROVED BY:

Peng Ning	William Enck
Rudra Dutta	Ting Yu Chair of Advisory Committee

DEDICATION

To my parents.

BIOGRAPHY

Wei Wei earned his B.S. degree in Computer Science from Huazhong University of Science and Technology in 2002, and M.S. degree in Computer Science from Shanghai Jiaotong University in 2006. In August 2008, he started his Ph.D. study at North Carolina State University in the Department of Computer Science. His research focuses on distributed system security. He devotes himself to designing new protocols and developing new techniques to protect distributed systems that are running in open environments.

ACKNOWLEDGEMENTS

I would like to thank all those who have supported and encouraged me during my Ph.D. study. First, I would like to thank my advisor, Dr. Ting Yu. I have benefited a lot from his guidance and suggestions of my work. I would like to thank him for his patience, encouragement, and financial support. I would like to thank my committee members, Dr. Pend Ning, Dr. William Enck, and Dr. Rudra Dutta, for their valuable feedback and comments on my research. Second, I would like to thank Secure Open Systems Initiative (SOSI) and National Science Foundation (NSF) for its funding support. Third, I would like to give my thanks to my friends for their help during my Ph.D. study: Juan Du, Jagan Srinivasan, Junjie Ni, Zach Jorgensen, Entong Shen and Attila Altay Yavuz. Finally, I would like to give my special thanks to my families for their encouragement and support during my Ph.D. study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The amount of data has been exploding in the world. The capability of processing large data sets, so-called big data, is becoming a key basis of competition, underpinning new waves of productivity growth, research innovation, preventing diseases, and combating crime. Big data requires exceptional technologies to efficiently process large amount of data within a reasonable time, which include distributed parallel data processing, distributed file systems, cloud computing platforms, and scalable storage systems. Deploying these technologies over open cloud is a cost-effective and practical 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 parties in open systems usually comes from different domains, and are not always trusted, several security issues arise, including confidentiality, integrity and availability, for example how to transmit data securely through public network, how to verify the integrity of data received from other parties, how to know if a task is preformed correctly, and how to detect malicious behavior during big data processing.

> *Although it presents challenges and difficulties to provide practical integrity assurance while deploying these techniques over open cloud due to the insecurity and complexity of open systems, by carefully examining their designs and protocols, they bring opportunities to design effective communication protocols, create novel schemes and simplify technique deployment to make integrity protection practical while deploying them over open systems.*

This thesis focuses on the discussion of providing practical integrity assurance while deploying these techniques over open systems. It includes three works toward providing practical integrity assurance while deploying these techniques over open systems. The first two works focus on providing practical integrity assurance for two important big data processing techniques respectively, one is MapReduce, a parallel data processing system [22] and the other is Bigtable, a scalable storage system [20]. Approaches proposed in the first two works require modifying existing systems, which may not be desirable since it introduces extra deployment cost and also hampers the adoption of our approaches. To

resolve this issue, we started to investigate on schemes that can provide integrity assurance without requiring modifying existing systems. The third work is our first work toward addressing this issue, which designed schemes to address this issue for outsourced databases. The scheme proposed in the third work is promising for addressing this issue for Bigtable [20], which is one of the future work. We elaborate details of those works in Chapters 2, 3, and 4. Chapter 5 compares our work with related work. Then, we discuss possible future work in Chapter 6, and finally conclude the thesis in Chapter 7. In the following, we give our motivations of three works respectively.

1.1 SecureMR: A Service Integrity Assurance Framework for MapReduce

MapReduce is a parallel data processing model, proposed by Google to simplify parallel data processing on large clusters [22]. Recently, many organizations have adopted the model of MapReduce, and developed their own implementations of MapReduce, such as Google MapReduce [22] and Yahoo's Hadoop [6], as well as thousands of MapReduce applications. Moreover, MapReduce has been adopted by many academic researchers for data processing in different research areas, such as high end computing [41], data intensive scientific analysis [28], large scale semantic annotation [39] and machine learning [21].

Current data processing systems using MapReduce are mainly running on clusters belonging to a single administration domain. As open systems, such as Service-Oriented Architecture (SOA) [15, 29], Could Computing [1] and Volunteer Computing [10, 16], increasingly emerge as promising platforms for cross-domain resource and service integration, MapReduce deployed over open systems will become an attractive solution for large-scale cost-effective data processing services. As a forerunner in this area, Amazon deploys MapReduce as a web service using Amazon Elastic Compute Cloud (EC2) and Amazon Simple Storage Service (Amazon S3). It provides a public data processing service for researchers, data analysts, and developers to efficiently and cost-effectively process vast amounts of data [2]. However, in open systems, besides communication security threats such as eavesdropping attacks, replay attacks, and Denial of Service (DoS) attacks, MapReduce faces a data processing service integrity issue since service providers in open systems may come from different administration domains that are not always trustworthy.

Several existing techniques such as replication (also known as double-check), sampling, and checkpoint-based verification have been proposed to address service integrity issues in different computing environments like Peer-to-Peer Systems, Grid Computing, and Volunteer Computing (e.g., [26, 27, 31–33, 54, 68]). Replication-based techniques mainly rely on redundant computation resources to execute duplicated individual tasks, and a master (also known as supervisor) to verify the consistency of results. Sampling techniques require indistinguishable test samples. The checkpoint-based verification focuses

on sequential computations that can be broken into multiple temporal segments.

In this work, we present SecureMR, a practical service integrity assurance framework for MapReduce. SecureMR provides a decentralized replication-based integrity verification scheme for ensuring the integrity of MapReduce in open systems. Our scheme leverages the unique properties of the MapReduce system to achieve effective and practical security protection. First, MapReduce provides natural redundant computing resources, which is amenable to replication-based techniques. Moreover, the parallel data processing of MapReduce mitigates the performance influence of executing duplicated tasks. However, in contrast to simple monolithic systems, MapReduce often consists of many distributed computing tasks processing massive data sets, which presents new challenges to adopt replication-based techniques. For example, it is impractical to replicate all distributed computing tasks for consistency verification purposes. Moreover, it is not scalable to perform centralized consistency verification over massive result data sets at a single point (e.g., the master).

To address these challenges, our scheme decentralizes the integrity verification process among different distributed computing nodes who participate in the MapReduce computation.

1.2 iBigTable: Practical Data Integrity for BigTable in Public Cloud

BigTable [20] 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 [7] and Cassandra [3]. HBase is used to power the messages infrastructure at Facebook [12], and also used as a data storage for Hadoop [6] and MapReduce [22] 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 [23, 40, 48]. Some others are probabilistic approaches, which do not require to modify existing systems but may inject some fake data into

outsourced databases [55, 63, 64]. 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 work, 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.

1.3 IAODB: Integrity Assurance for Outsourced Databases without DBMS Modification

Database outsourcing has become increasingly popular as a cost-effective solution to provide database services to clients. In this model, a *data owner* (DO) outsources data to a third-party *database service provider* (DSP), which maintains the data in a DBMS and answers queries from *clients* on behalf of the data owner. However, it introduces one of the most important security concerns, data integrity. Usually, DSPs are not fully trusted by data owners. Thus, data owners have to protect the integrity of their own data when outsourcing data to DSPs. Specifically, when clients retrieve data from a DSP, they should be able to verify that the returned data is what should be returned for their requests on behalf of data owners, i.e., no data is maliciously modified by DSPs and DSPs return all data clients request.

In the field of database outsourcing there are many techniques proposed to address integrity issues, including correctness, completeness and freshness. These techniques usually can be divided into two categories. Approaches belonging to the first category are based on *authenticated data structures* (ADSs) such as Merkle hash tree (MHT) [23, 40, 44] and Signature Aggregation [40, 46, 48, 50]. Existing ADS-based approaches require modifying a DBMS so that it can generate a *verification object* (VO) when executing a query and return the VO along with the actual result to clients so that clients can verify the integrity of the query result returned. Such modification is usually costly and hard to be deployed in a third-party service provider, which hampers the adoption of database outsourcing [63]. The second category uses a probabilistic approach [55, 63, 64], which injects some fake data into outsourced databases. Although the probabilistic approach does not require the modification of DBMSs, its integrity guarantee is significantly weaker than those based on ADSs.

In this work, we explore the feasibility of utilizing approaches of the first category to provide integrity assurance *without requiring any modification of DBMSs running at the server side*. In existing approaches, DBMSs are modified to be ADS-aware. That is, they are enhanced with special modules (as well as special data structures other than relations) that efficiently manage ADSs and facilitate the generation of VOs. Unfortunately, as mentioned above, it is often hard to convince database service providers to make such modifications to their DBMSs. In fact, up to today, to the best of our knowledge, no existing cloud database services support integrity checking [51]. Thus, for clients who care about query integrity, it is desirable to have integrity assurance techniques over "vanilla" DBMSs (i.e., without any special features for integrity). The general approach is straightforward: the data owner would have to store authenticated data structures along with data in relations, and retrieve appropriate integrity verification data besides issuing queries. And all these have to be done through the generic query interface (usually SQL) of the DBMS. Though the basic idea is simple, the challenge is to make it practical: we need to design appropriate schemes to convert ADSs into relations and form efficient queries to retrieve and update authentication information, without imposing significant overhead.

In this work, we present an efficient and practical scheme based on Merkle B-tree, which provides

integrity assurance without requiring special support from database service providers. Our scheme serializes a Merkle B-tree based ADS into relations in a way, where the data in the ADS can be retrieved and updated directly and efficiently using existing functionality provided by DBMSs, that is, SQL statements.

Many modern relational databases also have built-in support for XML. One seemingly promising approach is to represent Merkle B-tree as XML, store the XML representation into the DBMSs, and utilize their built-in XML support to retrieve authentication data for integrity verification. However, as can be seen from the performance result presented in Section 4.5, the XML-based solutions do not provide a good performance compared with our scheme, which is mainly because the XML features are not targeting at providing efficient operations of MHT-based integrity verification. Note that although we describe our scheme based on relational DBMSs, it is not hard to see that our scheme can also be applied to Non-SQL databases such as Bigtable [20], Hbase [7].

1.4 Summary of Contributions

Our major contributions are summarized as follows:

- SecureMR: First, we propose a new decentralized replication-based integrity verification scheme for running MapReduce in open systems. Our approach achieves a set of security properties such as non-repudiation and resilience to DoS attacks and replay attacks while maintaining the data processing efficiency of MapReduce. Second, we have implemented a prototype of SecureMR based on Hadoop [6], an open source implementation of MapReduce. The prototype shows that the security components in SecureMR can be easily integrated into existing MapReduce implementations. Third, we conduct security analytical study and experimental evaluation of performance overhead based on the prototype. Our analytical study and experimental results show that SecureMR can ensure the service integrity while imposing low performance overhead.

- iBigTable: First, 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. Second, 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*. Third, We build a prototype of iBigTable based on HBase [7], an open source implementation of BigTable. The prototype shows that the security components in iBigTable can be easily integrated into existing BigTable implementations. Fourth, 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.

- IAODB: First, we propose a novel scheme called Radix-Path Identifier to identify each piece of authentication data in a Merkle B-tree based ADS so that the MBT can be serialized into and de-serialized from a database, and design an efficient and practical mechanism to store all authentication data of a Merkle B-tree in a database, where the authentication data in the MBT can be retrieved and updated efficiently. Second, we explore the efficiency of different methods such as MultiJoin, SingleJoin, ZeroJoin and RangeCondition, to retrieve authentication data from a serialized MBT stored in a database, create appropriate indexes to accelerate the retrieval of authentication data, and optimize the update processing for authentication data. Third, we build a proof-of-concept prototype and conduct extensive experiments to evaluate the performance overhead and efficiency. The results show that our scheme imposes a low overhead for queries and a reasonable overhead for updates while providing integrity assurance.

Chapter 2

SecureMR: A Service Integrity Assurance Framework for MapReduce

2.1 Background

As a parallel data processing model, MapReduce is designed to run in distributed computing environments. Figure 2.1 depicts the MapReduce data processing reference model in such an environment. The data processing model of MapReduce is composed of three types of entities: a distributed file system (DFS), a master and workers. The DFS provides a distributed data storage for MapReduce. The master is responsible for job management, task scheduling and load balancing among workers. Workers are hosts who contribute computation resources to execute tasks assigned by the master. The basic data processing process in MapReduce can be divided into two phases: i) a map phase where input data are distributed to different distributed hosts for parallel processing; and ii) a reduce phase where intermediate results are aggregated together. To illustrate the two-phase data processing model, we use a typical example, WordCount [14] that counts how often words occur. The application is considered as a job of MapReduce submitted by a user to the master. The input text files of the job are stored in the DFS in the form of data blocks, each of which is usually 64MB. The job is divided into multiple map and reduce tasks. The number of map tasks depends on the number of data blocks that the input text files have. Each map task only takes one data block as its input.

During the map phase, the master assigns map tasks to workers. A worker is called a mapper when it is assigned a map task. When a mapper receives a map task assignment from the master, the mapper reads a data block from the DFS, processes it and writes its intermediate result to its local storage. The intermediate result generated by each mapper is divided into r partitions $P1, P2, ..., Pr$ using a partitioning function. The number of partitions is the same with the number of reduce tasks r. During the reduce phase, the master assigns reduce tasks to workers. A worker is called a reducer when it is assigned a reduce task. Each reduce task specifies which partition a reducer should process. After a

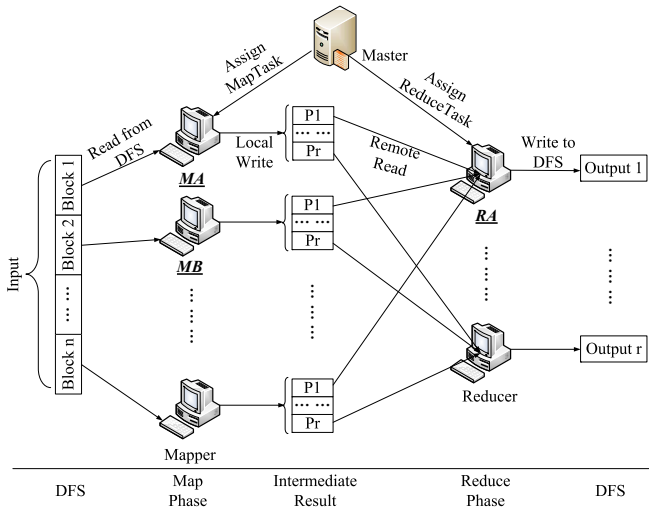


Figure 2.1: The MapReduce data processing reference model.

reducer receives a reduce task, the reducer waits for notifications of map task completion events from the master. Upon notified, the reducer reads its partition from the intermediate result of each mapper who finishes its map task. For example, in Figure 2.1, RA reads $P1$ from MA, MB and other mappers. After the reducer reads its partition from all mappers, the reducer starts to process them, and finally each reducer outputs its result to the DFS.

In fact, the MapReduce data processing model supports to combine multiple map and reduce phases into a MapReduce chain to help users accomplish complex applications that cannot be done via a single Map/Reduce job. In a MapReduce chain, mappers will read the output of reducers in the preceding reduce phase, except mappers in the first map phase, which read data from the DFS. Then, the data processing enters into the map phase with no difference from the normal map phase. Similarly, reducers will read intermediate results from mappers in the preceding map phase and generate outputs to DFS or their local disks like what mappers do, which is different from a single Map/Reduce data processing model. For reducers in the middle of data processing, they may store their results in their local disks to improve the overall system performance. Eventually, the final results go into the DFS.

2.2 System Model

2.2.1 MapReduce in Open Systems

MapReduce can be implemented to run in either closed systems or open systems. In closed systems, all entities belong to a single trusted domain, and all data processing phases are executed within this domain. There is no interaction with other domains at all. Thus, security is not taken into consideration for MapReduce in closed systems. However, MapReduce in open systems presents two significant

differences:

- The entities in MapReduce come from different domains, which are not always trusted. Furthermore, they may be compromised by attackers due to different vulnerabilities such as software bugs, and careless administration.
- The communications and data transferred among entities are through public networks. It is possible that the communications are eavesdropped, or even tampered to launch different attacks.

Therefore, before MapReduce can be deployed and operate in open systems, several security issues need to be addressed, including authenticity, confidentiality, integrity, and availability. In this work, we focus on protecting the service integrity for MapReduce. Since the data processing model of MapReduce includes three types of entities and two phases, to provide the service integrity protection for MapReduce, it naturally boils down to the following three steps:

1. Provide mappers with a mechanism to examine the integrity of data blocks from the DFS.
2. Provide reducers with a mechanism to verify the authenticity and correctness of the intermediate results generated by mappers.
3. Provide users with a mechanism to check if the final result produced by reducers is authentic and correct.

The first step ensures the integrity of inputs for MapReduce in open systems. The second step provides reducers with the integrity assurance for their inputs. The third step guarantees the authenticity and correctness of the final result for users. Finally, the combination of three ensures the MapReduce data processing service integrity to users. Since the first step has been addressed by existing techniques in [17, 30], we will go through the rest of steps in the following sections.

2.2.2 Assumptions and Attack Models

MapReduce is composed of three types of entities: a DFS, a master and workers. The design of SecureMR is built on top of several assumptions that we make on these entities. First, each worker has a public/private key pair associated with a unique worker identifier. Workers can generate and verify signatures, and no worker can forge other's signatures. Second, the master is trusted and its public key is known to all, but workers are not necessarily trusted. Third, a good worker is honest and always returns the correct result for its task while a bad worker may behave arbitrarily. Fourth, the DFS for MapReduce provides data integrity protection so that each node can verify the integrity of data read from the DFS. Fifth, if a worker is good, then others cannot tamper its data (otherwise, the worker is compromised and should be considered as a bad one). Since each worker can have its own access control mechanism to protect data from being changed by unauthorized workers, the assumption is reasonable.

Based on the above assumptions, we concentrate on the analysis of malicious behavior from bad workers. In open systems, a bad worker may cheat on a task by giving a wrong result without computation [27] or tamper the intermediate result to mess up the final result. Moreover, a bad worker may launch DoS attacks against other good workers. For example, it may keep sending requests to a good worker and asking for intermediate results or it may impersonate the master to send fake task assignments to workers. Furthermore, it may initiate replay attacks against good workers by sending old task assignments to keep them busy. In addition, it may eavesdrop and tamper the messages exchanged between two entities so that the final result generated may be compromised. Here, we classify malicious attacks into the following two models:

Non-collusive malicious behavior. Workers behave independently, which means that bad workers do not necessarily agree or consult with each other when misbehaving. A typical example is that, when they return wrong results for the same input, they may return different wrong results.

Collusive malicious behavior. Workers' behavior depends on the behavior of other collusive workers. They may communicate, exchange information, and make an agreement with each other. For example, when they are assigned tasks by the master, they can know if their colluders receive tasks with the same input blocks. If so, they return the same results so that there is no inconsistency among collusive workers. By doing so, they try to avoid being detected even if they return wrong results.

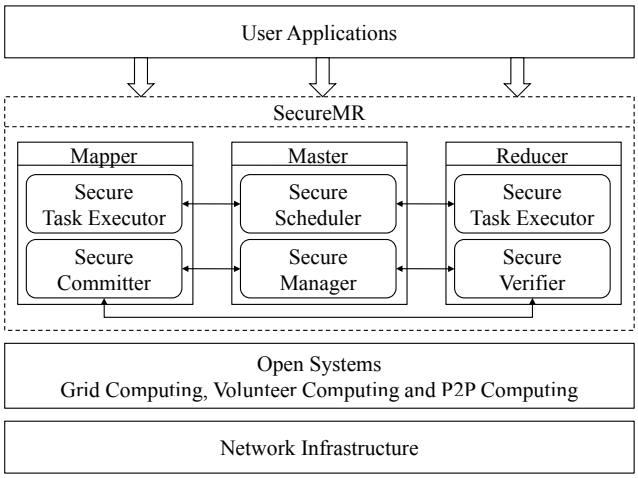
2.3 System Design

In this section, we present the detailed design of our decentralized replication-based integrity verification scheme.

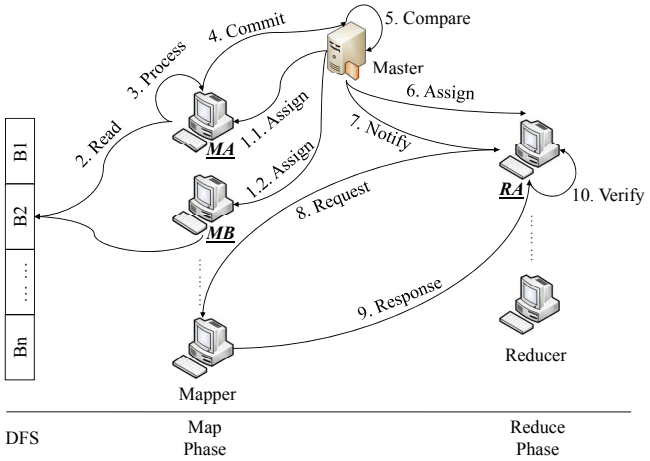
2.3.1 Design Overview

SecureMR enhances the basic MapReduce framework with a set of security components, illustrated by Figure 2.2. To validate the integrity of map/reduce tasks, our basic idea is to replicate some map/reduce tasks and assign them to different mappers/reducers. Any inconsistent intermediate results from those mappers/reducers reveal attacks. However, due to scalability and efficiency reason, though the master is trusted in our design, consistency verification should not be carried out only by the master. Instead, in our design, this responsibility is further distributed among workers. Our design must ensure properties such as non-repudiation and resilience to DoS and replay attacks, as well as efficiency. Further, our design should preserve the existing MapReduce mechanism as much as possible so that it can be easily implemented and deployed with current MapReduce systems. We introduce the design of SecureMR from two aspects: architecture and communication.

