Fall 2014

Trustworthy data from untrusted databases

Rohit Jain
Purdue University

Graduate School Form 30
(Updated 11/20/2014)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Rohit Jain

Entitled
TRUSTWORTHY DATA FROM UNTRUSTED DATABASES

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Sunil Prabhakar

Mikhail J. Atallah

Walid G. Aref

Dongyan Xu

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification/Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Sunil Prabhakar

Approved by Major Professor(s): _____

_____

Approved by: Sunil Prabhakar/William J. Gorman 12/03/2014

Head of the Department Graduate Program Date

TRUSTWORTHY DATA FROM UNTRUSTED DATABASES

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Rohit Jain

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2014

Purdue University

West Lafayette, Indiana

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Sunil Prabhakar, who has been a great mentor to me both in academia and life in general. It has been a privilege working with him. Sunil has painstakingly and warmly listened to my cooked and uncooked ideas and thoughts, let me wander until they made sense, and gave me valuable advice when I needed it. He has been a role model, and I have learnt so much from him. I am forever indebted to him.

I would like to thank Dr. Walid Aref, Dr. Mikhail Atallah, and Dr. Dongyan Xu for serving in my committee, giving me useful feedback, and discussing research ideas with me. I would like to thank Chris Mayfield, Yinian Qi, and Sarvjeet Singh for their discussions with me and for helping me whenever I got stuck.

A special thanks goes to my partner, Mariheida Córdova Sánchez. Her support, encouragement, and patience during ups and downs in my research and personal life were crucial ingredients for this dissertation. I would also like to thank my friends, Pawan Prakash, Soumyadip Banerjee, Ankit Jain, and Sindhura Balireddy for their constant support and help throughout this process. Their presence made this journey a lot more enjoyable than it would have been otherwise.

I would like to thank all the staff members of the department of Computer Science, especially, Dr. William J. Gorman, Renate Mallus, late Amy Ingram, Nicole Piegza, Nick Hirschberg, Daniel Trinkle, Steven Plite, Mike Motuliak, and Tammy Muthig. They were very approachable, patient, and helpful in keeping away every small and big hurdle so I could focus on research.

I would like to thank my parents, Jain Pal Jain and Kalpana Jain for their constant encouragement and guidance. They have always been supportive of what I chose to do, and helped me when I felt low. I would like to thank my uncles, Ramesh C. Jain and Ravindra K. Jain, who introduced me to engineering and scientific methods. I

would like to thank my brother, Anuj Jain, who was there whenever I needed to let off steam. Last and not least, I thank my boricua family for their love, support, and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

SYMBOLS

h	a one-way hash function
$h(x)$	the value of a one way hash function over x
$\Phi(n)$	label of node n in MB-Tree
H_i	label of the i^{th} node in the MB-tree
$a\|\|b$	concatenation of a and b
t	a tuple in a relation
t_i	the i^{th} tuple of a relation
$S_{Carol}(h(M))$	digital signature produced by $Carol$ for message M
$S_{Carol}(M)$	signed message M signed by $Carol$
VO	verification object
$Proof$	the MB-tree root label
C_h	cost of computing one hash
C_{IO}	cost of one disk IO
C'_{IO}	cost of searching and retrieving one block from history
DB_i	the consistent DB state after the i^{th} commit
MBT_i	the proof structure after the i^{th} commit
$Proof_i$	the MB-tree root label after the i^{th} commit
S_h, S_n, S_t	size of a hash value, MB-tree node, and a tuple in the user table respectively
T_i	i^{th} transaction in the commit order
R	range set for access control
r_i	i-th range in the range set, R
S_i, K_i	state and key for range r_i
$Enc_k(x)$	encryption of x using symmetric-key k

B_n	access control bitmap for node n
$t(a)$	a tuple with key value a
$t(a, v)$	v^{th} version of a tuple with key value a
Q_j	j^{th} read statement in the transaction
DB_{qj}	latest consistent state seen by Q_j
TS_s	the timestamp when the transaction was submitted
TS_i	the timestamp when the i^{th} transaction was committed

ABBREVIATIONS

ACID	Atomic, Consistent, Isolation, and Durable
AES	Advanced Encryption Standard
API	Application Programming Interface
BGLS	Boneh-Gentry-Lynn-Shacham Scheme
CVS	Concurrent Versioning System
DBMS	Database Management System
EMB	Embedded Merkle B+ Tree
FPGA	Field Programmable Gate Array
HIPAA	Health Insurance Portability and Accountability Act
IO	Input Output
MBT	Merkle B+ Tree
MHT	Merkle Hash Tree
MVCC	Multi-version Concurrency Control
PL/SQL	Procedural Language/Structured Query Language
SQL	Structured Query Language
VO	Verification Object
XML	Extensible Markup Language

ABSTRACT

Jain, Rohit Ph.D., Purdue University, December 2014. Trustworthy Data from Untrusted Databases. Major Professor: Sunil K. Prabhakar.

Increasingly, data are subjected to environments which can result in invalid (malicious or inadvertent) modifications to the data. For example, when we host the database on a third party server, or when there is a threat of insider attack or hacker attack. Ensuring the trustworthiness of data retrieved from a database is of utmost importance to users. In this dissertation, we address the question of whether a data owner can be assured that the data retrieved from an untrusted server are trustworthy. In particular, we reduce the level of trust necessary in order to establish the trustworthiness of data. Earlier work in this domain is limited to situations where there are no updates to the database, or all updates are authorized and vetted by a central trusted entity. This is an unreasonable assumption for a truly dynamic database, as would be expected in many business applications, where multiple users can access (read or write) the data without being vetted by a central server. The legitimacy of data stored in a database is defined by the faithful execution of only valid (authorized) operations. Decades of database research has resulted in solutions that ensure the integrity and consistency of data through principles such as transactions, concurrency, ACID properties, and access control rules. These solutions have been developed under the assumption that the threats arise due to failures (computer crashes, disk failures, *etc.*), limitations of hardware, and the need to enforce access control rules. However, the semantics of these principles assumes complete trust on the database server. Considering the lack of trust that arises due to the untrusted environments that databases are subjected to, we need mechanisms to ensure that the database operations are executed following these principles. In this disserta-

tion, we revisit some of these principles to understand what we should expect when a transaction execution follows those principles. We propose mechanisms to verify that the principles were indeed followed by the untrusted server while executing the transactions.

1 INTRODUCTION

Database services are often hosted in an untrusted environment which can lead to invalid execution of database operations. Such possibilities clearly arise when we employ a third party server to host the databases services, *e.g.*, cloud servers, as we lack complete control over the hardware and the software running at the server. Even when the server is trusted, such possibilities can arise due to a malicious insider or an intruder who manages to compromise the server or communication channels. With the advent of Cloud Computing, there is increasing interest to move databases onto a cloud platform. Although Cloud Computing holds great promise, the possibility of malicious activities raises a number of security and privacy concerns. It is of utmost importance to the data owner to ensure that the database operations are executed faithfully. Due to the possibilities of malicious activities, there is a reluctance to blindly trust the server. In this dissertation, we address the question, *is it possible to reduce the level of trust required in order to trust the data retrieved from the untrusted server ?*

In most settings, the server is likely to be honest, *i.e.*, it would not intentionally compromise the integrity of the database. However, a server may be improperly configured, or inadvertently left open to hacker attacks. There is also the concern about the server being attacked by an external or an internal entity that can compromise the integrity of the data or database operations despite the best efforts of the server. Even though the service provider is likely to be honest, it may try to hide its failure. Currently, users have no recourse but to trust the server blindly or rely on legal agreements. Even with such agreements, it is difficult for a user to discover, let alone prove, any foul play by the server.

In this dissertation, we consider the following general model. There are three main entities involved: **Alice**, the data owner; **Bob**, the (untrusted) database server;

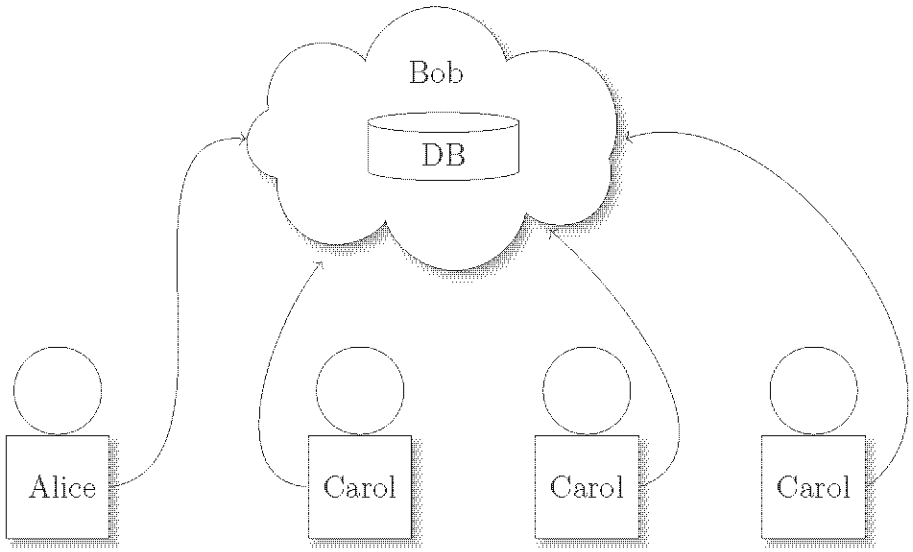


Figure 1.1.: The various entities involved: The database owner (Alice); The database server(Bob); and Authorized users (Carol).

and **Carol**, the user(s) that access the database hosted on the server. Figure 1.1 shows the various entities involved in this model. Alice (and Carol) is concerned about the trustworthiness of the database operations and would like to be ensured that the user operations were executed faithfully by the server. Bob is interested in hosting the services for Alice (possibly in return for a fee) and will thus make efforts to ensure that Alice is convinced about the fidelity of the hosting. Bob has complete control over the hardware and the software that is used to host the database. Bob can intercept all communication between the users and the server and may modify user requests, or server responses. Note that our assumptions about Bob are minimal. In most settings, the server is likely to be at least semi-honest – *i.e.*, it will not maliciously run invalid operations or modify a server response. However, due to poor implementation, failures, over-commitment of resources, or other reasons, some loss of data or updates may occur. Given the lack of direct control over the server, Alice should not assume that Bob is infallible.

Since the data server and users communicate over the Internet, an important concern is the security and privacy of the data and the database operations. Alice

needs to decide if she wants to encrypt the data and the operation logic. On the one hand, encryption of the data and the operations would provide security and privacy from malicious entities. On the other hand, it restricts the range of operations that can be performed by the server. Even though providing reasonable solutions for privacy preserving computation over encrypted data is an important problem, most of this dissertation focuses on ensuring integrity of the data and database operations. We do not address privacy of data or database operations except to the extent enabled by storing encrypted data at the server.

The main challenge for ensuring faithful execution of database operations comes from the dynamic nature of databases. A typical database is accessed by multiple users independently. Users interact with the database directly using transactions, often concurrently. These transactions can read or modify the data without prior knowledge of the data owner. Past works in this domain put the burden of applying updates to the data on the data owner. This is an undesirable assumption for a dynamic database. We proposes solutions with which authorized users can execute transactions at the server concurrently without being vetted by the data owner. Users can ensure that the database operations were executed following database principles. This is done by engaging the server in a protocol that requires the server to declare the database state(s) on which the transaction was executed and the database state that the transaction produced. Using our solutions, we shift the necessity for trust on the database server to the data owner. The data owner stores a small information which is enough to verify that the database operations were executed faithfully. This can be done by a third-party server without the knowledge of the domain logic.

Specifically, this dissertation makes the following contributions:

- We identify the problem of establishing transactional integrity for databases hosted at untrusted servers. We propose solutions to establish that the user transactions were executed faithfully. Our solutions enable the data owner to detect malicious modifications to the database and prove that the modifications were indeed invalid. Assured provenance of the database and indemnity for the

server from false claims of foul play is provided as well. A proof-of-concept implementation on Oracle is provided. We provide an empirical evaluation of the solutions.

- Access Control is an important part of database systems. Most authentication mechanisms fail in the presence of access control rules. We propose solutions that allow the data owner to enforce access control rules without having to trust the server. With our solutions, database operations can be authenticated in the presence of access control rules. We propose some encryption schemes to encrypt the data so that only the authorized users can read or write the data. We implemented our solutions on top of Oracle. An empirical evaluation of our solutions shows the efficiency our solutions.

- In practice, a database allows transactions to run at different isolation levels to improve efficiency. Different isolation levels risk showing the transactions an inconsistent database state. Ensuring that the transaction execution followed the semantics of the given isolation level is required to ensure the trustworthiness of the transaction execution. We revisit the semantics for different isolation levels to understand requirements for ensuring that a particular transaction was executed at a certain isolation level and expand our solutions to provide verification mechanisms for the users to ensure that a given transaction execution followed the isolation level semantics honestly.

The rest of this dissertation is organized as follows. Chapter 2 presents the related work done in this area. Chapter 3 presents some preliminary tools that are necessary for this work and summarizes existing results that form the basis of our solution. Chapter 4 presents our proposed solutions for ensuring transactional integrity. Our solutions to enable the data owner to enforce access control rules are discussed in Chapter 5. Chapter 6 expands our solutions to support different isolation levels. Finally, Chapter 7 discusses some future work and concludes the dissertation.

2 RELATED WORK

The problem of ensuring authenticity of database operations when the database is hosted on an untrusted server has been explored by several researchers. In this chapter, we review these work.

2.1 Authenticity and Integrity of Query Results

Much work has been done towards ensuring the authenticity and integrity of query results from an untrusted (*e.g.*, outsourced) database service [1–7]. Ensuring authenticity and integrity of a query result requires verifying two important aspects: *correctness* and *completeness*. Correctness of a query result requires that all data items in the query results do exist in the database. Completeness requires that no data item is omitted from the query result, *i.e.*, all data items that should have been part of the query result are indeed present in the query result. Some of the earlier work proposed solutions to ensure only correctness of query results [1,8], while later works considered both correctness and completeness [2,4]. A few of these works [2,4,7,9] have also considered updates. In most of these works, it is assumed that a single, central entity executes the updates on the database, while all other entities only read the data. This is an unreasonable assumption for many applications. Only limited work has been done for the situation where multiple users can update the data [4].

2.1.1 Static Databases

As mentioned before, most of the work done towards ensuring authenticity and integrity of query results has been limited to static database setting, where either no updates are applied on the database or a single entity updates the database. In these

works, it is assumed that the updates are performed at the data owner's site and the correct values of the updated data are therefore known directly to the data owner. The owner then participates in a protocol with the server to ensure that these new values are added to the outsourced database, and the necessary authentication data structures are updated correctly. Thus the data owner bears a significant burden for each update. [7] proposes an authentication data structure based on skip lists which allows a user to perform membership queries on a dataset, *i.e.*, users can query the server asking if a particular data item exists in the database. The proposed mechanisms allow the user to verify that the query result was correct. The solution allow the data owner to insert or delete data items from the database. [2] offers another example of a single updater solution. It proposes an embedded Merkle tree (EMB tree) for query correctness and completeness. An EMB tree is an embedded B+-tree similar to a Merkle Hash tree. The root hash of the tree is made available to users. With the help of this root hash, users can prove the correctness and completeness of their query results. Updates are performed only by the data owner and the updated root label is then distributed to the users.

2.1.2 Dynamic Databases

A typical database supports a multi-updater model where a large number of authorized users read and write data directly on the database. In such setting, it is infeasible for the database owner to determine the correct updates. Only limited work has been done towards ensuring authenticity and integrity of query results in a multi-updater model [4]. [4] proposes a solution for ensuring authenticity and integrity of query results in the multi-updater model. The solutions use BGLS [10] signature scheme to sign tuples to ensure authenticity of data items. BGLS signature scheme provides a multi-party signature scheme with which users can sign the data they generate with their own private keys. To verify the authenticity of a set of data items, the signatures of the data items are aggregated together and verified in a single

step. When a user updates the data, [4] requires users to sign the new value using their key. For proving completeness, signature chains are used. Each tuple is signed together with the tuple just before and after it in sorted order. The approach has been shown to be orders of magnitude slower than hashing based schemes [2].

Even though signatures can ensure authenticity of data, the server can still present the updates in different order, or it may drop an update. As a result, different users can view different (inconsistent) versions of the databases. To avoid this, [11, 12] propose protocols based on the notion of *fork-consistency*. Fork-consistency ensures that a user's view of the data is derived by the correct application of updates from other users. If an update from another user was omitted from a user's view, the user will never see any future update from that user. Eventually, users can communicate among each other to verify if they are able to see each other's updates. If not, they have detected a malicious activity. Under the given solutions, the server has no way to "fix" the malicious activities. Thus, once a malicious activity occurs, the server cannot hide it in case the users decide to compare each other's views. This makes the detection of such malicious events easier. However, the proposed work handles only single read or write operations and the operations that require multiple reads and writes, *e.g.*, typical database SQL queries or transactions, are not handled.

2.1.3 Other Data Types

Researchers have also begun to consider similar problems with respect to generic data. For example, [13] presents a solution to the problem of ensuring that large files that have been outsourced are indeed available for download in their entirety. The solution provides a means to ensure that if parts of such files become corrupted or missing, then the user is able to discover this with high probability. [14] presents a solution to the problem of ensuring correct execution of a CVS server. In this work, it has been proved that to prove the execution of CVS (which includes both read and write), users have to communicate with each other. The need for communication

arises to ensure the *freshness* of the data. [15] uses the notion of fork-consistency to ensure integrity of revision control systems.

2.2 Trusted Hardware

Some work has also been done towards ensuring the integrity of databases using trusted hardware [16–18]. In these systems, trusted hardware (*e.g.*, a secure co-processor) is used to execute queries correctly and with privacy. [16], for example, provides mechanisms to execute SQL queries with honesty without having to trust the server. This is achieved by using server-hosted tamper-proof hardware. A significant advantage of using tamper-proof hardware is that the expressiveness of the SQL queries (or transactions) are not compromised. However, the tamper-proof hardware have limited resources compared to commodity computing hardware. To execute a SQL query, the user encrypts the query using the trusted hardware's public key and sends it to the server. Since the query is encrypted, only the trusted hardware can read the query. Since the trusted hardware has limited storage capacity, it stores the data on the untrusted server hard-disk in an encrypted form. Based on the query, the trusted hardware reads the appropriate part of the encrypted data from the server hard-disk, decrypts it and processes the query. Since the trusted hardware has limited main memory, mechanisms have also been proposed to store the intermediate data produced during query processing to the untrusted storage in encrypted form.

[18] proposes mechanisms for fine-grained integration of the trusted hardware (using FPGAs) and untrusted commodity servers. Fine-grain integration enables a smaller footprint for the trusted hardware while still leaving room for generic queries. The proposed solutions encrypt only the necessary parts of the database. The encryption scheme can be chosen based on the expected database operations. For example, a deterministic encryption scheme can be used to encrypt a table column on which a equi-join is expected. However, if the encryption scheme does not allow a particular

operation, the trusted hardware can be used to decrypt the data and execute the operation on the raw data.

2.3 Provenance

Much work has been done towards data provenance and tamper-proofing of data [19–23]. While most works focus on storing and querying provenance [19, 22], some have considered the problems of privacy and trustworthiness of data provenance [20, 21]. [24,25] propose mechanisms to store the provenance of the workflow in a database. [25] allows queries on this provenance to understand the role of each component in the workflow [26], and to understand the intermediate data.

Due to regulatory requirements, some application require to store provenance [27]. [21] defines *secure* provenance as the one which ensures that authenticity of provenance, and access control, *i.e.*, users can selectively preserve the privacy of provenance records and only authorized users can access the provenance. [28] propose solutions for tamper detection of an audit log of the database that records all changes. This work does not address outsourcing or privacy concerns and assumes that the database owner is a trusted entity.

2.4 Privacy

Other orthogonal avenues of research focus include hiding data from outsourced databases [29–34], hiding access patterns from the database servers [35], privacy preserving data publishing [36–38], and, private outsourcing of computation [39–43]. [44,45] propose solutions to distribute computation tasks across untrusted nodes on a public network. The proposed solutions ensure privacy of the computation task and data while also ensuring trustworthiness of such computation.

[33] proposes solutions to execute SQL queries over encrypted data. The data are encrypted before sending it to the server. The data are partitioned into ordered buckets to support range queries. user queries are rewritten so that the query can

be executed on the partitioned (encrypted) data. Since the data are encrypted, expressiveness of SQL is compromised when rewriting the query. The remainder of the query is executed at the user after decryption of the data. [34] proposes solutions to alleviate some of these problems. It proposes a collection of SQL-aware encryption schemes to support various SQL features, *e.g.*, deterministic AES to support equi-joins, order-preserving encryption [29] for range queries, homomorphic encryption for summation, *etc.*. The data are encrypted in multiple layers. As necessary, the layers are decrypted by supplying the layer key to the server. By doing so, the data owner can ensure that the data are encrypted as much as possible while still being able to execute the queries. In the worst case, if its required, the data can be decrypted completely to support any SQL query.

Much work has been done towards private outsourcing of computation, where the data owner does not want to reveal the data [40–43]. [43] provides mechanism to evaluate a function F at an untrusted server, without revealing the input or the output of the function to the untrusted server. [40] proposes a solution to the problem of computing edit script to convert a string u to another string v.

2.5 Access Control

Most of the solutions proposed for ensuring authenticity and integrity of query results leak extra information at the time of verification. The extra information is necessary for the user to be able to verify the correctness and completeness of the query result. This is undesirable in the presence of access control rules as the extra information required for verification may not be accessible to the user. Some work has been done towards ensuring correctness and completeness in presence of access control rules [9, 46–49]. While [9] supports one-dimensional range queries and data updates, [47] supports multi-dimensional range queries and does not handle updates. Both these solutions do not provide privacy against the server. [48, 49] focus on the access control problems with data authenticity for XML data.

Much work has been done towards key management [50–54]. [50–52] propose key management solutions for access hierarchies. [50] proposes a solution to not only restrict user access to limited data, but also till limited time. [53, 54] consider lazy revocation model where the user can maintain access to an old version of data when they are evicted from group. However, any new value that is accessible to the group will not be accessible to the evicted user. This is done by using key derivations which allows user to derive previous encryption keys but does not allow user to derive future keys. When a user is evicted from the group, the data is not re-encrypted immediately using the new key. All future updates are encrypted using the new key. This saves a lot of computation and I/O cost whenever access control rules are changed.

3 PRELIMINARIES

In this chapter we present some basic tools that we use for building our solutions, and also discuss existing solutions for ensuring completeness and correctness.

3.1 Tools

We use three data security tools in this chapter: *one-way hash functions* [55], *Merkle Trees* [56], and *digital signatures* [57].

3.1.1 One-Way Hashing

A one-way hash function h takes as input a data item x and produces as output the hash of the data item $y = h(x)$. Important requirements for a one-way hash function are:

- Given a hash value y, and the hash function h, it is infeasible to find x such that $h(x) = y$
- It is infeasible to find two different data items, x and y, such that $h(x) = h(y)$.

