Ensuring Trustworthiness of Transactions under Different Isolation Levels

Rohit Jain, Sunil Prabhakar

Department of Computer Sciences, Purdue University
West Lafayette, IN, USA
{jain29,sunil}@cs.purdue.edu

***Abstract*—Given the complexity and cost of managing data, many organizations are moving towards outsourcing their data management activities. However, since the data owner looses control over the data when the data services are hosted on an outsourced server, there is a reluctance to blindly trust the server. The problem of trust escalates further when the data is allowed to be updated by multiple authorized users, as in a typical dynamic database setting where authorized users can query and update the database using transactions. The data owner needs to be assured that all user transactions are executed faithfully, and no unauthorized changes are made to the database. Earlier works in this domain have been limited to those cases where the data is either not updated at all, or all changes to the database are vetted by a central trusted entity. In our previous work [1], we proposed solutions to ensure trustworthy transaction executions under the requirement of strict serializable isolation levels without requiring the data owner to trust the service provider. In this paper, we generalize those solutions to support other isolation levels. In addition to developing the necessary protocols, we discuss a proof-of-concept implementation in the form of a middleware on top of Oracle. The middleware provides an API to read/write data from the database. Users can create transactions using this API and can verify the faithfulness of transaction execution. This prototype implementation establishes the feasibility of our solutions on existing commercial database systems.**

I. Introduction

Data are often subjected to untrusted environments, for example, when data services are outsourced to a third party and the data owner lacks direct control over the software and hardware running at the servers. Even in the absence of outsourcing, data are often susceptible to malicious tampering by insiders or hackers, and inadvertent errors. Ensuring that the integrity of the data has not been compromised is of utmost importance to a data owner. Decades of database research has resulted in solutions that ensure the integrity and consistency of data through principles such as transactions, ACID properties, and access control rules. These solutions have been developed under the assumptions that the threats arise due to failures (computer crashes, disk failures, etc.), limitations of hardware, and the need to enforce access control based upon user authorization. However, the semantics of these principles are defined with the assumption of trust on the database server. Considering the lack of trust that arises due to the untrusted environments that the databases are subjected to, these principles need to be revisited to understand exactly what we should expect when a transaction execution follows those principles and how we can verify that the principles were indeed followed by the untrusted server while executing the transactions.

Much work has been done towards data integrity and query verification. However, most of these works are limited to static data where the data are either not modified at all, or the data are modified by only one entity. Some works consider the dynamic setting as well, where the data can be modified by multiple users, however they are not easily applicable to databases due the complex nature of the underlying principles, for example, serializability or freshness of data.

In our previous work [1], [2], we developed solutions to the problem of ensuring the authenticity and integrity of transactions executed on a dynamic relational database hosted on an untrusted server. Our previous solutions required strict serializability. 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. In this paper, we revisit the semantics for different isolation levels without assuming that the server is trusted, and clarify their definition 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. Valid consistent states are determined based on the chosen 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 tacit 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 paper is organized as follows. Section II discusses related work. Section III presents some preliminary tools that are necessary for this work and summarizes existing results that form the basis of our solution. Section IV presents our solutions. A discussion of the implementation of the solution and empirical evaluation is presented in Section V. Section VI concludes the paper.

II. Related Work

Much work has been done towards ensuring authenticity and integrity of query results from an untrusted database server [1], [3], [4], [5], [6], [7], [8], [9]. Most of these works consider only those cases where the data is either not updated at all, or it is updated by a single entity [3], [4], [5], [10]. However, this is unreasonable as a typical dynamic database accepts queries and updates from multiple authorized users. Only limited work has been done towards the situation where multiple users can update data [1], [7], [11].

Some work has also been done towards exploiting trusted hardware to provide trustworthy databases [12], [13]. Since the trusted hardware have very limited resources, it brings significant challenges to process typical database workloads. Furthermore, these techniques are only applicable when trusted hardware is available. In our solutions, we do not make this assumption.

[14] proposed a signature scheme using which multiple signatures from different users can be aggregated together and verified in a single step. [7] uses [14] and signature chains to provide mechanisms to verify correctness and completeness of data. Combined with Merkle Hash Tree, this solution can provide freshness guarantees as well. However, the solution is orders of magnitude slower than other hash based solutions [5].

[5] proposes to use B+-trees rather than binary trees for Merkle Hash Trees. Merkle Binary Trees can provide mechanisms to verify the correctness of query results, however, completeness cannot be ensured. Since a B+-tree orders the data at the leaf level, completeness guarantees can be provided as well. [5] further optimizes the solution by embedding B+ tree nodes with another tree. This solution is applicable in cases where only the data owner can modify the data.