Architecture Design. Figure 2.2a shows the architecture design of SecureMR, which comprises five security components: Secure Manager, Secure Scheduler, Secure Task Executor, Secure Committer and



(a) SecureMR Architecture Design.



(b) SecureMR Communication Design.

Figure 2.2: SecureMR Design Overview.

Secure Verifier. They provide a set of security mechanisms: task duplication, secure task assignment, DoS and replay attack protection, commitment-based consistency checking, data request authentication, and result verification.

Secure Manager and Secure Scheduler are deployed in a master mainly for task duplication, secure task assignment, and commitment-based consistency checking. Secure Task Executor is running in both mappers and reducers to prevent DoS and replay attacks that exploit fake or old task assignments. In mappers, Secure Committer takes the responsibility of generating commitments for the intermediate results of mappers and sending them to Secure Manager in the master to complete the commitment-based consistency checking. Secure Verifier running in a reducer collaborates with Secure Manager to verify a mapper's intermediate result. For simplicity, we quote all components using names without Secure in the following sections, for example Manager, Scheduler, Task Executor and so on.

Communication Design. Figure 2.2b shows how the entities in SecureMR communicate with each other to provide security protection for MapReduce. Communications among them are further organized into two protocols: *Commitment* protocol and *Verification* protocol. In Figure 2.2b, communications from 1 to 5 form the commitment protocol while communications from 6 to 10 form the verification protocol.

In the commitment protocol, to avoid checking the intermediate results directly (which is expensive), mappers only send commitments (which will be described in detail later) to the master, which can be used to detect inconsistency efficiently. However, this introduces another vulnerability. Mappers may send the master the right commitments but the wrong results to reducers. For this reason, we further ask reducers to check the consistency between the commitment and the result in the verification protocol. Note that this does not add much extra effort to the reducer as it has to retrieve the intermediate result for data processing anyway.

In the following two sections, we will discuss the details of communications between the five security components of SecureMR, which happen in the commitment and verification protocols.

2.3.2 Commitment Protocol

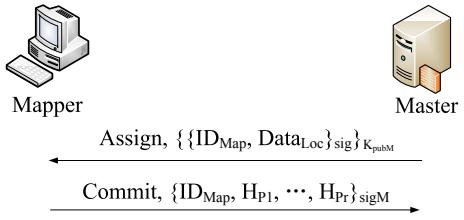


Figure 2.3: The Commitment Protocol.

As mentioned in Section 2.2.2, the master is a trusted entity. However, since the intermediate result is usually tremendous, it is impractical to require the master to check all intermediate results generated by different map tasks in different jobs, which will overload the master and lead to low system performance. Thus, instead of examining intermediate results directly, the master requires mappers to generate commitments for their intermediate results, and then check commitments [27].

Protocol design

Since we assume that the DFS provides data integrity protection, we do not discuss the communications between mappers and the DFS. Figure 2.3 shows the communications between a mapper and the master in the commitment protocol. The specific steps are described as follows.

Assign. The Scheduler in the master sends the *Assign* message to the Task Executor in a mapper to assign a map task to the mapper. Regarding task duplication, the Scheduler may assign the same map task to different mappers. For example, in Figure 2.2b, MA and MB are assigned the same map task. The *Assign* message includes a monotonically increasing identity ID_{Map} of a map task and an input data block location $Data_{Loc}$, which is signed by the master and encrypted using K_{pubM}, the public key of the mapper. After the Task Executor receives the task assignment message, the Task Executor decrypts and verifies the signature of the message. Then, the Task Executor reads an input block according to $Data_{Loc}$ from the DFS. In Figure 2.2b, since MA and MB receive the same task, they both read the same data block $B2$ from the DFS.

Commit. After the mapper processes the input block, the Committer of the mapper makes a commitment to the master by generating a hash value for each partition of its intermediate result and signing those hash values. We use $\{...\}sigM$ to denote a signed message of a mapper. When the Manager of the master receives the commitment, the Manager verifies the signature using the mapper's public key K_{pubM}. If the Manager has received more than one commitments for the same map task from different mappers, the Manager will compare new commitment with an old one to see if they are consistent with each other.

Note that in this work, we focus on expose suspicious activities. How to exactly pinpoint malicious ones is the next step and some existing techniques may be applied [67].

Protocol analysis

In this protocol, since the task assignment message is signed by the master and encrypted using the mapper's public key, the integrity and confidentiality of the *Assign* message is well protected. It also ensures that the mapper is the only entity that can decrypt the *Assign* message and the master is the only entity that can create it. In this case, malicious mappers cannot know task assignments of other good mappers or arbitrarily assign fake tasks to a mapper to launch DoS attacks. Furthermore, to prevent replay attacks which send old task assignments, a monotonically increasing identity ID_{Map} is associated

with each map task, which is automatically generated using timestamp or sequence number by the Scheduler. The Task Executor in the mapper records the ID_{Map} for the last map task that it processed. In this way, the Task Executor can determine if a task assignment is an old one by comparing the ID_{Map} with the latest recorded ID_{Map}. Regarding the *Commit* message, the integrity of the commitment is assured since the *Commit* message is signed using the mapper's private key. Moreover, ID_{Map} is needed so that the master knows which map task this commitment is for.

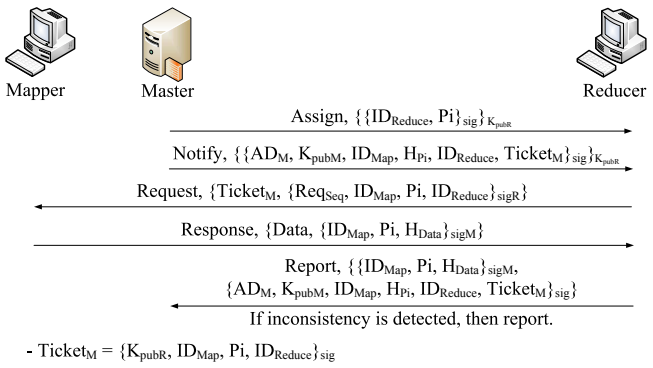


Figure 2.4: The Verification Protocol.

2.3.3 Verification Protocol

In the verification protocol, reducers further help the master to verify if intermediate results generated by mappers are consistent with commitments submitted to the master. The verification protocol is built on existing MapReduce communication mechanisms. There are no additional messages introduced to MapReduce.

Protocol design

Figure 2.4 shows how the master, a mapper, and a reducer communicate with each other in the verification protocol. We illustrate each step as follows.

Assign. The master signs the *Assign* message and encrypts it using K_{pubR}, the public key of a reducer. In the message, ID_{Reduce} is a monotonically increasing identity of a reduce task, and Pi indicates the partition of intermediate results that the reducer will process. When the Task Executor in the reducer receives the task assignment, the Task Executor first verifies the integrity and authenticity of the task assignment. Then, the Verifier of the reducer will wait for notifications from the Manager.

Notify. When the Manager receives the completion event with a commitment from the Committer of a mapper, the Master sends a notification to the Verifier of each reducer, which includes the mapper's

address AD_M, the mapper's public key K_{pubM}, ID_{Map}, the ticket $Ticket_M$ for the mapper signed by the master and the hash value H_{Pi} for the Pi partition committed by the Committer. The ticket $Ticket_M$ is used for data request authentication in the *Request* message.

Request. After the Verifier in a reducer gets notified, the Verifier sends a data request to the Committer of the mapper, which includes the ticket $Ticket_M$ as evidence of an authentic data request authorized by the master, the reducer's public key K_{pubR}, a sequence number Req_{Seq} and Pi which indicates which partition is requested.

Response. After the Committer verifies the authenticity of the request by verifying the ticket from the master and the reducer's signature, the mapper sends a response to the Verifier, which includes ID_{Map}, Pi, the data $Data$ and H_{Data}, the hash value of $Data$. To verify the integrity of the response, the Verifier first verifies the signature in the *Response* message, then regenerates a hash value H'_{Data} for the data, and compares H_{Data} with H'_{Data} to make sure that the data is not tampered during the *Response* communication. Finally, the Verifier compares H'_{Data} with H_{Pi} committed to the master to check if any inconsistency occurs.

Report. When the Verifier detects an inconsistency, the Verifier sends two signatures as evidence to the Manager to report the inconsistency. After the Manager receives and verifies the two signatures, the Manager can compare H_{Data} with H_{Pi} to confirm the reported inconsistency.

Protocol analysis

Similar to the commitment protocol, the reduce task assignment mechanism prevents both DoS and replay attacks against reducers. However, in the verification protocol, a mapper faces DoS attacks when others request data from it. To countermeasure this kind of DoS attacks, the mapper needs to authenticate data requests from reducers. The data request authentication is achieved by requiring that a reducer shows a ticket from the master. If the mapper sees a ticket at the first time, the mapper can make sure that the request must come from an authorized reducer who holds the ticket issued by the master. However, if the first attempt of data request fails somehow, attackers may get the ticket by eavesdropping the communications between the mapper and the reducer. In this case, since the mapper will record the latest request sequence number Req_{Seq} associated with a ticket, the mapper will check if this data request is an old one by comparing the two Req_{Seq} numbers when the mapper receives another data request with the same ticket. In this way, replay attacks can be defeated.

2.3.4 SecureMR Extension

So far, we have discussed how SecureMR provides reducers with a mechanism to verify the authenticity and correctness of the intermediate results generated by mappers. In this section, we present how SecureMR applies the replication-based verification scheme to reducers and MapReduce chain to provide users with a mechanism to check if the final result produced by reducers is authentic and correct.

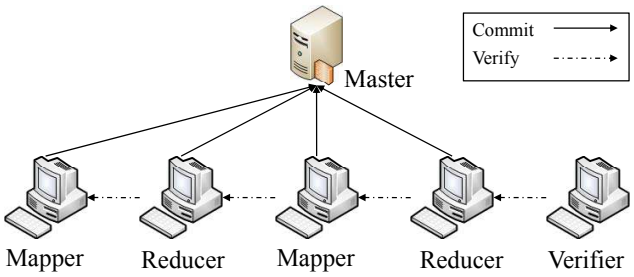


Figure 2.5: SecureMR Extension for MapReduce Chain.

Extension for Reducers. Similar to mappers, the Scheduler in the master may duplicate reduce tasks and assign them to multiple reducers. Reducers assigned the same task will read the same partition of the intermediate results from mappers. However, we observe that reducers are not configured with a Secure Committer component in current architecture described in Figure 2.2a, which means they cannot make a commitment to the master. In order for reducers to make commitments, we can easily deploy a Secure Committer component for reducers. Another problem to apply the verification scheme to reducers is that there are no other entities to complete the verification protocol since reducers are in the last phase. To address this problem, we extend the MapReduce model to include an additional phase called *Verify* phase. In the verify phase, the master involves several workers with a Secure Verifier component, called verifiers to complete the verification protocol. Another alternative is to install a Secure Verifier component into MapReduce user applications and ask them to complete the verification protocol by themselves after their jobs are done.

Extension for MapReduce Chain. Similarly, the verification scheme can be applied to MapReduce chain since each map and reduce share the similar procedure of data processing. Figure 2.5 shows the design overview of how SecureMR applies the verification scheme to MapReduce chain. As we can see from the figure, the design is like a *Commit-Verify* chain between the master, mappers and reducers. If mappers make commitments to the master, reducers will take the role of verifiers to verify the consistency between intermediate results and commitments of mappers. If reducers make commitments to the master, mappers will take the role of verifiers to verify the consistency between outputs and commitments of reducers except the last phase, *Verify* phase. The verify phase has been discussed in the above. In order for mappers to be able to fulfill the verification protocol, the only thing that we need to do is to plug a Secure Verifier component into each mapper.

2.4 Analysis and Evaluation

In this section, we discuss the security properties of SecureMR, and then evaluate the performance overhead both analytically and experimentally. Note that in Section 2.4.1 and 2.4.2, we focus on the

discussion for mappers due to the similarity of the analysis between mappers and reducers.

2.4.1 Security Analysis

There are two kinds of inconsistencies for mappers in MapReduce. One is an inconsistency between results returned by different mappers that are assigned the same task. The other is an inconsistency between the commitment and the result generated by a mapper. The former can only be detected by the master in the commitment protocol and the latter can only be detected by a reducer in the verification protocol. We claim that SecureMR provides the following two properties. We also provide arguments for our statement in the following.

- **No False Alarm.** For any inconsistency detected by SecureMR, it must happen between good and bad mappers, between bad mappers or on a bad mapper. It cannot occur between good mappers or on a good mapper.
- **Non-Repudiation.** For any inconsistency that can be observed by a good reducer or the master, SecureMR can detect it and present evidence to prove it.

Arguments of No False Alarm. The assumptions in Section 2.2.2 guarantee that good mappers always produce correct and consistent results. We prove by contradiction that SecureMR provides *No False Alarm* property in terms of the two kinds of inconsistencies.

First, suppose that an inconsistency between two good mappers is detected by the master. In this case, the master must get two different sets of hash values from the commitments of two good mappers, which means that the two commitments the master received must be tampered somehow since two good mappers will not produce inconsistent results. However, if the master accepted a commitment of a mapper, the master must have confirmed the integrity and freshness of the commitment. Thus, the commitment is neither a bad commitment nor an old one. From the arguments, we can infer that there is no way to tamper a commitment of a mapper without being detected by the master. And the hypothesis implies that the master already accepted the commitments, which means it is impossible that the commitments that the master received have been tampered. Therefore, the hypothesis that an inconsistency between two good mappers is detected by the master is not true.

Second, suppose that an inconsistency between the commitment and the intermediate result of a good mapper is detected by a reducer. If the reducer is good, it can be inferred that the message received by the reducer must be tampered somehow. Since the reducer knows ID_{Map} and Pi, the reducer will not accept the message unless the reducer confirms the integrity of the message. ID_{Map} can also be the proof of the freshness of the signatures. For the same reason, it is impossible that the message has been tampered. Thus, the case that an inconsistency on a good mapper is detected by a good reducer cannot be true. If the reducer is a bad reducer, the reducer can report an inconsistency even if there is no inconsistency. But, the verification protocol requires that the reducer present the evidence to the master,

which is described in Figure 2.4. And the reducer cannot forge evidence without being detected by the master. Hence, the case that an inconsistency on a good mapper is detected by a bad reducer cannot be true, either. Therefore, the hypothesis that an inconsistency on a good mapper is detected by a reducer is not true.

Arguments of Non-Repudiation. We prove by contradiction that SecureMR provides *Non-Repudiation* property in terms of the two kinds of inconsistencies. Suppose that an inconsistency is observed by the master or a good reducer. Both the master and the good reducer definitely report the inconsistency since they both tell the truth. Meanwhile, the master holds the commitments of workers, which cannot be denied, and the good reducer has the signatures of mappers. They both can present the commitments or the signatures of mappers to prove the inconsistency they detect. Thus, SecureMR provides the *Non-Repudiation* property in terms of the two kinds of inconsistencies.

2.4.2 Attacker Behavior Analysis

We analyze the behavior of the following attackers under the two kinds of behavior models defined in Section 2.2.2. When we analyze the collusive attacks, we consider the worst case that all malicious entities are colluding with one another.

- Periodical Attackers: they misbehave with a certain probability p_m. Since a naive attacker is a special case of periodical attacker with p_m equal to 1. Thus, we discuss these two kinds of attacker's behavior together.
- Strategic Attackers: with the assumption that they know the duplication strategy, they may not behave maliciously until they definitely know that they will not be caught due to the collusion, which means that all duplicates are assigned to the collusive group.

Definition 2.4.1 *(Detection Rate) We define the detection rate, denoted D_{rate}, as the probability that the inconsistency between results caused by the misbehavior of a mapper is detected during l jobs.*

Since each map task processes one block, the duplication of a map task is the same as the duplication of a block. The following discussion may use both terms, block duplication and map task duplication exchangeably. Suppose MapReduce consists of one master and n workers, and m out of n workers ($m < n$) are malicious workers. For simplicity, we assume that the input of each job has the same number of blocks b, no two blocks are the same and each worker only processes one task in one job. The percentage of blocks that will be duplicated in each job is p_b. Thus, the number of duplicated blocks is $b \cdot p_b$. SecureMR randomly chooses one block from the original b blocks to duplicate for each duplication. It uses a *naive task scheduling* algorithm, which launches all map tasks together, including duplicated map tasks. In the following, we analyze the detection rate for periodical attacker without and with collusion, and the probability that strategic attackers can misbehave in a job.

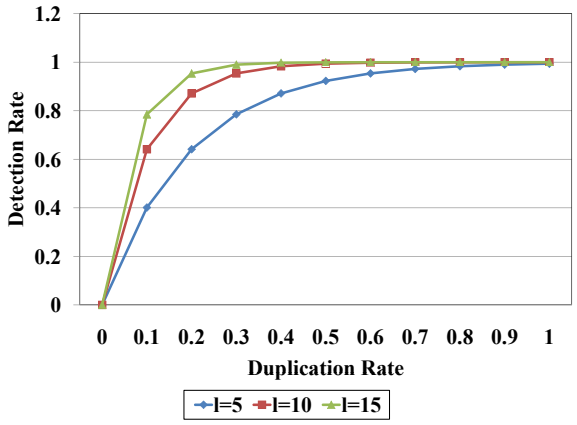


Figure 2.6: Detection Rate for Non-Collusion Naive Attacker.

Periodical attackers without collusion. For simplicity, we assume that they return different results when they misbehave on the same input. Thus, without collusion, the detection rate of a malicious mapper is the same as the probability that the block processed by the mapper is duplicated. Therefore, the detection rate is calculated as follows:

$$D_{rate} = 1 - (1 - (1 - (1 - 1/b)^{b \cdot p_b}) \cdot p_m)^l \tag{2.1}$$

In Equation 2.1, $(1 - (1 - 1/b)^{b \cdot p_b}) \cdot p_m$ denotes the probability that the misbehavior of the malicious mapper is detected during one job. Figure 2.6 shows detection rate for a naive attacker without collusion, where b is equal to 20 and l is 5, 10 and 15. Figure 2.7 shows detection rate for a periodical attacker with 0.5 misbehaving probability. Both of them demonstrate that as the number of tasks that a malicious mapper processes increases, high detection rate can be achieved even if the duplication rate is only 20%, which means that the chance for an attacker to cheat without being detected in the long run is very low.

Periodical attackers with collusion. With collusion, the maximum number of entities that collude with each other is m. Let $P(B_i)$ denote the probability that a block will be duplicated i times and $P(D)$ denote the probability that the inconsistency caused by the misbehavior of a malicious mapper will be detected. In this case, the detection rate is:

$$\begin{aligned} D_{rate} &= 1 - (1 - \sum_{i=0}^{b \cdot p_b} P(D|B_i) \cdot P(B_i))^l \\ &= 1 - (1 - \sum_{i=0}^{b \cdot p_b} P(D|B_i) \cdot \binom{b \cdot p_b}{i} (\frac{1}{b})^i \cdot (1 - \frac{1}{b})^{b \cdot p_b - i})^l \end{aligned} \tag{2.2}$$

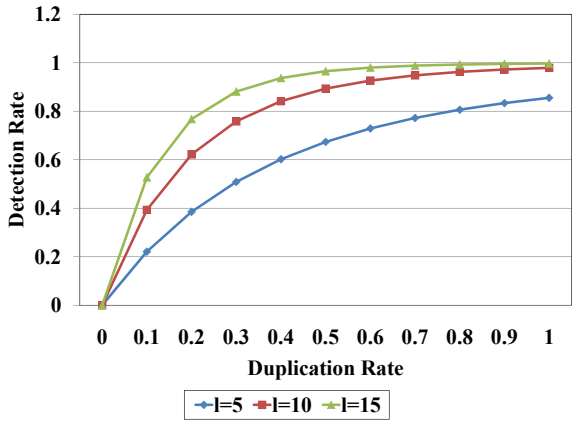


Figure 2.7: Detection Rate for Non-Collusion Periodical Attacker.

where

$$P(D|B_i) = \begin{cases} 0 & \text{if } i = 0, \\ (1 - \binom{m-1}{i}/\binom{n-1}{i}) \cdot p_m & \text{if } i > 0 \text{ and } i < m, \\ p_m & \text{if } i >= m. \end{cases}$$

In Equation 2.2, the detection rate is computed using the law of total probability. The inconsistency cannot be detected only if all duplicates for the block that the malicious mapper processes are assigned to its collusive parties. $P(D|B_i)$ is the probability that the inconsistency is detected when the block that the malicious mapper processes is duplicated i times. If $i >= m$, at least one duplicate will not be assigned to its collusive parties. Figure 2.8 shows how the detection rate changes as the duplication rate and the percentage of malicious workers change given n, p_m, b, l equals to 50, 0.5, 20 and 15, respectively. From the figure, we observe that as long as the majority of workers are good, 90% detection rate can be achieved with 40% duplication rate.