One-way hash functions ensure that it is infeasible to compute inverse of the function, *i.e.*, given a desired hash value, it is computationally impossible to find the input value. However, verifying that the input value does produce the given hash value is easy.

There are many well-known and commonly used strong one-way hash functions such as SHA-256.

3.1.2 Merkle Hash Trees

A Merkle Hash Tree (MHT), or Merkle Tree, is a binary tree with labeled nodes. Each leaf node represents a data item. We represent the label for node n as $\Phi(n)$. If n is an internal node with children n_{left} and n_{right}, then

$$\Phi(n) = h(\Phi(n_{left})||\Phi(n_{right})) \tag{3.1}$$

, where $||$ is concatenation and h is a one-way function. Labels for leaf nodes are computed using data values depending upon the application, *e.g.*, in case of a database relation, a label is calculated as the hash of the tuple represented by that leaf.

3.1.3 Digital Signatures

A digital signature serves to ensure the legitimacy of a digital message. In particular, it gives the following assurances to the recipient:

- *Authenticity:* the message was created by the given sender.
- *Non-repudiation:* the sender cannot deny having sent the message.
- *Integrity:* the message was not altered in transit.

There are many well known digital signature schemes such as RSA based digital signature. In order to sign a message, M, the sender first computes the hash of the message using a public one-way hash function, h, and then encrypts the hash value using their private key: $S_{sender}(h(M))$. We will refer to the message M, and the signature $S_{sender}(h(M))$, together as a *signed message*, or $S_{sender}(M)$. To verify the authenticity and integrity of a signed message, the recipient applies the same hash function to the message to compute $h(M)$, and decrypts the signature using the sender's public key. If the decrypted value matches the hash value that they computed, the recipient is certain that the message was indeed signed and sent by the sender.

3.2 Correctness and Completeness

We begin by discussing the use of Merkle Trees to prove correctness. And then further discuss a variant of it, MB-tree, which we use as a building block for our overall solution.

3.2.1 Correctness

Correctness requires that all values that are part of a query result are indeed part of the database. In other words, this implies that the values returned are indeed from the right consistent state of the authentic database, and not *manufactured* by the server or an attacker.

Merkle trees can be used to establish the correctness of query results from an outsourced database. Initially, when Alice chooses to outsource a database relation, she computes a Merkle tree over the relation. Each leaf node in the Merkle Tree represents a tuple in the database relation. The data are then sent to Bob for servicing queries. Bob computes the same Merkle tree structure over this data. Alice and Bob ensure that the hash values of the Merkle tree that they independently computed match. Once Alice and Bob agree on the state of the initial data and Bob starts services queries from Alice (and Carol). After this, Alice saves the value of the root label and can delete the data and the Merkle Tree. Alice distributes the root label to the users.

Figure 3.1 shows an example of the tree produced. In the figure, the nodes with labels $t_1, t_2, \ldots, t_8$ represent the tuples. Other nodes represent the nodes in the Merkle Tree. Each leaf node (*e.g.*, H_8, H_9) represent a single unique tuple in the database.

The correctness of a tuple, t_i, in the result of any query is established as follows: Alice (or Carol) requests the *Verification Object* (VO) for tuple t_i. Consider t_5 as an example. Let H_{12} be the label of the leaf node that represents t_5. The VO for t_i consists of all sibling nodes along the path from x to the root of the Merkle tree. In Figure 3.1, these nodes are shaded and are: H_{13}, H_7, H_2. Bob returns to Alice the

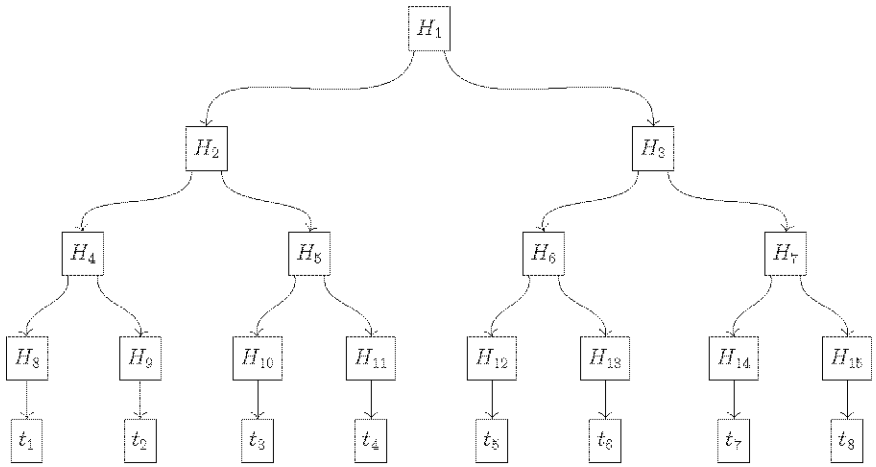


Figure 3.1.: An example Merkle Tree

VO for t_5. Alice uses the value of t_5 to compute the label H_{12}. She uses this label and the label H_{13} from the VO to compute the label of the parent, H_6. Alice further computes H_3 using the previously computed H_6 and H_7 that was present in the VO, and so on, all the way to the root label. If the computed value of the root label is same as the one initially computed by Alice before sending the data to the database server, she is convinced that t_5 must be part of the original table. If not, then Alice has detected a malicious activity.

Note that the hash function has to be public. The label of the root saved by Alice is known as the *proof*. Of course, it is also possible for Carol to submit queries and verify the correctness of each answer similarly. All she needs to know is the value of the root label initially computed by Alice. Since we use one-way hash functions, Bob cannot cheat the verification process.

3.2.2 Completeness

Completeness requires that all values that should be in the answer to a query are indeed present in the answer. In other words, Bob has evaluated the query correctly and has returned all tuples that are produced as a result without dropping any out.

Table 3.1.: Sample Data Table

tupleID	A
1	23
2	29
3	35
4	48
5	59
6	63
7	65
8	70

Note that correctness and completeness together ensures the correctness of a read-only query.

As explained before, a Merkle Tree is a binary tree. However, it can be easily extended to use a B+-tree instead [2]. An MB-tree behaves just like a regular B+-tree with its nodes extended with child hash values. Similar to the Merkle Tree, the label of each non-leaf node is computed by hashing concatenation of labels of its children. Label of a leaf node is computed by hashing concatenation of hash values of each tuple entry in the node.

We use MB-tree to establish the completeness of the results of a query. Figure 3.2 shows an example MB-tree built on sample data in Table 3.1. Consider a range query which returns tuples $t_4 \ldots t_6$. To establish the completeness of tuples satisfying a range, the VO consists the tuple just before the range (t_3) and the tuple just after the range (t_6). VO then includes the path from those two tuples to the root.

To establish correctness, Alice computes the labels for all tuples in the results and the two surrounding values, *i.e.*, $h(t_3), h(t_4), \ldots, h(t_7)$. She then computes the labels of ancestor nodes working her way up the tree. The VO contains the labels of nodes that she needs to compute the ancestor label and also which tuples belongs to the same leaf node. Finally, she compares the computed *proof* with the *proof* she stored earlier to determine if the result set contains all the tuples that should have been returned. If this is the case, she is assured that all valid tuples were returned to her.

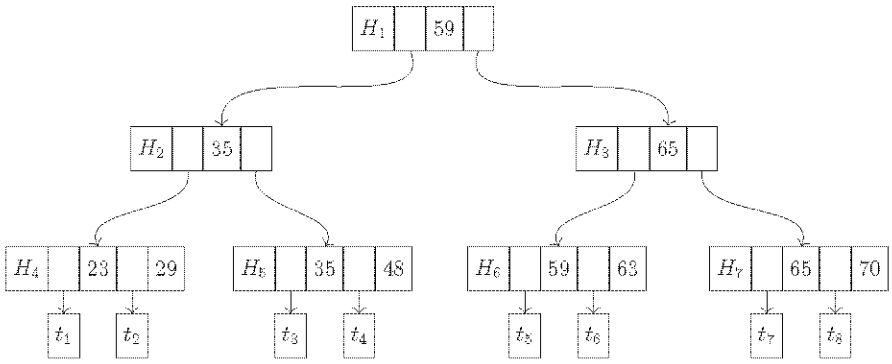


Figure 3.2.: An MB-tree on attribute A of Table 3.1

Alice verifies that t_0 and t_{r+1} are outside the query range, ensuring that the query results were indeed complete.

Updates Generated By Alice

The above solutions for correctness and completeness work for single-updater model where all updates to the database are computed at the data owner site, and the computed updates are then sent to the server. In order to update the data, Alice recomputes the Merkle Tree structure keeping the updates in mind and then sends the updates to Bob. Bob applies the updates to the database and updates the Merkle tree independently. Alice and Bob compare the updated root labels that they computed. If they agree on the state of the data after the update, Alice distributes the updated root label to the users, and Bob starts answering the subsequent queries using the updated data. The verification is now down against the new *proof*.

This solution requires that all updates are vetted by the data owner (Alice). Also, any update to the database needs to be verified before executing any further updates. In the next section, when we discuss a more general solution for updates that are not generated directly by Alice, we will remove both of these requirements.

4 TRANSACTIONAL INTEGRITY

A typical relational database is accessed (reads and writes) by multiple users using transactions. The database server is expected to execute these transactions ensuring that the ACID properties are satisfied. Even though two transactions were executed concurrently, their affect is same as if they were executed one after the other. In the scenarios where the database server is not trustworthy, the database owner needs to ensure that a transaction submitted by a user was executed correctly – the transaction read the correct and complete data and the generated updates were applied correctly and future transactions were executed on the updated data. In the past, most work in this domain focussed on cases where the users did not update the database (just read queries). Some later works considered updates from a single entity, where the updates applied at the data owner's site and then sent to the server. Some work also considered updates from multiple sources, however they are applicable to complex systems like relational databases, which support transactions. In this chapter, we present solutions that allows the database users to run the transactions on the database. The solutions allow the users to be able to verify that the transaction was executed honestly by the database server.

In particular, we provide solutions to force an untrusted (relational) database server to provide trustworthy data [58]. This is achieved by engaging the server in a protocol that makes it impossible for it to hide unfaithful execution. A key challenge for this work arises from the fact that multiple, independent users can access and make valid updates to parts of the data using SQL. In order to ensure authenticity, it is necessary to guarantee:

- **Correctness**: All answers to a query do indeed come from the authentic database.

- **Completeness**: The query answer contains all relevant tuples (*i.e.*, no part of the answer is dropped).

- **Transactional Integrity**: The database always reflects a valid consistent state – *i.e.*, the state corresponding to the initial state followed by the correct application of all previous valid transactions in the correct order. Furthermore, each new transaction executes against the latest (or freshest) state.

As mentioned before, correctness and completeness of query results have been studied in earlier work. In most earlier work it was assumed that the data were either not modified, or the updates were authenticated by the data owner and then sent to the database server. Thus, the legitimacy of any data in the database could be established directly by the data owner. However, in a dynamic database setting this is an unacceptable assumption. A typical database supports a large number of authorized users to run transactions directly on the database. Updates to the data are computed by execution of these transactions on the database. The database owner or the users cannot always determine the correct updates produced by each transaction. The validity of these updates (*i.e.*, what items are modified, and their new values) is determined by the faithful execution of a transaction semantics over the latest valid state. Due to the lack of trust on the database server, there is a need to assure the data owner that the user transactions were indeed executed honestly by the server.

Many applications may also require, *e.g.*, due to regulatory compulsions, to keep the provenance of the operations. This can be particularly important to check if malicious activity occurred in the past. In addition to these requirements from the data owner's perspective, there is an additional requirement from the service provider. The server should be able to prove its innocence if it has faithfully executed all transactions.

To the best of our knowledge, this problem of ensuring transactional integrity over an untrusted database server, as critical as it is for outsourced databases, has not been addressed in earlier work. Given that the goal of the outsourcing is to alleviate the

burden on the data owner, satisfactory solutions need to minimize the overhead and role of the data owner in establishing the authenticity and integrity of the solutions. In this regard, existing solutions are inadequate as they require the data owner to play a significant role. In addition, existing solutions also place a significant overhead on the service provider. While this may work for some applications, many outsourced databases are expected to have both a large size and high rate of querying and updates. Thus it is necessary to explore more efficient solutions with low overheads for all involved parties. The contributions of this work are:

- Identification of the problem of ensuring *Transactional Integrity* for databases hosted on untrusted servers.
- *Novel authentication mechanisms* that ensure correctness, completeness, and transactional integrity.
- Solutions that provide *indemnity for the server*, and also *trustworthy provenance* for the database.
- A demonstration of the feasibility of the solution through a *prototype in Oracle*, and its evaluation.

The rest of the chapter is organized as follows. Section 4.1 explains our models and assumptions. Section 4.2 presents our requirements and proposed protocols. In Section 4.3 discusses how our solutions can provide assured provenance. A discussion of the implementation of the solution and empirical evaluation is presented in Section 4.4. Finally, Section 4.5 concludes the chapter.

4.1 Assumptions and Model

We consider the general model as described in Chapter 1. The data owner (Alice) wants to host the database on the database server (Bob) who is not trusted. The users (Carol) interact with the database using transaction. These transactions can

read or update data. users are authorized by Alice and can independently authenticate themselves with the server. As in a typical database, multiple users can run transactions concurrently. These transactions can modify the data as well.

To ensure the legitimacy of the communication (for example, a transaction execution request sent by a user, or transaction results sent by the server) between different entities, the communication is digitally signed. Each entity has a public key for digital signatures. A user's private key should remain secret else an intruder can compromise the database.

To ensure that a transaction was executed faithfully, the user or the data owner can ask the server to provide some extra information. Using this extra information, the user or the data owner can ensure the transaction was executed faithfully. It is desired that if Bob makes any invalid changes to the database or the results of queries, Alice (or Carol) should be able to discover this and be able to prove his error. Similarly, from the database server's perspective, it is required that if Bob does indeed faithfully operate the database then he should be able to establish his innocence, *i.e.*, Bob is indemnified against invalid claims by Alice or Carol if he faithfully executes the database.

In a typical database, a transaction can be aborted for multiple reasons, *e.g.*, when the transaction cannot be serialized. When Bob receives a transaction request, it executes the transaction and decides to either commit or abort the transaction. For this work, we assume that the users are not interested in knowing the reasons behind aborting a transaction, however, expect the *Quality of Service* agreement to provide certain guarantees on performance quality. users need not be trusted. In the case when a particular user(s) is not trusted, Alice will verify the transaction authorized by the untrusted user, else she can expect the user to verify the transactions submitted by her. We assume that the users will not collude with the server.

Bob runs a relational database that supports only *Replayable Transactions*. A replayable transaction is one which is deterministic – *i.e.*, the outcome of the transaction (the outputs it generates, the values of its updates and its decision to commit

or abort) is completely determined by the values that it reads from the database and its input parameters. In other words, a replayable transaction will produce the same output and updates when executed on the same consistent state. This assumption is no different than one that is currently made by database systems in order to ensure the well known ACID properties of databases (*e.g.*, the ability for the database to automatically restart a running transaction if it is deadlocked). For this work, we assume that all transactions are executed at *strict serializable* isolation level.

4.2 Transactional Integrity

As seen in the previous section, solutions for ensuring correctness and completeness have been proposed in earlier work. In this section we present our solution for ensuring transactional integrity in a dynamic database where multiple users can independently run transactions.

Transactional integrity requires that each transaction is run against a consistent database containing all, and only the changes of all previously committed transactions, in order of their commits. Furthermore, any updates generated by the correct execution of transaction are reflected in the updated database upon commitment.

4.2.1 Requirements

To ensure the integrity of the database, each transaction must be authorized by Alice or Carol. Alice needs to be ensured that the transaction was faithfully executed by Bob without any tampering of the results. Furthermore, any subsequent queries should be answered against this resulting database following the transaction's commitment. The problem is difficult because we must ensure that Bob does not:

- drop a transaction, *i.e.*, claims to execute it, but does not;
- add an invalid transaction not authorized by Carol;
- alter the order of execution of transactions; or

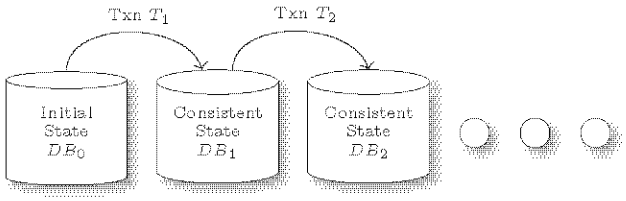


Figure 4.1.: A simplified view of database consistency

- alter the data read by or modified by a transaction.

In addition, we also require indemnity for Bob: that is, if he faithfully executes all transactions received from Alice or Carol, it is impossible for Alice or Carol to implicate him, *i.e.*, he must be able to prove his innocence.

4.2.2 Key Idea

Databases are complex systems, but they are built to ensure that the following simple definition of consistency is satisfied [59]. Figure 4.1 shows this graphically. The initial state of the database (DB_0) is considered to be a consistent state. The correct (isolated, atomic) execution of a transaction(T_i) over a consistent state(DB_{i-1}) takes the database to a new consistent state(DB_i). Of course, this is only a conceptual notion – in reality multiple transactions execute concurrently and can have various SQL isolation levels. Thus, in practice, the database is in an inconsistent state represented by the partial execution of concurrent transactions. However, when a transaction is allowed to commit it is certain that its execution is equivalent to having run in isolation against the consistent state produced by the execution of all transactions that have committed earlier (in the order of commits).

Our initial solution focuses on strict serializability. If a user is willing to run a transaction with a weaker isolation level, than our solution can be applied with an appropriately relaxed notion of correctness. Since strict serializability is the most stringent condition for correctness and completeness (as required by many applications, *e.g.*, Banking), we limit our discussion to this case.

Our solution is built upon the fact that in strict serializable isolation level, each transaction "sees" a consistent, committed state of the database corresponding to the state produced after completing earlier committed transactions. All its reads are from this consistent state. If the transaction is able to successfully execute and commit, it generates a set of updates which must all be installed atomically to produce the next consistent state.

Based on this observation, from the conceptual point of view, the integrity and authenticity of a transaction's execution can be divided into three sub-components:

- Establishing that all values read by the transaction come from a single consistent state (in particular, one that reflects the updates of all prior committed transactions, in correct order).
- Faithful execution of the transaction using these values – determination of the correct values of the updates.
- Establishing that all updates generated by this execution have been applied to the database.

Under the assumption of replayable transactions, Bob does not need to prove the second point listed above (faithful execution) since this can be verified by a simple re-execution of the transaction over the same consistent state that was visible to the transaction when it was run by Bob. We need to establish that a given transaction, T_i, ran against a particular consistent state, DB_{i-1} and produced a particular consistent state, DB_i. This will be achieved by using MB-tree structure discussed earlier. Although the database at any time is in flux and contains inconsistent states, we will only update the MB-tree structures at the time of transaction commitment. Thus each new MB-tree reflects the commitment of exactly one transaction. In other words, we maintain a one-to-one correspondence between the conceptual consistent states and the proof structure – a new version of the structure is generated in one step only upon the commitment of a transaction. This new structure is computed from the previous structure by applying the changes made by exactly one transaction

– the one that has just committed. Bob has to declare this structure at the time of commitment of a transaction (by sending out a signed copy of the new root label, $Proof_i$, beyond his control), and be able to use this structure when asked to verify the correct execution of the transaction (which will also involve the structure before the commitment of the given transaction).

4.2.3 The Protocol

We now discuss our solution for ensuring Transactional Integrity. We discuss the initialization steps taken by Alice and Bob, the execution of a single transaction by Carol (similar for Alice), and the verification steps to establish the validity of a given transaction. For ease of exposition, we discuss only the case of one relation. The extension to multiple relations is straightforward and omitted due to lack of space. All signed messages are verified by the recipient – this is not explicitly mentioned in the discussion below.

While it is certainly feasible for Carol (or Alice) to validate every single transaction, this results in a heavy burden. In practice, we expect that they will randomly validate transactions with a frequency that reflects their distrust of Bob. In order for this to achieve the goals of this work, it is essential that they be able to verify any past transaction without any forewarning to Bob during the execution of the transaction. Moreover, the solution must ensure that once they request the validation of a transaction, it is impossible for Bob to go back and "fix" any errors or omissions with respect to the execution of that transaction. These requirements are met by our solution.

Initialization

Algorithm 1 describes the initialization step. In the beginning, Alice and Bob independently compute MBT_0 – the MB-tree structure over the initial state of the

Algorithm 1 Initialization

1: Alice sends the initial database, DB_0 to Bob
2: Bob computes MBT_0, and sends $S_{Bob}(Proof_0)$ to Alice.
3: Alice independently computes $Proof_0$ and verifies what Bob has sent.
4: Alice retains $Proof_0$. (She may now choose to discard her copy of the database.)

relation. Alice retains only the proof, $Proof_0$. Bob sets up the database with this initial table and opens shop, ready for processing transactions from Alice and Carol.

Transaction Execution

Algorithm 2 describes this step and Figure 4.2 shows the steps graphically. To execute a transaction, Carol sends to Bob a signed message containing the identity of the transaction (as discussed above), all necessary parameters (*e.g.*, account numbers), and a unique transaction sequence number (SID). This sequence number must be unique for each transaction submitted by Carol (if there are multiple Carols, *i.e.*, multiple authorized users, the number needs to be unique for each such user, not necessarily across all users). Bob verifies the signature to be that of Carol and examines the message. He rejects a transaction request from Carol if the sequence number is not larger than the earlier request from the same user. The sequence numbers prevent *replay* attacks by Bob. That is, Bob will be prevented from re-using a transaction request to run that transaction multiple times. The details will become clear later in the section.

Bob then runs the requested transaction. He keeps track of the data items written by the transaction. If the transaction successfully commits, Bob installs the updates produced by the transaction into the current proof structure. Since transactions are committed sequentially, let i be the ordinal position of this transaction's commitment since the initial outsourcing. This transaction will be identified by Bob as T_i – *i.e.*, the i^{th} transaction to commit. Concurrency control will ensure that this transaction's reads were consistent with DB_{i-1} – the consistent state corresponding to all earlier

Algorithm 2 Transaction Execution

1: Carol generates the unique transaction sequence number SID and sends $S_{Carol}(transaction, SID)$ to Bob.
2: Bob records this message after verifying the signature and executes the transaction.
3: If the transaction successfully commits, Bob computes MBT_i and sends $S_{Bob}(SID, i, Proof_{i-1}, Proof_i, RSet)$ to Carol. i is the transaction's commit sequence number, and $RSet$ is the result set produced by the transaction.
4: Bob also sends to Alice $S_{Bob}(i, Proof_{i-1}, Proof_i, S_{Carol}(transaction, SID))$.
5: Alice verifies Bob's signature and then adds $Proof_i$ to its chain of proofs after verifying that $Proof_{i-1}$ is at the end of the current chain. Alice also checks that the SID for this user is in increasing order.
6: Carol sends $S_{Carol}(Proof_{i-1}, Proof_i)$ to Alice.
7: Alice checks that $Proof_{i-1}$ and $Proof_i$ are contiguous proofs in its chain.

committed transactions. If this is not the case, then the transaction will not be allowed to commit (recall that we are assuming only strict, serializable executions [59]). Once it commits, its changes will be included in the next conceptual consistent state, DB_i. Bob stores the authorization message from Carol along with the transaction's commit position, i.

Bob needs to declare that T_i was applied on DB_{i-1} and produced DB_i. He does this by computing the corresponding MB-tree structures MBT_{i-1} and MBT_i. As part of the proof of the commitment of T_i, he send to Carol a signed message containing (i) the sequence number submitted by Carol, (ii) the transaction's commit sequence number, i, (iii) the label of the root of MBT_{i-1}, *i.e.*, $Proof_{i-1}$, (iv) the label of the root of MBT_i, *i.e.*, $Proof_i$, and (v) RSet, the result set produced by the transaction. He also sends to Alice a signed message containing (i) the transaction commit sequence number, i, (ii) $Proof_{i-1}$, (iii) $Proof_i$, and (iv) the Carol's transaction request – $S_{Carol}(transaction, SID)$. Note that for a read-only transaction, Bob need not send anything to Alice.

Alice uses the messages from Bob to maintain the sequence of proofs: $Proof_0$, $Proof_1$, ..., $Proof_i$ that Bob claims to be the sequence of consistent states that the database has gone through. She ensures that $Proof_{i-1}$ is currently the last value in

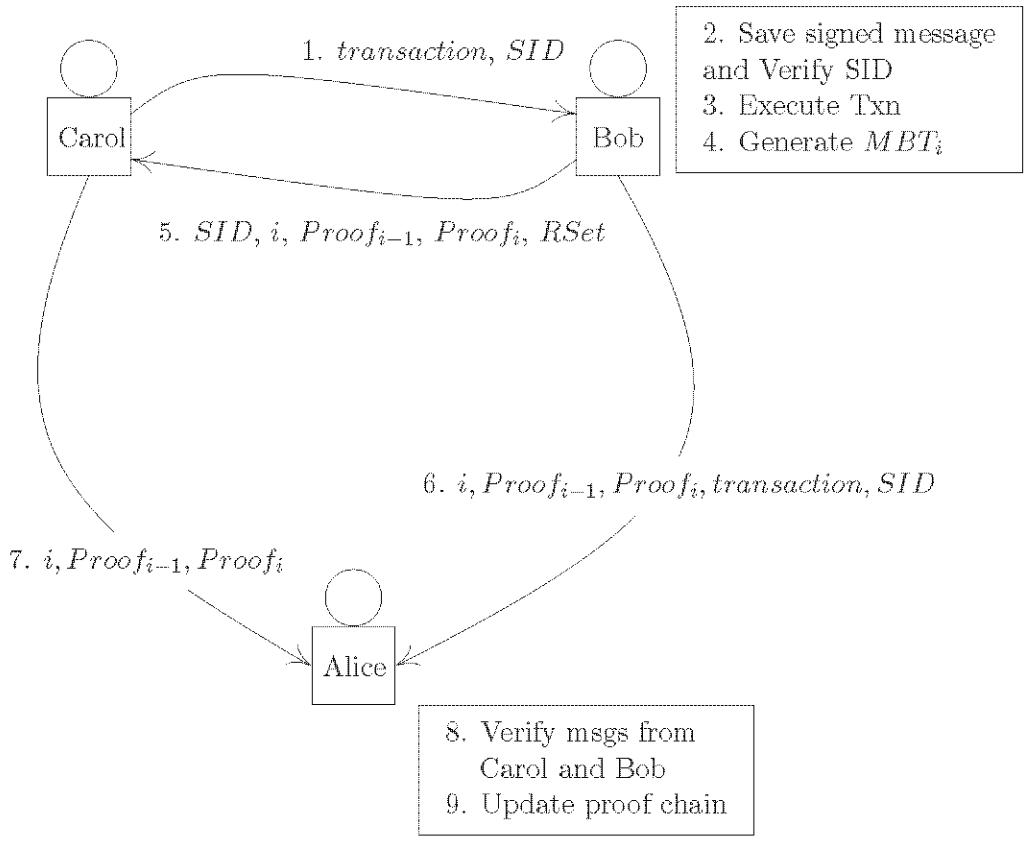


Figure 4.2.: Transaction execution

its proof chain. If this is the case, she adds $Proof_i$ to the end of the chain. She also retains the latest SID used by each user. She checks to see that the SID has not been used by this user earlier and is in increasing order for the user. Alice receives $Proof_{i-1}$ and $Proof_i$ from Carol as well, and ensures that $Proof_{i-1}$ precedes $Proof_i$ in the sequence received from Bob. If not, Alice has detected a problem.

Transaction Verification

Algorithm 3 explains the verification protocol formally. Following the execution of a transaction, Carol (or Alice) can arbitrarily decide if she wants to verify a given transaction (current or past). To verify transaction T_i, three requirements need to be established.

Firstly, Bob has to show that all values read by the transaction were indeed from DB_{i-1}[1](Step 2 and 3). To do this, he needs to produce the verification objects for the reads from MBT_{i-1}. For now, let us assume that Bob maintains a copy of MB-tree for each consistent state. We will revisit this issue later and propose a more efficient solution. Carol uses the Correctness and Completeness mechanisms discussed earlier to verify the reads against $Proof_{i-1}$(Step 4 and 5).

Secondly, Carol needs to know the correct values of all updates generated by the given (replayable) transaction when run on a database corresponding to DB_{i-1}. Given a replayable transaction and the values read by the transaction (as declared by Bob and validated in the previous step), we need to determine the values of its updates. Given the replayable transaction and the values that it reads, Carol (or Alice) can determine the values of its output and updates (Step 6).

Thirdly, Carol needs to establish that the updates of the transaction were faithfully recorded in the database and used for subsequent transactions. To do this, Bob needs to show that MBT_i differs from MBT_{i-1} by exactly the modifications of T_i. For this, Bob sends the node values from MBT_{i-1} which were modified at the time of commit of transaction T_i. Bob also sends other nodes values required for Carol to generate the $Proof_i$ (Step 7). Then, Carol can update the partial MBT_{i-1} to include the changes applied by the transaction and verify the new proof (Step 8). Carol then ensures that the new proof ($Proof_i$) is indeed what Bob claimed it to be at the time of transaction commit. This is verified by comparing it with $Proof_i$ value obtained from Alice in Step 4.

Notice that in our solutions, the overhead for Alice is minimal. Alice just maintains the proof chain and stores the latest SID used by each user. Alice incurs cost of verification only if she wants to. The users can verify the transactions independently. There is a communication cost for Alice, as both Bob and Carol updates Alice about the execution of a transaction. However, it does not stop Bob or Carol from executing

[1]As is always the case, if T_i updates a data value and subsequently reads it, it will read the value it wrote, not the one from DB_{i-1}. This should be taken care of during the transaction verification.

Algorithm 3 Transaction Verification

1: Carol asks Bob to verify a transaction T_i
2: Bob computes MBT_{i-1} and MBT_i.
3: Bob sends to Carol the verification objects for all values read by T_i based on MBT_{i-1}.
4: Carol obtains $Proof_{i-1}$ and $Proof_i$ from Alice.
5: Carol verifies the correctness and completeness of T_i's reads.
6: Carol determines the outputs and updates for T_i (replays T_i) given these reads.
7: Bob sends to Carol the verification objects for T_i's updates based on MBT_i.
8: Carol verifies that MBT_i contains these updates.

further transactions, hence, the communication can occur at available time rather than instantly.

4.2.4 Discussion of Correctness

We now show that the proposed protocol meets our requirements. We show how a failure on the part of Bob will be detected by our protocols.

Lemma 1 *If Bob (or an intruder) maliciously modifies a set of tuples, mSet, after transaction T_i (with corresponding proof $Proof_i$), then $Proof_{i+1}$ will not authenticate the malicious version of mSet.*

Proof Any tuple value after executing T_i can be authenticated using $Proof_i$, which is declared by Bob after execution of T_i. If the values in the tuple set $mSet$ were modified after T_i, $Proof_i$ will not authenticate the updated values in $mSet$, *i.e.*, the server will not be able to prove that the new values in $mSet$ were indeed part of the database after executing T_i. Thus transaction T_{i+1} will not read those values (if it does, the server will not be able to authenticate those values). Since the execution of the transaction depends solely on the data that it reads, the verifier can generate the updates that the T_{i+1} generated. Thus the calculation of the new proof will not include the malicious changes to $mSet$. Hence, $Proof_{i+1}$ will not authenticate the changes in $mSet$. ■

Theorem 4.2.1 *If Bob modifies mSet after transaction T_i and if T_k is the first transaction after T_i that accesses the tuples in mSet, then verification of T_k will fail.*

Proof For the verification of T_k, the server has to authenticate the tuples that T_k reads against $Proof_{k-1}$. Using lemma 1, we can say that T_{i+1} can not authenticate the malicious values of $mSet$. Applying the same lemma again on T_{i+1} ensures that T_{i+2} will not authenticate $mSet$ either, and so on. Hence, T_{k-1} will not authenticate the values in $mSet$ either. Hence, the verification will fail. ■

Theorem 4.2.1 proves that each transaction reads data from a consistent state which reflects the updates applied by previously committed transactions. The proof chain stored at Alice establishes the order of commitment (serialization order) of the transactions. We now show different malicious events that may compromise the trustworthiness of data, and discuss how our solution ensures that such malicious events will be detected.

If Bob drops a transaction:

Consider a transaction submitted by Carol that Bob pretends to execute (*i.e.*, sends unauthentic responses to Carol, but does not actually execute the transaction). Bob has to notify Carol that it executed the transaction and her transaction modified the proof from $Proof_{i-1}$ to $Proof_i$ (for some i). As part of the protocol, Carol will send this information to Alice (Algorithm 2, Step 6). If Bob drops this transaction, Bob will claim that the next transaction (sent by the same user or some other user Carolina) moved the proof from $Proof_{i-1}$ to $Proof_i^{'}$. When Carolina sends this information to Alice, she will detect that Bob executed two transactions on the same consistent state, which breaks the consistency of the database.

If Bob executes an unauthorized transaction:

Bob can execute an unauthorized transaction in two ways: i) Bob could manufacture a new transaction and pretend that a user sent it to him; or ii) Bob could replay a transaction that it already executed. To manufacture a new transaction, Bob has to forge a user's signature as the protocol requires Bob to execute only signed transactions – this is computationally infeasible.

To prevent a replay of an old valid request, the protocol requires a unique, increasing identifier (SID) as part of the signed request. Hence an attempt to reuse an old signature will be caught when Alice receives $S_{Carol}(transaction, SID)$ value that does not show an increase in SID for the given user (Algorithm 2, Step 5).

If Bob does not run the transactions in the claimed sequence:

The chain of proofs maintained by Alice prevents this from happening. Bob informs Carol of the commit order, i, for each transaction. The corresponding pair of proofs, $Proof_{i-1}$ and $Proof_i$ must validate this transaction. If Bob does not specify these correctly, verification of T_i will fail.

4.2.5 Indemnity for Bob

We also require that if Bob is honest and faithfully executes all transactions submitted by Alice and Carol, then he can prove his innocence. This is indeed the case for this solution.

We first consider Bob's indemnity from Carol. In order to verify a transaction submitted by Carol, Bob needs the following from Carol: (i) the request for running the transaction including the transaction name, its parameters, and a sequence number, and (ii) Carol's ability to replay a transaction faithfully. Carol cannot repudiate her request for running a transaction since she signs the request with all the necessary information. If Carol does not replay a transaction correctly, Bob can check that him-

self by replaying it and implicate Carol. This is possible because each transaction and parameters are known to each party, the values read from the database are known to Bob and he can verify that they are consistent with the consistent state corresponding to the proof value he sent to Carol in response to the transaction request. Thus, it is not possible for Carol to falsely implicate Bob.

Next we consider Bob's indemnity from Alice. Bob relies on Alice to maintain the chain of proofs and also to check that a given SID has not been used earlier for a given user. Alice cannot modify the chain with impunity. If she adds a proof that Bob has not provided, she would have to produce a signed message from Bob containing the old and new proofs. She cannot manufacture such a signed message. Similarly, she cannot delete any proof ($Proof_i$) from the chain, as she has to produce a signed message containing ($Proof_{i-1}$, $Proof_{i+1}$). If Alice claims that an SID value for a user is being reused by Bob, she can once again be challenged to produce the prior message from Bob containing this SID and user pair. If he has never sent her such a message, she will be unable to produce it.

Thus, Bob is protected from baseless claims of wrongdoing from either Alice or Carol, as desired.

4.2.6 History

Recall that we assumed above that Bob maintains a copy of each successive MBT_i corresponding to the commit of each transaction. This is expensive and unnecessary. Instead, Bob can maintain a base structure and record incremental updates to the structure after each commit. Alternatively, he can maintain the latest version of the structure and maintain enough information to work backwards to an earlier version.

To reduce the storage cost, each tuple in the database and each node in MB-tree is assigned a unique id. Each tuple in the relation is also assigned a version number. A *history* stores each value that a tuple or a node takes as the database evolves. At the start, for each tuple and MB-tree node, the history stores only one value

– the value in the database or MB-tree after the initialization step. Initially each tuple value is assigned version number 0. When a tuple or a node is modified, the version number is incremented by one and the new value is added to the history along with the transaction's sequence number which modified the value. As the database evolves, the history stores all the values that a particular tuple or MB-tree node takes on consistent states. When a user wants to verify an old transaction, the database server can use the history to generate the values that the server read when executing this transaction.

4.2.7 Efficiency

For ease of exposition, the protocol discussed above intentionally omits several possibilities for gaining efficiency. We now discuss some of the possible optimizations.

As discussed before, in our solutions, the overhead for Alice is minimal. Alice just maintains the proof chain and stores the latest SID used by each user. In practice, the size of the proof chain will be small compared to the database size. Alice incurs the cost of verification only if she wants to. The users can verify the transactions independently. There is a communication cost for Alice, as both Bob and Carol update Alice upon the commitment of a transaction. However, the proposed solution does not prevent Bob or Carol from executing further transactions before sending updates to Alice. Hence, transaction processing does not stall while updates are sent. Updates can be delayed as long as they arrive before verification is performed. Similarly, verification can be performed at any later time, and is not limited to immediately after transaction execution.

In the above protocol, Bob has to maintain *history*, which records the values read and written by all transactions, and the information necessary to generate the MBT structures for any transaction in the past. The size of this information can grow to be very large. If space is a problem, we can introduce *verification checkpoints*. A verification checkpoint corresponds to a statute of limitations for Alice – *i.e.*, we do

not allow Carol or Alice to verify any transaction beyond a certain time in the past (*e.g.*, a month). Thus they have up to one month to challenge any transaction. After this point, if the transaction is not challenged, Bob can assume that Alice accepts it and he can discard any data necessary to verify that transaction.

4.2.8 Analysis

We now analyze some of the overheads introduced by our protocol. We provide a treatment similar to that in [2] which introduced the MB-tree and serves as a base case for us since it provides a solution that meets the requirements of correctness and completeness, and also allows for centralized updates through Alice. It does not provide a solution that meets the Transactional Integrity requirement. In our solution, the user's cost of verifying an update or read is the same as that with an MB-tree, as users still verify an update or query against an MB-tree.

Authentication Structure Construction Time

For a database with n tuples and fanout f, the cost of construction of our data structures involves calculating hashes for each tuple and each node in the tree. Also, it requires the cost of writing those nodes to disk. For a tree of height d, the number of nodes in the tree will be:

$$m = \frac{f^d - 1}{f - 1} = O(n) \tag{4.1}$$

Hence, the cost of construction is :

$$nC_h + 2mC_h + 2mS_nC_{IO} + nS_tC_{IO} \tag{4.2}$$

where S_n and S_t are the sizes of a tree node and a tuple, respectively. C_h is the cost of computing a hash value, and C_{IO} is the IO cost for one block. Thus, like an MB-tree, the construction time is $O(n)$ – linear in the size of the database. This cost

exceeds that of an MB-tree by $mS_nC_{IO} + nS_tC_{IO}$, which represents the overhead of the history.

Update Time

To insert or delete a tuple from the tree, the path from the leaf node to the root has to be updated. Also, the updated values have to be added to the history. Hence, the cost of an update is

$$C_h + dC_h + 2dS_nC_{IO} \tag{4.3}$$

This is $O(\log n)$.

VO Construction Cost

To construct a VO, the server has to go through the history and find the values that it read while executing a transaction. If, C'_{IO} is the cost of finding the value of a tuple or node which the transaction read, then cost of VO construction is:

$$2dC'_{IO} \tag{4.4}$$

Since the height of the tree is d, the server has to find the rightmost path and the left most path to construct VO. C'_{IO} will increase as the transaction count increases, since it increases the history size hence search space.

Storage Overhead

Since our proposed data structure also stores the history of each tuple and tree node, the server keeps an extra copy of the tree and the relation. Thus, in the start we need twice as much disk space as that required by an MB-Tree. For each update,

the server keeps a copy of the updated data in history. Thus, after k updates, the storage cost is:

$$2nS_t + 2mS_n + kdS_n + kS_t \tag{4.5}$$

Thus the overhead for Bob is $O(n + klog(n))$ – *i.e.*, linear in the size of database and updates. On the other hand, Alice has to store the proof chain (one hash and one transaction ID per transaction) and the largest SID values for each user. This requires a disk space of:

$$(|h| + |tID| + |SID| + |User|)t \tag{4.6}$$

Thus the overhead for Alice is minimal and is linear in the number of transactions and users.

In the section next, we discuss implementation details and an empirical evaluation of the proposed solutions.

4.3 Secure Provenance

Many applications may also require, *e.g.*, due to regulatory compulsions, to keep the provenance of updates to the database. This can be particularly important to check if a malicious activity occurred in the past. Since the data is stored and edited at an untrusted server, ensuring the provenance is trustworthy is important. Our solution provides secure provenance of the database. The provenance can be produced at the database level, or at tuple level which shows how a particular tuple evolved over time. It can be verified that every change in the tuple value was reflected correctly in the tuple provenance.

The history also provides a secure provenance of the data. When a user asks for the provenance of a tuple, the server responds with a set of provenance records(history). Each provenance record has three components: the id of the transaction that created

that value; the version of the tuple; and the value of the tuple. Lemma 2 shows that the existence of any provenance record returned by the server can be verified. Further, Lemma 3 establishes that the completeness of the provenance records for a tuple (*i.e.*, no record that should have been in the provenance is missing), can be verified. Theorem 4.3.1 then proves that the data provenance of a tuple returned by the server is indeed trustworthy. If not, the user will be able to detect the error.

Lemma 2 *Any provenance record returned is indeed an authentic record.*

Proof A provenance record includes the transaction that modified the tuple and the new tuple value. The authenticity of a provenance record can be ensured by verifying the transaction that generated that tuple value. ■

Lemma 3 *Any provenance record that should have been returned is indeed in the provenance.*

Proof Since each tuple value is attached with its version, when the server returns the provenance of a tuple, $< T_{i1}, 0, v_0 >, < T_{i2}, 1, v_1 > \ldots < T_{ij}, j, v_j >$, the version numbers have to be contiguous, starting with 0. Any missing version number will mean incompleteness. To ensure that v_j is indeed the last version of the tuple, the user can verify that by asking the server to verify the authenticity of v_j against the latest proof (and not necessarily T_{ij}). ■

Theorem 4.3.1 *The provenance of a tuple given by the server is correct and complete. If not, the user can detect the error.*

Proof The theorem is a direct consequence of lemma 2 and lemma 3. ■

4.4 Proof-of-Concept Implementation

To demonstrate the feasibility and evaluate the efficiency of the proposed solution, we implement our protocols with the MB-Tree in Oracle. Our implementation is built

on top of Oracle without making any modifications to the internals. All our experiments are run with Snapshot Isolation. *Note that for the empirical evaluations, we chose operations which behave the same under Snapshot Isolation and Strict Serializable isolation level. Thus, our solutions are applicable for this study.* The protocols are implemented in the form of database procedures using Pl/SQL. While we expect that the ability to modify the database internals or to exploit the index system will lead to a much more efficient implementation, our current goal is to establish the feasibility of our approach and to demonstrate the ease with which our solution can be adopted for any generic DBMS. Users are implemented using python.

4.4.1 Setup and Implementation Details

We implement the MB-tree in the form of a database table. Each tuple in the MB-tree table represents a node in the tree. A better way to maintain the MB-Tree would be to use the B+ index trees of the database. However, that will require internal modifications to the index system of the database. We leave that for future work. Table *uTableMBT* stores the MB-tree for the data in *uTable*. Each MB-tree node is identified by a unique *id*. Each node stores keys in the range $[key\_min, key\_max)$. *level* denotes the height of the node from the leaf level, *i.e.*, leaf nodes have $level = 0$, and the root has the highest level. The *keys* field stores the keys of the node, and the *children* field stores the corresponding child ids and labels. Finally, *Label* stores the label of the node. This table is updated at the time of transaction commit.

Tables *uTableHistory* and *uTableMBTHistory* are used to store the history of the tables *uTable* and *uTableMBT* respectively. When a tuple is modified in *uTable* or *uTableMBT*, a new tuple is inserted in the corresponding history table to store current values. For example, when a new tuple is created in *uTable* by a transaction with transaction ID *tID*, an entry is added to *uTableHistory* with the value of the new tuple and transactionID as *tID*. At the time of a commit, the transaction modifies the MB-tree to update the proof. The updated node values are inserted into

Table 4.1.: Relations and Indexes in the Database

Table	**Attributes**	**Indexes**
uTable	TupleID, A, Version#	A
uTableHistory	TransactionID, TupleID, A, Version#	(TupleID, TransactionID)
uTableMBT	id, level, Label, keys, children, key_min, key_max	id, (key_min, key_max, level)
uTableMBTHistory	TransactionID, id, level, Label, keys, children, key_min, key_max	(TransactionID, id)
Transaction	id, query, finalLabel	id

uTableMBTHistory with transactionID *tID*. Updates to the MB-tree are made level by level, beginning at the leaves and working to the root. Once the root is updated, the transaction is committed.

Setup

We create a synthetic database with one table *uTable* containing one million tuples of application data. *uTable* is composed of a table with two attributes ($TupleID$ and A). The table is populated with synthetic data with random values of A between -10^7 and 10^7. All necessary structures for the protocol are maintained using other tables. Table 4.1 describes the different tables and indexes used in our prototype. An MB-tree is created on attribute A (integer). We consider three *replayable* transactions implemented as stored procedures, namely *Insert*, *Delete*, and *Select*. *Insert* creates a new tuple with a given value of attribute A. *Delete* deletes the tuples which have the given value of attribute A and *Select* is a range query over attribute A. In practice, transactions will be more complex than a single insert or delete. Our solution can handle complex transactions as well. However, for simplicity, we consider only simple transactions. The experiments were run on an Intel Xeon 2.4GHz machine with 12GB RAM and a 7200RPM disk with a transfer rate of 3Gb/s, running Oracle 11g on Linux. We run Oracle with Snapshot Isolation. As mentioned before, the

transactions considered for this evaluation behave similarly under Snapshot Isolation and Strict Serializable. Thus, our solutions are applicable here. We use a standard block size of 8KB.