Strategic attackers. Since the misbehavior of attackers cannot be detected, we discuss the probability $P(F)$ that the intermediate result that reducers receive is tampered, which is the same as the misbehaving probability of a strategic attacker. In this case, we analyze the strategic attacker's behavior in the following two steps:

1. The master assigns b input blocks to b mappers before any duplication is made.

2. The master duplicates $b \cdot p_b$ input blocks after assignments for the original b blocks. For each duplication, the master randomly chooses one block from the original b blocks to duplicate.

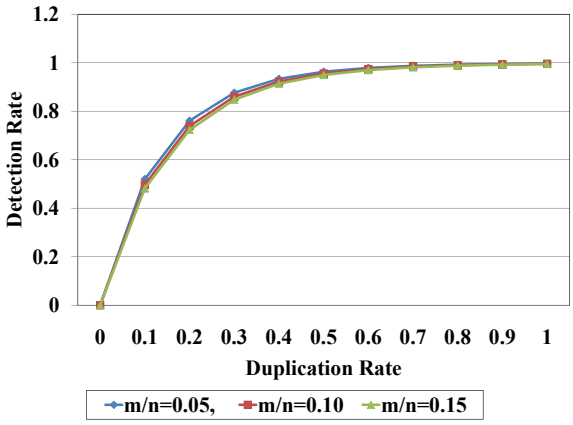


Figure 2.8: Detection Rate for Collusion Periodical Attacker.

Therefore, $P(F)$ can be calculated by the following formula:

$$\begin{aligned} P(F) &= \sum_{i=0}^{x} P(F|E_i) \cdot P(E_i) = \sum_{i=0}^{x} P(A_i) \cdot P(M_i) \cdot P(E_i) \\ &= \sum_{i=0}^{x} (i/b)^{b \cdot p_b} \cdot \left(\binom{m-i}{b \cdot p_b} / \binom{n-b}{b \cdot p_b} \right) \cdot \binom{m}{i} \cdot \binom{n-m}{b-i} / \binom{n}{b} \end{aligned} \tag{2.3}$$

where

$$x = \begin{cases} m & \text{if } m < b, \\ b & \text{if } m >= b. \end{cases}$$

Note that E_i and $P(E_i)$ denote the event that mappers contain i collusive mappers before input block duplication and the probability that E_i happens, respectively. $P(F|E_i)$ denotes the probability that the result is tampered by some mappers when E_i occurs. $P(A_i)$ and $P(M_i)$ denote the probability that all duplicated blocks $b \cdot p_b$ belong to the set of blocks that the i collusive mappers process and the probability that all duplicated blocks $b \cdot p_b$ are assigned to the rest of m_i's collusive workers. Figure 2.9 shows the misbehaving probability of a strategic attacker when duplication rate and the percentage of malicious workers change, where n, b, l equals to 50, 20 and 15, respectively. The result implies that the misbehaving probability of a strategic attacker is pretty low even if the duplication rate is only 10%.

Since strategic attackers can exchange information of tasks with their collusive entities when they decide whether or not to cheat in tasks, sometimes they can misbehave without being detected. In order to address this vulnerability, we propose a *commitment-based task scheduling* algorithm. Basically, the *commitment-based task scheduling* algorithm will launch the duplicates of a task only after the task has been committed. In this case, when a strategic attacker initially processes a task, there is no way for it to know any duplication information about the task that it handles because no duplicated tasks have been assigned yet. Later when its collusive entities receive the duplicated tasks, they need to return the same

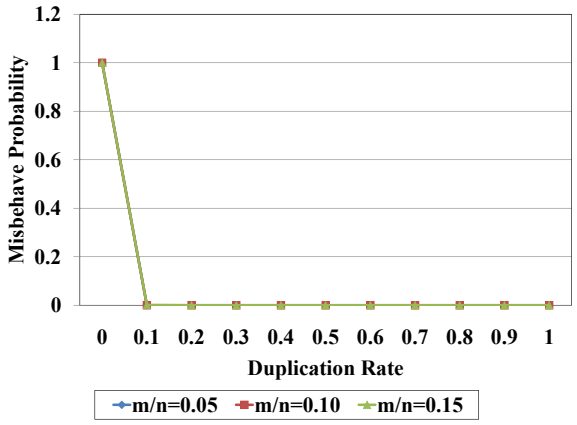


Figure 2.9: Misbehaving Probability vs Duplication Rate.

results with the initial result. Otherwise, inconsistency will be produced, which can be detected by the master. Thus, the strategic attacker cannot misbehave because it is always possible that the misbehavior could be detected as long as there are duplicated tasks. However, intuitively, it delays the execution of duplicated tasks, which may bring down the performance of the system. In the following section, we will evaluate the performance overhead of SecureMR under both the *naive task scheduling* algorithm and the *commitment-based task scheduling* algorithm.

2.4.3 Experimental Evaluation

System Implementation. We have implemented a prototype of SecureMR based on one existing implementation of MapReduce, Hadoop [6]. In our prototype, we have implemented both *naive task scheduling* algorithm and *commitment-based task scheduling* algorithm mentioned in previous sections. Regarding consistency verification, we have implemented a *non-blocking* replication-based verification scheme, which means that reducers do not need to wait for all duplicates of a map task to finish and users do not need to wait for all duplicates to finish. Finally, users will be informed if an inconsistency is detected after all duplicates finish.

Experiment Setup. We run our experiments on 14 hosts provided by Virtual Computing Lab (VCL), a distributed computing system with hundreds of hosts connected through campus networks [13]. The Hadoop Distributed File System (HDFS) is also deployed in VCL. We use 11 hosts as workers that offer MapReduce services and one host as a master, and HDFS uses 13 nodes, not including the master host. We adopt the duplication strategy discussed in Section 2.4.2. All hosts used have similar hardware and software configurations (2.66GHz Intel Intel(R) Core(TM) 2 Duo, Ubuntu Linux 8.04, Sun JDK 6 and Hadoop 0.19). All experiments are conducted by using Hadoop WordCount application [14].

Table 2.1: Performance Overhead on Entities

Type	Cost Estimation	Estimated Time
Master	$4 \cdot T_{sig} + 3 \cdot T_{Epub} + T_{ver}$	$20ms$
Mapper	$2 \cdot T_{sig} + T_{Dpub} + 3 \cdot T_{ver} + r \cdot T_{hash}$	$14 + (r + 1) \cdot 40ms$
Reducer	$2 \cdot T_{Dpub} + 3 \cdot T_{ver} + T_{hash}$	$51ms$

Table 2.2: Communication Overhead between Entities

Type	Cost Estimation	Additional Bytes
Master-Mapper	$2 \cdot D_{sig} + r \cdot D_{hash}$	$256 + r * 20bytes$
Master-Reducer	$3 \cdot D_{sig} + D_{hash} + D_{pub}$	$532bytes$
Mapper-Reducer	$3 \cdot D_{sig} + D_{hash} + D_{pub}$	$532bytes$

Performance Analysis. First, we estimate the additional overhead introduced by SecureMR in Table 2.1 and 2.2. Table 2.1 shows the performance overhead of SecureMR on the master, a mapper and a reducer. Table 2.2 shows the additional bytes to be transmitted on each communication between them. Note that there are no additional messages introduced. Here, T and D denote the time and data transmission cost for different secure operations such as encryption, decryption, signature, verification and hash. r is the number of reducers. The size of each partition is around 14MB. We use SHA-1 to generate hash values, and RSA to create signature or encrypt/decrypt data. The estimation shows that the cost of communication is negligible and the cost on each entity is small.

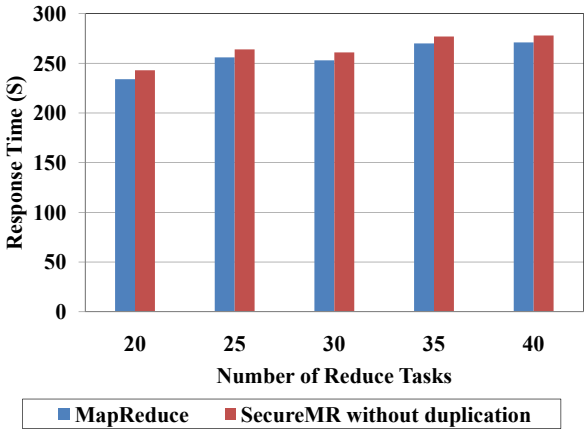


Figure 2.10: Response Time vs Number of Reduce Tasks.

We also conduct experiments to evaluate the performance overhead caused by SecureMR. Figure

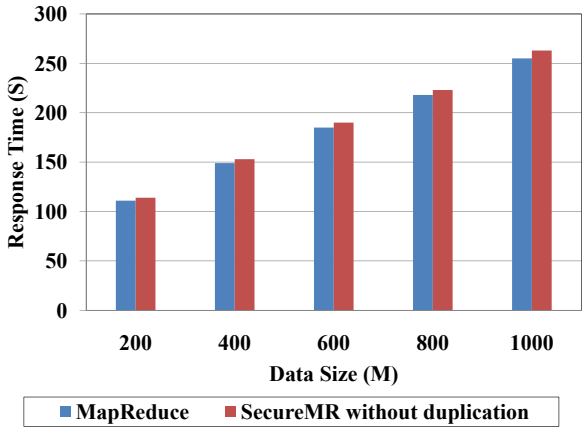


Figure 2.11: Response Time vs Data Size.

2.10 shows the response time versus the number of reduce tasks under two scenarios, MapReduce and SecureMR without duplication, where the number of map tasks is 60 and the data size is 1GB. The result shows that the overhead of SecureMR is below 10 seconds, which is small compared with the response time which is about 250 seconds. Figure 2.11 shows the response time versus the data size, where the number of map tasks is 60 and the number of reduce tasks is 25. Since the data size only affects the time to generate hash values, it shows a similar overhead in Figure 2.10.

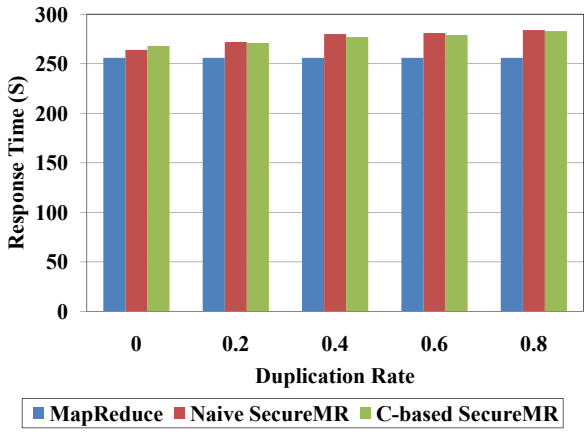


Figure 2.12: Response time vs Duplication Rate.

Regarding the performance overhead by executing duplicated tasks, we compare the response time in three cases: MapReduce, SecureMR with naive scheduling, and SecureMR with commitment-based scheduling. Figure 2.12 shows the response time versus the duplication rate. Since we adopts a *non-*

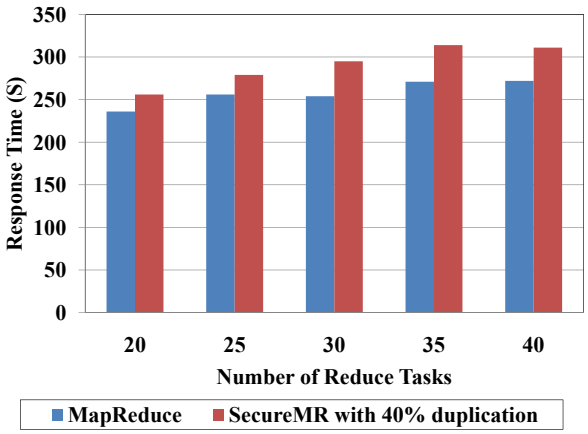


Figure 2.13: Response time vs Number of Reduce Tasks.

blocking verification mechanism, the difference between two scheduling algorithms is very small. The result shows that the time overhead increases slowly with the increase of duplication rate. Figure 2.13 shows the response time versus the number of reduce tasks under the two scenarios, MapReduce and SecureMR with 40% duplication rate, where the number of map tasks is 60 and the data size is 1GB. Compared with the no-duplication case in Figure 2.10, the performance overhead caused by executing duplicated tasks ranges from 5% to 12%.

Chapter 3

iBigTable: Practical Data Integrity for BigTable in Public Cloud

3.1 Background

3.1.1 Cryptographic Primitives

One-way hash function. A one-way hash function, denoted as $H(.)$, is a function that is easy to compute an output $H(m)$ from an input m, but hard to find the original input m from the output $H(m)$. MD5 [9] and SHA1 [11] are typical one-way hash functions used in many application areas. We use the terms hash, hash value and digest interchangeably.

Digital signature. A digital signature algorithm is a mathematical way to authenticate the integrity of a message as well as its origin. A signer keeps a private key used to sign messages, and publish the corresponding pubic key for the public to verify the integrity of signed messages. The successful verification indicates that messages are not tampered and really come from the signer. RSA [53] and DSA [4] are two commonly-used signature algorithms. The terms signature and digital signature are used interchangeably.

Merkle Hash Tree. 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 3.1 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.

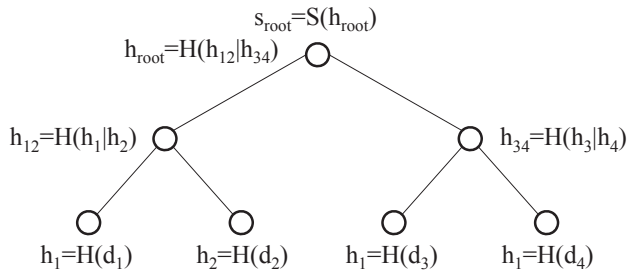


Figure 3.1: A Merkle Hash Tree Example.

3.1.2 BigTable - Distributed Storage System

Bigtable is a distributed storage system for managing structured data. A table in BigTable is a sparse distributed, persistent, multidimensional sorted map [20]. 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.

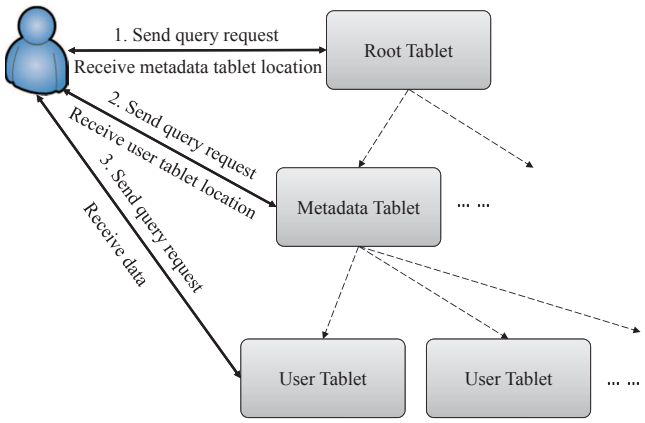


Figure 3.2: BigTable: Tablet Location Hierarchy.

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 3.2 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.

3.2 System Model

3.2.1 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 work, 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.

3.2.2 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.

3.3 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 Section 3.1 for details.

3.3.1 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 3.3(a) shows a dis-

tributed 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 3.3(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.

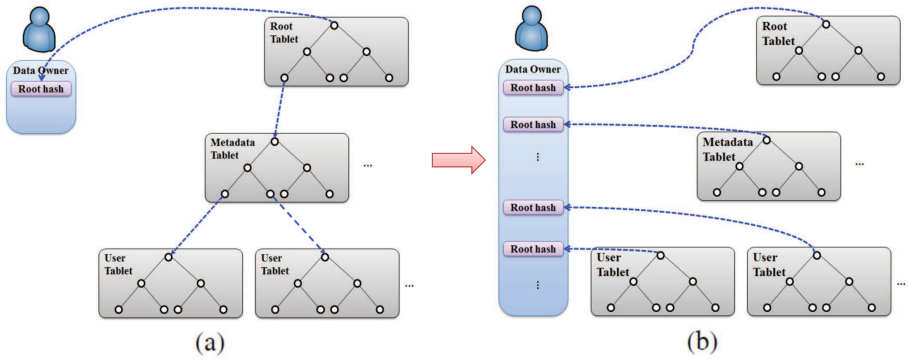


Figure 3.3: Distributed Authenticated Data Structure Design.

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 3.3(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 [20]. 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 work is based on this design.

3.3.2 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 3.4. 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 3.4 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.

As can be seen from Figure 3.4, 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 3.5.3.

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

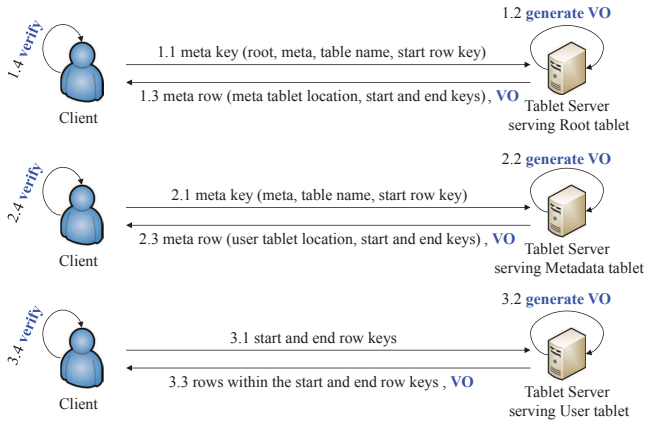


Figure 3.4: Decentralize Integrity Verification.

applied to our design, which do not requires that the data owner be online to handle the root hash retrieval requests [40, 42, 64]. 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 work, for simplicity, we still assume that the data owner stores the root hashes and handles the root hash retrieval requests.

3.3.3 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 [40, 48] and Merkle Hash Tree(MHT) based approaches [23, 40]. Although both of them can guarantee correctness and completeness, it is unknown how to efficiently guarantee freshness using signature aggregation based approaches [40]. 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 [40]. Thus, we focus on exploring MHT-based authenticated data structures in the following.

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 3.5. 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,

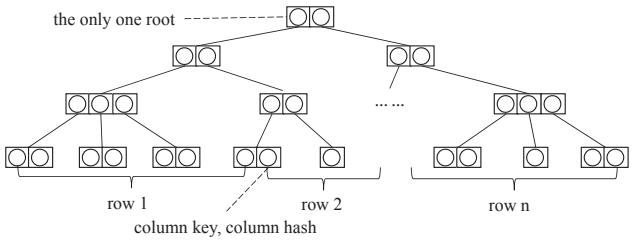


Figure 3.5: SL-MBT: Single-Level Merkle B+ Tree.

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:

$$\begin{aligned} H_r &= (f-1) \cdot (h_t - h_p) \\ &= (f-1) \cdot \log_f(n_r/n_q) \end{aligned} \tag{3.1}$$

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:

$$H_c = \sum_{i=0}^{\log_f(n_q \cdot n_{cf} \cdot n_c)} f^i + \log_f(n_r/n_q) \tag{3.2}$$

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:

$$H_r = n_q \cdot (f-1) \cdot h_t \tag{3.3}$$

$$H_c = n_q \cdot (1 + h_t) \tag{3.4}$$

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

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 3.6:

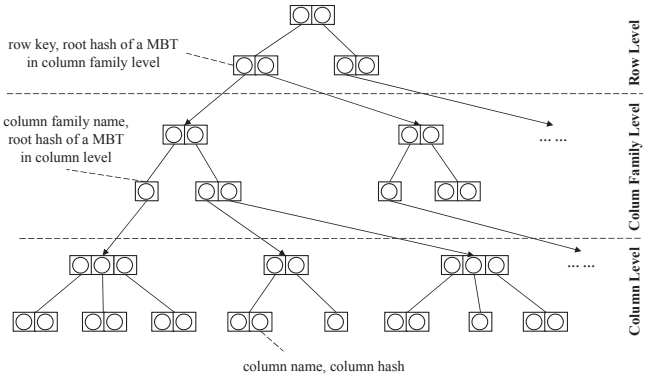


Figure 3.6: ML-MBT: Multi-Level Merkle B+ Tree.

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:

$$H_c = \sum_{i=0}^{\log_f n_q} f^i + \log_f(n_r/n_q) \tag{3.5}$$

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 3.7 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.

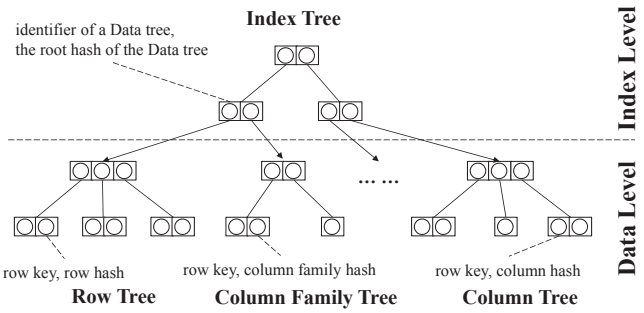


Figure 3.7: TL-MBT: Two-Level Merkle B+ Tree.

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:

$$H_r = (f-1) \cdot (h_m + \log_f(n_r/n_q)) \tag{3.6}$$

$$H_c = \sum_{i=0}^{\log_f n_q} f^i + \log_f(n_r/n_q) + h_m \tag{3.7}$$

In Equation 3.6 and 3.7, 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.

3.4 Data Operations

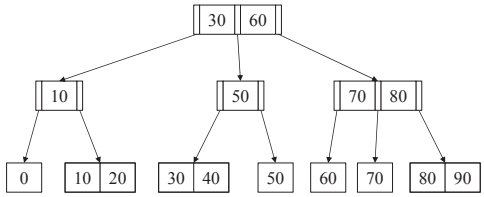
Based on TL-MBT, we describe how clients ensure the integrity of the major data operations of BigTable. We address several challenges, including range query across multiple tablets, efficient batch updates, tablet merge and split. In our design, the data owner stores the root hash of each tablet, and 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.

3.4.1 Data Access

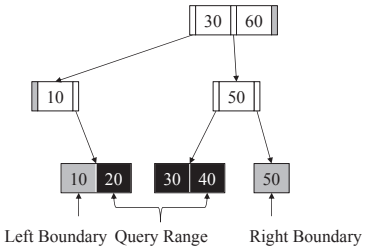
We start our discussion from range queries[1]. In Section 3.3.2, we illustrate a general process to run query with integrity protection in iBigTable. The execution of range queries within a tablet follows exactly the same process shown in Figure 3.4. However, we need to handle range queries across multiple 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 3.8a and 3.8b 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

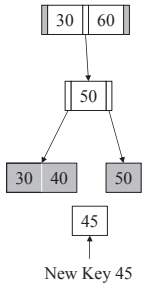
[1]A single row query as a special case of range query can be handled in the same way that a range query is executed.



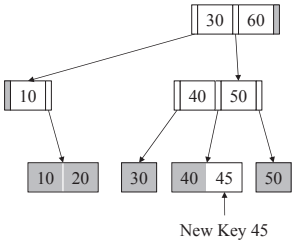
(a) Initial MB+ row tree.



(b) Range query from 15 to 45.



(c) Insert a row with key 45.



(d) Insert after range query.

Figure 3.8: A running example.

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 3.4. 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.

Single Row Update. In iBigTable, we support integrity protection for dynamic updates such as insertion, modification and deletion. In the work, 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 3.9.

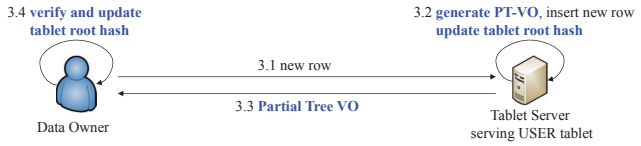


Figure 3.9: Single Row Insert with Integrity Protection.

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 3.8c shows the PT-VO returned for an insertion at key 45. As can be seen from Figure 3.9, 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.

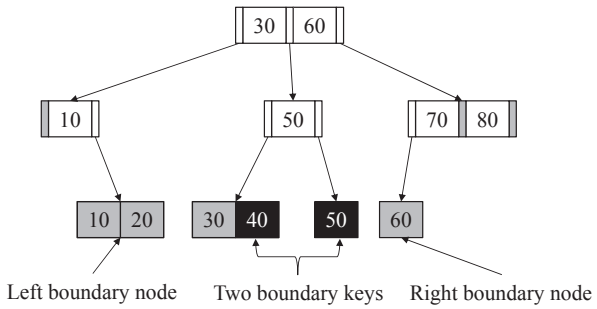
In the second case where we assume that a PT-VO for a range is already cached in the data owner side, the data owner does not need to request it. As long as we have the PT-VO for a range, we do not need any VO from servers if we insert rows within the range. For example, Figure 3.8b and 3.8d show such 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.

3.4.2 Tablet Changes

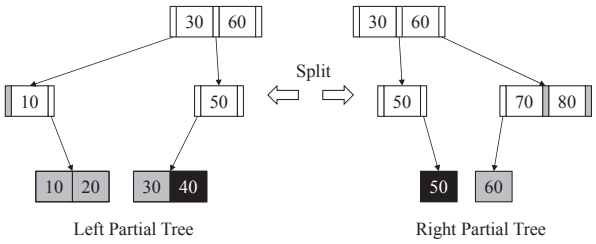
As the size of a tablet changes, the data owner may want to split a tablet or merge two tablets for load balancing. For both tablet split and merge, we need to rebuild an authenticated data structure and update the root hashes for newly created tablets. One straightforward way is to retrieve all data 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.

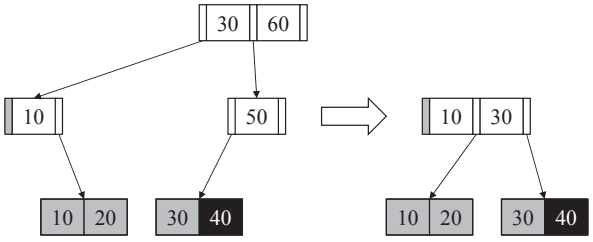
The main advantage of the above approach is its simplicity. It can be easily implemented. However,



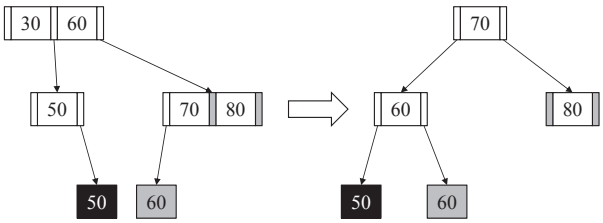
(a) Partial tree returned to the client.



(b) Split it into two partial trees.



(c) Adjust left partial tree.



(d) Adjust right partial tree.

Figure 3.10: Split the tablet at key 45.

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** $T_a \leftarrow T_l$

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 3.10a. 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 3.10b. 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) 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 3.10c and 3.10d. Due to the similarity of adjustments for both trees, we only describe the process for left tree in Algorithm 1.

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

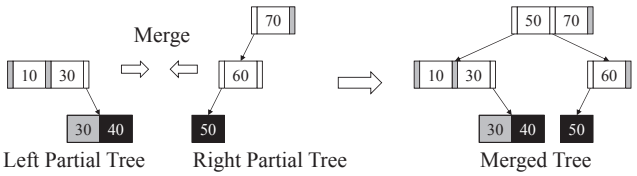


Figure 3.11: Tablet Merge.

shown in Figure 3.11; 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**

3.5 Analysis and Evaluation

3.5.1 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.

3.5.2 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.

3.5.3 Experimental Evaluation

System Implementation. We have implemented a prototype of iBigTable based on HBase [7], 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 [13]. 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 [20]. We use MD5 [9] to generate hashes.

Baseline Experiment. Before we evaluate the performance of write and read operations in iBigTable,

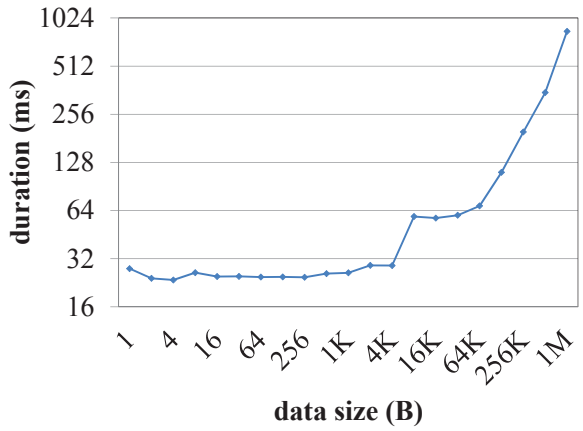


Figure 3.12: Time to Receive Data from Server.

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 3.12 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 3.13 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 3.13 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 3.14 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 3.15 shows the breakdown of performance overhead introduced by iBigTable, which shows the client computation overhead, the

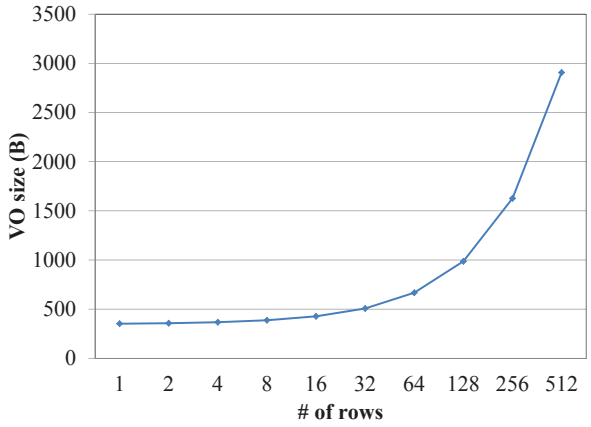


Figure 3.13: VO Size vs Number of Rows.

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%.

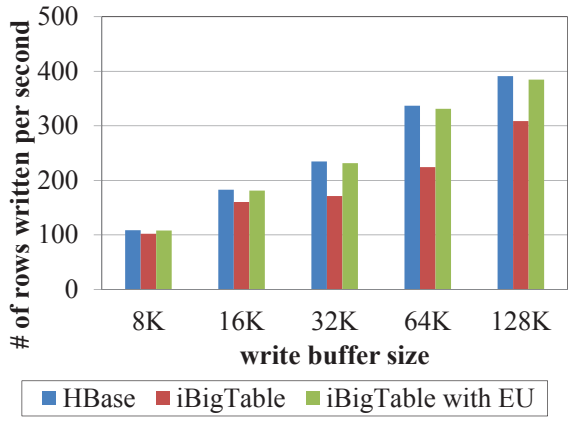


Figure 3.14: Write Performance Comparison between BigTable and iBigTable.

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 3.12. iBigTable with EU

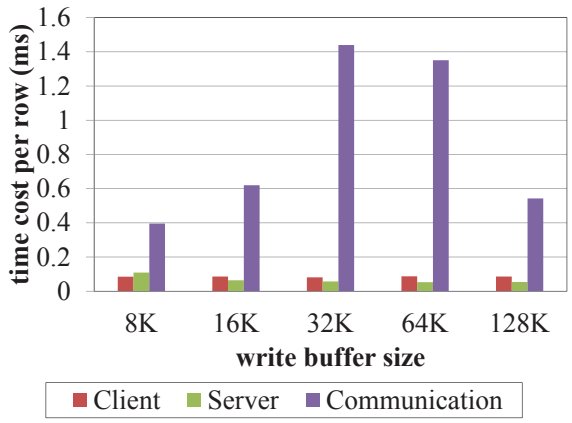


Figure 3.15: The Breakdown of iBigTable Write Cost.

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.

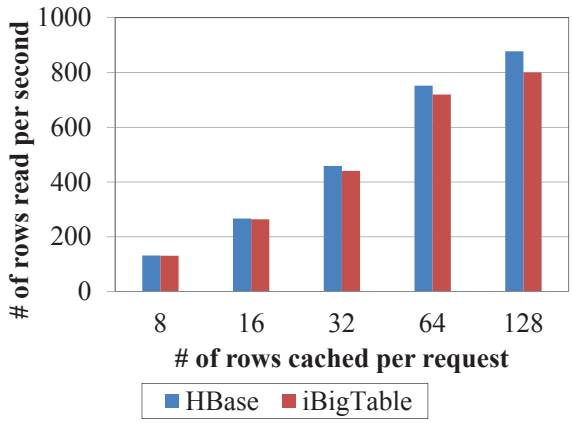


Figure 3.16: Read Performance Comparison between BigTable and iBigTable.

Read Performance. We also run experiments to evaluate the read performance overhead of iBigTable. Figure 3.16 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 3.17 shows the breakdown of iBigTable read overhead. The communication overhead can be explained by the result shown in Figure 3.12 because the total amount of data transmitted for the first

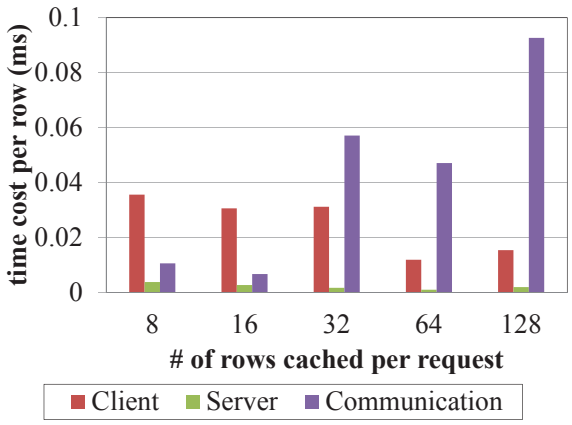


Figure 3.17: The Breakdown of iBigTable Read Cost.

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 3.18 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.

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

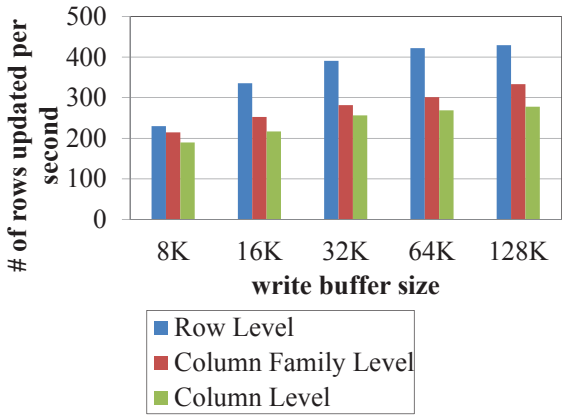


Figure 3.18: TL-MBT Update Performance in iBigTable.

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 3.19 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.

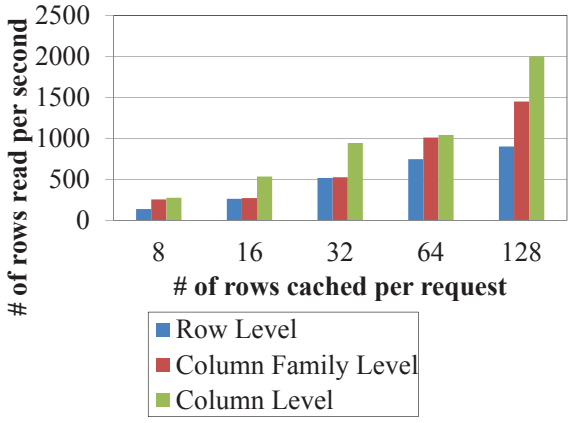


Figure 3.19: Projection Query with TL-MBT Support in iBigTable.

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. 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.

Chapter 4

IAODB: Integrity Assurance for Outsourced Databases without DBMS Modification

4.1 System Model

4.1.1 Database Outsourcing Model

Figure 4.1 shows our database outsourcing model with integrity protection. There are three types of entities: *data owner*, *database service provider* (DSP) and *clients*. A data owner uploads a database with data and authentication data to a DSP, which provides database functionality on behalf of the data owner. Clients send to the DSP queries to retrieve data and a *verification object* (VO), which can be used for data integrity verification.

In our outsourcing model, we leave the DSP without any change to its DBMS. Basically, the DSP does not even know where and how to store authentication data and when and how to return authentication data to clients for integrity verification. Everything related to data integrity verification is done at the client side, and data and authentication data updates are done by the data owner. In this way, data owners can provide integrity assurance for their outsourced databases without any special support from DSPs. Therefore, the adoption of database outsourcing with integrity assurance is completely decided by data owners themselves.

4.1.2 Assumptions and Attack Models

We make a few assumptions for the target database outsourcing model before we discuss our attack models. First, we assume that data owners and clients do not fully trust the services provided by DSPs. Second, since our scheme relies on digital signatures to provide integrity protection, we assume that the

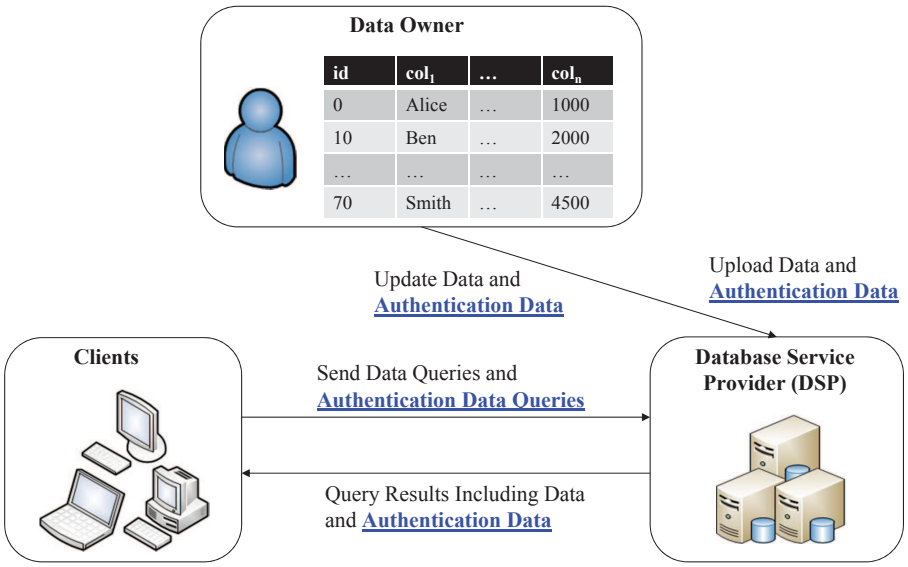


Figure 4.1: Non-Intrusive Database Outsourcing Model.

data owner has a pair of private and public keys for signature generation and verification. The public key is known to all clients. Moreover, like in many existing work [35, 40, 46, 50], we assume that the data owner is the only entity who can update its data. In addition, we assume that communications between DSPs and clients are through a secure channel (e.g., through SSL). Thus, DSPs and clients can detect any tampered communication.

Regarding attack models, we focus ourselves on the malicious behavior from a DSP since it is the only untrusted party in our target database outsourcing model. We do not have any assumption about what kind of attacks or malicious behavior a DSP may take. A DSP can behave arbitrarily to compromise data integrity. Typical malicious behaviors include, but not limited to, modifying a data owner's data without the data owner's authorization, returning partial data queried to clients and reporting non-existence of data even if data does exist. Further, it could return stale data to clients instead of executing queries over the latest data updated by the data owner [61].

4.1.3 Security Goals

We focus on addressing the integrity issues for database outsourcing. There are three aspects in data integrity: correctness, completeness, and freshness. First, the correctness checks if all records returned in a query result come from the original data set without being maliciously modified, which is usually achieved using digital signatures that authenticate the authenticity of records. Second, the completeness checks if all records satisfying conditions in a query are completely returned to clients. Note that it is possible to meet the correctness requirement without meeting the completeness requirement by only

returning a partial set of the query result. Third, the freshness checks if all records in a query result are the up-to-date data instead of some stale data. It is a challenging task to guarantee freshness since stale data is still valid data at some past point of time.

Regarding freshness, we propose mechanisms for data owners to efficiently compute signatures of updated data and guarantee the correctness of the signatures, which is the key to provide freshness guarantee. In the work, we do not focus on discussing how the latest signatures are propagated to clients for integrity verification purpose, where existing techniques [42,64] can be easily applied in our system.

4.2 Running Example

In this section, we take an example, which will be referred to throughout the work to ease the understanding of our scheme. Without loss of generality, we assume that a data owner has a database with a table called "data", which stores the data owner's data, shown in Figure 4.2.

id	col_1	col_2	col_3	col_4	…	col_n
0	Alice	F	20	NC	…	1000
10	Ben	M	30	NY	…	2000
20	Cary	F	42	CA	…	1500
30	Lisa	F	15	CA	…	3000
40	Kate	F	18	NY	…	2300
50	Mike	M	24	SC	…	4000
60	Nancy	F	36	VA	…	2300
70	Smith	M	12	TA	…	4500

Figure 4.2: A Relational Data Table.

The table has several columns. The *id* column is a unique key or indexed. Besides, there are n columns $\{col_1, ..., col_n\}$ containing arbitrary data. We do not assume that specific types of data should be stored in the table. This example will be used throughout the work to explain our ideas, design decisions, and verification steps.

To provide integrity protection, there are several things we need to do. First, we need to decide which ADS we should use to organize the authentication data for the data stored in the table. Second, we need to figure out how to store the ADS into the database where the data table resides so that we can support integrity verification without modifying existing DBMSs. We will illustrate how data owners can enable integrity protection for their outsourced databases without requiring DSPs to modify anything at their end.

4.3 System Design

4.3.1 Authenticated Data Structure

One of the major design choices we need to make is Authenticated Data Structure (ADS), which is used to organize authentication data. There are two options: signature aggregation based ADS and Merkle hash tree based ADS. Since our goal is to provide the integrity protection without modifying existing DBMSs, we need to figure out a way to retrieve authentication data without requiring some functionality that does not exist in current DBMSs. We observe that there are several disadvantages of developing a scheme based on signature aggregation based ADS. First, to minimize communication cost, signature aggregation operation needs to be done dynamically in DBMSs, which unfortunately is not supported. Moreover, although signature aggregation based ADS can guarantee correctness and completeness, it is unknown how to efficiently guarantee freshness using signature aggregation based approaches [40]. Additionally, 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 [40].

Thus, we choose to adapt MHT-based ADS, in particular, Merkle B-tree (MBT) [40]. MHT-based ADS can not only guarantee correctness and completeness, but also provide efficient freshness protection since only one root hash needs to be maintained correctly. Furthermore, the relationship between parent node and child nodes may bring us some opportunities to design an efficient scheme to retrieve authentication data. Figure 4.3 shows a Merkle B-tree created based on the table introduced in Section 4.2. The values in the *id* column are used as keys in the MBT. A hash h_i is associated with a pointer in an internal node or a record in a leaf node. For simplicity, the hashes associated with pointers and records in nodes of the MBT are not shown in the figure. The hash of a record in a leaf node is the hash value of the data record in the data table. The hash associated with a pointer in an internal node is the hash of concatenating all hashes in the node pointed by the pointer.