4.4.2 Results

We now present the results of our experiments. To provide a base case for comparison, we compare the performance of our protocol with a regular MB-tree based protocol [2] where all updates are routed through the data owner. Furthermore, this solution does not provide indemnity for the server or secure provenance. We analyze the costs of construction for the authentication data structures, execution of a transaction, and verification of a transaction. We also study how our solution scales with multiple users concurrently running transactions.

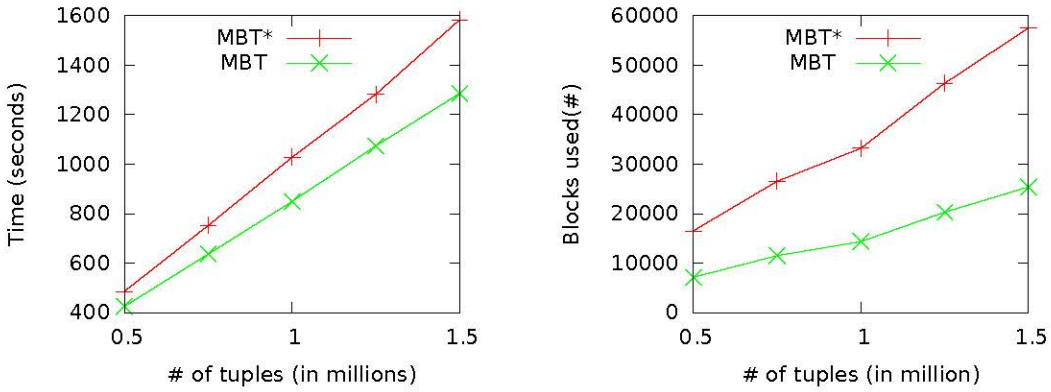
The fanout for the authentication structure is chosen so as to ensure that each tree node is contained within a single disk block. In each experiment, time is measured in seconds, IO is measured as the number of blocks read or written as reported by Oracle, and storage usage is measured in number of file blocks. The reported times and IO are the total time and IO for the entire workload. Each experiment was executed 3 times to reduce the error – average values are reported. In the plots, MBT represents the protocol from [2] where updates are always routed through Alice, and MBT^* represents our protocol where updates are sent directly from the users to the server. In experiments that measure the effect of multiple users concurrently running transactions, we keep the total number of transactions constant. We divide the workload equally among multiple users.

Construction Cost

First, we consider the overhead of constructing (bulk loading) the proposed data structure. For our solution, there is an extra cost for storing the history of the database, which increases the storage cost and construction time. Figures 4.3(a)

and 4.3(b) show the effect of data size on construction time and storage overhead, respectively. As predicted by our analysis (Eqns. 4.2 and 4.5), both costs increase linearly with the size of the database. In addition to the MB-tree, we maintain an extra copy of the MB-tree and the database table in the history files. Hence, our solution incurs a 100% storage overhead. The construction time has two components; time to compute hashes and time for IO (Eqn. 4.2). Our protocol needs twice the amount of IO as compared to MB-tree. However, the IO cost is superseded by hash computation. Hence, the construction time is not much higher than maintaining just an MB-tree.

In past work, the verification of a transaction was only allowed immediately after the execution of a transaction before any other transactions are executed. *Our work removes this restriction and enables the verification of past transactions. This provides much greater flexibility and reduces the need to immediately verify transactions.* Of course, the added functionality comes at an additional storage cost for the history tables.



(a) Time vs data size

(b) Blocks used vs data size

Figure 4.3.: Construction time and storage overhead

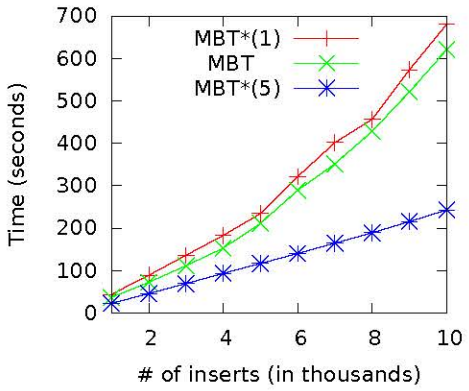
Insert Cost

We now discuss the cost of inserting and deleting tuples. Since both operations show similar costs, we only present the results for insertion due to lack of space. For this experiment no verification is performed. In the first experiment, we study the performance as the number of *Insert* transactions is increased. Figures 4.4(a), 4.4(b) and 4.4(c) show the results. As expected, with a single user, our protocol incurs a much higher overhead for storage and IO for maintaining the history information. These costs increase linearly with the number of transactions (Eqn. 4.5) Surprisingly, this does not translate into a significant increase in the running time. This represents the computational overhead of hashing and concatenations which dominate the cost (see Eqn. 4.3).

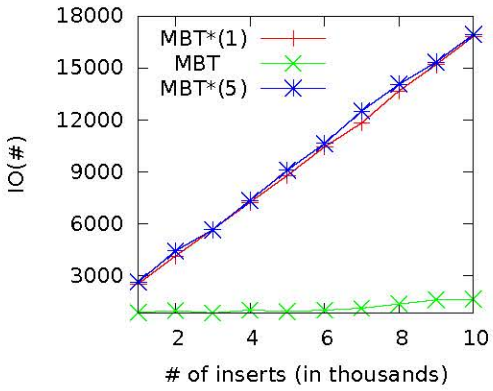
A key advantage of our protocol comes to light as we begin to increase the number of concurrent users, as seen in Figure 4.4(a) where the running time for our protocol drops significantly when 5 users run the same number of transactions in total, *i.e.*, each user runs one fifth of total number of transactions. In order to better study the impact of concurrent users, we ran another set of experiments where a varying number of users ran a total of 1000 *Insert* transactions. The results are shown in Figures 4.5(a) and 4.5(b). As we can see, the MB-tree solution which needs to process all updates through a single node (Alice) sees no gain in performance, whereas our solution results in improved performance with greater concurrency (even though it is performing a much larger amount of IO).

Verification Cost

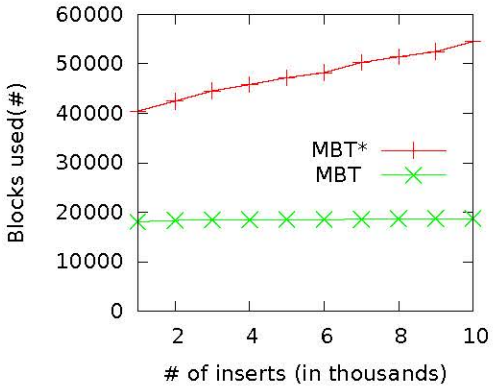
We now demonstrate the overhead of transaction verification on the system. We run 1000 *Insert* transactions with increasing fractions of transactions that are verified. The percentage of transactions that are verified reflects the data owner's distrust of the server. Figure 4.6(a) and 4.6(b) show the results. As the verification percentage increases, we observe that the execution time of the transactions increases. However,



(a) Time vs # of inserts



(b) IO vs # of inserts

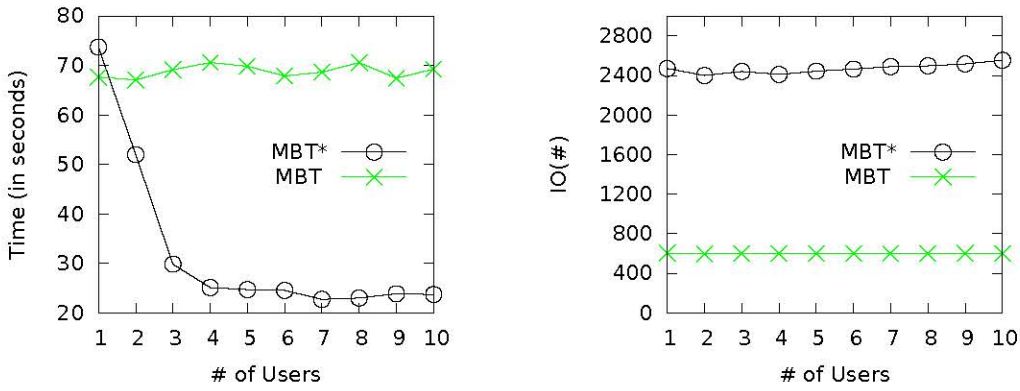


(c) Space used vs # of inserts

Figure 4.4.: Cost of insert

disk IO does not increase as rapidly since there is little extra IO for verification as compared to running the transaction itself. Verification also takes advantage of already cached MB-tree nodes.

Figure 4.6(c) shows the effect of the increase in number of users on execution time. For this experiment, we run 1000 *Insert* transactions and each transaction is verified. As before, our prototype scales much better to the increase in the number of users. This is because verification can be done independent of transaction execution, hence, verification executes while other transactions are running. Whereas for the case of just an MB-tree, the verification of a transaction prevents other transactions from running.



(a) Time vs # of users

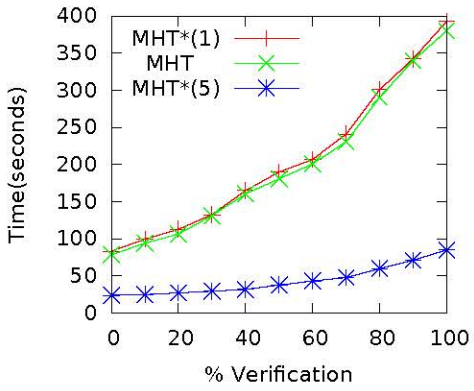
(b) IO vs # of users

Figure 4.5.: Cost of insert vs number of users

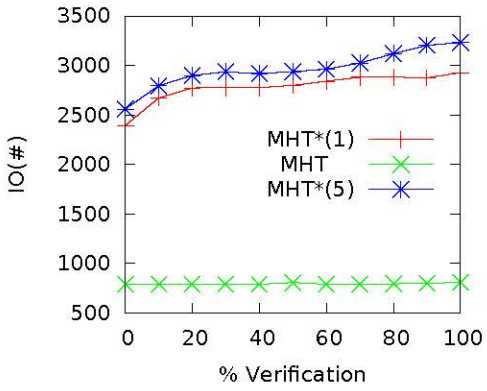
Search Cost

Now we evaluate the performance of our solution for range queries (*Search*). This cost is influenced by both the size of the result (larger results will be more expensive to verify) and the size of the history that needs to be searched for generating the proof. We conduct two experiments to study this behavior. In the first experiment, we run 1000 *Insert* transactions on the database to populate history. Then, we run 100 *Search* transactions with 100% verification for different ranges (thereby with different result set size). Figures 4.7(a) and 4.7(b) show the results. As the result set size increases, execution time and the amount of IO increase. For verification, the server has to return the right and left most paths of the range. Along with this, the server also has to return which tuples belong to which leaf nodes, as that is crucial information for the user to be able to verify the result set. Hence, as the result set size increases, the verification object size increases which results in an increase in verification time. The performance of our solution is comparable to that of an MB-tree alone.

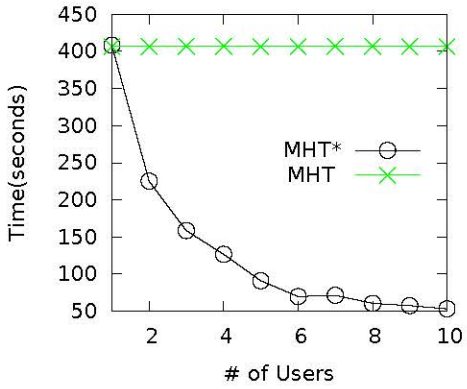
In the second experiment, we vary the history size by executing varying numbers of *Insert* transactions before running a fixed search and verification. Figures 4.8(a) and 4.8(b) show the effect of increase in history size. The x-axis in both graphs represents number of *Insert* transactions executed before running the search transactions. As



(a) Time vs verification %



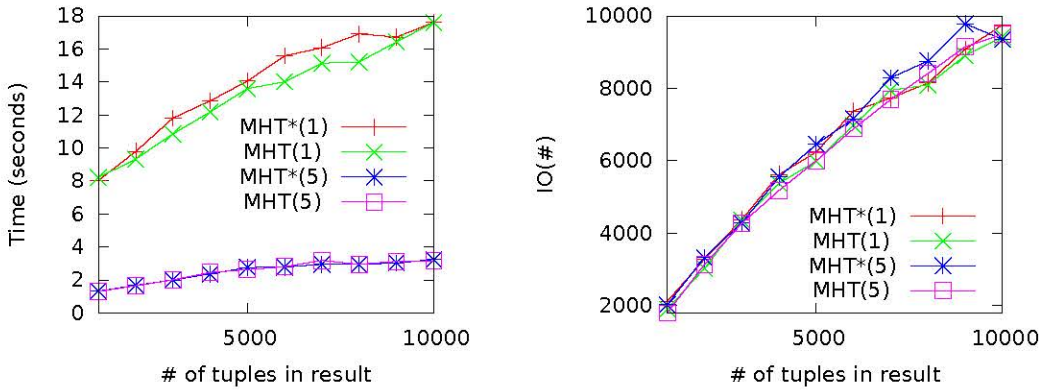
(b) IO vs verification %



(c) Time(100%verification) vs # of users

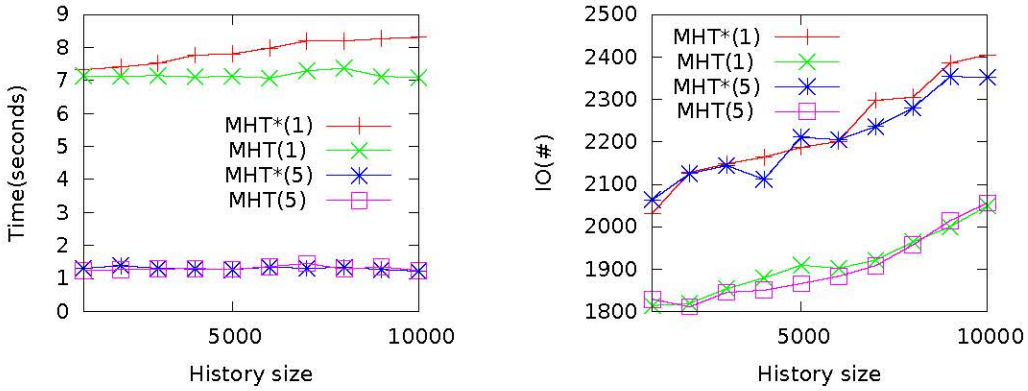
Figure 4.6.: Cost of insert and verification

expected, increase in the history size increases the verification time for our solution. However, it does not increase rapidly. Note that the search time without verification for our solution and MB-tree was the same, hence we do not report it separately. Overall, we see that the proposed ideas can be easily implemented on top of an existing DBMS. Even with this simple implementation, the overhead for ensuring transactional integrity is reasonable and actually less than the cost of the state of the art for ensuring only correctness and completeness.



(a) Time vs # of tuples (b) IO vs # of tuples

Figure 4.7.: Cost of search+verification vs number of tuples in the result



(a) Time vs # of tuples (b) IO vs # of tuples

Figure 4.8.: Cost of search+verification vs History size (number of inserts before doing search)

4.5 Chapter Summary

In this chapter, we introduced the problem of ensuring the authenticity and integrity of dynamic transactional database hosted on an untrusted server where the data owner may not have any direct control over the database. To the best of our knowledge, this problem has not been identified in earlier work. We develop novel solutions for this problem. Our protocol makes it possible to detect any failures on the part of the server to faithfully host a transactional database with multiple inde-

pendent users. Furthermore, the solutions also provide indemnity for the outsourcing server against false claims of erroneous processing, and provide assured provenance for the data managed by the untrusted server. These solutions are the first to address the problem of transactional integrity of an untrusted database. We demonstrate that the solutions can be implemented over an existing database system (Oracle) without making any changes to the internals of the DBMS. Our results show that we are able to remove the need to trust the server and provide support for independent users at a cost comparable to earlier work that does not provide either of these guarantees. We believe that the efficiency of the solutions can be further improved by modifying the internals and also developing proof structures that have better disk performance (*e.g.*, using GiST like indexes). We plan to explore these issues in future work.

The results of this chapter were published in [58, 60].

5 ACCESS CONTROL

Data privacy issues are becoming increasingly important. Many regulations now mandate responsible management of sensitive data, for example, Health Insurance Portability and Accountability Act (HIPAA). Database systems provide mechanisms to handle such requirements using access control policies. Access control mechanisms are an important part of a database system with which the data owner limits a user's access to a subset of the data. Traditionally, the server is assumed to be trustworthy and the data owner assumes that the access control policies will be faithfully enforced by the server. However, when the server is not trusted, or there is a threat of hacker attacks, the data owner cannot blindly assume that the access control policies will be honestly enforced by the server.

A naive approach to ensuring that only the authorized users read the data and no insider, attacker or the server can read the data, an encryption scheme could be used. Even though a simple encryption of the data before transferring it to the server ensures that only authorized entities who have the private key can access the data, it has many drawbacks. Encryption alone does not ensure that the retrieved query results are trustworthy (*e.g.*, retrieved values are the latest values and not stale). A simple encryption cannot enforce access control policies where each entity has access rights to only a certain part of the database. Another important issue is change in access control rules. In a typical setting, the database server enforces access control policies by rewriting user queries to limit access to the authorized subset. When the data owner wants to revoke or grant a user, access to a certain part of the data, the data owner does that by informing the server. Most solutions that provide mechanisms to enforce access control policies using encryption require the data to be re-encrypted when access control policies are changed. However, this is not desired as it incurs significant computation and communication cost. To avoid this, we use

Lazy Revocation Model, in which when the access control policies are changed, only the new data values are encrypted using newer set of keys. Previous values in the database are not re-encrypted. However, under this model, a subset of data could be encrypted using different keys, thus key management becomes a non-trivial task. We employ a scheme such that the user can desire previous keys used to encrypt the subset of the data using the latest key. Thus, a user needs only one key to be able to decrypt data in a particular subset.

In Chapter 4, we proposed solutions to ensuring that an untrusted database executed transactions honestly. However, these solutions require the server to reveal extra data from the database for the user to be able to verify trustworthiness of transaction execution. In the presence of access control rules, these solutions will not work as the extra information required for verification may not be accessible to the user. Also, a curious server or a hacker who manages to compromise the server could read the data stored in plain text.

In this chapter, we provide solutions that ensure that a data item in the database hosted on an untrusted server is read and modified only by authorized users, and none other (including the server). The data encrypted by our solution is still queriable. Our solution provides mechanisms to verify the trustworthiness of query results in the presence of access control rules. For this, we extend our previous work [58], as discussed in Chapter 4, on ensuring the trustworthiness of data retrieved from an untrusted database that can be modified by multiple entities. The contributions of this work are:

- A novel mechanism to enforce access control rules without trusting the server
- Solutions that allow users to verify the correctness and completeness of query results in the presence of access control rules
- A demonstration of the feasibility of the solution through a *prototype in Oracle*, and its evaluation

The rest of this chapter is organized as follows. Section 5.1 describes our assumptions and our model for this work. Section 5.2 presents our solutions. A discussion of the implementation of the solution and an empirical evaluation is presented in Section 5.3. Finally, Section 5.4 concludes this chapter.

5.1 Preliminaries

In this section, we start with explaining our model and assumptions. Then, we explain the problem of data leakage in Merkle B+ trees.

5.1.1 Assumptions and Model

We consider a similar model as presented in Chapter 4. The data owner, Alice, hosts the database on an untrusted database server, Bob. Authorized users, Carol, access this database using transactions. As in the previous model, Alice wants to ensure that the database server executes the user transactions faithfully, *i.e.*, (i) a transaction reads correct and complete data from the latest consistent state, (ii) the transaction is executed on this data to produce updates, (iii) updates produced by the transaction are applied to the database to produce the next consistent state. For this work, we focus on strict serializable isolation level. Apart from transactional integrity, Alice also wants to enforce access control policies to limit a user's access to only a part of the database. Alice also wants to ensure that the server, an insider and a hacker is not able to read the data. Alice and Carol want to ensure that the query results were indeed correct and complete in the presence of access control policies. We consider the following fine-grained access control policies.

Access Control Policy: We consider a fine-grained access control policy that exposes a user to only a subset of a database table. The subset that is accessible to a user is defined by query ranges (this is the approach adopted by some commercial systems like Oracle VPD as well). For example, in the example table, 3.1, the data

owner could define the access control policy which restricts a user to only those tuples which has attribute A value in the certain range. Table 5.1 shows one example. Consider User 2 in the example which is allowed to access only those tuples which has attribute A value in the ranges $[0, 39]$ or $[55, 64]$. A user query is rewritten by the server so as to restrict the user query to the subset of the data accessible to them. For example, for the user with the above restrictions, a query that asks for all tuples in the range $[20, 60]$, *i.e.* $\sigma_{20 \leq A \leq 60}$, should return tuples with tupleIDs $1, 2, 3, 4$, and, 5 in the absence of access control rules. However, with the access control rules, the query will be rewritten as tuples in the range $[0, 39]$ and $[55 - 64]$, *i.e.*, $\sigma_{(20 \leq A \leq 39) || (55 \leq A \leq 60)}$, which returns tuples with tupleIDs $1, 2, 3$, and, 5, thus ensuring that all and only those tuples that are in the range defined by the user query and also in ranges authorized for the user in the access control rules are returned as query result.

To ensure that the server or a hacker is not able to read the data, encryption of data is desired by the data owner. Encryption should disable Bob from being able to read the data. However, Alice and Carol should still be able to execute queries and run updates on the encrypted data. An acceptable solution should allow Alice to grant or revoke access to a user at any point in time, without much work. To make access control rules easily adaptable, we use Lazy Revocation Model as defined below.

Lazy Revocation Model: Under lazy revocation model, the data is encrypted such that only authorized users can read a particular data item. This is done by encrypting the data using a set of keys. Based on which part of the data is accessible to a user, the user given some keys so that the user could access (read or write) that part of the data and no other. If the user's access is revoked from that subset, the data are not re-encrypted immediately. Instead, the new values in that subset are encrypted with a new version of the key so that the evicted user can no longer read the new values in that subset. Since the user had access to the old data before eviction,

it can be assumed that the user had cached that data, hence it is not important to re-encrypt old values.

An acceptable solution should ensure that the users are able to run queries on the encrypted data while still being able to verify transactional integrity of their transactions in the presence of access control rules and encryption. Alice should be able to make changes to the access control policies at any point and the changes should be enforcible, *i.e.*, once the access control rules have been updated, the future transactions should follow the new rules, and the users should be able to verify that the rules are being followed. In our model, we assume that a user and the server will not collude. A user can be untrusted otherwise. In the case when a user is untrusted, the data owner should be able to verify the untrusted user's transaction by herself.

5.1.2 Data Leakage in MB-tree

For this work, we build upon MB-tree as described previously. To verify correctness and completeness of query result using MB-tree, the user receives Verification Object. The verification object includes some MB-tree node labels and two extra tuples with values just smaller and just higher than the query range in the sorted order. Consider our previous example table, Table 3.1. Figure 3.2 shows a sample MB-tree structure built on the attribute A of the table.