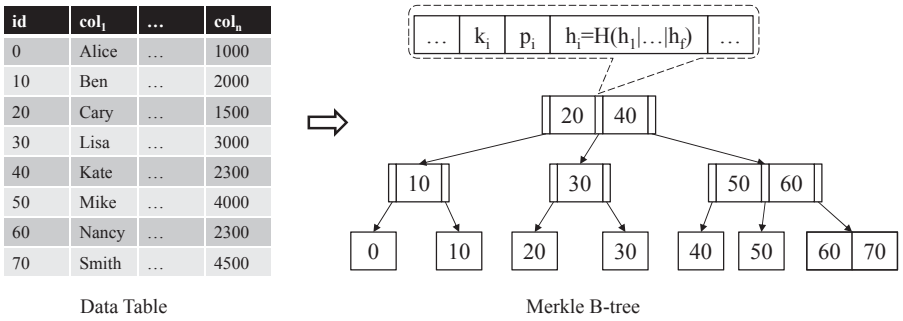


Figure 4.3: Data Table to Merkle B-tree.

Now the question is how to upload the authentication data along with data to a DSP so that the authentication data can be retrieved using existing functionality provided by the DSP. One straightforward solution is to serialize the MBT into a binary stream and store all data in the stream into a blob field of a data table. Since the DSP knows nothing about the internal structure of the data stored in a blob field and there is no known way to extract the authentication data for a data record from the blob field, the only thing clients can probably do is to retrieve all data in the blob field along with a data record queried so that they can rebuild the MBT, find the authentication data by themselves and verify the integrity of the record returned from the DSP. However, it incurs significant communication cost between clients and the DSP and computation cost in clients since the binary data of the MBT could be very large depending on the number of records in the data table, which renders the scheme impractical. In the following, we will explore different choices to upload authentication data into a database so that we can retrieve necessary authentication data efficiently.

4.3.2 Identify Authentication Data

Based on the above analysis, we have to think about how to dissect the tree structure so that we can store authentication data and retrieve necessary authentication data efficiently. The first thing we need to do is to identify pointers in internal nodes and records in leaf nodes of a MBT since each pointer or record is associated with a piece of authentication data, that is, a hash. And also we need to model their parent-child and sibling relationships. Besides, we need to preserve the ordering of pointers or records in a node of a MBT.

Existing Approaches. There are a few widely-used models such as adjacency list, path enumeration, nested set and closure table to store tree-like hierarchical data into a database [19,60]. With an adjacency list, it is easy to find the parent of a pointer or record since it models the parent-child relationship directly, but to find its ancestor, we have to go through the parent-child relationship step by step, which could make the process of retrieving VO inefficient. And also the adjacency list model does not consider the order of pointers or records in a node, which is important for hash generation and integrity verification. The path enumeration model uses a string to store the path of each pointer or record, which is used to track the parent-child relationship. Unlike the adjacency list model, it is easy to find an ancestor of a pointer or record in a node. But same as the adjacency list, the path enumeration does not consider the order of pointers or records in a node. In the nested set model, it is very inefficient to find a parent of a pointer or a record as it requires joining two tables. Similarly, the closure table does not consider the ordering of pointers or records in a node, and needs to consume more storage than other models. Besides, Miguel [5] proposed a different scheme called Genealogical Identifier, which contains the complete genealogy of a node similar to the path enumeration model. What different is that it considers the ordering of nodes with a common parent. In terms of performance, many operations discussed in [5] are based on string operations and index scan of the tree instead of index seek, which is inefficient.

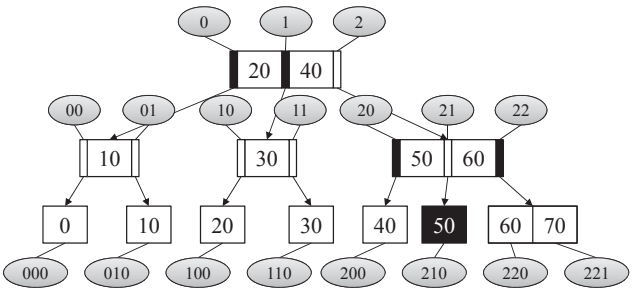


Figure 4.4: Radix-Path Identifier.

data_auth (max level - 2)

id	rpid	hash	level
-1	0	TvJtus	2
20	1	asdwS	2
40	2	DFsQ	2
-1	0	Kjdaw	1
10	1	Ujrw	1
-1	4	JHds	1
30	5	iueDs	1
-1	8	Jdiw.	1
50	9	.dkaw	1

id	rpid	hash	level
60	10	Udew	1
0	0	nudg	0
10	4	Q9ej	0
20	16	wVi2	0
30	20	kidDs	0
40	32	Kdie*	0
50	36	8dFes	0
60	40	Iurw	0
70	41	KJdw	0

(a) Single Authentication Table (SAT)

data_mapping

level	table
2	data_auth2
1	data_auth1
0	data

data_auth2 (Level 2)

id	rpid	hash
-1	0	TvJtus
20	1	asdwS
40	2	DFsQ

data_auth1 (Level 1)

id	rpid	hash
-1	0	Kjdaw
10	1	Ujrw
-1	4	JHds
30	5	iueDs
-1	8	Jdiw.
50	9	.dkaw
60	10	Udew

data (Level 0)

id	col_1	…	col_n	rpid	hash
0	Alice	…	1000	0	nudg
10	Ben	…	2000	4	Q9ej
20	Cary	…	1500	16	wVi2
30	Lisa	…	3000	20	kidDs
40	Kate	…	2300	32	Kdie*
50	Mike	…	4000	36	8dFes
60	Nancy	…	2300	40	Iurw
70	Smith	…	4500	41	KJdw

(b) Level-based Authentication Table (LBAT)

Figure 4.5: Authentication Data Organization.

Radix-Path Identifier. To address the disadvantages of existing approaches, we propose a novel and efficient scheme called *Radix-Path Identifier*. The basic idea is to use numbers based on a certain radix to identify each pointer or record in a MBT. Figure 4.4 shows all identifiers as base-4 numbers for pointers or records in the tree based on a radix equal to 4. Given a MBT, the *Radix-Path Identifier* of a pointer or record depends on its level and position in the MBT. To illustrate this scheme, suppose that the fanout of a MBT is f. The radix base r_b could be any number larger than or equal to f. l denotes the level where a node resides in the MBT. The l of the root node is 0. i denotes the index of a pointer or record in a node, ranging from 0 to f. The *Radix-Path Identifier* $rpid$ of a pointer or record can be computed using the following equation:

$$rpid = \begin{cases} i & \text{if } l == 0, \\ rpid_{parent} * r_b + i & \text{if } l > 0. \end{cases} \tag{4.1}$$

Note that $rpid_{parent}$ is the *Radix-Path Identifier* of its parent pointer in the tree. Equation 4.1 models not only the relationship among pointers or records in one node, but also the parent-child relationship among nodes. The identifier of each pointer or record in the root node is i. With identifiers in the root node, we can use the second part of Equation 4.1 to compute identifiers of pointers or records in their

child nodes in the tree. In this way, all identifiers can be computed starting from the root node to the leaf nodes in the tree.

The proposed *Radix-Path Identifier* scheme has several important properties:

- Identifiers of pointers or records in a node are continuous, but not continuous between two sibling nodes. For example, the base-4 numbers $20, 21, 22$ are continuous.
- From an identifier of a pointer or record in a node, we can easily find the identifier of its parent pointer based on the fact that $rpid_{parent}$ equals to $\lfloor rpid/r_b \rfloor$.
- From an identifier of a pointer or record in a node, we can easily calculate the min and max identifiers in the node, which are $(\lfloor rpid/r_b \rfloor) * r_b$ and $(\lfloor rpid/r_b \rfloor) * r_b + (r_b - 1)$.
- From an identifier of a pointer or record in a node, we can easily compute the index i of the pointer or record in the node, which is $rpid\%r_b$.

The above properties will be utilized for efficient VO retrieval and authentication data updates. We will elaborate those details later in Section 4.4.

4.3.3 Store Authentication Data

Once we identify each pointer or record in nodes of a MBT, the next step is how we can store the authentication data associated with them into a database. In the following, we propose two different designs - Single Authentication Table (SAT) and Level-based Authentication Table (LBAT), and discuss their advantages and disadvantages.

SAT: Single Authentication Table. A straightforward way is to store all authentication data as data records called *Authentication Data Record* (ADR) into one table in a database, where its corresponding data table is stored. Figure 4.5(a) shows all authentication data records in a single table for the data table described in the running example. The name of the authentication table adds a suffix "_auth" to the original table name "data". The authentication table has 4 columns: *id*, *rpid*, *hash* and *level*. *id* column stores values from *id* column of the data table, which are keys in the B+ tree except "-1". Note that since the number of keys is less than the number of pointers in the internal nodes in a B+ tree node, we use "-1" as the *id* for the left-most pointers in the internal nodes. *rpid* records identifiers for pointers or records in the B+ tree. *hash* column stores the hash values of pointers or records in the B+ tree, which is essential for integrity verification. *level* stores values indicating the level of a pointer or record in the B+ tree. The *level* value is necessary for searching the *rpid* for a data record given an *id* of the data record because the *rpid* values could be same in different levels. The level of a leaf node is 0, and the level of the root node is the maximum level.

Although SAT is simple and straightforward, it has several disadvantages, which makes it an inefficient scheme. First, queries to retrieve ADRs could be inefficient since indexes are built based on all

ADRs. For example, even if we want to retrieve an ADR associated with a pointer in the root node, we still need to search an index built based on all ADRs in the MBT. If we store ADRs in the root node in another table, then the index to be searched is built on the ADRs in the root node. Second, updates could be inefficient since one data record update usually requires updating ADRs in different levels. With table level lock, it is not allowed to concurrently execute ADR updates since all ADR updates have to be executed over the only one table. Although concurrent updates can be enabled with row level lock, it may consume much more database server resources, which may not be desired. Third, it may require using join query to find the *rpid* a data record since the data table is separated from the only one authentication data table. Fourth, updates to a data record and its ADR in the leaf level are not able to be merged to improve the performance since they go to different tables.

LBAT: Level-based Authentication Table. To resolve the above issues, we propose a Level-based Authentication Table (LBAT). In this scheme, instead of storing all ADRs into one table, we store ADRs in different levels to different tables. We create one table per level for a MBT except the leaf level and also create a mapping table to indicate which table corresponds to which level. For nodes in the leaf level of the MBT, since each data record corresponds to an ADR in leaf nodes, we extend the data table by adding two columns - *rpid* and *hash* to store ADRs instead of creating a new table, which reduces the redundancy of *id* values and also the update cost to some extent. Figure 4.5(b) shows all tables created or extended to store ADRs and the mapping table for the data table described in the running example. Tables for different levels have different number of records. For the root level, it may only contain a few records. Also, the number of records in the mapping table is equal to the number of levels in the MBT. We name those tables by adding a suffix such as "_mapping", "_auth0", etc, based on table types and levels.

The proposed LBAT scheme presents several advantages. First, the indexes built based on those tables are more efficient than those in SAT because the number of records of each table in LBAT are much smaller than that in SAT. Thus, it could improve the efficiency of queries to a great extent. Second, since ADRs in different levels are stored in different authentication tables, it makes concurrent updates possible with table level lock, which also allows to design efficient concurrent update mechanisms. Third, since we store ADRs in the leaf level along with data, it makes it straightforward to retrieve the *rpid* of a data record. Fourth, due to the same advantage, it is easy to merge updates for a data records and its ADR in the leaf level for performance improvement.

In the work, we use LBAT to store authentication data records. In the following sections, we will elaborate how to retrieve authentication data records along with data records efficiently to provide a practical integrity protection for exist DBMSs without modifying them.

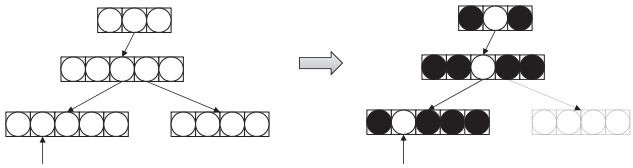


Figure 4.6: Retrieval of Authentication Data.

4.3.4 Extract Authentication Data

Figure 4.6 shows an example of what authentication data need to be retrieved for a data record. In the left side of Figure 4.6, it only shows the necessary part of a MBT as an example and an arrow pointing to the record for which we want to retrieve authentication data to verify its integrity. The right side of Figure 4.6, shows the authentication data records that need to be retrieved in black in order to compute the root hash of the MBT for integrity verification. To extract the ADRs for the record based on LBAT, we can utilize the properties of our *Radix-Path Identifier*. First, given a *rpid* of a record, we can easily compute the *rpid* of its parent pointer. Second, we can find the ADRs of records or pointers in the same node based on the fact that they have the same parent *rpid*, which can be computed by using any one of their *rpid*. Once we receive all related ADRs, we can compute the root hash since we can infer the tree structure from the *rpid* values, which uniquely models the relationship among pointers, records and nodes in the MBT.

Note that since the DSP is only assumed to provide standard DBMS functionalities, all the above operations have to be realized by SQL queries issued by the client. Here we explore four different ways - Multi-Join, Single-Join, Zero-Join and Range-Condition, to find the authentication data records for a record based on LBAT in the following, for each of which we will use specific examples to show how it works. All examples are based on the data presented in the running example. Suppose that we want to verify the integrity of the data record with the *id* 50. The ADRs needs to be returned shown as the black parts in Figure 4.4, which is also highlighted with a black background in Figure 4.5(b).

Multi-Join. In this scheme, we try to minimize the number of queries to retrieve all necessary authentication data for a record by joining multiple authentication tables together. Here we only need to use the following one query to retrieve all related authentication data for the record with id 50. The following shows the SQL statement we use to retrieve the authentication data of the record. Note that 4 is the radix base r_b we choose for the running example and 50 is the id of the record.

```
-- retrieve all authenticatoin data in one query with joins
select da0.rpid as rpid0,da0.hash as hash0,
   da1.rpid as rpid1,da1.hash as hash1,
   da2.rpid as rpid2,da2.hash as hash2
from data t0
left join data da0 on da0.rpid/4 = t0.rpid/(4)
```

```
left join data_auth1 da1 on da1.rpid/4 = t0.rpid/(4*4)
left join data_auth2 da2 on da2.rpid/4 = t0.rpid/(4*4*4)
where t0.id=50;
```

Although we can use only one query to retrieve all authentication data, redundant data will be returned since the SQl statement uses “left join” across several authentication tables, which definitely increases the communication cost. Moreover, since redundant data is returned, in the client side, we have to do a little bit more work to filter out redundant data and find necessary authentication data and compute the root hash.

Single-Join. Since the Multi-Join scheme retrieves a lot of redundant data, the Single-Join scheme aims at eliminating the redundant data in the query result. Instead of using one query with multiple join tables, it uses multiple queries, each of which joins one authentication table. The following shows the SQL statements we use to retrieve the authentication data using Single-Join.

```
-- level 2, 1, 0 (from root level to leaf level)
select l1.rpid,l1.hash from data t0
left join data l1 on l1.rpid/4 = t0.rpid/(4) where t0.id=50;

select l1.rpid,l1.hash from data t0
left join data_auth1 l1 on l1.rpid/4 = t0.rpid/(4*4) where t0.id=50;

select l1.rpid,l1.hash from data t0
left join data_auth2 l1 on l1.rpid/4 = t0.rpid/(4*4*4) where t0.id=50;
```

As we can see from the above SQL statements, multiple result sets will be returned for authentication data, but the redundancy is minimized dramatically compared with the Multi-Join scheme. Each query doing a join between the data table and one authentication table retrieves the authentication data in one node based on the “left join” condition.

Zero-Join. We observe from the Single-Join scheme that in each join query what we actual need is the *rpid* of the record 50. If we know its *rpid*, we can eliminate the “left join” completely from the SQL statements. The following shows the SQL statements we use to retrieve the authentication data without joining any table.

```
-- find the rpid of the data record with the id 50
declare @rowrpid AS int;
set @rowrpid=(select top 1 rpid from data where id=50);

-- level 2, 1, 0 (from root level to leaf level)
select rpid,hash from data where rpid/4=@rowrpid/(4);
select rpid,hash from data_auth1 where rpid/4=@rowrpid/(4*4);
select rpid,hash from data_auth2 where rpid/4=@rowrpid/(4*4*4);
```

The major difference is that we declare a "rowrpid" variable to store the *rpid* of the record, which is retrieved from the first query. After that, we use the "rowrpid" for other queries to retrieve the authentication data for nodes in different levels. Although it needs to execute one more query, it eliminates the "join" clause completely.

Range-Condition. We observe that the execution of the above queries does not utilize the indexes created on the *rpid* field in the authentication tables. Instead of doing an index seek, each of them actually does an index scan, which is inefficient and incurs a high computation cost in server side. To utilize indexes, we propose a new method called Range-Condition to retrieve authentication data for records. The following shows the SQL statements we use to retrieve the authentication data for the record 50 using Range-Condition.

```
-- find the rpid of the data record with the id 50
declare @rowrpid AS int;
set @rowrpid=(select top 1 rpid from data where id=50);

-- level 2, 1, 0 (from leaf level to root level)
select rpid,hash from data
where rpid>=(@rowrpid/(4))*4 and rpid<(@rowrpid/(4))*4+4;

select rpid,hash from data_auth1
where rpid>=(@rowrpid/(4*4))*4 and rpid<(@rowrpid/(4*4))*4+4;

select rpid,hash from data_auth2
where rpid>=(@rowrpid/(4*4*4))*4 and rpid<(@rowrpid/(4*4*4))*4+4;
```

As can be seen from the figure, the major difference from Zero-Join is the *where* condition. Instead of using equality, the Range-Condition uses a range query selection based on the *rpid* column. The range query retrieves the same set of results with the equality condition used in Zero-Join. Thus, they both return the same set of authentication data records, and Single-Join does that too. However, with the range query on the *rpid* field, it can utilize indexes built on the *rpid* column, which minimizes the computation cost in server side.

In Section 4.5, we will compare the performance overhead caused by different methods of authentication data retrieval. In the following, our discussion will use *Range-Condition* by default to retrieve authentication data.

4.4 Data Operations

In this section, we will illustrate the details of handling basic queries such as *select*, *update*, *insert* and *delete* with integrity protection efficiently based on our design using the running example. Without loss

of generality, we assume that clients always have the latest root hash of the table for integrity verification, and we focus on discussing how to retrieve authentication data from DSPs.

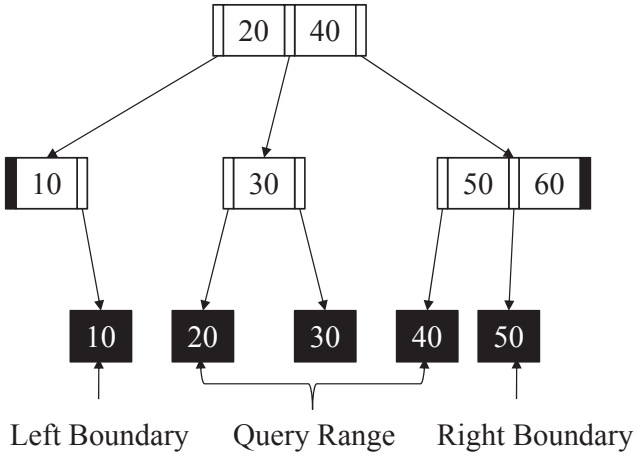


Figure 4.7: Range Query with Integrity Protection.

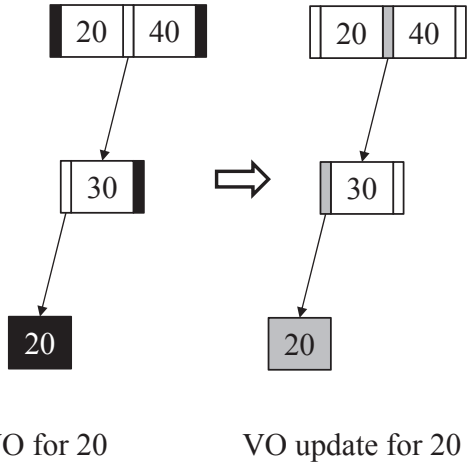


Figure 4.8: Update with Integrity Protection.

4.4.1 Select

As discussed in Section 4.3.4, we can retrieve authentication data for a *Unique Select* query, which returns only one data record based on a unique key selection. Thus, we focus on the discussion of how to handle a *Range Select* query with integrity protection, which retrieves records within a range.

The verification process for *Range Select* queries is different from *Unique Select* queries. First, we need to find the two boundary keys for the range of a range query. For example, for a range query with a range from 15 to 45, we need to identify its two boundaries, which are 10 and 50 in this case. Although DBMSs do not provide a function to return the boundary records directly, we can use the following two queries to figure out what the left and right boundaries are for a query range:

```
select top 1 id from data where id < 15 order by id desc
select top 1 id from data where id > 45 order by id asc
```

Then, to retrieve the authentication data for the range query, we only need to retrieve the authentication data for both boundaries, which is similar to the way we use to retrieve authentication data object for a data record since the authentication data for records within the range are not necessary and they will be computed by using the returned records. Figure 4.7 shows the authentication data records and the data records that need to be retrieved for the range query from 15 to 45.

To execute the range query with integrity protection, we need to rewrite the range query by adding SQL statements shown in the following. Then, we execute all SQL statements in one database transaction. Once the result with authentication data is returned, we verify the integrity of the query result using the authentication data. If the verification succeeds, the data result is returned to the client as before; otherwise, an integrity violation exception could be thrown to warn the client of the integrity verification failure.

```
-- find the left boundary key and the left boundary rpid
declare @keyLB AS int,@rpidLB AS int;
select top 1 @keyLB=id,@rpidLB=ripd from data
where id<15 order by id desc;

-- find the right boundary key and the right boundary rpid
declare @keyRB AS int,@rpidRB AS int;
select top 1 @keyRB=id,@rpidRB=ripd from data
where id>45 order by id asc;

-- retrieve authentication data records for the left boundary from leaf level to root level
select ripd,hash from data
where ripd>=(@rpidLB/4)*4 and ripd<(@rpidLB/4)*4+4;
```

```
select ripd,hash from data_auth1
where ripd>=(@rpidLB/4*4)*4 and ripd<(@rpidLB/4*4)*4+4;

select ripd,hash from data_auth2
where ripd>=(@rpidLB/4*4*4)*4 and ripd<(@rpidLB/4*4*4)*4+4;

-- retrieve authentication data records for the right boundary from leaf level to root level
select ripd,hash from data
where ripd>=(@rpidRB/4)*4 and ripd<(@rpidRB/4)*4+4;

select ripd,hash from data_auth1
where ripd>=(@rpidRB/4*4)*4 and ripd<(@rpidRB/4*4)*4+4;

select ripd,hash from data_auth2
where ripd>=(@rpidRB/4*4*4)*4 and ripd<(@rpidRB/4*4*4)*4+4;
```

The overhead to provide data integrity for range queries consists of both computation and communication cost. The computation cost in the client side includes two parts: rewriting range query and verifying data integrity. The computation cost in the server side is the execution of additional queries for authentication data retrieval. The communication cost between them includes the text data of additional queries and the authentication data returned along with the data result.

This process can also handle *Unique Select* queries. However, it requires to retrieve authentication data for both left boundary and right boundary, which may not be necessary. If the key does not exist, we have to resort to the process of handling range queries, where we can check left boundary and right boundary to make sure the record with the key does not exist.

4.4.2 Update

Single Record Update

When a data record is updated, we need to update its authentication data (mainly hash values) so that we can guarantee data integrity. For updating a record, we assume that the record to be updated already exists in the client side and the VO for the updated record cached in the client too. Otherwise, we retrieve the data record and the VO first, then update it and its authentication data. It consists of two round-trips, which may cause a large overhead.

Figure 4.8 shows the VO in black for the record 20 in the left side and the hash values in gray to be updated once the record is updated. Each data update requires an update on each authentication data table. It means if the MBT tree's height is h, then the total Number of update queries is $h + 1$. In this case, we need to actually update 4 records. One of them is to update the data record and three of them is to update the authentication data records. The following shows the three additional update statements.

The generation of update queries for authentication data is simple since we know the *rpid* of the data record to be updated, and then we can easily compute its parent *rpid* and generate update queries.

```
update data set [hash]='aNcAh0i1qC2ebq/z7jn3pw==' where [ripd]=16;
update data_auth1 set [hash]='5XGM7Y0PqxdbZnb8atpL/Q==' where [ripd]=4;
update data_auth2 set [hash]='VOUDDXH+bxoYBVevXDpY/Q==' where [ripd]=1;
```

Since the authentication data table for the leaf level of a MBT is combined with the data table, we can combine two update queries into one to improve the performance. Thus, in this case we only need 3 update queries instead of 4. All update queries are executed within one transaction. So, the consistency of data records and authentication data is guaranteed by the ACID properties of DBMSs, and the integrity is also guaranteed since the verification and the root hash update are done directly by the data owner.

Batch Update and Optimization

Suppose that we want to update x records at one time. Actually, as the number of records to be updated increases, the total number of update queries we need to generate to update both data and authentication data linearly increases. In this case, the total number of update queries is $x * h$. We observe from those update queries that several update queries try to update the same authentication data record again and again due to the hierarchical structure of a B+ tree. We also notice that each update SQL statement only updates one authentication record in one table. Actually, we just need to get the latest hash of the authentication data record, and do one update. To do that, we need to track all update queries for each table, find the set of queries to update one authentication data record in an authentication table, and remove all of them except the latest one. In this way, the number of necessary update queries could be much less than the number of update queries we generate before. We believe that this process called *MergeUpdate* improves the performance of batch update to a great extent, which will be shown in Section 4.5.

4.4.3 Insert

Single Record Insert

Inserting a record is much more complicated than updating a record since an insertion may result in the MBT structure changes such as node split, split propagation to parent nodes, which may require *rpid* changes for many data and authentication records since the *rpid* values are continuous in one node. We need to maintain this property for data integrity verification. Before that, we use a new type of VO - Partial Tree VO (PT-VO) [61], which has both hashes and keys 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 a data owner receives a PT-VO from a DSP, it can directly update the PT-VO locally to compute the new root hash of

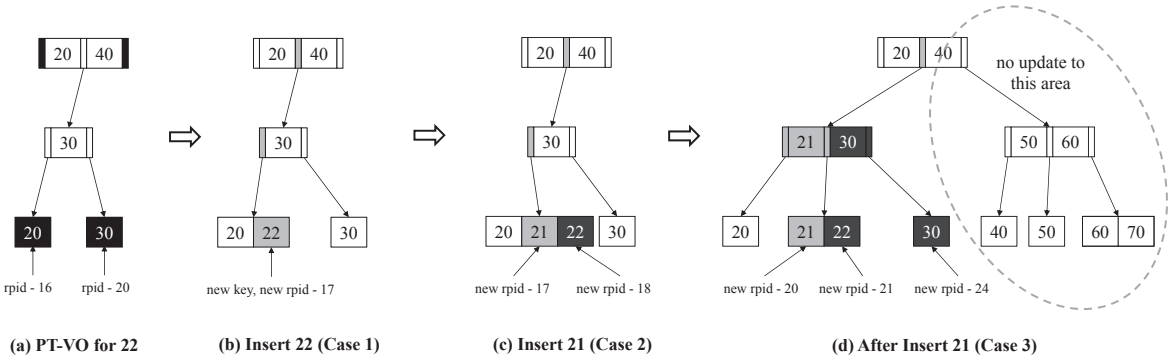


Figure 4.9: Insert With Integrity Protection.

the original authenticated data structure. As an example, Figure 4.9(a) shows the PT-VO for the record 22, which does not exist in the data table. In general, steps to insert a record into a table with integrity protection are described as follows:

1. Retrieve the PT-VO for the new record to be inserted and verify the non-existence of the new record.

2. Generate update and insert SQL statements for both the new record and the authentication data records.

3. Send all SQL statements to a DSP in one round-trip and execute all of them in one transaction at the DSP.

The above steps includes two network round trips between a client and a DSP. One is to retrieve the PT-VO and the other is to insert a new record and update its corresponding authentication data records. During this process, the difficult part is to generate the SQL statements to update authentication data records. There are three cases we need to consider: 1) insert one record without node split and any *rpid* changes of other records; 2) insert one record without node split, but with *rpid* changes of other records; 3) insert one record with node split and rpid changes of other records. We discuss how to generate necessary SQL statements to update authentication data records for each case in details. To ease the understanding of each case, we use specific examples derived from the running example, shown in Figure 4.9.

Case 1. It is pretty similar to update a data record. The only difference is that we insert a new data record instead of updating an existing one. The authentication data update process is almost same with that for data record update, as shown in Figure 4.9(b). Here we can see one benefit of the *Radix-Path Identifier* that there are available identifiers for new records between sibling nodes in a MBT. Thus, we do not need to update *rpid* values of other authentication records sometimes.

Case 2. In this case, we need to update the *rpid* values for some records. For example, after 22 is inserted we insert 21, shown in Figure 4.9(c). Since the 21 is inserted in the middle of 20 and 22, the *rpid* of 22 should be updated to 18 from 17, and the 17 will be the *rpid* of the newly inserted record 21. We also need to pay attention to the execution order of update queries. In this case, we should update the *rpid* values for existing records first, and then insert the new record 21.

Case 3. We continue the example in Case 2. Since the B+ tree in our example only allows two records in a leaf node, we need to split the leaf node into two leaf nodes. Figure 4.9(d) shows the partial B+ tree structure after the leaf node is split. In the figure, the gray and dark parts of authentication data need to be updated correctly, and we do not need to update the authentication data within the dash circle since nothing is changed in that part. We observe from this case that even if a node is split or even if the node split is propagated to parent nodes, it does not necessarily mean that we need to update the *rpid* values of authentication records that are in the right of the inserted new record.

For the above all cases, the update and insert queries are executed within one transaction like how we handle updating a record. Thus, the consistency between data and authentication data is guaranteed by the ACID properties of existing DBMSs. The following shows the generated update and insert SQL statements based on the example used for the above cases.

```
-------------------------------------------------------------------------------
-- update queries for authentication data for inserting 22
-------------------------------------------------------------------------------
update [dbo].[data] set [ripd]=17,[hash]='WFL32gcL49vgu3FGTnrGjA==' where [id]=22;
update [dbo].[data_auth1] set [hash]='y6Mg6yFJgy7xtMC+XOlV4w==' where [ripd]=4;
update [dbo].[data_auth2] set [hash]='5aKaSct3okrKJoQ+U6teOA==' where [ripd]=1;

-------------------------------------------------------------------------------
-- update queries for authentication data for inserting 21
-------------------------------------------------------------------------------

-- update ripd
update [dbo].[data] set [ripd]=[ripd]+1 where [ripd]>=17 and [ripd]<=18;

-- update the hash for newly inserted record
update [dbo].[data] set [ripd]=17,[hash]='BGbQuR2ngL3CzjR2YvkhZw==' where [id]=21;

-- update the hash of the node where children is split
update [dbo].[data_auth1] set [hash]='740mHexVWcUeeKVL40yCgw==' where [ripd]=4;

-- update ripd since a new node is inserted into the MBT because of node split
update [dbo].[data_auth1] set [ripd]=[ripd]+1 where [ripd]>=5 and [ripd]<=7;
```

```
-- update ripd for child nodes
update [dbo].[data] set [ripd]=[ripd]+4 where [ripd]>=20 and [ripd]<=31;

-- insert a new authentication record for the newly created node because of node split
insert into [dbo].[data_auth1] ([id],[ripd],[hash]) values(21,5,'T3AiQkvu3n8YM0NSWPO9OA==');

-- update ripd for children of newly created node because of node split
update [dbo].[data] set [ripd]=[ripd]+3 where [ripd]>=17 and [ripd]<=19;

-- update the hash of the pointer with ripd 1
update [dbo].[data_auth2] set [hash]='FjYV6cRitM2vwfsRPjX8lQ==' where [ripd]=1;
```

Based on our discuss so far, one important step for integrity protection is to rewrite user queries by adding additional queries to retrieve authentication data or update authentication data depending on user queries. The rewrite process for *select* is relatively easier than that for *update*, *insert* or *delete*, and it is especially complicated for *insert* and *delete*. To generate the SQL statements for authentication data when *update*, *insert* or *delete* happens, we need to handle different events from the MBT, for example node update, key insert, key delete, root split, etc. One trick part is that we treat an internal node split as a new key insert in its parent node. In the work, we only discuss one of the most important parts for *insert*, which is the process to handle key insert event in a MBT. Algorithm 3 shows the pseudo-code to explain the general process to handle key insert event, which calls a sub-process described in Algorithm 4.

Batch Insert and Optimization

We observe the fact from single record insert that a data owner is able to compute the new root hash based on a PT-VO. Based on this observation, we can insert multiple data records and update authentication data records in one database transaction to maximize the performance of batch insert and maintain the consistency between data records and authentication data records.

As we can see from the discussion above, it is much more complicated to handle insertions than to handle updates when we need to provide integrity protection. First, it may generate more SQL statements compared with the update case. Second, it may not only update hash values of authentication records, but also update the *rpid* values of authentication records. Third, regarding computation cost, it may need to update much more authentication data records compared with the update case. Although it is more complicated than the update case, there are still redundant SQL statements that can be merged.

The process to merge SQL statements for *insert* is a little bit more complicated than that for *update* since the SQL statements may update the **rpid** values of authentication records. To merge SQL statements generated for insertions, we proceed the *MergeUpdate* process table by table like what we do for update, where all SQL statements are strictly ordered by the generation time. One major differ-

Algorithm 3 HandleKeyInsertEvent(n, sm_n, l_n, k, ip, h)

Require: n - node, sm_n - start rpid of node, l - node level, k - key, ip - inserted position, h - hash

if n is leaf node **then**
  if $ip < (n.numOfKeys - 1)$ **then**
    $sm = sm_n + ip$ {sm - start rpid}
    $em = sm_n + n.numOfKeys - 1$ {em - end rpid}
    $mc = 1$ {mc - rpid change}
    update $rpid$ by adding mc from sm to em
  end if
  update $rpid$ and hash for the inserted data record
else
  if $ip < n.numOfKeys$ **then**
    $sm = sm_n + ip$
    $em = sm_n + r_b - 1$
    $mc = 1$
    for $i = l \rightarrow maxLevel$ **do**
      update $rpid$ by adding mc from sm to em

      $sm = sm * r_b$
      $em * r_b + r_b - 1$
      $mc = mc * r_b$
    end for
  end if
  InsertAuthenticationRecord(n, sm_n, l, k, ip, h)
end if

Algorithm 4 InsertAuthenticationRecord(n, sm_n, l, k, ip, h)

Require: n - node, sm_n - start rpid of node, l - node level, k - key, ip - inserted position, h - hash

insert a new authentication record with k, $sm_n + ip$ and h
$sm \leftarrow$ new child node start rpid
$em \leftarrow$ new child node max rpid
$mc \leftarrow$ initial rpid change

for $i = l + 1 \rightarrow maxLevel$ **do**
  update $rpid$ by adding mc from sm to em

  $sm = sm * r_b$
  $em * r_b + r_b - 1$
  $mc = mc * r_b$
end for

ence is that we should not run the *MergeUpdate* process across all SQL statements for a table. Instead, we should start from the first update statement, find a set of continuous SQL statements that update the hash values of authentication records, apply the *MergeUpdate* process to the set of statements just found, and continue this process until all SQL statements have been scanned. In this way, we can reduce the number of SQL statements while correctly updating authentication data without breaking the integrity protection.

4.4.4 Delete

Deleting a record is also complicated like inserting a record. When a record is deleted in a B+ tree, some nodes may need to be merged together, where we may need to update the *rpid* values of some authentication records depending on specific cases. Due to the similarity between the processes to handle deletion and insertion, we do not elaborate details in the work.

Here we would like to discuss an alternate solution, which could make the deletion process simpler and more efficient. In this solution, when a data record is deleted, we also delete the authentication data record, but we do not merge leaf nodes even if there are no records in leaf nodes. In this case, there will be no merge and merge propagation on the Merkle B-tree structure during the deletion of a record. However, data owners may need to run a background process like clean-up or garbage collection to reconstruct the Merkle B-tree so that the height of the MBT can be reduced after a lot of deletions happen.

4.5 Experimental Evaluation

System Implementation. We have implemented the Merkle B-tree and the query rewrite algorithms for clients, which is the core of generating select, update and insert SQL statements to operate authentication data. We also built a tool to create authentication tables and generate authentication data based on a data table in a database. Data owners can run this tool on all data tables in a database before outsourcing the database to a DSP. Once the authentication data is created for the database, they can upload the database to the DSP. Besides, we have implemented all four different ways - *MultiJoin*, *SingleJoin*, *ZeroJoin* and *RangeCondition* to retrieve authentication data for performance overhead evaluation. Our implementation is based on .NET and SQL Server 2008. In addition, we implemented two XML-based schemes: OPEN-XML and DT-XML, which utilize built-in XML functionality of SQL Server, for efficiency analysis and comparison. In both OPEN-XML and DT-XML schemes, we use a hierarchical XML structure to represent the authentication data of a Merkle B-tree and store the XML string into a database. The OPEN-XML scheme uses OPENXML function provided in SQL Server to retrieve VO data from the XML string, and the DT-XML uses XPath and nodes() methods to retrieve VO data from an indexed XML data field, where the XML string is stored.

Experiment Setup. We use a synthetic database that consists of one table with $100,000$ records. Each record contains multiple columns, a primary key *id*, and is about 1KB long. For simplicity, we assume that an authenticated index is built on *id* column. We upload the database with authentication data to a third-party cloud service provider, which deploys the SQL Server 2008 R2 as a database service, and run experiments from a client through a home network with 30Mbps download and 4Mbps upload. To evaluate the performance overhead of integrity verification and the efficiency of the proposed mechanisms, we design a set of experiments using the synthetic database. We use MD5 [9] to generate hashes.

Baseline

Height of MBT. Figure 4.10 shows the height of a MBT for different fanouts, which is built on $100,000$ data records. The height is computed based on the fact that we insert the corresponding authentication data records of those data records into a MBT with a certain fanout one by one. It is clear that as the fanout of a MBT increases, the height of the MBT decreases, which means that the number of authentication tables that need to be created decreases since we create an authentication table for each level in a MBT. We can also see that sometimes the height may not change when the fanout changes, for example from 32 to 64.

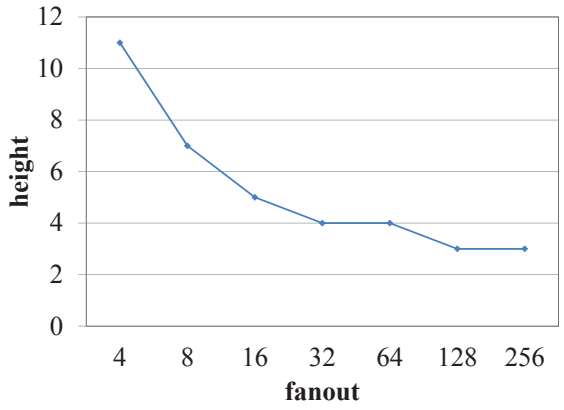


Figure 4.10: Height vs Fanout.

VO Size. Figure 4.11 shows how the VO size changes as the fanout of a MBT changes for *Unique Select* and *Range Select*. The results clearly show that as the fanout increases, the VO size increases, and the VO size of *Range Select* is almost twice of that of *Unique Select* since the VO of *Range Select* includes the VO of two boundaries of the range. Note that for *Range Select*, its VO size almost stays same no matter how many records are returned in a *Range Select* since its VO only include the VO of

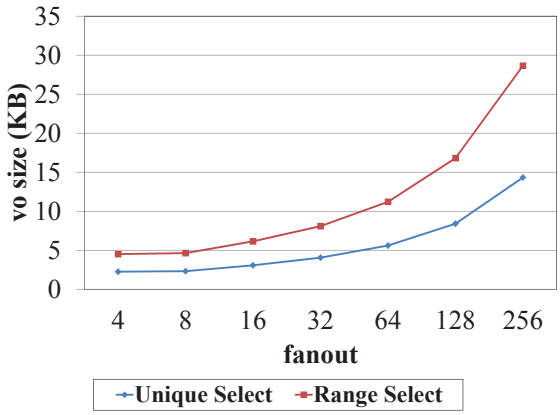


Figure 4.11: VO Size vs Fanout.

two boundaries of a range.

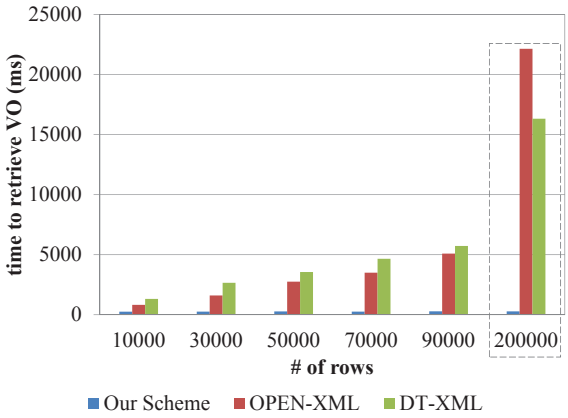


Figure 4.12: VO Retrieval Time.

VO Retrieval. Figure 4.12 shows the time to retrieve a VO for our scheme using RangeCondition and two XML-based schemes when the number of rows in the data set changes. As can be seen from the figure, when the data size is small, three schemes show a similar time to retrieve the VO. However, as the data size increases, two XML-based schemes show linear increases in terms of the VO retrieval time. When the data size goes up to $200,000$ records, the XML-based schemes take more than 15 seconds to retrieve a VO for one single record. In this case, our scheme is about 100 times faster than two XML-based schemes. The result indicates that a well-design scheme could be much more efficient than a

scheme using built-in XML functionality in DBMSs.

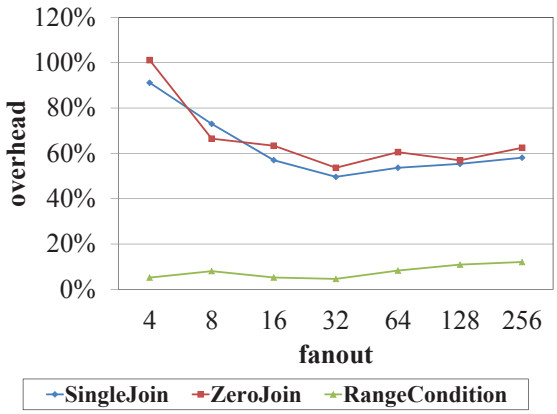


Figure 4.13: Unique Select Overhead.