For a user query $\sigma_{30<A<60}$, the result would include t_3, t_4, and t_5. To verify the correctness and completeness of the query results, the server sends VO to the user which includes the tuples just preceding and just following the query ranges (*i.e.*, t_2 and t_6). The VO also includes any node labels that are required to compute the root label (*i.e.*, $h(t_1)$ and H_7). Using VO, the user can generate the $Proof$. If the computed proof matches with the proof value computed by Alice, the user is assured that the query results were correct and complete.

Table 5.1.: Sample Access Control Rules

User	Accessible Ranges
User 1	$[40 - 100]$
User 2	$[0 - 39]\&[55 - 64]$
User 3	$[55 - 100]$

5.2 Access Control

As mentioned before, access control rules allow the data owner to restrict a user's access to a certain part of the database. The database owner may also want to hide the data from the server as well using encryption, while still allowing the users to read and query the data. The difficulty introduced by using access control rules is two fold. Firstly, most existing verification mechanisms do not work in the presence of access control rules. In Subsection 5.2.1, we extend our solutions (as discussed in Chapter 4) so that a user could be assured that the partial database table visible to the user is indeed correct and complete. Secondly, the data have to be encrypted so the user sees only the allowed data and the data remain private from the server or an intruder. The server should still be able to run queries on this data. In Subsection 5.2.2, we propose an encryption scheme so that a data item can be accessed by only those users who have been authorized by the data owner.

As explained before, we consider a fine-grained access control policy. The access control policy is defined by exposing the user to a certain subset of the database using ranges. Table 5.1 shows one such policy defined for Table 3.1 for three users. We divide the domain of the attribute into disjoint ranges so that each range is either completely accessible by a user, or none of the tuples in the range is accessible by the user. In particular, we consider the following system for defining access control rules: $R = \{r_i \mid 0 \leq i \leq k\}$ is a set of ranges on an attribute that partitions the data into k disjoint subsets. Each user is allowed access (read and write) to a part of the database table defined by a subset of R, *i.e.*, the user, Carol, can access tuples $\{t_i \mid t_i \in \cup r_{i_l}\}$, where $\{r_{i_l}\} \subset R$ is the set of ranges accessible to Carol. Table 5.2 shows the disjoint

ranges for the example rules in Table 5.1. Access control rules are rewritten in terms of the disjoint range set. Table 5.3 shows the rewritten access control rules for Table 5.1.

5.2.1 Verification in Presence of Access Control

Given a range query, all tuples that satisfy the range query may not be accessible to the user. In such case, the verification using the regular MB-tree *VO* will not work. Also, verification of query results usually involves reading extra tuple values [2, 4]. These tuples may not be accessible to the verifier due to the access control rules. In such case, the verifier will not be able to verify a query or a transaction. Suitable adjustments to the authentication data structures are required to enable the verification of a query in the presence of access control rules.

To solve this problem, we modify the MB-tree as follows:

- Each node, n, is extended with an access control bitmap, B_n, in which the i^{th} bit is "on" if there is a tuple in the subtree that belongs to the range r_i. Leaf node access control bitmap has only one bit "on" corresponding to the range that the tuple it represents is in. For non-leaf nodes, the bitmap is computed as a binary OR of the bitmaps of the child nodes. Equation 5.1 presents it formally.

$$B_n = B_{child\_1}||\ldots||B_{child\_k} \tag{5.1}$$

- Node labels are computed by concatenating both the bitmaps and labels of the child nodes, as shown in Equation 5.2.

$$\Phi(n) = h(B_{child\_1}||\Phi(n_{child\_1})||\ldots||B_{child\_k}||\Phi(n_{child\_k})) \tag{5.2}$$

As an example, consider the access control ranges on attribute A shown in the Table 5.2. Under these access control ranges, each access control bitmap will have

Table 5.2.: Access Control Ranges

Label	Range
r_1	[0, 39]
r_2	[40, 54]
r_3	[55, 64]
r_4	[65, 100]

Table 5.3.: Updated Access Control Rules

User	Accessible Ranges
User 1	r_2, r_3, r_4
User 2	r_1, r_3
User 3	r_3, r_4

four bits, one for each access control range. Figure 5.1 shows an augmented MB-tree, as described above, built on Table 3.1. In the augmented MB-tree, B_2's value is 0011 as the subtree rooted at this node has tuples in the ranges r_1 and r_2, thus the first and the second bits in the access control bitmap is "on".

The *VO* now contains the nearest tuple value just preceding and the nearest tuple value just following the query range such that these tuples are accessible to the user. *VO* also contains all the tree nodes required to prove the correctness of these tuples and to prove that the tuples that were left out of the query results were indeed inaccessible to the user.

When User 2, who is authorized to access r_1 and r_4, executes a range query $\sigma_{25<A<50}$, tuple t_2 and t_3 will form the query result. Tuple t_4 will not be part of the query result as it is not accessible to the user. To verify the correctness and completeness of the query result, the user has to verify that the missing tuples were indeed inaccessible to her. The user will also have to verify that the nearest tuple just before the query range was indeed t_1 and the nearest tuple just following the query range (and also accessible to the user) is indeed t_7.

To prove the completeness of the query result, the *VO* of this query will include t_1 and t_7. To prove that the omitted tuples (*i.e.*, t_4, t_5, and t_6) were indeed inaccessible

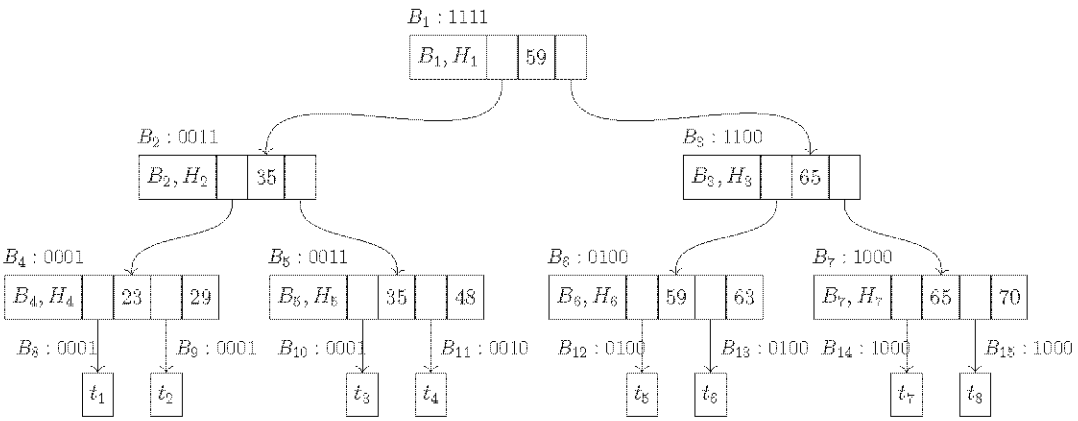


Figure 5.1.: Augmented MB-tree to allow access control

to the user, the VO will also include the bitmaps B_6 and B_{11}. Using B_6, the user can be convinced that tuples t_5, and, t_6 were indeed inaccessible to her. Similarly, using B_{11} the user can be assured that Tuple t_4 was inaccessible to her. As in the case of regular MB-tree, the VO will include all other necessary labels required to calculate the root label.

5.2.2 Enforcing Privacy for Access Control

In this subsection, we present our solutions to encrypt the database, so that a user can read/write only the subset of the data that she has been authorized to access. The server (or any intruder) cannot read the data. As mentioned before, in this work we consider the *lazy revocation model*. Under this model, once a range, r_i, is removed from a user's accessible ranges, the future tuples in r_i are encrypted using a new key. All remaining and future users who can access r_i will be distributed the new key. Any pre-existing tuples in r_i are not necessarily re-encrypted with the new key. To decrypt the data in the range r_i, the user may need the current or previous keys of that range. Only the data owner decides which ranges are accessible to the user.

The Key Regression scheme [53] provides a mechanism for versioning encryption keys for symmetric-key encryption. Given a version of the key, the user can compute

all previous versions of the key. However, future versions of the key can not be derived using the current key. In the start, all data items in an access control range are encrypted using the first version of the key. Each user authorized to access the range is given that key. When a user is evicted from the range, the key is updated to a newer version. All future data items in the range are now encrypted using the new version of the key. Since the users cannot generate the new version of the key, the evicted user cannot read future tuples in the range. A Key Regression scheme is defined using four algorithms. Algorithm *setup* is used by the data owner to setup the initial state. Algorithm *wind* is used to generate the next state. Algorithm *unwind* is used to derive the previous state, and *keyder* is used to generate the symmetric key for a given state. The tuples are encrypted using the symmetric key. We consider a particular key regression scheme that uses RSA to generate states.

Consider an RSA scheme with private key $< p, q, d >$, public key $< N, e >$, and security parameter k, such that p and q are two k-bit prime numbers, $N = pq$, and $ed = 1(mod\varphi(N)$ where $\varphi(N) = (p-1)(q-1)$. For each range r_i, a secret random number $S_i \in Z_N^*$ is selected as the initial state.

Algorithm *wind*, *unwind* and *keyder* are defined in Algorithms 4, 5, and, 6 respectively.

Algorithm 4 wind (N, e, d, S_i)

$nextS_i = S_i^d \ (mod\ N)$
return $nextS_i$

For each range, r_i, in range set R, the data owner generates a secret state $S_i \in Z_N^*$. The data owner sends the initial secret state of each state to all those users who are authorized to access the range. The user stores the current states for each range it has access to. Using the current state of a range, the user can compute the corresponding symmetric-key to encrypt or decrypt the data in that range. Whenever the data owner adds of removes a range from a user's accessible ranges, the data owner updates the state corresponding to that range to the next state and all users who still have access

Algorithm 5 unwind (N, e, S_i)

$prevS_i = S_i^e \ (mod\ N)$
return $prevS_i$

Algorithm 6 keyder(S_i)

$K_i = SHA1(S_i)$
return K_i

Table 5.4.: Bucketized Data Table Based on Table 3.1

tupleID	A^*	**Enc(A)**
1	[20-30)	$Enc_{K_1}(23)$
2	[20-30)	$Enc_{K_1}(29)$
3	[30-40)	$Enc_{K_1}(35)$
4	[40-50)	$Enc_{K_2}(48)$
5	[50-60)	$Enc_{K_3}(59)$
6	[60-70)	$Enc_{K_3}(63)$
7	[60-70)	$Enc_{K_4}(65)$
8	[70-80)	$Enc_{K_4}(70)$

to that range are informed about the new version of the state. If a tuple in a range is encrypted using a newer key, the user requests the new state from the data owner.

Due to encryption, the server cannot execute range queries. To be able to execute range queries on data, we use bucketization to divide the data into multiple buckets. Range queries are then suitably modified to search data among these bucket ranges.

Bucketization: Bucketization involves partitioning the attribute domain into multiple equi-width or equi-depth partitions. Attribute values are then converted from a specific value in the domain to the bucket labels. Table 5.4 is an example of equi-width bucketization of Table 3.1 where each partition width is 10. The bucket labels are ordered, *i.e.*, two values in different buckets can be compared. Using equi-width bucketization reveals the density in each bucket. Equi-depth, on the other hand, requires frequent adjustments (which requires communication with the user) when

database is updated frequently. [30, 61] show that only limited information can be deduced due to bucketization.

As shown in Table 5.4, the attribute A^* represents the bucket labels after bucketization, and the encrypted tuple value is kept in a separate attribute. User queries are rewritten to be executed on A^*. For example, a user query for tuples in the range $[25, 35]$ is rewritten to return tuples in buckets $[20 - 30)$ and $[30, 40)$. Note that the rewritten query increases the query range. The user receives the tuples and filters them so that only the tuples that satisfy the original query are present. To verify the correctness and completeness of query results, our augmented MB-tree can be built on top of the bucketized field, A^*. The verification process will remain the same, except now tuples will be inserted in the tree according to A^*. Since the attribute is encrypted, the access control ranges have to be defined in terms of A^*, rather than A.

Thus, combining the solutions proposed in Subsections 5.2.1 and 5.2.2, the data owner and the users can be convinced that the data were not maliciously modified, and the data were accessed by the user that had appropriate authorizations. In case the data owner is not interested in privacy against the server or a hacker, our solution in Subsection 5.2.1 can still be used alone to ensure that the transactional integrity can be ensured in the presence of access control rules.

5.3 Proof-of-Concept Implementation

To demonstrate the feasibility and evaluate the efficiency of the proposed solutions, we implement our solutions on top of Oracle. The solutions are implemented in the form of database procedures using PL/SQL and no internal modifications were done on the database. While we expect that the ability to modify the database internals or to exploit the index system will lead to a much more efficient implementation, our current goal is to establish the feasibility of our approach and to demonstrate

the ease with which our solution can be adopted for any generic DBMS. Users are implemented using Python.

5.3.1 Setup and Implementation Details

The MB-tree has been implemented in the form of a database table – each node in the MB-tree is represented by a tuple in the MB-tree table ($uTableMBT$). Ideally, the MB-tree should be maintained as a B+ index trees of the database. However, that requires internal modifications to the index system of the database. We leave that for future work. Each MB-tree node, identified by a unique id, stores $uTable$ tuples in the range $[key\_min, key\_max)$. $level$ denotes the height of the node from the leaf level, *i.e.*, leaf nodes have $level = 0$, and the root has the highest level. The $keys$ field stores the keys of the node, and the $children$ and $childLabels$ fields store the corresponding child ids and labels respectively. $Label$ stores the label of the node. When access control mechanisms are in place, two more attributes, $accessBitmap$ and $childAcessBitmaps$, are added to store the access control bitmap of the node and access control bitmaps of the child nodes respectively.

We create a synthetic database with one table $uTable$ containing one million tuples of application data. $uTable$ is composed of a table with two attributes ($TupleID$ and A). The table is populated with random values of A between -10^7 and 10^7. When tuples are encrypted, the ciphertext is stored in attribute $EncA$. Table 5.5 describes the different tables and indexes used in our prototype. An MB-tree is created on attribute A (integer). We consider three transactions implemented as stored procedures, namely *Insert*, *Delete*, and *Select*. *Insert* creates a new tuple with a given value of attribute A. *Delete* deletes the tuples which have a given value of attribute A and *Select* is a range query over attribute A. The experiments were run on an Intel Xeon 2.4GHz machine with 12GB RAM and a 7200RPM disk with a transfer rate of 3Gb/s, running Oracle 11g on Linux. We run Oracle with a strict serializable isolation level. We use a standard block size of 8KB.

Table 5.5.: Relations and Indexes in the Database

Table	**Attributes**	**Indexes**
uTable	TupleID, A, EncA[2], keyVersion[2]	A
uTableMBT	id, level, Label, keys, children, childLabels, key_min, key_max, accessBitmap[1], childAccess-Bitmap[1]	id, (key_min, key_max, level)
AccessControlRanges	id, key_min, key_max	
AccessControlRules	AcessControlRule_id, Use_id	

5.3.2 Results

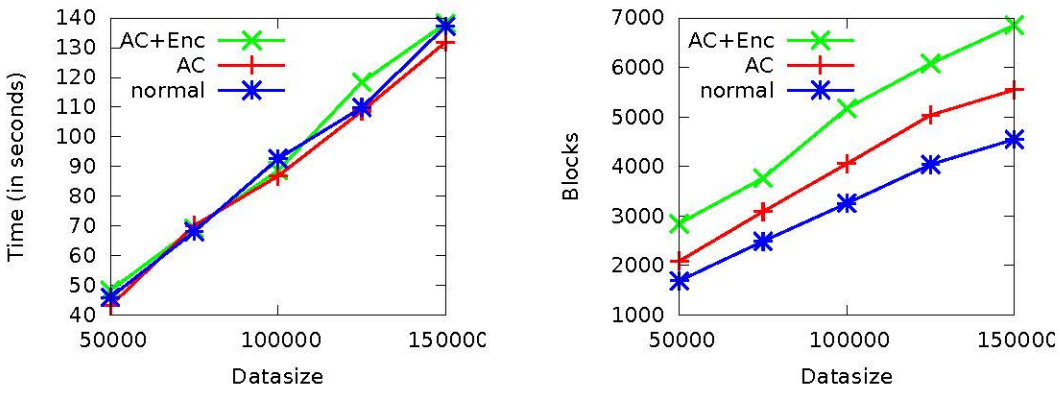
We now present the results of our experiments. To provide a base case for comparison, we compare the performance of our solutions with a regular MB-tree based solution [2], where access control rules are not supported. This solution leaks information for transaction verification. Furthermore, this solution does not provide privacy against a malicious server. We analyze the costs of construction for the authentication data structures, execution of a transaction, and verification of a transaction.

The fanout for the authentication structure is chosen so as to ensure that each tree node is contained within a single disk block. In each experiment, time is measured in seconds, storage and IO is measured as the number of blocks read or written as reported by Oracle.The reported times and IO are the total time and IO for the entire workload. Each experiment was executed 3 times to reduce the error – average values are reported. In the plots, *Normal* represents the solution from [2], AC represents our solution where access control bitmaps are added to the nodes to support access control rules, and $AC + Enc$ represents our solution that encrypts the tuple values and uses bucketization. AC and $AC + Enc$ both allow query verification in the presence of access control rules. $AC + Enc$ also provides privacy against the server or an intruder. When bucketization is used, we divide the data into 1000 buckets. We use 200 access control ranges.

Construction Cost

First, we consider the overhead of constructing (bulk loading) the proposed data structures. To support access control rules, our solution requires augmenting MB-tree nodes with additional values that store the access control bitmaps. To provide privacy from the server, key regression is used that allows different versions of the encryption key. This requires storing additional attributes to store the ciphertext and

[1]Used when supporting Access Controls
[2]Used when supporting Access Controls with Encryption



(a) Time vs data size

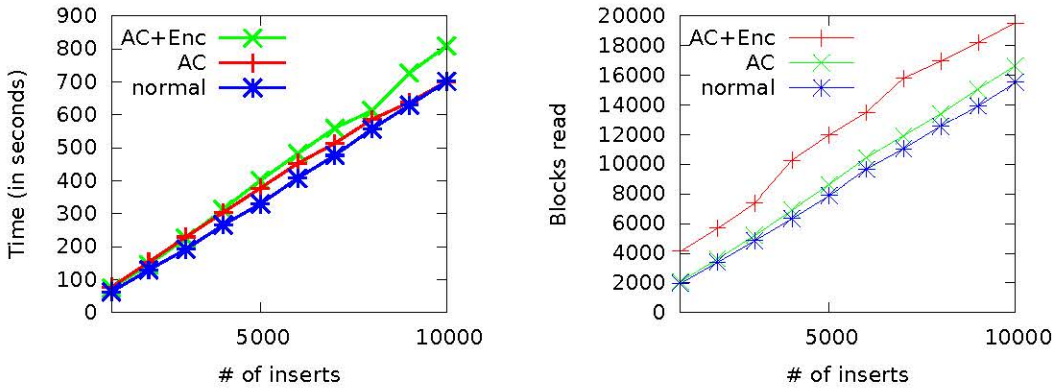
(b) Blocks used vs data size

Figure 5.2.: Construction time and storage overhead

the key version. Figures 5.2(a) and 5.2(b) show the effect of data size on construction time and storage overhead, respectively. As expected, the storage cost is higher for our solutions. However, the construction time does not change significantly as the additional computation required for encryption is done by the user, keeping the computation cost for the server similar to just maintaining the MB-tree.

Insert Cost

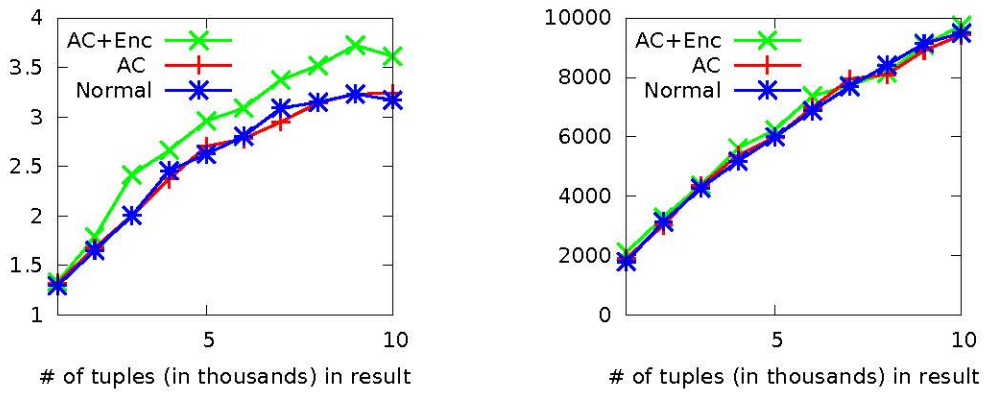
We study the performance as the number of *Insert* transactions is increased. For this experiment no verification is performed. Figures 5.3(a) and 5.3(b) show the results. As expected, our solution incurs a higher overhead for IO as it requires keeping additional data. These costs increase linearly with the number of transactions. Surprisingly, this does not translate into a significant increase in the running time. This represents the computational overhead of hashing and concatenations which dominates the cost. Delete operation shows similar costs (not presented due to lack of space).



(a) Time vs # of inserts

(b) I/O vs # of inserts

Figure 5.3.: Insert time and I/O overhead



(a) Time vs # of tuples

(b) IO vs # of tuples

Figure 5.4.: Cost of search+verification vs number of tuples in the result

Search Cost

Search cost is influenced by both the size of the result (larger results will be more expensive to verify) and number of access control rules as that requires verifying that the tuples that were dropped from the query result were indeed not accessible to the user. To evaluate the performance of our solution for range queries (*Search*), we run 100 *Search* transactions for different ranges (thereby with different result set size) and verify all transactions. Figures 5.4(a) and 5.4(b) show the results. As the result set size increases, execution time and the amount of IO increase. For the regular MB-tree solution, the query range is divided into multiple sub-ranges based on access control rules. Each sub-range that is accessible to the user is returned as query result. For verification, the server has to return the right and left most paths of each sub-range. However, in our solution, an access control bitmap is enough to verify that the sub-range is not accessible. This decreases the *VO* size and computation cost. As shown in the figure 5.4(a), our solution performs slightly better than MB-tree as our solution requires lesser *VO* size. As the result set size increases, the verification object size increases which results in an increase in verification time. The performance of our solution is comparable to that of an MB-tree alone.

Verification Cost

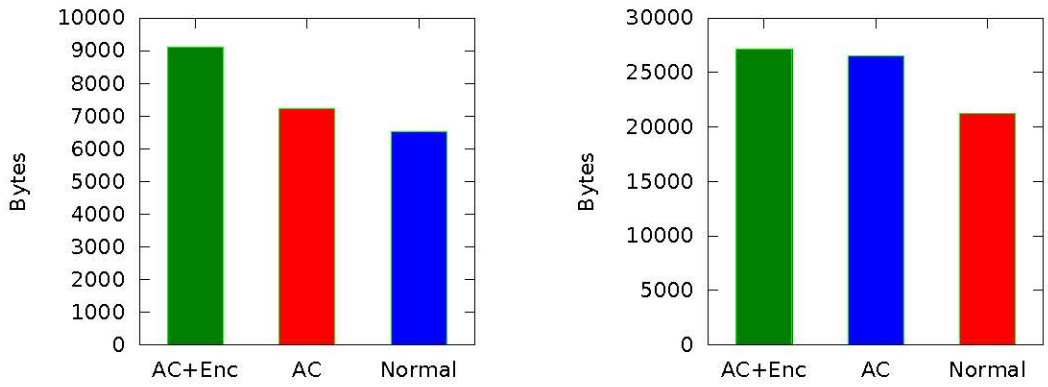
Our solution changes the *Verification Object* significant as our solution does not require bordering tuples outside the accessible range. However, since the node labels now include access control bitmaps, it increases the VO size. We now demonstrate the change in VO size in our solutions. To demonstrate the overhead of insert verification, we run 1000 *Insert* transactions and verify them. Average VO size is reported in figure 5.5(a). As expected, the VO size is higher for our solutions as it requires additional information, like access control bitmaps and key versions.

To demonstrate the overhead of search query verification on the system, we run 1000 *Search* transactions with varying ranges and varying access control ranges. The average VO size is reported in Figure 5.5(b). As discussed before, in a normal MB-tree, to support access control, a query range has to be divided into multiple sub-ranges so that the query accesses only the part of the data that are accessible to the user. For each sub-range, the VO includes the tuple just before and just following the sub-range. VO also includes all necessary nodes that are required to verify that the bordering tuples indeed existed the database. However, in our solution, this is not necessary. Each node contains information if the descendant tuples are accessible or not. Hence, VO does not always require the bordering tuples. Figure 5.5(b), that shows the effects of our solution on the VO size validates this. VO size for AC is smaller than the normal MB-tree. VO size for $AC + Enc$ is comparable to MB-tree. This is due to the ciphertexts.

Overall, we observe that our solutions are efficient and provide mechanisms for access control with reasonable overheads and perform better than current solutions in some cases.

5.4 Chapter Summary

In this chapter, we considered the problem of implementing access control policies on an untrusted database server, while ensuring that the query results are trustworthy.



(a) VO Size for insert

(b) VO Size for search

Figure 5.5.: Verification object size

With our solution, the data owner can be assured, without having to blindly trust the server, that the data will be read and modified by only those users that were authorized by her apriori. Furthermore, the data owner and the users can be assured of the trustworthiness of the query results without violating the access control policies. We demonstrate that the solutions can be implemented over an existing database system (Oracle) without making any changes to the internals of the DBMS. Our results show that the solutions do not incur heavy costs and are comparable to current solutions for query verification (that do not support access control rules).

The results of this chapter were published in [62].

6 ISOLATION LEVELS

In Chapter 4, we presented solutions to the problem of ensuring the authenticity and integrity of transactions executed on a dynamic relational database hosted on an untrusted server. These solutions assume that the transactions will be executed at strict serializable isolation level. However, many applications prefer to use weaker isolation levels for better concurrency. Many object databases and NoSQL databases exploit weaker isolation levels for better performance as well. Weaker isolation levels increase transaction throughput, however risk showing transactions a fuzzy or inconsistent database state. Most database systems allow users to pick isolation levels on a per transaction basic, *i.e.*, some transactions can execute at the strictest isolation level while other concurrent transactions run at weaker isolation levels.

The semantics for different isolation levels have been defined with the assumption of trust on the server. These definitions describe how the database systems should be built. Some isolation levels risk showing inconsistent database state, for example when transactions allow *Dirty Read*. When *Dirty Reads* are allowed, a transaction can read uncommitted data created by another concurrent transaction which may be temporarily in an inconsistent state. Even when *Dirty Read* is not allowed, two statements in a transaction can read from different consistent states, leading to a "fuzzy" view of the database. Considering that the weaker isolation levels risk showing a "fuzzy" database state, from the user's point of view, it is not clear what they can expect. In this chapter, we revisit the semantics for different isolation levels without assuming that the server is trusted, and clarify their definitions in this setting in terms of what users should expect when a transaction is executed under various isolation levels. We also design mechanisms that allow a user to verify that the transactions were executed faithfully in accordance with the chosen isolation levels.

In particular, we present solutions that require an untrusted database server to prove the trustworthiness of its data and query results. This is done by engaging the server in a cryptographic protocol that forces the server to reveal some key aspects of the execution of each transaction. Once revealed, the database server has no way to go back and claim a different execution scenario. A key challenge for this work arises from the fact that different isolation levels risk showing the transactions an inconsistent database state. Another key challenge is that multiple, independent users can read and modify parts of the data concurrently. In order to ensure trustworthiness of a transaction execution under a given isolation level, we need to ensure that the transaction read correct, and complete data from "valid" consistent states. What constitutes a valid consistent state depends on the isolation level. We also need to ensure that the transaction was executed honestly on this data to produce its outcomes and the updates produced by the transaction were applied correctly to the database so that subsequent transactions could see them.

To the best of our knowledge, this problem of ensuring transactional integrity over an untrusted database server under isolation levels other than strict serializable has not been addressed in earlier work. In particular, the contributions of this work are:

- Definition of isolation levels in terms of verifying faithful execution of transactions under a given isolation level without the assumption of trust of the server.
- Mechanisms that ensure trustworthy execution of transactions on the database in the presence of different flavors of isolation levels.
- A demonstration of the feasibility of our solutions through a prototype implementation in Oracle and its evaluation.

The rest of this chapter is organized as follows. Section 6.1 discusses our model and assumptions. Section 6.2 presents our solutions. A discussion of the implementation of the solution and empirical evaluation is presented in Section 6.3. Finally, Section 6.4 summarizes the chapter.

6.1 Assumptions and Model

For this work, we relax the model in Chapter 4 further to support isolation levels weaker than strict serializable. Similar to the models used in the previous chapters, the database *owner*, **Alice**, authorizes user(s), **Carol**, to access the data from the *untrusted* server, **Bob**. Users can independently authenticate themselves with the server. A user can read or write data to the database using transactions.

Alice and Carol want to ensure that the user transactions were executed faithfully, – *i.e.*, the transaction execution followed ACID semantics. For each transaction, the user chooses the isolation level that the database server should follow. Multiple concurrent transactions can run at different isolation levels. For this work, we consider the following commonly used isolation levels: Strict Serializable, Snapshot, Repeatable Read, and Read Committed. We do not allow *Dirty Write*, as that can lead to an inconsistent state[1].

6.2 Isolation Levels

As discussed before, we can use MB-trees to ensure the correctness and completeness of the query results on a static database. We further discussed our previous solution which provides mechanisms to verify transactional integrity of a relational database under strict serializable isolation level. In this section, we re-examine the semantics of different types of isolation levels without the assumption of trust on the database server, and the implications for verifying faithful execution under these isolation levels.

As in [58], we consider only *Replayable Transactions*. A replayable transaction is one whose output is determined solely by the transaction parameters and the data it reads from the database. In other words, there is no randomness in the transaction,

[1]For example, if the database has a *unique key* constraint on *id*, and two transactions T1 and T2 execute actions on tuples A, B as follows: (1) T1 writes A.id = 1; (2) T2 writes B.id = 1; (3) T2 writes A.id = 2; (4) T2 commits; (5) T1 writes B.id = 2; (6) T1 commits;. In this case, the unique key constraint is broken, and the database will be in an inconsistent state.

or dependence on anything except the input parameters and values read from the database.

The ANSI/ISO SQL standard [63] defines isolation levels in terms of three phenomena that may or may not be permitted for a given isolation level. These phenomena are: *Dirty Read*, *Non-repeatable Read*, and *Phantom Read*. Most definitions and implementations of the isolation levels are defined around these terms. For this work, we do not allow *Dirty reads* – *i.e.*, a transaction reads only those data items that are committed and no uncommitted data created by others transactions is visible. As mentioned before, we also do not allow *Dirty Writes* as that can lead to an inconsistent state.

6.2.1 Main Ideas

Different isolation levels can lead to different possible outcomes for the same transaction. However, when *Dirty Writes* are not allowed, databases ensure that the following simple definition of consistency is satisfied. A database goes through different consistent states as the transactions are executed. Each consistent state is produced by a single transaction. Even though multiple transactions are running in parallel, when these transactions commit, it appears as if one transaction committed after the other producing a sequence of consistent states. When a transaction is executed, it reads data from one or more of these consistent states based on the isolation level. Figure 6.1 shows this graphically. $r_i(a)$ represents that transaction T_i read tuple a from the database. $w_i(a)$ represents that the transaction T_i updated tuple a. Notice that this is different than strict serializability. Strict serializability requires that it should appear as if one transaction committed after the other, but also that the transaction reads data from the previous consistent state – it appears as if all data read by the transaction came from one consistent state, and when the transaction committed, it produced the next consistent state. The initial state of the database (DB_0) is considered to be a consistent state. The database can go through multiple

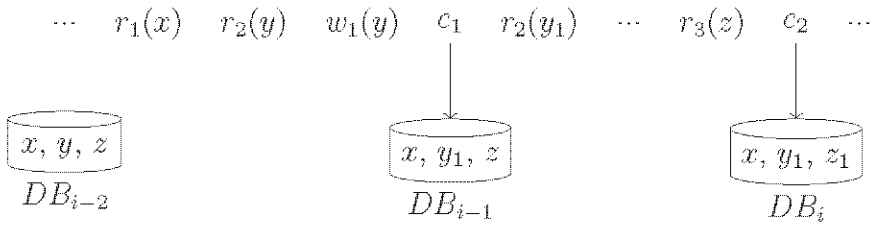


Figure 6.1.: A conceptual view of database consistent stats

consistent states during the execution of a transaction (as other concurrent transactions commit). For example, in Figure 6.1, while transaction T_2 was being executed, other transactions committed (T_1). Also, a transaction can see multiple consistent states depending upon the chosen isolation level. Only the correct (isolated, atomic) execution of a transaction (T_i) takes the database to a new consistent state(DB_i) when allowed to commit. Of course, this is only a conceptual notion – in reality multiple transactions execute concurrently. Thus, in practice, the database is in an inconsistent state represented by the partial execution of concurrent transactions. However, when a transaction is allowed to commit it is certain that its execution is equivalent to each transaction committing and producing the consistent state in the order of commits.

Even though it is not stated directly, there is an implicit assumption that the transaction will read *fresh* data. *Freshness* of data means that the transaction reads the latest data rather than some stale data which has been modified/deleted. However, the exact conditions to establish the freshness of data depend upon the given isolation level. Since the transactions are executed in an untrusted environment, we need to reexamine each isolation level and define the freshness conditions for different isolation levels.

Based on these observations, establishing the trustworthiness of a transaction execution requires the following conditions:

- Each statement inside the transaction reads committed and *fresh* data
- The transaction is executed using these data to produce the updates

- The updates produced by the transaction are applied to the database to produce the next consistent state

To ensure that the above conditions were satisfied, we use two key ideas to develop our solutions. These ideas are explained below.

Consistent State

As mentioned before, to verify that a transaction was executed faithfully under the chosen isolation level, we need to verify that (1) all data read by the transaction came from "valid" consistent states (2) all data that should have been read by the transaction was indeed read by the transaction and (3) all updates produced by the transaction were applied correctly to produce a new consistent state. We propose to represent the consistent states by the root label of the MB-tree. Although the database can be in flux at any time, it is assured that when a transaction is allowed to commit, it appears to have committed in an isolated manner, *i.e.*, conceptually, the database evolves from one consistent state to another and each consistent state is produced by a single transaction. We maintain a one-to-one mapping between the consistent states and the MB-tree representing that state. To achieve this, we update the MB-tree at the time of transaction commitment. Thus, each new MB-tree root label represents a consistent state produced by a single transaction.

History

To verify the correctness, completeness, and freshness of data read by the transaction, we need to know from which consistent state a particular data item was read. For some isolation levels, where the transaction can read data from multiple consistent states, we also need to know if the data item was modified between two consistent state. For example, under the *repeatable read* isolation level, we need to verify that if two statements inside the transaction read the same data item, it must not be modified between the execution of these two statements. Similarly, a data item that

is to be updated under the snapshot isolation must not have been updated between the start and the commit of the transaction. To achieve this, each tuple and each MB-tree node is assigned a unique id. Each tuple (or object, in the case of object databases) in the database is also assigned a version number. The version number keeps track of how many times the tuple has been modified since it was first created.

To verify the execution of a transaction, the verifier would need to verify the data at previously consistent states, so we keep a history of the tuples and the MB-tree node values as well. Whenever a tuple of the MB-tree node value is changed, the new version is stored in the history. This way, when a user wants to verify a transaction, the database server can look at the history and compute the data necessary to verify the authenticity of the data read or written by the transaction.

As discussed above, the definitions for correctness and completeness of data read by a transaction depend upon the chosen isolation level. In the next subsections, we define the correctness and completeness of data for a given isolation level in terms of consistent states and history.

6.2.2 Strict Serializable

This is the strictest isolation level. Under this isolation level, even when multiple transactions execute concurrently, when the transactions are allowed to commit, it is ensured that they are equivalent to a serial execution. Figure 6.2 shows this graphically. Read operatation of transaction T_2, $r_2(y)$, in Figure 6.2 reads some data before c_1. However, the database ensures that it would appear as if $r_2(y)$ was exected against DB_{i-1} (*i.e.*, the consistent state produced by c_1). In more formal terms, the initial state (DB_0) is considered to be a consistent state. The correct execution of a transaction (T_i) takes the database state from the previous consistent state (DB_{i-1}) to the next consistent state (DB_i). All reads for T_i come from the previous consistent state DB_{i-1}, and all updates that the transaction produces are applied atomically

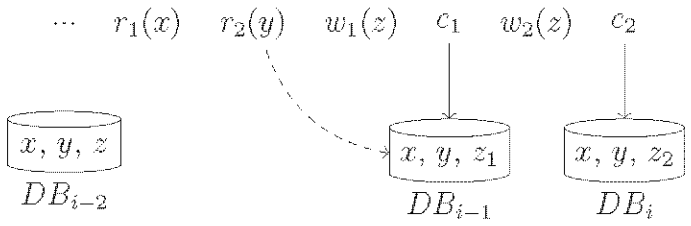


Figure 6.2.: An example of strict serializable isolation level

to produce the next consistent state DB_i. Based on these ideas, the verification of transactional integrity can be divided into the following parts:

- Verify that the data read by the transaction came from a single consistent state (DB_i) and were correct and complete
- Verify that the transaction was executed faithfully using this data to produce updates
- Verify that the updates produced by the transaction were applied to produce the next consistent state, DB_{i+1}

If we can verify these three parts, we can be assured that the transaction was faithfully executed at the strict serializable isolation level.

6.2.3 Snapshot

Even though many database systems (*e.g.*, Oracle) call this form of isolation level "Serializable", it is a weaker isolation level than the Strict Serializable isolation level. At this isolation level, a transaction reads from a single consistent snapshot of the database. Two transactions can execute against the same snapshot. However, when the transaction is allowed to commit, it is ensured that no two concurrent transactions wrote to the same data item (*i.e.*, *Dirty Writes* are not allowed). Figure 6.3 shows this graphically. Even though c_1 is executed before $r_2(y, z)$, and DB_{i-1} is the freshest

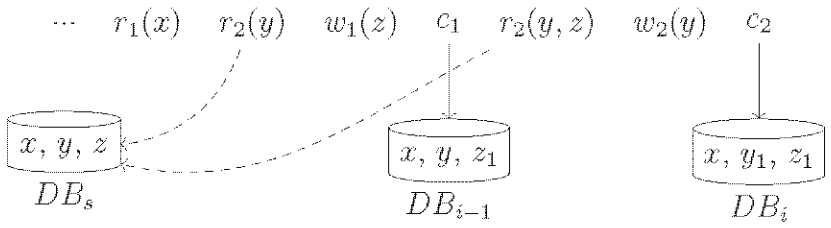


Figure 6.3.: An example of snapshot isolation level

state, $r_2(y, z)$ reads from the previous consistent state, DB_{i-2}, from which $r_2(y)$ read data.

Formally, a transaction T_i is executed against a consistent state DB_s. If the transaction is allowed to commit, its updates are applied to produce the new consistent state DB_i. Under the strict serialization isolation level, we would have $s = i - 1$. This is not necessary under the snapshot isolation level. However, T_i cannot update any data item that was created by those transactions that produced the consistent states between DB_s and DB_i (excluding DB_s and DB_i), as that would require a dirty write.

For example, in Figure 6.3, 2_2 starts at DB_{i-1}, and modifies y to y_1. Since no other concurrent transaction was modifying y, this is acceptable. If T_2 was to modify z, after c_1, that is not allowed.

Based on this explanation, to verify that a transaction followed snapshot isolation, we need to verify the following:

- The data read by the transaction came from a unique consistent state DB_s
- The transaction was executed faithfully using this data to produce updates
- All updates produced by the transaction were applied on DB_{i-1} to produce the consistent state DB_i
- The data items modified by the transaction were not updated in consistent states between DB_s and DB_i (excluding DB_s and DB_i)

To verify that the data items modified by the transaction were not updated before, we use the version number of the data items. For any data that was modified by the transaction, we ensure that its version number in DB_{i-1} is same as its version in DB_s.

6.2.4 Repeatable Read

Repeatable read requires all data items that have been read by the transaction should have the same values if the transaction reads those data items again. However, *phantom* data are allowed. Consider Figure 6.4. When transaction T_2 executes $r_2(y)$, it reads from a consistent state DB_{i-2}. However, its next read $r_2(y, z)$ reads from a newer consistent state DB_{i-1}, and reads value of z as z_1. Repeatable read ensures that a data item that has been part of a transaction read, should not be modified until the transaction commits. Formally, we need to verify the following to ensure that the transaction followed repeatable read:

- The data read by a statement in the transaction came from a unique consistent state DB_j
- The transaction was executed faithfully using this data to produce updates
- All updates produced by the transaction were applied to produce a consistent state DB_i
- The data items read by the transaction were not modified by another transaction between the consistent states it was read from and DB_i

To verify the fourth condition, we use version of the data items at the time when it was read and at the consistent state produced by the transaction commit. This step is similar to that in snapshot isolation level, except that the initial consistent state for the data item is not the initial snapshot but the first consistent state from which the data item was read.

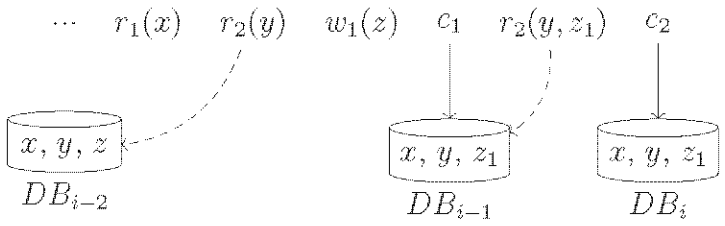


Figure 6.4.: An example of repeatable read isolation level

6.2.5 Read Committed

This is a weaker isolation level than the other isolation levels discussed before. The read committed isolation level defines the behavior at the statement level rather than the transaction level. The read committed isolation level requires that each statement inside the transaction should read only committed data. However, different implementations vary in specifics about whether all data read by a statement comes from a single consistent state or not. We consider both these forms here.

Strict Form

This is the most common form of read committed isolation level among popular relational databases that use multi-version concurrency control. Under this isolation level, each statement inside the transaction reads data from a single consistent state. Two statements in the same transaction can read from different consistent states. Figure 6.5 shows an example. Transaction T_2 issues two $r_2(y)$ statements, and they execute against two different consistent states DB_{i-2} and DB_{i-1} respectively. However, it is ensured that each statement is executed against a consistent state that is at least as fresh as the previously seen state by the transaction. In Figure 6.5, the second $r_2(y)$ reads from DB_{i-1} which is fresher than DB_{i-2}.

To verify the faithful execution of a transaction in this isolation level, the verifier needs to do the following:

- For each statement inside the transaction:

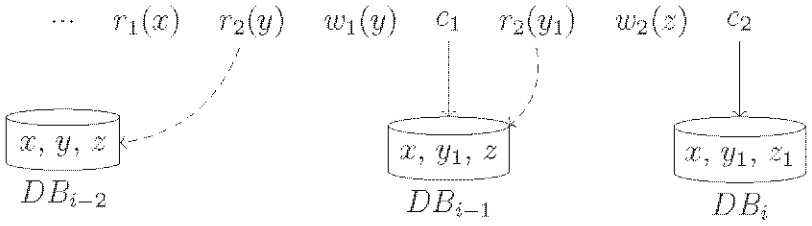


Figure 6.5.: An example of stricter read committed isolation level

  - Verify that the consistent state against which this statement was executed is not older than the previous consistent state seen by the transaction
  - Verify that all the data were read from the consistent state as reported by the server

- The transaction was executed faithfully using the data read by the statements to produce updates
- Verify that all updates produced by this transaction were applied to produce the next consistent state

Weaker Form

Some databases use a weaker semantics for Read Committed isolation level. Even though the database server is supposed to ensure that the transaction reads only committed data items, it does not guarantee that all the data items read by a statement come from a single consistent state. Many database systems, particularly NoSQL and object oriented databases like DB4O use this form of isolation level.

A naive approach for verification could be to just confirm that each data item read existed in at least one consistent state. However, this does not guarantee freshness of data which is an important part of faithful execution of transactions. Consider Figure 6.6 as an example. T_2 statement $r_2(x_1, y, z_1)$ should not read x_0 from DB_{i-3} even though it is a committed value. This is because a previous statement of T_2 has

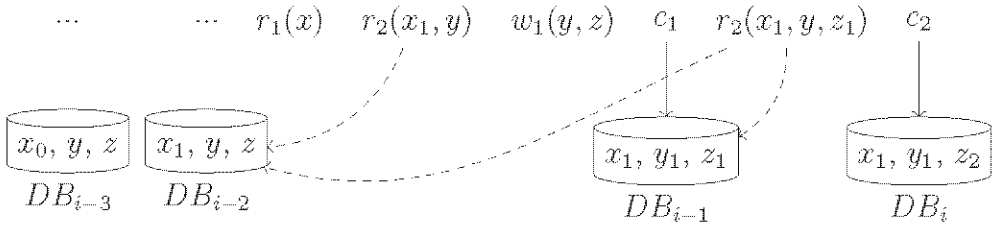


Figure 6.6.: An example of weaker read committed isolation level

already seen DB_{i-2}. Thus, all future statements must see either DB_{i-2} or newer states. $r_2(x, y, z)$ can see either y or y_1, and z or z_1. However, it must see x_1, as x_1 is present is all possible consistent states from which $r_2(x, y, z)$ could read. Formally, if DB_s is the state of the database when the transaction started, DB_i is the consistent state it produces when allowed to commit, and DB_{ri} is the freshest state seen by the statement R_i, then $ri \leq rj$ for $i < j$, *i.e.*, each statement should read data from a state which exists on the last seen state or after that.