Data Operations

Unique Select. We conduct experiments to see how different fanouts of a MBT and different methods of retrieving VO could affect the performance of *Unique Select* queries, where we vary the fanout of a MBT and compare the performance overhead caused by different VO retrieval methods, shown in Figure 4.13 The results show that the overhead of SingleJoin and ZeroJoin is much higher than that of RangeCondition. When the fanout is 32, the overhead of SingJoin or ZeroJoin is about 50%, but the overhead of RangeCondition is 4.6%. The communication cost for the three different methods is almost same, and the major performance difference is caused by the computation cost in the server side. As we can see from this figure, when the fanout increases from 4 to 32, the overhead of both SingleJoin and ZeroJoin drops, and when the fanout is larger than 32, their overhead increases. It is because in general the VO size increases and the number of queries to be executed to retrieve authentication data decreases as the fanout increases, and when the fanout is less than 32 the computation cost dominates the overhead and when the fanout is larger than 32 the communication cost dominates the overhead. Note that we do not show the overhead caused by MultiJoin since its performance is tens of times worse than that of other methods, which is caused by the large amount of VO data generated. Based on current experiment environment, the 32 fantout shows a better performance compared with other fanouts. In the following experiments we use 32 as the default fanout if we do not specify the fanout.

Range Select. We also run experiments to explore how the overhead changes when the number of records retrieved increases. Figure 4.14 shows the response time of retrieving different number of records in range queries, where NoVeri denotes range queries without integrity verification support,

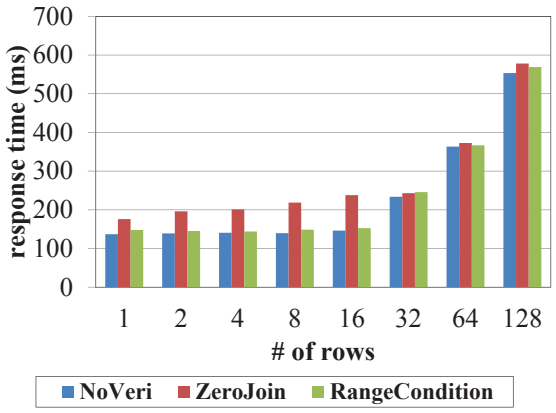


Figure 4.14: Range Select Response Time.

ZeroJoin and RangeCondition denote rang queries with integrity verification but using VO retrieval method ZeroJoin and RangeCondition respectively. The results show two points: 1) the RangeCondition is much better than ZeroJoin when the number of rows to be retrieved is small, which is because the computation cost dominates the overhead caused by different VO retrieval methods; 2) once the number of records to be retrieved is larger than a certain number, the response time of all three is almost same. In our algorithm, the overhead caused by different VO retrieval methods does not change as the number of retrieved records increases. Thus, as the number of retrieved records increases, the overhead percentage becomes smaller and smaller. Besides, we also conduct experiments to show how the overhead changes as the database size increases, where we run range queries to retrieve 512 rows from databases with different number of data records. As shown in Figure 4.15, the overhead is about 3% even if the number of data records goes up to 1.6 million.

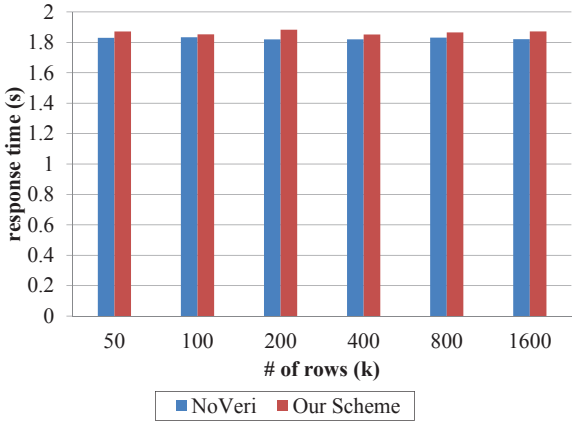


Figure 4.15: Scalability of Range Select.

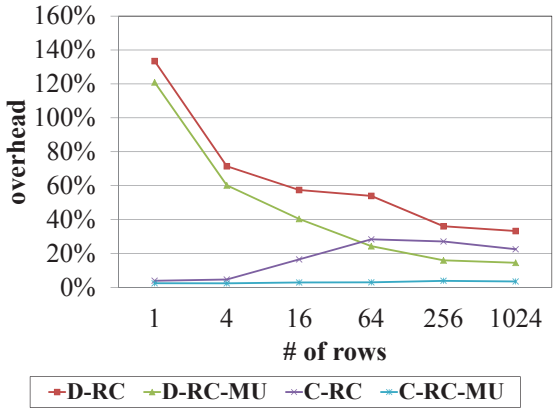


Figure 4.16: Direct and Cached Update Overhead Comparison.

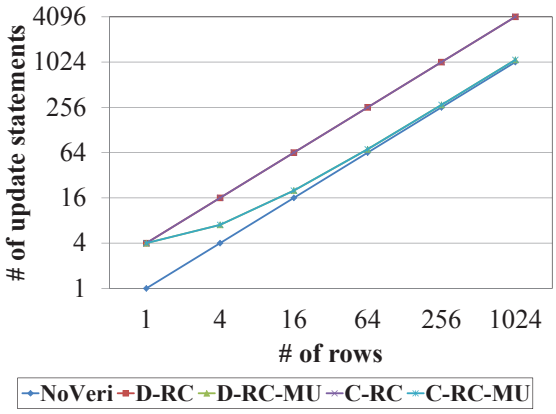


Figure 4.17: Number of Generated Update Statements.

Update. We evaluate the performance overhead caused by two different update cases - Direct Update and Cached Update, discussed in section 4.4.2. For Direct Update, we first retrieve the data to be updated and verify its data integrity, and then we generate update queries for both data and authentication data and send them to the sever for execution. For Cached Update, we assume that the data to be updated is already cached in the memory, we just need to generate update queries and send them to the server for execution. Figure 4.16 shows the overhead versus the number of rows to be updated. In the figure, 'D' denotes Direct Update, C denotes Cached Update, "RC" denotes RangeCondition, and "MU" denotes MergeUpdate, which indicates if a MergeUpdate process is used to reduce the number of SQL statements generated for updating authentication data records. The results show that when we

directly update only a few records with integrity protection, the overhead could go above 100%, but if we update cached records, the overhead is about 2.5%. In this case, the additional round-trip time in Direct Update dominates the response time of the whole update process. As the number of updated rows increases, the overhead percentage of Direct Update decreases because the response time is dominated by the update time in the server side. The major overhead for Cached Update comes from the execution of update statements to update authentication data in the server side. The results also show that the performance of C-RC-MU is comparable to the performance of NoVeri without integrity protection, but without optimization, the overhead of C-RC ranges from 3% to 30% shown in the figure.

Figure 4.17 shows the number of update statements to be executed for different cases by varying the number records to be updated. If no verification is required, then the number of update statements is equal to the number of records to be updated. Since the height of the MBT with fanout 32 is 4, to update each row we need to generate 4 update statements as discussed in Section 4.4.2, the number of update statements generated in RangeCondition is 4 times of the number of rows to be updated in this case. After applying the MergeUpdate process to the set of generated update statements, this number could become much closer to the number of records to be updated shown in the figure.

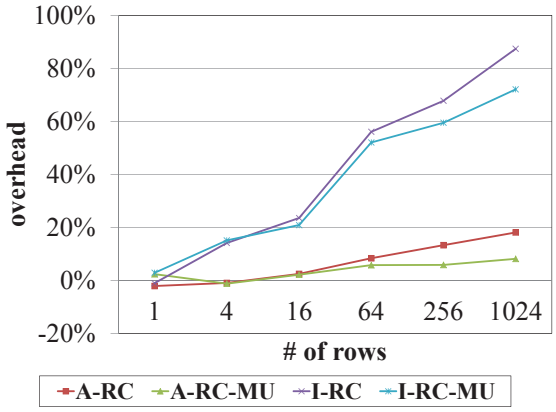


Figure 4.18: Append and Insert Overhead Comparison.

Append and Insert. We run two different experiments to evaluate the performance of insert with integrity protection. In both experiments, we create an empty table and insert different number of records into the table. In the first experiment called Append, we append each record at the end of the table while in the second experiment called Insert we insert each record at the beginning of the table. To assure integrity, they need to generate different set of update statements to update authentication data.

Figure 4.18 compares the overhead for both Append and Insert, where ‘A’ denotes Append that always inserts data at the end of the table and ‘I’ denotes Insert that always inserts data the beginning

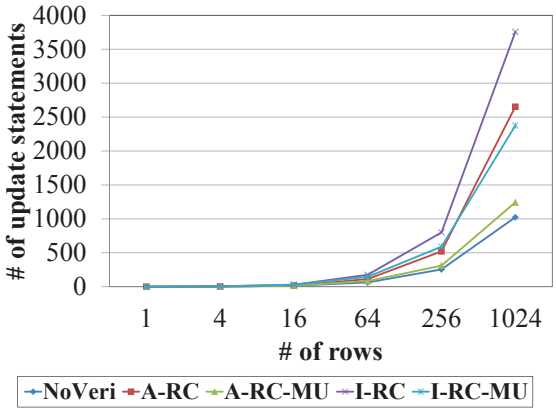


Figure 4.19: Number of Generated Update Statements.

of the table. First, the results show that the overhead caused by Append is much lower than that caused by Insert. Second, the optimization of MergeUpdate improves the performance a lot for both Append and Insert when a large number of records are appended or inserted. Third, as the number of records increases, the overhead for all cases increases. Based on current results, the A-RC-MU overhead is less than 10% and the I-RC-MU overhead is more than 70%. The major performance overhead comes from the number of update statements to be executed in the server side.

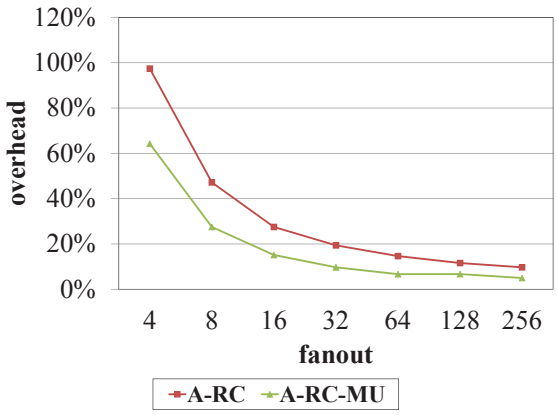


Figure 4.20: Append Overhead vs Fanout.

To understand more about the overhead caused by different cases, we quantify the number of update statements generated for each case, shown in Figure 4.19. As we can see from the figure, the number of update statements generated for Insert is much larger than that for Append, and the MergeUpdate

process does remove unnecessary update statements to a great extent, which improve the performance a lot.

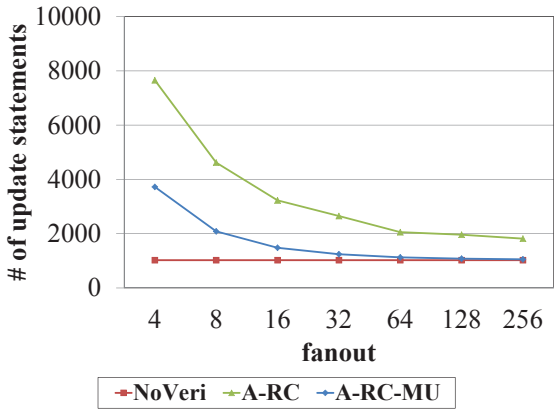


Figure 4.21: Number of Generated Update Statements vs Fanout.

Append with Different Fanouts. We also conduct experiments to understand how fanout could affect the performance of adding new records, where we append 1K rows into an empty table and generate authentication data using different fanouts ranging from 4 to 256. And we compare the performance overhead for A-RC and A-RC-MU in this experiment. Figure 4.20 shows how the overhead changes as the fanout changes. The results show that the overhead for both A-RC and A-RC-MU decreases consistently as the fanout increases. It is because the number of update statements decreases as the fanout increases, which is shown in Figure 4.21. In this sense, to improve the performance for Append or Insert, one simple way is to increase the fanout of the MBT. Actually, it is also true for Cached Update. Although we can improve the write performance by increasing the fanout of a MBT, it may downgrade the read performance as shown in Figure 4.13.

Summary

Overall, our scheme imposes a small overhead for *Select*, *Update* and *Append* and a reasonable overhead for *Insert* and *Delete*, which also depends on the fanout chosen for a MBT. Based on our experimental environment, the 32 fanout shows a good performance overall. However, the selection of fanout relies on several factors such as the access pattern of data in the table since the read and write may require different fanouts for their best performance, the number of records in a table, the size of a typical record, network bandwidth, client and server computation capability, etc. As we can see from our experimental results, the major overhead caused by our scheme comes from the computation cost of executing the

additional number of SQL statements in the server side, which either tries to retrieve authentication data or update authentication data. The communication cost can be amortized when a large number of records are retrieved, updated, inserted or deleted. We do not present the performance overhead for Delete since the Delete process is similar to the Insert process.

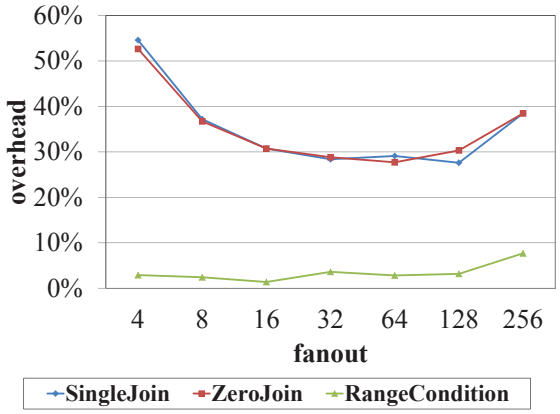


Figure 4.22: Unique Select Overhead in Campus Network.

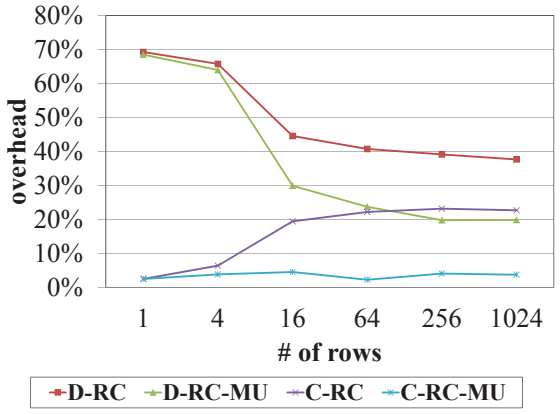


Figure 4.23: Direct and Cached Update Overhead Comparison in Campus Network.

Note that we also run some experiments in a campus network to see how the performance of our scheme changes in a different network environment. As can be seen from Figure 4.22 and 4.23, our scheme achieves a similar performance, and also the performance overhead of our scheme in the campus

network show a similar trend compared with that in the home network.

Chapter 5

Related Work

Previous work that is related to the research in the report is divided into two parts based on the two work we have done. We summarize the related work for each work in the following respectively.

5.1 MapReduce and Service Integrity

MapReduce recently has received a great amount of attention for its simple model and parallel computation capability for data intensive computation in different application and research areas. Chu et al. [21] applied MapReduce to the multicore computation for machine learning. Ekanayake et al. [28] applied MapReduce technique for two scientific analyses, High Energy Physics data analyses and Kmeans clustering. Mackey et al. [41] utilized MapReduce for High End Computing applications. Most of them focus on how to utilize MapReduce to solve issues or problems in specific application domains. Few work pays attention to the service integrity protection in MapReduce. SecureMR provides a set of practical security mechanisms to ensure MapReduce data processing service integrity.

Service integrity issues addressed in this work also share similarity with the problem addressed in [26,27,31–33,54,68]. Du et al. [27] used sampling techniques to achieve efficient and viable uncheatable grid computing. Zhao et al. [68] proposed a scheme called Quiz to combat collusion for result verification. Sarmenta et al. [54] introduced majority voting, and spot-checking techniques, and presented credibility-based fault tolerance. Although several existing techniques have been proposed to address the service integrity issues in different application areas [10,27,59], the integrity assurance for MapReduce data processing service presents its unique challenges like massive data processing and multi-party distributed computation. SecureMR adopts a new decentralized replication-based integrity verification scheme to address these new challenges, which fully utilizes the existing architecture of MapReduce.

Regarding system security, Srivatsa and Liu proposed a suite of security guards and a resilient network design to secure content-based publish-subscribe systems [56]. PeerReview [38] system ensures that Byzantine faults observed by a correct node are eventually detected and irrefutably linked to a faulty

node in a distributed messaging system. Swamynathan et. al. proposed a scheme to improve the accuracy of reputation systems using a statistical metric to measure the reliability of a peer's reputation [58]. Different from previous works, SecureMR is based on a trustworthy master and leverages natural redundancy of map and reduce services and existing MapReduce data processing mechanisms to perform comprehensive consistency verification.

5.2 BigTable and Data Integrity

As a large scale distributed data storage, BigTable recently has received a great amount of attention from both industries and academia for its efficiency and reliability. Several open-source, distributed data storages have been implemented modelled after BigTable, for example HBase [7], Cassandra [3], and Hypertable [8], which are widely used for both academia research and commercial companies. Carstoiu et al. [18] focused on the performance evaluation of HBase. You et al. [66] 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. [57] 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 [23, 34, 36, 48, 52, 63]. Different authenticated data structures have been proposed to address the integrity issues, for example skip list [52], signature aggregation [46], and Merkle B+ tree [23, 40] , etc. 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.

Besides, Xie et al. [63] 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. [65] discussed different approaches to address some join queries for outsourced database, which is not a case in BigTable. Xie et al. [64] analyzed the different approaches to provide freshness guarantee over different integrity protection schemes, which is complimentary to our work for BigTable.

Additionally, Zhou et al. [62] discussed the data integrity verification in the cloud, and proposed an appoach 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 is not desirable and 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.

5.3 Outsourced Database Integrity

Extensive research efforts have focused on security issues of database outsourcing. Hakan et. al. [37] focuses on addressing privacy issues for database outsourcing. It explores the efficiency of different schemes using both hardware and software encryption. A later work from Hakan et. al. [36] explores techniques to execute SQL queries over encrypted data so that the data privacy could be protected even from the DSPs. Both work does not consider the problem of data integrity. Different from those work, we focus on ensuring data integrity for outsourced databases.

Researchers have investigated on data integrity issues for years in the area of database outsourcing [24, 35, 40, 45, 48, 49, 55, 63, 64]. Mykletun et. al. [45] analyzed several signature methods for data authentication. Although they brought forth the completeness problem, they did not propose a design to solve the problem. Some work [24, 48] studied the problem of verifying the completeness aspect of data integrity. Pang et. al. [48] proposed a signature aggregation based scheme that enables a user to verify the completeness of a query result by assuming an order of the records according to one attribute. Devanbu et. al. [24] uses Merkle hash tree based methods to verify the completeness of query results. But they do not consider the freshness aspect of data integrity.

Moreover, Sion [55] proposed a scheme for query execution assurance over outsourced databases by providing query execution proofs that show queries were actually executed. It can handle arbitrary types of queries with a reasonable overhead, but it only focuses on read-only queries. Xie et al. [63] proposed a probabilistic approach by inserting a small amount of fake records into outsourced databases so that integrity can be effectively audited by analyzing the inserted records in the query results. Unfortunately this approach only protects integrity probabilistically.

Li et. al. [40] first brought forward the freshness issue as an aspect of data integrity. It verifies if data updates are correctly executed by DSPs so that queries will be executed over the up-to-date data instead of old data. Based on Merkle hash tree, Li et. al. proposed a scheme by generating a time-stamped signature for the root node of the tree, which is inspired by the work on certificate validation and revocation [42]. Xie et al. [64] analyzed different approaches to ensuring query freshness. The aggregated signature based approaches [46, 48] require to modify signatures of all the records, which renders it impractical considering the number of signatures.

Besides, Miklau et. al. [43] designed a scheme based on interval tree to achieve guarantees of integrity when interacting with a vulnerable or untrusted database server. However, several disadvantages are mentioned in the work [25], which dealt with a similar issue based on authenticated skip list [34]. That work does not well explain how authentication data is retrieved completely. It claims that only one query is required for integrity verification while it also mentions that multiple queries are necessary to retrieve all authentication data, which is confusing. In addition, how the completeness is handled based on their scheme is not clear, and experiments seem limited. Following this one, Palazzi et. al. [47] proposed approaches to support range queries based on multiple attributes, which is complementary to our

work.

Compared with previous work, our scheme is able to provide integrity assurance for database outsourcing, including all three aspects: correctness, completeness and freshness. More importantly, one significant advantage of our scheme over existing approaches is that existing approaches need to modify the implementation of DBMSs in order to maintain an appropriate authenticated data structure and generate VOs. Such requirement often renders these approaches impractical to be deployed in real-world applications [63]. Our work provides a strong query integrity guarantee (instead of probabilistic guarantee [63]) without requiring DBMSs to perform any special function beyond query processing.

Chapter 6

Future Work

Big data processing has become increasingly important. As a complete big data processing system, it relies on not only scalable storage system to store large data sets, but also powerful data processing system to analyze large data sets. Deploying it over open cloud is a cost effective and practical solution for small companies and researchers who lack capabilities to obtain their own powerful clusters. In open cloud, storage and computation resources may be contributed by participants from different domains. Large scale clusters can be formed dynamically to process large amount of data in open cloud. Our goal is to deploy a complete big data processing system with practical integrity assurance over open cloud. In the following, we elaborate new issues for the whole system, and also discuss the approaches and challenges of addressing the issues.

6.1 Introduction

Big data processing has become more and more important for business companies and academic researchers. It not only relies on scalable storage system to store large data sets, but also requires powerful data processing system to analyze large data sets. Such large scale systems are usually deployed in large clusters with hundreds or thousands of machines, which is impractical for small companies and researchers since it is prohibitively expensive to deploy and maintain such large scale systems on their own. Cloud computing emerges as a promising and cost-effective computing platform. We envision that an open cloud could be dynamically formed by numerous of participants from all over the world. They contribute and share their storage and computation resources with others in the cloud. Deploying such systems over open cloud is a cost-effective and practical solution for small companies and researchers who are not able to build their own large clusters.

In open cloud, as participants may come from different domains, they may not always trust each others. Moreover, malicious participants who may want to save storage and computation resources discard data or return result without actually doing their job. Thus, before we can safely deploy such systems

over open cloud and make the best use of them, several security issues need to be addressed, including confidentiality, integrity and availability, without sacrificing scalability provided by existing systems. We focus ourselves on the discussion of providing practical integrity assurance for a complete big data processing system including both scalable storage system and data processing system.

On one hand, as a practical service integrity assurance framework for MapReduce, SecureMR not only ensures MapReduce service integrity as well as to prevent replay and Denial of Service (DoS) attacks, but also preserves the simplicity, applicability and scalability of MapReduce. To achieve scalability, it designs a decentralized replication-based integrity verification scheme for ensuring the integrity of MapReduce in open systems. The scheme decentralizes the integrity verification process among different distributed computing nodes who participate in the MapReduce computation. However, SecureMR only provides the integrity assurance of data processing.

On the other hand, iBigTalbe, 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 query execution. It also designs efficient schemes to merge and split authenticated structures among multiple nodes. iBigTable tries to utilize the unique properties of BigTable to reduce the cost of integrity verification and preserve its interfaces to applications as much as possible. However, iBigTalbe only focuses on the data integrity.

In future, we plan to design new protocols and simplify technique development to provide practical integrity assurance for a complete big data processing system over open cloud. Although SecureMR and iBigtable provide integrity assurance for data processing and data storage respectively, it is not straightforward to build a complete data processing system while providing integrity assurance based on SecureMR and iBigTable. Several issues and challenges need to be addressed, which are described in the following. And also approaches proposed in SecureMR and iBigTable require modifying existing systems, which may not be desirable since it introduces extra deployment cost and also hampers the adoption of our approaches.

6.2 Problem Summary

In this section, we illustrate the issues of providing practical integrity assurance for a complete data processing system based on SecureMR and iBigTable.

Firstly, iBigTable provides practical data integrity for BigTable in public cloud. It assumes that the cloud provider deploys iBigTable and exposes it as a service for the public. The public can verify the correctness, completeness and freshness of data retrieved from the cloud provider through iBigTable service. Any integrity violation must be detected by clients. Once the violation is detected, clients is convinced that there is something wrong with the cloud provider. The violation may be caused by the malicious version of iBigTable installed in the cloud, by an attacker, due to data corruption because of

machine crash, or tampered by the cloud provider intentionally. iBigTable does not try to identify why it happens in the cloud. In general, we believe that something is wrong with the cloud provider since all nodes of iBigTable in the public cloud are from the same domain.

However, in open cloud we target, parties may come from different domains and form a cluster dynamically to provide a BigTable service for clients. In such an environment, it is very important to identify who behaves maliciously if any integrity violation is detected so that the malicious party will be punished in a way, for example being fined or being kicked out. It also helps good contributors build their reputation in open cloud, and bad contributors are punished. Thus, one property we aim to achieve is *non-repudiation*. With this property, any malicious behavior of any participant can be identified and proved by evidence from other participants.

Furthermore, iBigTable assumes that the master (or data owner) is the only party who makes updates to the data stored in BigTable because the master is the only party who can generate signature for new updates, which guarantees the data integrity. But it is not scalable to require the master to write all data to iBigTable. And also it may avoid the benefits of the parallel processing model. It is necessary to delegate the write permission to multiple parties. However, malicious parties may write garbage data to iBigTable on behalf of the master. The issue is how we can authorize other parties to make updates to the data stored in iBigTable while providing data integrity assurance.

Additionally, iBigTable requires to modify exiting implementation of BigTable to enable the generation of authentication data for integrity verification. It has at least two major disadvantages. First, existing service providers may not be willing to update their existing services to provide integrity protection since it requires a lot of work to make it work. Second, even if they adopt iBigTable solution, it could be a problem to keep the BigTable service up to date. Either, each service provider will extend the integrity protection for each version of BigTable or the creators of iBigTable need to upgrade iBigTable based on BigTable implementation. Both of them will hamper the adoption of iBigTable.

Secondly, SecureMR provides a practical service integrity assurance framework for MapReduce, but it focuses on providing integrity assurance for data computation process. It assumes that DFS provides integrity assurance for the input of MapReduce, which can be provided by iBigTable easily. However, when reducers write data to DFS, it is unclear how to make sure that reducers correctly write data to DFS without maliciously altering data. As mentioned in SecureMR, an additional phase *Verify* is introduced into the MapReduce data processing. The verifiers can help the master complete the verification protocol in the *Verify* phase to have a certain guarantee that reducers do their tasks correctly. Even if verifiers are supposed to write data to DFS instead of reducers, it exposes the same vulnerability as we ask reducers to write data to DFS since verifiers are not trusted and can behave maliciously.

Moreover, the additional *Verify* phase introduce additional data processing delay, which downgrades the performance of MapReduce. It is good to remove the *Verify* phase and boost the performance of MapReduce. However, it is unclear how to complete the verification protocol against reducers without introducing verifiers in SecureMR. Another alternative described in SecureMR is to ask MapReduce

user applications to complete the verification protocol after their jobs are done. But this alternative does not fit for applications requiring a real-time data integrity verification since it is time-consuming to complete the verification protocol against all reducers by a single user application itself. It is also unknown how to provide a scalable real-time data integrity protection without adding additional *Verify* phase.

6.3 Approaches and Challenges

In this section, we discuss possible ideas, approaches and challenges for the above issues, respectively.

First, like non-repudiation guaranteed in SecureMR it is necessary to guarantee the non-repudiation property for the whole system so that we can identify who does something bad when inconsistency is detected. It is especially important to resolve disputes in open systems since parties come from different domains and are not always trusted each other. A common way to achieve non-repudiation is to deploy Public Key Infrastructure (PKI), ask each party to sign their messages using private key and encrypt them using other's public key before they are sent, request acknowledgement from receiver, and record all communications in their local side so that any dispute can be resolved by replaying communication logs, which also provide evidence for any malicious behavior. We face a few challenges and issues for this solution: 1) PKI may not be available in open cloud, such a dynamic environment; 2) it may be not efficient to use public key to encrypt messages for their communications; 3) it is not practical to keep logs all the time to achieve non-repudiation.

One alternative to PKI is to ask the master to generate public/private key pairs for each party got involved in the system. To speed up the communication, the master can generate secret keys for the encryption and decryption of messages during the communications, but they still need to use private key to sign their message for non-repudiation. To discard logs at certain time, we may adopt techniques similar to checkpoint in the database or session management. In this way, logs before a checkpoint or after session ends can be discarded without influencing the integrity violation detection and malicious behavior identification. To resolve those issues, specific protocols and designs need to be further investigated, for example how we setup keys for other parties, when we create a checkpoint for the system, and how a session starts and ends.

Second, it is necessary to delegate the write permission to other parties so that the master will not be overloaded by a lot of updates to BigTable and become a bottleneck of the system. To resolve this issue, we may design a Kerberos-like authentication protocol to authorize other parties to update data on behalf of the master by issuing a ticket, which implies the write permission authorization. In Kerberos authentication protocol, the Key Distribution Center (KDC) does not need to communicate with service providers, but different from Kerberos the master may need to talk to both reducers and tablet servers, and to achieve non-repudiation the public/private keys and session keys are used for communication instead of only relying on session key. Although the general idea or protocol may be

similar to Kerberos, the extension and revision of the protocol seem inevitable to address this issue efficiently. Once multiple writers are allowed, the synchronization of multiple writes to the same tablet need to be carefully examined so that the authenticated data structure can be correctly updated.

Third, our third work presents a practical solution to provide integrity protection without modifying DBMSs. Based on ideas in this work, it is possible to design a similar scheme to resolve this issue for BigTable-like distributed storage systems, which only support a limited query interface. With such a distributed storage system, it may be easy to distribute authentication data across nodes in the system, which is not like a centralized DBMS. However, due to the limited query interface, for example HBase only supports forward scan, but not backward scan, it could be hard to design an efficient way to find two boundaries for a range query. One possible approach is to build a chain based on data keys. Although it may help find the boundaries for a range, it also introduces redundancies. Since nodes are distributed and updates may go to different nodes, data consistence will be another big concern and it is a challenge to maintain data consistence in this case. And also it may be impossible to fulfill an operation with integrity protection within one transaction or round-trip, which usually can be done in DBMS.

Fourth, instead of requiring a set of verifiers, user applications or the master, the verification protocol needs to be revised in the last *Reduce* phase so that it can be fulfilled by existing parties involved in the data processing without overloading a single party in the system. Since the data will be written into tablet servers in BigTable, tablet servers can be a perfect candidate to help the master complete the verification protocol against reducers so that the malicious behavior of reducers can be detected, for example tampering the final result. But it is vulnerable to attacks from tablet servers, for example even if tablet servers receive correct data from reducers, they can store whatever data they want because there is no way for the master to check. This is because the verification protocol only provides a way for tablet servers in BigTable to verify the integrity of data written by reducers, but does not provide a way for the master to check if data is correctly written into BigTable. One naive way to help the master check that is to ask reducers who performance duplicated tasks to read data from BigTable and generate the hash again to make sure the data is correctly stored in BigTable. But it is vulnerable to the collusion attack conducted by reducers and tablet servers. To counter such attacks, one promising approach is to further change the commitment protocol to ask tablet servers to make commitments to the master beside reducers.

In short, to address the issues and challenges, we plan to extend and revise existing protocols and mechanisms in SecureMR, iBigTable and IAODB so that it can be safely and easily deployed over open cloud. Specific consideration need to be taken for other security and performance concerns.

Chapter 7

Conclusion

This thesis includes three works building security mechanisms and designing novel schemes to provide integrity assurance for big data processing deployed over open cloud.

In the first work, we have presented SecureMR, a practical service integrity assurance framework for MapReduce. We have implemented a scalable decentralized replication-based verification scheme to protect the integrity of MapReduce data processing service. To the best of our knowledge, our work makes the first attempt to address this problem. Based on Hadoop [6], we have implemented a prototype of SecureMR, proved its security properties, evaluated the performance impact resulted from the proposed scheme, and tested it on a real distributed computing system with hundreds of hosts connected through campus networks. Our initial experimental results show that the proposed scheme can ensure data processing service integrity while imposing low performance overhead.

In the second work, 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 [7], 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.

In the third work, we have presented an efficient and practical scheme based on Merkle B-tree, which provides integrity assurance without modifying the implementation of existing DBMSs. We have proposed a novel approach called Radix-Path Identifier, which makes it possible to serializes a Merkle B-tree into a database while enabling highly efficient authentication data retrieval for integrity verification. And also we have explored the efficiency of different methods such as MultiJoin, SingleJoin, ZeroJoin and RangeCondition, to retrieve authentication data from a serialized MBT stored in a database. We have implemented a proof-of-concept prototype and conducted extensive experimental evaluation. Our initial experimental results show that our scheme imposes a small overhead for *Select*, *Update* and *Append* and

a reasonable overhead for *Insert* and *Delete*.

REFERENCES

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

[2] Amazon Elastic MapReduce. http://docs.amazonwebservices.com/Elastic-MapReduce/latest/DeveloperGuide/index.html.

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

[4] Digital Signature Algorithm. *http://en.wikipedia.org/wiki/Digital_Signature_Algorithm*.

[5] Genealogical Representation of Trees in Databases. *http://www.tetilab.com/roberto/pgsql/postgres-trees.pdf*.

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

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

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

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

[10] SETI@home. *http://setiathome.ssl.berkeley.edu/*.

[11] SHA-1. *http://en.wikipedia.org/wiki/SHA-1*.

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

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

[14] WordCount, Hadoop. http://wiki.apache.org/hadoop/WordCount.

[15] G. Alonso amd F. Casati, H. Kuno, and V. Machiraju. Web Services Concepts, Architectures and Applications Series: Data-Centric Systems and Applications. *Addison-Wesley Professional*, 2002.

[16] D. P. Anderson. Boinc: a system for public-resource computing and storage. pages 4–10, 2004.

[17] Mikhail J. Atallah, YounSun Cho, and Ashish Kundu. Efficient data authentication in an environment of untrusted third-party distributors. In *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 696–704, Washington, DC, USA, 2008. IEEE Computer Society.

[18] 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.

[19] Joe Celko. *Joe Celkoś Trees and Hierarchies in SQL for Smarties*. Morgan Kaufmann, 2004.

[20] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26:4:1–4:26, June 2008.

[21] Cheng T. Chu, Sang K. Kim, Yi A. Lin, Yuanyuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In Bernhard Schölkopf, John C. Platt, and Thomas Hoffman, editors, *NIPS*, pages 281–288. MIT Press, 2006.

[22] Jeffrey Dean and Sanjay 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.

[23] Premkumar Devanbu, Michael Gertz, Charles Martel, and Stuart G. Stubblebine. Authentic data publication over the internet. *J. Comput. Secur.*, 11:291–314, April 2003.

[24] Premkumar T. Devanbu, Michael Gertz, Charles U. Martel, and Stuart G. Stubblebine. Authentic third-party data publication. In *Proceedings of the IFIP TC11/ WG11.3 Fourteenth Annual Working Conference on Database Security: Data and Application Security, Development and Directions*, pages 101–112, Deventer, The Netherlands, The Netherlands, 2001. Kluwer, B.V.

[25] Giuseppe Di Battista and Bernardo Palazzi. Authenticated relational tables and authenticated skip lists. In *Proceedings of the 21st annual IFIP WG 11.3 working conference on Data and applications security*, pages 31–46, Berlin, Heidelberg, 2007. Springer-Verlag.

[26] Patricio Domingues, Bruno Sousa, and Luis Moura Silva. Sabotage-tolerance and trust management in desktop grid computing. *Future Gener. Comput. Syst.*, 23(7):904–912, 2007.

[27] Wenliang Du, Jing Jia, Manish Mangal, and Mummoorthy Murugesan. Uncheatable grid computing. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 4–11, Washington, DC, USA, 2004. IEEE Computer Society.

[28] Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox. Mapreduce for data intensive scientific analysis. In *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pages 277–284, 2008.

[29] T. Erl. Service-Oriented Architecture (SOA): Concepts, Technology, and Design. *Prentice Hall*, 2005.

[30] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. *ACM Trans. Comput. Syst.*, 20(1):1–24, 2002.

[31] Cécile Germain-Renaud and Dephine Monnier-Ragaigne. Grid result checking. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 87–96, New York, NY, USA, 2005. ACM.

[32] Philippe Golle and Ilya Mironov. Uncheatable distributed computations. In *CT-RSA 2001: Proceedings of the 2001 Conference on Topics in Cryptology*, pages 425–440, London, UK, 2001. Springer-Verlag.

[33] Philippe Golle and Stuart Stubblebine. Secure distributed computing in a commercial environment. In *5th International Conference Financial Cryptography (FC*, pages 289–304. Springer-Verlag, 2001.

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

[35] Michael T. Goodrich, Roberto Tamassia, and Nikos Triandopoulos. Super-efficient verification of dynamic outsourced databases. In *Proceedings of the 2008 The Cryptopgraphers' Track at the RSA conference on Topics in cryptology*, CT-RSA'08, pages 407–424, Berlin, Heidelberg, 2008. Springer-Verlag.

[36] Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad 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.

[37] Hakan Hacigumus, Bala Iyer, and Sharad Mehrotra. Providing database as a service. In *Proceedings of the 18th International Conference on Data Engineering*, ICDE '02, pages 29–, Washington, DC, USA, 2002. IEEE Computer Society.

[38] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: practical accountability for distributed systems. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 175–188, New York, NY, USA, 2007. ACM.

[39] Michal Laclavík, Martin Šeleng, and Ladislav Hluchý. Towards large scale semantic annotation built on mapreduce architecture. In *ICCS '08: Proceedings of the 8th international conference on Computational Science, Part III*, pages 331–338, Berlin, Heidelberg, 2008. Springer-Verlag.

[40] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid 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.

[41] Grant Mackey, Saba Sehrish, Julio Lopez, John Bent, Salman Habib, and Jun Wang. Introducing mapreduce to high end computing. In *Petascale Data Storage Workshop Held in conjunction with SC08*, 2008.

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

[43] Gerome Miklau and Dan Suciu. Implementing a tamper-evident database system. In *Proceedings of the 10th Asian Computing Science conference on Advances in computer science: data management on the web*, ASIAN'05, pages 28–48, Berlin, Heidelberg, 2005. Springer-Verlag.

[44] Kyriakos Mouratidis, Dimitris Sacharidis, and Hweehwa Pang. Partially materialized digest scheme: an efficient verification method for outsourced databases. *The VLDB Journal*, 18:363–381, January 2009.

[45] Einar Mykletun, Maithili Narasimha, and Gene Tsudik. Authentication and integrity in outsourced databases. *Trans. Storage*, 2:107–138, May 2006.

[46] Maithili Narasimha and Gene Tsudik. Authentication of outsourced databases using signature aggregation and chaining. In *In International Conference on Database Systems for Advanced Applications (DASFAA*, pages 420–436. DASFAA, 2006.

[47] Bernardo Palazzi, Maurizio Pizzonia, and Stefano Pucacco. Query racing: fast completeness certification of query results. In *Proceedings of the 24th annual IFIP WG 11.3 working conference on Data and applications security and privacy*, DBSec'10, pages 177–192, Berlin, Heidelberg, 2010. Springer-Verlag.

[48] HweeHwa Pang, Arpit Jain, Krithi Ramamritham, and Kian-Lee 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.

[49] HweeHwa Pang and Kian-Lee Tan. Authenticating query results in edge computing. In *Proceedings of the 20th International Conference on Data Engineering*, ICDE '04, pages 560–, Washington, DC, USA, 2004. IEEE Computer Society.

[50] HweeHwa Pang, Jilian Zhang, and Kyriakos Mouratidis. Scalable verification for outsourced dynamic databases. *Proc. VLDB Endow.*, 2:802–813, August 2009.

[51] Lawrence Pizzette and Toby Cabot. Database as a service: A marketplace assessment. 2012.

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

[53] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21:120–126, February 1978.

[54] Luis F. G. Sarmenta. Sabotage-tolerance mechanisms for volunteer computing systems. *Future Generation Computer Systems*, 18(4):561–572, 2002.

[55] Radu 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.

[56] Mudhakar Srivatsa and Ling Liu. Securing publish-subscribe overlay services with eventguard. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 289–298, New York, NY, USA, 2005. ACM.

[57] Kyungho Jeon Steven Y. Ko and Ramses Morales. The hybrex model for confdentiality and privacy in cloud computing. 2011.

[58] G. Swamynathan, B.Y. Zhao, K.C. Almeroth, and S.R. Jammalamadaka. Towards reliable reputations for dynamic networked systems. In *Reliable Distributed Systems, 2008. SRDS '08. IEEE Symposium on*, pages 195–204, Oct. 2008.

[59] D. Szajda, B. Lawson, and J. Owen. Toward an optimal redundancy strategy for distributed computations. In *Cluster Computing, 2005. IEEE International*, pages 1–11, Sept. 2005.

[60] Vadim Tropashko. Nested intervals tree encoding in sql. *SIGMOD Rec.*, 34(2):47–52, June 2005.

[61] Wei Wei, Ting Yu, and Rui Xue. ibigtable: Practical data integrity for bigtable in public cloud. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, CODASPY '13. ACM, 2013.

[62] Zhou Wenchao, Marczak William R., Tao Tao, Zhang Zhuoyao, Sherr Micah, Thau Loo Boon, and Lee Insup. Towards secure cloud data management. *University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-10-10.*, Feb 2010.

[63] Min Xie, Haixun Wang, Jian Yin, and Xiaofeng 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.

[64] Min Xie, Haixun Wang, Jian Yin, and Xiaofeng 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.

[65] Yin Yang, Dimitris Papadias, Stavros Papadopoulos, and Panos 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.

[66] Jinguo You, Jianqing Xi, Chuan Zhang, and Gengqi 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.

[67] Qing Zhang, Ting Yu, and Peng Ning. A framework for identifying compromised nodes in wireless sensor networks. *ACM Trans. Inf. Syst. Secur.*, 11(3):1–37, 2008.

[68] Shanyu Zhao, Virginia Lo, and Chris GauthierDickey. Result verification and trust-based scheduling in peer-to-peer grids. In *P2P '05: Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing*, pages 31–38, Washington, DC, USA, 2005. IEEE Computer Society.