In particular, verification involves the following steps:

- For each statement inside the transaction:
  - Verify that all data items read by the statement existed on and after the previous consistent state seen by the transaction
  - Verify that all data items that were part of all consistent states between the previous consistent state and the most recent consistent state were read by the statement
- The transaction was executed faithfully using the data read the statements to produce updates
- Verify that all updates produced by this transaction were applied to produce the next consistent state

6.2.6 The Protocol

We now discuss our solution for ensuring Transactional Integrity under a given isolation level. Our solutions allow multiple transactions to run concurrently under different isolation levels. We will discuss the initial steps that Alice (data owner) and Bob (server) take to setup the database server. Then we discuss the protocol which engages the user (Carol) and Bob so as to run Carol's transaction. With this protocol, Bob commits to the "environment" under which Carol's transaction was executed – *i.e.*, the consistent states that the transaction read from or committed to. Finally, we discuss the verification protocol to establish the integrity of the transaction execution. Our solutions are independent of the type of database – the solutions are equally applicable to relational or object databases.

Since multiple transactions could be running in parallel, it is essential that Carol or Alice are able to verify any past transaction. Using *history*, Bob can regenerate the environment under which a given transaction was executed. However, we need to ensure that it is impossible for Bob to go back and change his claims. Our transaction execution protocol makes Bob commit to a small amount of data sufficient to ensure that Bob cannot hide any dishonest changes to data or transaction execution.

All communication between the data owner, the users and the servers are signed by their corresponding keys so that the users could prove that Bob made a certain claim and the server could prove that a particular transaction was indeed authorized by a given user. For readability, we do not explicitly state it when describing the protocols.

Initialization

To allow the server to start serving the transaction requests of authorized users, Alice sends the initial database (DB_0) to Bob. Alice and Bob compute the MB-tree and agree on the initial state of the database (represented by the initial root label of the MB-tree, $Proof_0$. Once they have agreed on the initial state, Bob can setup the

Algorithm 7 Initialization

1: Alice sends the initial consistent state of the database, DB_0, to Bob.
2: Bob and Alice independently compute MBT_0, and verify that they agree on $Proof_0$.
3: Alice retains $Proof_0$ and discards her copy of the database.

database and start accepting transactions from the users. Algorithm 7 describes the initialization step.

Transaction Execution

When Carol wants to execute a transaction, she sends the transaction, along with the current timestamp (TS_s) to Bob. The transaction should also specify the desired isolation level. The current timestamp should be unique to the user, *i.e.* user cannot submit two transactions with the same timestamp. Bob must ensure that the timestamp is larger than all earlier timestamps used by Carol. The timestamps are used for two purposes. Firstly, they prevent *replay attacks*, *i.e.*, the server cannot execute an authorized transaction multiple times. This is because once a user uses a timestamp, it cannot be used again for a new transaction, to run the same transaction again, the server has to use a different timestamp and fake the transaction request signature (which is computationally impossible). Secondly, they are used to provide freshness guarantees.

Once Bob is satisfied with the authenticity of the transaction request, Bob executes the transaction while tracking the updates produced by the transaction (in *history*) and the consistent states seen by each transaction statement. If the transaction is allowed to commit, Bob applies the updates produced by the transaction and updates the MB-tree producing the next consistent state. Bob reports these consistent states to the user. In particular, for each statement inside the transaction, Bob reports the freshest consistent state seen by the transaction. Bob also reports the final consistent

Algorithm 8 Transaction Execution

1: Carol sends $transaction, TS_s$ to Bob.
2: Bob records this message after verifying the signature and TS_s, and starts executing the transaction.
3: While executing the transaction, Bob keeps track of the freshest consistent state that each statement sees.
4: If the transaction successfully commits at timestamp TS_i, Bob computes MBT_i
5: Bob sends $< i,\ TS_s,\ Proof_s,\ \{DB_{rj}\ for\ each\ statement\ R_j\},\ TS_i,\ Proof_i,\ RSet >$ to Carol.
6: Bob sends $transaction, TS_s, i, Proof_s, Proof_i, TS_i$ to Alice
7: Carol sends $i, TS_s, Proof_s, Proof_i, TS_i$ to Alice.
8: Alice verifies that Carol's version of commit information is same as Bob's.
9: Alice verifies that the TS_s is unique for the user and adds $< Proof_i, TS_i >$ to its chain of proofs.

state produced by the transaction at the time of commit. Algorithm 8 describes this step and Figure 6.7 shows the steps graphically.

Let DB_{rj} be the freshest consistent state seen by a statement in T_i, R_j. Then Bob declares the following information to both Alice and Carol: (1) the transaction commit sequence number, *i.e.*, i, (2) the timestamp used by Carol, TS_s, (3) the initial consistent state $Proof_s$, (4) DB_{rj} for all statements R_j, (5) the final consistent state $Proof_i$, and, (6) $RSet$, the set of values user expects as return value.

Alice uses the information from Bob to update the proof chain. The proof chain is used to keep track of different consistent states that the database goes through. Whenever Alice receives information about a transaction commit from Bob and Carol, she adds $(Proof_i, TS_i)$ to the proof chain. If k transactions have been committed, the proof chain would have $< Proof_0, TS_0 >, < Proof_1, TS_1 >, \ldots, < Proof_k, TS_k >$, where $Proof_i$ represents the root label of the MB-tree when transaction T_i committed and TS_i is the timestamp at the time of commit. The proof chain is used by the transaction verification protocol to establish freshness conditions. Alice also retains the timestamps used by each user. Alice has to ensure that a timestamp has not been used by Carol more than once.

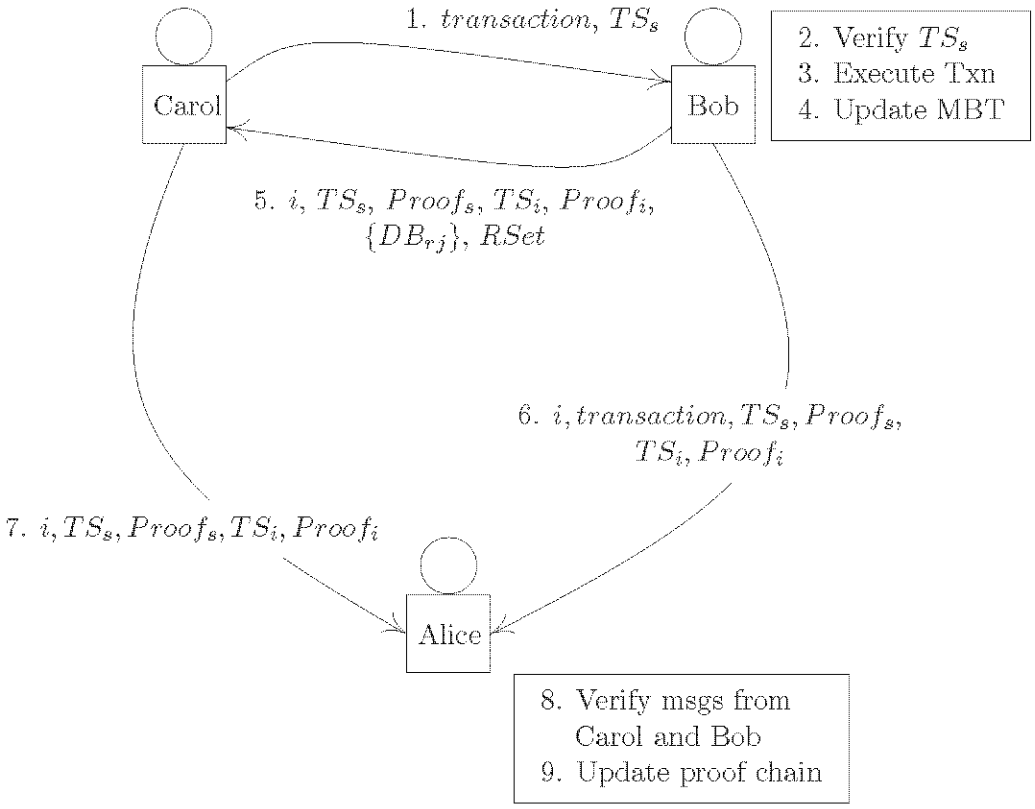


Figure 6.7.: Transaction execution

Transaction Verification

As mentioned before, our solution allows a user to randomly verify any previously committed transaction. To verify a transaction, the user needs to verify that (1) all values read by the transaction came from "valid" consistent states and were correct and complete, (2) the transaction was executed on this data to produce the updates (3) the updates were applied on the database to produce a new consistent state. We elaborate on these parts below. Algorithm 9 explains the verification protocol formally.

Firstly, the user needs to verify the trustworthiness of the data read by the transaction. Bob has to show that all values read by a transaction statement indeed came from the consistent state it reported at the time of commit. However, since a transaction can read from different consistent states based on the isolation level, Bob needs

Algorithm 9 Transaction Verification

1: Carol asks Bob to verify a transaction T_i
2: **for** each statement, R_j in T_i **do**
3: Bob computes MBT_{rj}
4: Bob sends to Carol the verification objects for all values read by R_j based on MBT_{rj}.
5: Carol verifies the correctness and completeness of R_j's reads based on the isolation level.
6: **end for**
7: Carol determines the outputs and updates for T_i (replays T_i) given these reads.
8: Bob sends to Carol the verification objects for T_i's updates based on MBT_i.
9: Carol verifies that MBT_i contains these updates.

to compute verification objects for the reads of each statement in the transaction from the corresponding consistent state. This can done using history. Carol uses the Correctness and Completeness mechanisms discussed earlier to verify the reads. Specific verification requirements for a given isolation level have been discussed in the previous subsections.

Secondly, Carol needs to compute the updates that the transaction is supposed to produce. As mentioned before, we allow only replayable transactions. Given the data read by each statement of the transaction, Carol can replay the transaction to compute the updates it produced.

Finally, Carol needs to ensure that the updates produced by the transaction were faithfully applied to the database to produce a new consistent state. For this, Bob computes the verification object from MBT_i for the updates produced by the transactions. Carol uses the verification object to ensure that all updates produced by the transaction were applied to produce DB_i and no other updates were applied.

To establish the freshness of the consistent states, we require Alice to perform a little task as well. When a transaction is committed, Bob and Carol independently report to Alice about the execution of the transaction. Alice maintains the proof chain and whenever a new state is produced by a transaction, the new proof is added to the chain. To ensure freshness, Alice checks that the first consistent state seen by

the transaction is at least the freshest state at the timestamp when the transaction was submitted (TS_s). For example, if k transactions had been committed till TS_s (Alice can figure this out using the proof chain), Alice ensures that the first consistent state seen by the transaction, DB_s, is either DB_k or fresher. Other than that, Alice stores the latest TS_s used by each user. Alice verifies TS_s has not been used by the user before. If it has been, she has detected a replay attack. Notice that the overall overhead for Alice is minimal.

6.2.7 Discussion

We now present some malicious scenarios, and show how our solutions handle them.

Bob executes an unauthorized transaction

Bob cannot manufacture a new transaction request as it requires Bob to create Carol's signature. Bob cannot replay a transaction either as we require TS_s to be unique.

Bob drops a transaction

$Proof_i$ is created by applying updates produced by T_i on $Proof_{i-1}$. If Bob drops a transaction, T_{i-1}, $Proof_i$ has to be computed using $Proof_{i-2}$. Alice will be able to detect that.

Bob does not run transactions in the claimed sequence

The proof chain maintained by Alice ensures that Bob cannot claim a different sequence of transaction commitment once the transaction has been committed.

Bob falsely claims a transaction execution environment

Each statement in the transaction reads data from a particular consistent state which is determined based on the isolation level. The execution protocol requires Bob to declare the consistent state from which the statements read data (Step 5). Since the transactions are replayable, only these consistent states define the environment which determines the outcome of the transaction. Thus, Bob cannot claim a different transaction execution environment once the transaction has been committed.

Multiple transactions run concurrently under different isolation levels

As mentioned before, databases allow users to pick the isolation level for the transactions and two (or more) concurrent transactions can execute at different isolation levels. Our solution does not stop users from doing that. This is because the semantics for trust is defined based on the consistent states and is isolated from other concurrent transactions.

Delayed transaction execution

Our solution does not stop the server from maliciously delaying a particular transaction. We expect that the quality of services agreement between the data owner and the service provider will address this issue.

Untrusted users (Carol)

Even in the presence of untrusted users, no unauthorized changes can be applied to the database as other users will be able to detect those changes if verified. However, it is possible for the server to maliciously prioritize or delay a particular user transaction. Carol can authorize updates that are incorrect, but she would be caught doing so when the updates are read and verified by other users.

In the next section, we discuss implementation details and an empirical evaluation of the proposed solutions.

6.3 Proof-of-Concept Implementation

We have implemented our solutions on top of Oracle. Our proof-of-concept implementation establishes the feasibility of our solutions and demonstrates the ease with which our solutions can be adopted for a commercial DBMS. In this section, we discuss our implementation details and present some empirical results.

6.3.1 Setup and Implementation Details

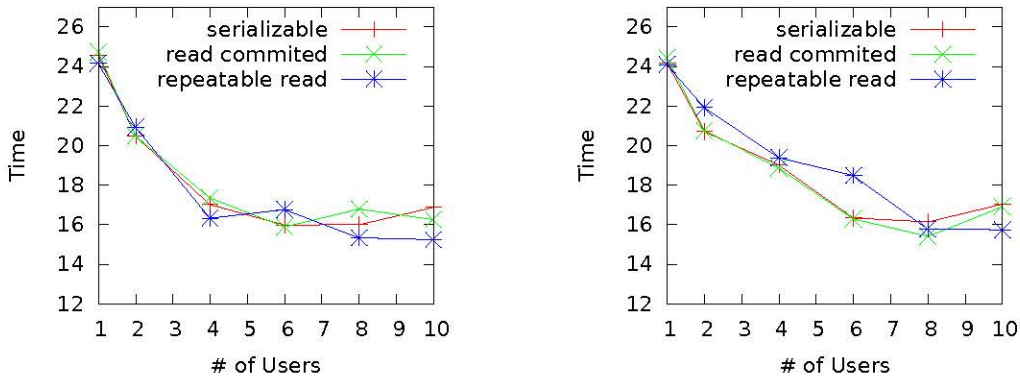
We implement the MB-tree in the form of a table. Each tuple in the MB-tree table represents a node in the tree. The history of the MB-tree and user tables are stored as a separate table. Ideally, we would exploit the database's internal structures, for example, the index structures, or Oracle's flashback technologies, however that requires internal changes to the database. We leave that for future work.

The solutions are implemented using java procedures on top of Oracle. The implementation provides an API containing the following methods: begin, commit, verify, insert, delete, update, and search. The implementation allows a transaction to be created using these methods. It is required that the data is accessed using these methods, and any processing of the data can be done by the transaction. We construct many sample transactions and execute them under different isolation levels to demonstrate the feasibility of our solutions. Table 6.1 lists different transactions used in the experiments. For example, transaction $T3$ performs 5 range queries, and inserts one tuple. users are implemented using Python.

A synthetic user table *SampleTable* was created which has two attributes, an integer *key* and a varchar *value*. *SampleTable* was populated with one million tuples with random values of *key* between -10^7 and 10^7. An MB-tree was created on *key*.

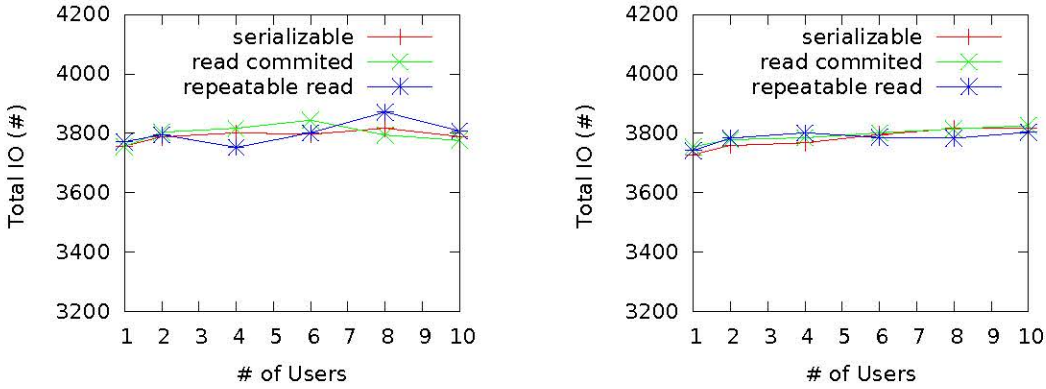
Table 6.1.: Transactions

Transaction	Description
T1	1 Insert
T2	5 Range Query
T3	5 Range Query, 1 Insert
T4	5 Range Query, 2 Insert
T5	5 Range Query, 3 Insert



(a) Time for T1 (b) Time of T1 and verification

Figure 6.8.: Insert (T1) time and verification overhead vs # of users



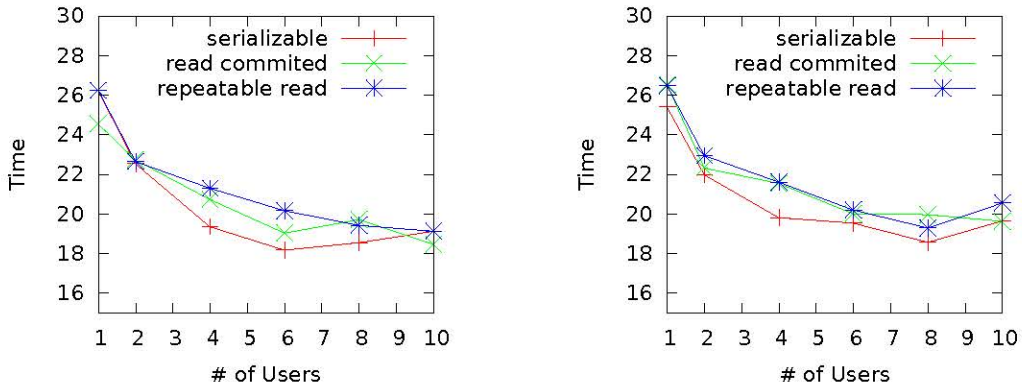
(a) IO for T1 (b) IO of T1 and verification

Figure 6.9.: Insert (T1) IO and verification overhead vs # of users

The experiments were conducted on a machine with Intel Xeon 2.4GHz processor, 12GB RAM and a 7200RPM disk with a transfer rate of 3Gb/s, running linux. Oracle 11g was used with a standard block size of 8KB.

6.3.2 Results

We now present the results of our experiments. As mentioned before, we created multiple sample transactions using our implemented API. We analyze the cost of



(a) Time for T3 (b) Time of T3 and verification

Figure 6.10.: T3: Execution time and verification overhead vs # of users

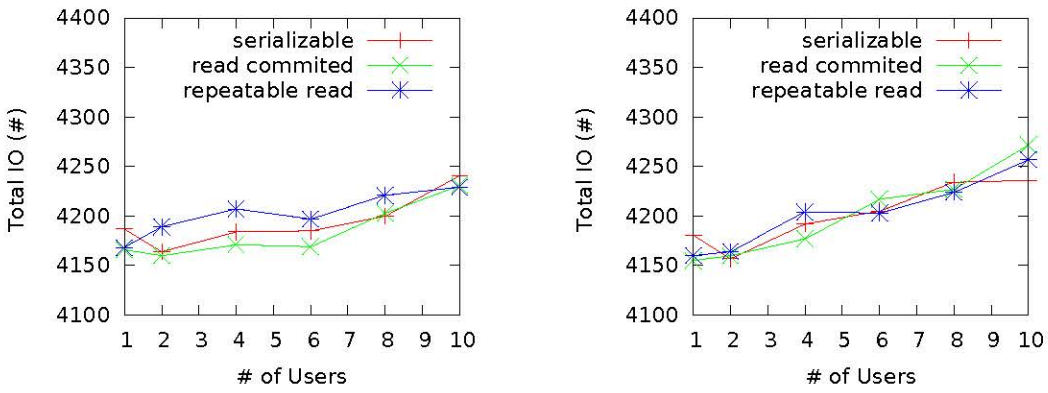
running those transactions in terms of the isolation level, concurrency and transaction verification. We present cost in terms of execution time and IO.

The fanout for the MB-tree is chosen so that each tree node is contained within a single disk block. Time is reported in seconds and IO is reported as the number of blocks read or written as reported by Oracle. Each workload ran 100 transactions. Each experiment was executed three times to reduce the error and the average values are reported. The reported times and IO are the total time and IO for the entire workload.

For experiments, we consider three isolation levels: Serializable (snapshot isolation), Read Committed, and Repeatable Read. Even though Oracle does not support Repeatable Read directly, it possible to achieve Repeatable Read isolation level. This is done by using Read Committed isolation level, and locking the tuples read by the transaction for updates, *i.e.*, by replacing *SELECT* with *SELECT FOR UPDATE* under Read Committed isolation level.

Effect of concurrency

First, we evaluate how our solutions scale with concurrency. For this, we first consider transactions that perform simple updates (insert, delete, and update). Since



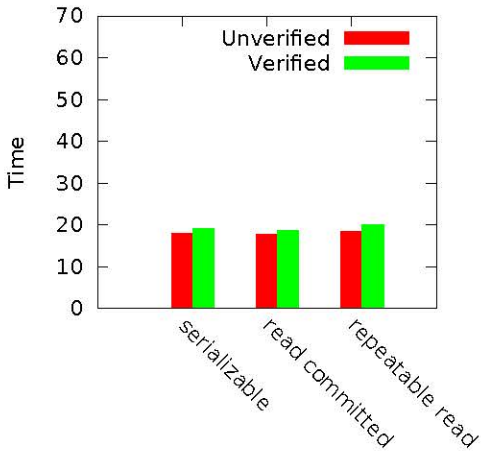
(a) IO for T3

(b) IO of T3 and verification

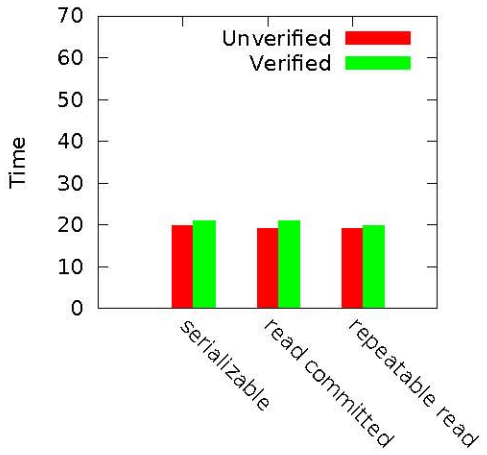
Figure 6.11.: T3: IO and verification overhead vs # of users

these operations show similar costs, we present the results for only inserts (transaction $T1$). As mentioned before, the decision to verify can be made independently per transaction. We present results for no verification, and complete verification of all transactions. Figures 6.8(a) and 6.8(b) show the transaction execution times as the number of concurrent users is changed. Figure 6.8(a) shows transaction execution times when the transactions are not verified, and Figure 6.8(b) shows transaction execution times when each transaction is verified. The same workload is divided among the concurrent users, and the time to complete the entire workload is reported. As we can see, our solutions are not negatively impacted by the increase in concurrency. Also, verification does not result in significant increase in execution time. Figures 6.9(a) and 6.9(b) further show the amount of IO performed by the database due to the workload. As we can see, the amount of IO does not change much as the number of concurrent users increases. Again, verification does not incur a significant increase in IO cost either.

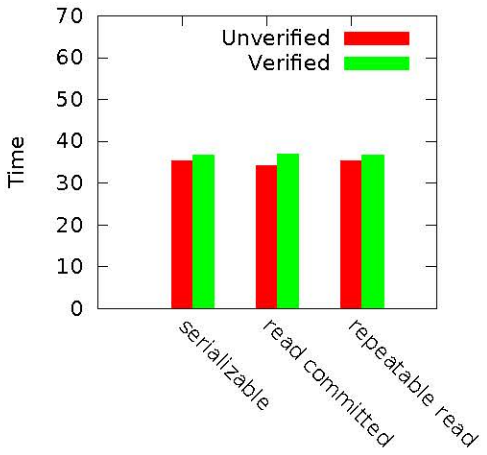
Similar results can be seen for $T3$. Figures 6.10(a) and 6.10(b) show transaction execution time as number of concurrent users is changed, and Figures 6.11(a) and 6.11(b) show the effect of concurrent users on IO.



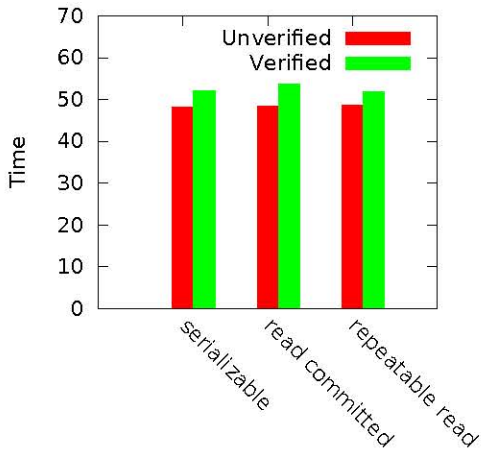
(a) Execution time of T2



(b) Execution time of T3



(c) Execution time of T4

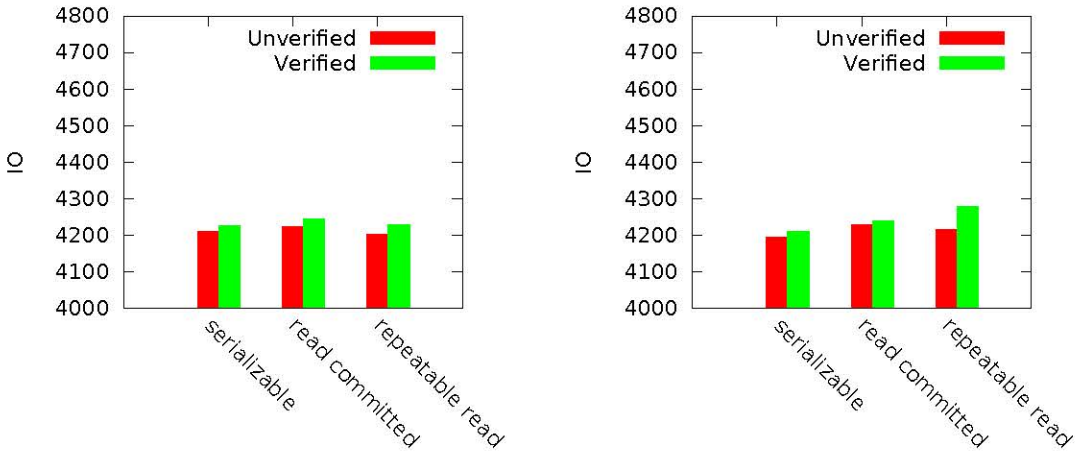


(d) Execution time of T5

Figure 6.12.: Execution time and verification overhead for different isolation levels

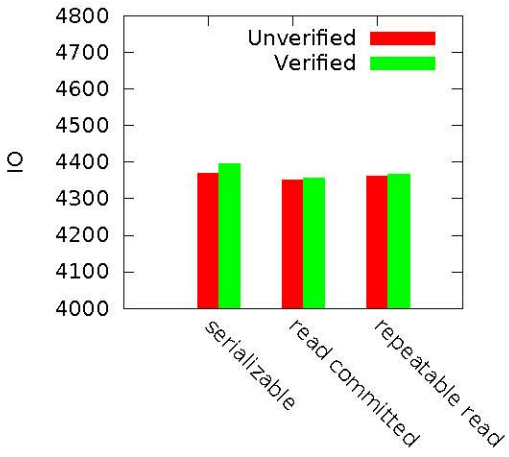
Effect on different types of transactions

Now, we consider more complicated transactions to understand the effect of isolation levels and verification on transaction execution. We consider transactions $T2$, $T3$, $T4$, and $T5$, each with increasing numbers of write operations. Each range query picks a random range with approximately 10 tuples. For these experiments, we used 5 concurrent users. Figure 6.12 shows the execution times of these transactions for

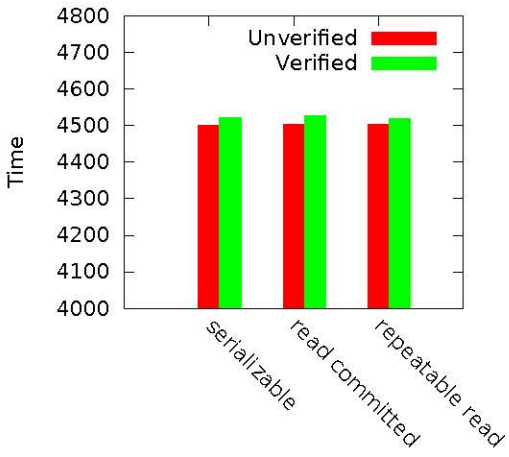


(a) IO for T2

(b) IO for T3



(c) IO for T4



(d) IO for T5

Figure 6.13.: IO Cost and verification overhead for different isolation levels

different settings (in terms of isolation levels and with or without verification). The results show that the verification cost is minimal. Even though weaker isolation requires verifying data against multiple consistent states, the results show that verification is not significantly affected by the chosen isolation level. Figure 6.13 shows the amount of IO done to execute the workload. As expected, when the transactions are verified, the amount of IO is higher, however, the overhead is small.

Overall, we observe that our solutions can be easily implemented and provide mechanisms for ensuring trustworthy execution of a transaction under a given isolation level.

6.4 Chapter Summary

In this chapter, we extended our solutions for ensuring trustworthiness of a dynamic transactional database to support weaker isolation levels than strict serializable. Since the weaker isolation levels can present a *fuzzy* database state, we revisited the weaker isolation levels to formally define which database states a transaction is expected to see at a given isolation level. We show that it is possible to ensure transactional integrity in the presence of weaker isolation levels. With our solutions, we can guarantee that the semantics of the given isolation level was followed by the server faithfully. Users can independently choose an isolation level for each transaction. Even though other users could be running transactions concurrent at different isolation levels, users can verify that their transactions followed the chosen isolation level. We implemented these solutions in Oracle. Our implementation shows the ease of adopting our solutions in existing, commercial databases without any need for modifying the database code. Empirical evaluation of the solutions show the feasibility and efficacy of the solutions.

7 FUTURE WORK AND CONCLUSION

In this chapter, we present some future work, and then conclude this dissertation.

7.1 Future Work

This section presents our ongoing and future work in this field.

Aggregate Queries An important part of SQL queries supported by database systems is aggregate queries. Our current solutions expect that all the data items that are needed to compute an aggregate query should be returned as part of the verification object. This is undesirable as such data can be large in size. One solution to alleviate this problem is to use the MB-trees to store known aggregate function values at the internal nodes. For example, for "SUM", each internal node can be augmented to store the summation (on an attribute) of all tuples in the subtree. Thus, to verify the faithfulness of an aggregate query result, we need only $O(log(k))$ partial sums, rather than all tuples that are used to compute the sum.

Ease of Verification Given that the goal of outsourcing is to alleviate the burden on the data owner and users, satisfactory solutions need to minimize the overhead and role of the data owner and users in establishing the authenticity and integrity of the solutions. In this regard, existing solutions are inadequate as they require the data owner and users to play a significant role. While this may work for some applications, many outsourced databases are expected to have both a large size and high rate of querying and updates. Thus it is necessary to explore more efficient solutions with low overheads for all involved parties. One particular cost that users incur when deploying such systems is the cost of verification. The current solutions

expect the verifier to setup a partial database and understand SQL semantics to be able to verify that the database operation was executed faithfully. For users with limited resources, it may not be feasible. One approach to reduce this burden is to outsource the verification process to a third party. Another approach would be to simplify the expectations of the database operations in order to be able to verify them without having to setup a temporary database.

Automatic Detection of Malicious Activity Current solutions expect the data owner or users to verify the transactional integrity or query results in order to detect any malicious activity. We expect the users to verify transactions randomly depending upon the level trust on the database server. However, if users had a mechanism to indicate which transaction execution seems more likely to be trustworthy, and which are not, they would able to pick transactions for verification more wisely. We propose the following two approaches.

- Defining transaction boundaries: Most transactions are simple in nature – it is clear before actually executing the transaction, which data items are likely to be read or written by the transaction. Given this information, if the transaction accessed data outside the expected boundary, we can decide to verify the transaction with greater likelihood.

- Defining constraints: Even though transactions are executed at the server, and it may not be directly known to the users which data is read or written by their transactions, they do expect certain properties in the data based on the application logic or otherwise. For example, if a customer deposits $100 in their bank account, the bank expects the total cash amount to be $100 higher. Many constraints, which are well defined, are already implemented by the database systems, for example, domain range on attribute values or foreign key constraints. However, based on the application logic and the knowledge of transactions executed on the database, the data owner and users can expect that the data satisfies certain constraints.

Combining Privacy Preserving Computation Even with encryption, a malicious server can infer information based on data access patterns, transaction semantics, *etc.*. Researchers have proposed solutions to outsource computing without leaking the input or output, or the computation. Other work in domain include oblivious-RAM with hides the data access patterns. However, these solutions cannot be combined with authentication data structures. Mechanisms that decrease the information leakage, while still providing assurance to the data owner and users about the fidelity of the data, would further reduce the required trust on the database server.

Efficient Implementation Our proof-of-concept implementation of the solutions were developed on top of an existing DBMS (Oracle), without any internal modifications. This implementation used database tables to store authentication data structures and used PL/SQL to implement the protocols. We believe that exploiting the internal structures and using more efficient structures, such as the internal index structures, will improve the performance significantly. Oracle, for example, offers mechanisms to keep old values in the database so that a user could look at an older state of the database. Exploiting these mechanisms rather than implementing our own history structure in the database tables will be helpful.

Automatic Rewrite To make our solutions easily deployable, it is desired that the transactions are automatically rewritten to use the authentication mechanisms. This will ensure that any application can use these solutions without having to rewrite the application. An ideal system will provide a middleware such that the application would send the user queries and transactions to the middleware. The middleware can follow the protocol, and rewrite the user transaction so that the transaction would be verified in future.

7.2 Conclusion

Given the complexity and cost of managing data, many organizations are looking towards outsourcing their data management activities. However, since the data owner lacks control over the hardware and software running at the server, there is a reluctance to blindly trust the server. In this dissertation, we identified and solved important problems that arise when the database is hosted in an untrusted environment. With our solutions, we reduce the level of trust required in order to trust an untrusted database to operate the database honestly.

We proposed solutions to assure the data owner and the database users that the user transactions were executed honestly. The proposed solutions allow multiple users to run transactions concurrently without being vetted by a central entity. Our solutions provide assured provenance of data. Furthermore, our solutions provide indemnity for the server against false claims of wrongdoing. We implemented our solutions on top of an existing DBMS (Oracle) without any internal modifications, thus demonstrating the ease of deployment. Our empirical evaluation shows that our solutions are comparable to existing solutions without providing transactional integrity.

We extended our solutions to support access control rules. Using our solutions, the data owner can be assured that the server implemented the access control rules, and the users can still verify transactional integrity in presence of access control rules. Our solutions provide encryption mechanisms so that only authorized users can read the data while the server can execute queries on the encrypted data. Our solutions allow the data owner to easily update the access control rules, while keeping the overhead for key management small. Our empirical evaluation of the solutions shows that the overhead of our solutions is minimal.

We further extended our solutions to support weaker isolation levels, thus allowing users to pick their isolation level before executing a transaction. Even though weaker isolation levels risk showing a fuzzy state, we design our solutions to ensure that

the user will be able to verify that the transaction execution is consistent with the isolation level requirements and guarantees. Our proof-of-concept implementation allows users to create transactions which can be verified at any point in future. This allows our solutions to be easily adoptable in existing database applications. Our empirical evaluation of the solutions show the feasibility and efficacy of the solutions.

Finally, we presented some of our ongoing and future work in this field. Despite the significant work already done in this field, numerous interesting problem remain to be solved in order to make it practical for databases. Solutions presented in this dissertation reduce that gap and raised some interesting problems to be solved yet. Overall, with our solutions, the data owner need not blindly trust the server. With our solutions, the data owner or the users can detect any malicious activity on the database, and also prove it. We show that our solutions can be easily adopted in an existing database.

REFERENCES

REFERENCES

[1] Premkumar T. Devanbu, Michael Gertz, Chip Martel, and Stuart G. Stubblebine. Authentic third-party data publication. In *Proceedings of the Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSec)*, 2000.

[2] Feifei Li, Marios Hadjieftheriou, George Kollios, and Leonid Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proceedings of ACM Special Interest Group on Management of Data (SIGMOD)*, 2006.

[3] Einar Mykletun, Maithili Narasimha, and Gene Tsudik. Authentication and integrity in outsourced databases. In *Network and Distributed System Security Symposium (NDSS)*, 2004.

[4] Maithili Narasimha and Gene Tsudik. Authentication of outsourced databases using signature aggregation and chaining. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*, 2006.

[5] Hweehwa Pang, Arpit Jain, Krithi Ramamritham, and Kian lee Tan. Verifying completeness of relational query results in data publishing. In *Proceedings of ACM Special Interest Group on Management of Data (SIGMOD)*, 2005.

[6] Sarvjeet Singh and Sunil Prabhakar. Ensuring correctness over untrusted private database. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2008.

[7] Michael T. Goodrich, Roberto Tamassia, and Andrew Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *Proceedings of the DARPA Information Survivability Conference & Exposition (DISCEX)*, 2001.

[8] HweeHwa Pang and Kian Lee Tan. Authenticating query results in edge computing. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2004.

[9] HweeHwa Pang, Arpit Jain, rithi Ramamritham, and Kian Lee Tan. Verifying completeness of relational query results in data publishing. In *Proceedings of ACM Special Interest Group on Management of Data (SIGMOD)*, 2005.

[10] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*, 2003.

[11] Christian Cachin, Abhi Shelat, and Alexander Shraer. Efficient fork-linearizable access to untrusted shared memory. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2007.

[12] David Mazières and Dennis Shasha. Building secure file systems out of byzantine storage. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2002.

[13] Ari Juels and Burton S. Kaliski Jr. PORs: Proofs of retrievability for large files. In *Proceedings of the ACM International Computer and Communication Security Conference (CCS)*, 2007.

[14] Muthuramakrishnan Venkitasubramaniam, Ashwin Machanavajjhala, David J. Martin, and Johannes Gehrke. Trusted CVS. In *Proceedings of the International Workshop on Security and Trust in Decentralized/Distributed Data Structures (STD3S)*, 2006.

[15] Christian Cachin and Martin Geisler. Integrity protection for revision control. In *Proceedings of the International Conference on Applied Cryptography and Network Security (ACNS)*, 2009.

[16] Sumeet Bajaj and Radu Sion. TrustedDB: A trusted hardware based database with privacy and data confidentiality. In *Proceedings of ACM Special Interest Group on Management of Data (SIGMOD)*, 2011.

[17] Einar Mykletun and Gene Tsudik. Incorporating a secure coprocessor in the database-as-a-service model. In *Proceedings of the International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA)*, 2005.

[18] Arvind Arasu, Spyros Blanas, Manas Joglekar, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, Prasang Upadhyaya, and Ramarathnam Venkatesan. Engineering performance and security with cipherbase. *Data Engineering Bulletin*, December 2012.

[19] Adriane P. Chapman, H. V. Jagadish, and Prakash Ramanan. Efficient provenance storage. In *Proceedings of ACM Special Interest Group on Management of Data (SIGMOD)*, 2008.

[20] Susan B. Davidson, Sanjeev Khanna, Sudeepa Roy, Julia Stoyanovich, Val Tannen, and Yi Chen. On provenance and privacy. In *Proceedings of the International Conference on Database Theory (ICDT)*, 2011.

[21] Ragib Hasan, Radu Sion, and Marianne Winslett. Introducing secure provenance: Problems and challenges. In *Proceedings of the ACM Workshop on Storage Security and Survivability (StorageSS)*, 2007.

[22] Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Querying data provenance. In *Proceedings of ACM Special Interest Group on Management of Data (SIGMOD)*, 2010.

[23] Jing Zhang, Adriane Chapman, and Kristen Lefevre. Do you know where your data's been? – Tamper-evident database provenance. In *Proceedings of the SIAM International Conference on Data Mining (SDM)*, 2009.

[24] Zoé Lacroix, Christophe Legendre, and Spyro Mousses. Storing scientific workflows in a database. *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2009.

[25] Shawn Bowers, Timothy McPhillips, Bertram Ludäscher, Shirley Cohen, and Susan B. Davidson. A model for user-oriented data provenance in pipelined scientific workflows. In *Proceedings of the International Conference on Provenance and Annotation of Data*, 2006.

[26] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system: Research articles. *Concurrency and Computation: Practice and Experience*, 2006.

[27] Ragib Hasan, Marianne Winslett, and Radu Sion. Requirements of secure storage systems for healthcare records. *Lecture Notes in Computer Science: Secure Data Management*, 2007.

[28] Richard Snodgrass Shilong, Richard T. Snodgrass, Shilong Stanley Yao, and Christian Collberg. Tamper detection in audit logs. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2004.

[29] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In *Proceedings of ACM Special Interest Group on Management of Data (SIGMOD)*, 2004.

[30] Bijit Hore, Sharad Mehrotra, and Gene Tsudik. A privacy-preserving index for range queries. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2004.

[31] Florian Kerschbaum and Julien Vayssiere. Privacy-preserving data analytics as an outsourced service. In *Proceedings of the ACM International Computer and Communication Security Conference (CCS)*, 2008.

[32] Hakan Hacigümüs, Bijit Hore, Bala Iyer, and Sharad Mehrotra. Search on encrypted data. *Advances in Information Security, Secure Data Management in Decentralized Systems*, 33, 2007.

[33] Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of ACM Special Interest Group on Management of Data (SIGMOD)*, 2002.

[34] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2011.

[35] Peter Williams and Radu Sion. Single round access privacy on outsourced storage. In *Proceedings of the ACM International Computer and Communication Security Conference (CCS)*. ACM, 2012.

[36] Ninghui Li and Tiancheng Li. t-Closeness: Privacy beyond k-anonymity and l-diversity. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2007.

[37] Cynthia Dwork. Differential privacy. In *International Colloquium on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science, 2006.

[38] Frank McSherry. Privacy integrated queries. *Communications of the ACM*, September 2010.

[39] Andrew C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, 1982.

[40] Marina Blanton, Mikhail J. Atallah, Keith B. Frikken, and Qutaibah M. Malluhi. Secure and efficient outsourcing of sequence comparisons. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2012.

[41] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Proceedings of the Annual Symposium on Foundations of Computer Science (SFCS)*, 1986.

[42] Susan Hohenberger and Anna Lysyanskaya. How to securely outsource cryptographic computations. In *Proceedings of the International Conference on Theory of Cryptography (TCC)*, 2005.

[43] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2010.

[44] Stanford University. Folding@home distributed computing. `http://folding.stanford.edu`.

[45] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: an experiment in public-resource computing. *Communications of the ACM*, November 2002.

[46] Ashish Kundu and Elisa Bertino. Structural signatures for tree data structures. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2008.

[47] Hong Chen, Xiaonan Ma, Windsor Hsu, Ninghui Li, and Qihua Wang. Access control friendly query verification for outsourced data publishing. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2008.

[48] Elisa Bertino, Barbara Carminati, Elena Ferrari, Bhavani Thuraisingham, and Amar Gupta. Selective and authentic third-party distribution of xml documents. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2004.

[49] Gerome Miklau and Dan Suciu. Controlling access to published data using cryptography. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2003.

[50] W. G. Tzeng. A time-bound cryptographic key assignment scheme for access control in a hierarchy. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2002.

[51] Selim G. Akl and Peter D. Taylor. Cryptographic solution to a problem of access control in a hierarchy. *ACM Transactions on Computer Systems (TOCS)*, 1983.

[52] Mikhail J. Atallah, Keith B. Frikken, and Marina Blanton. Dynamic and efficient key management for access hierarchies. In *Proceedings of the ACM International Computer and Communication Security Conference (CCS)*, 2005.

[53] Kevin Fu, Seny Kamara, and Tadayoshi Kohno. Key regression: Enabling efficient key distribution for secure distributed storage. In *Network and Distributed System Security Symposium (NDSS)*, 2006.

[54] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[55] Bart Preneel. *Analysis and design of cryptographic hash functions.* PhD thesis, Katholieke University Leuven, 1993.

[56] Ralph C. Merkle. Protocols for public key cryptosystems. *IEEE Symposium on Security and Privacy (SP)*, 1980.

[57] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21, 1978.

[58] Rohit Jain and Sunil Prabhakar. Trustworthy data from untrusted databases. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2013.

[59] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.

[60] Rohit Jain and Sunil Prabhakar. Guaranteed authenticity and integrity of data from untrusted servers. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2014.

[61] Alberto Ceselli, Ernesto Damiani, Sabrina De Capitani Di Vimercati, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. Modeling and assessing inference exposure in encrypted databases. *ACM Transactions of Information System Security (TISSEC)*, 2005.

[62] Rohit Jain and Sunil Prabhakar. Access control and query verification for untrusted databases. In *Proceedings of the Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSec)*, 2013.

[63] American national standard for information systems – Database language – SQL. *ANSI X3.135-1992*, November 1992.

VITA

VITA

Rohit Jain was born and raised in Ghaziabad, India. He joined the Indian Institute of Technology, Kanpur for his Bachelor of Technology degree in Computer Science and Engineering. He spent a summer as a research intern at Motorola India Research Labs, and another at Virginia Polytechnic Institute and State University. At IIT Kanpur, Rohit actively participated in different student outreach programs, including the annual technical festival Techkriti and the linux user group Navya.

After graduating from IIT Kanpur, Rohit joined Purdue University for his PhD in the department of Computer Science. He usually got excited about any problem related with data. At Purdue, he got to explore research problems in the areas of databases, data mining and algorithms. He spent one summer at Yelp, where he worked on some data mining problems. Along the way, he received a Master of Science degree in Computer Science as well. Rohit continued to participate in student outreach programs at Purdue. He chaired the CS Graduate Student Board and served in Purdue Graduate Student Government as well. After receiving his doctorate degree, Rohit joined Google.