An orthogonal avenue of research focuses on privacy preservation in outsourced databases [9], [10], [2], [15], [16], [17], [18], [19]. These solutions focus on the privacy aspect of data outsourcing. Some of these works address the problem of data-leakage in authentication data structures [2], [9], [10], while others focus on encrypting the data for privacy preservation [15], [16], [18], [19]. Data is encrypted before shipping it to the server. Mechanisms have been developed so that a user could run queries on the encrypted data. Much work has also been done towards key management and access control using encryption [20], [21], [22], [23].

Encryption of the data by the user ensures the authenticity of the data. However, encryption alone does not ensure that integrity of data, as the server could drop an update, or apply data updates in different orders. As a result, different users can view different (inconsistent) versions of the databases. [24], [25] propose protocols based on the notion of fork-consistency. Fork-consistency ensures that a user’s view of 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 updates from that user. This makes the detection of such malicious events easier. However, the proposed solution handles only single read or write operations and the operations that require multiple reads and writes, as in the case of a typical database transaction, are not handled.

In our previous work [1], we proposed initial solutions for ensuring trustworthiness of databases in the presence of transactions. Multiple users can run transactions on the database concurrently. Our solutions provide mechanisms for the users to verify faithful execution of any previously committed transaction. This is done by engaging the user and the server in a protocol which makes the server commit to a small amount of information sufficient to ensure that when a user decides to verify the transaction, the server cannot go back and “fix” its wrongdoing in order to pass the verification. However, these solutions require all transactions to be run in strict serializable isolation level.

In this paper, we propose solutions with which the data owner and the users can be assured that the untrusted database ran the transactions faithfully for a broader range of isolation levels.

III. Preliminaries

In this section, we start by explaining the different entities involved in our model. Then, we explain Merkle Hash Trees and Merkle B+ Trees which are building blocks for our solutions, and also discuss their use to verify the correctness and completeness of query results.

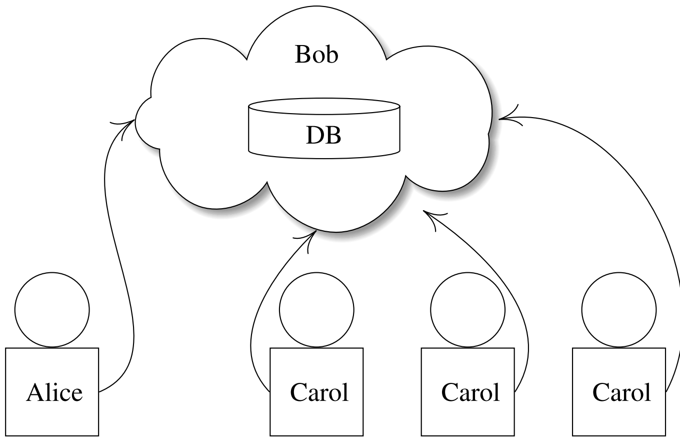


Figure 1: The various entities involved: The database owner (Alice); The database server(Bob); and authorized users (Carol).

Symbol	Description
t_i	the ith tuple of a relation
$h(x)$	the value of a one way hash function over x
$\Phi(n)$	label of node n in MB-tree or MHT
H_i	label of the ith node in the MB-tree
$a\|\|b$	concatenation of a and b
VO	a verification object
$Proof$	root label of the MB-tree
$t(a)$	a tuple with key value a
$t(a,v)$	v^{th} version of a tuple with key value a
T_i	i^{th} transaction in the commit order
DB_i	database state after i^{th} commit
MBT_i	Merkle B+-tree after i^{th} commit
R_j	j^{th} read statement in the transaction
DB_{rj}	Latest consistent state seen by R_j
TS_s	the timestamp when the transaction was submitted
TS_i	the timestamp when the ith transaction was committed

Table I: Symbol Table

A. Model

In our model, 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. Figure 1 shows the various entities in this model.

Alice and Carol want to ensure that the user transactions were executed honestly, – *i.e.*, the transaction execution followed ACID semantics. For each transaction, the user sets the isolation level that the database server should follow. Multiple concurrent transactions can run at different isolation levels. A typical database supports several 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].

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 compromise data integrity. However, due to unforeseen reasons or hacker or insider attacks the integrity of data can be compromised despite the best efforts of the server. Considering this, Alice should not blindly trust the server.

B. Correctness and Completeness

Correctness of query results requires that any data item in a query result should exist in the database. Merkle Hash Trees (MHT) have been used extensively to provide correctness guarantees of data. A Merkle Hash Tree is a binary tree with labeled nodes. Each leaf node represents a tuple and its label is computed as the hash of that tuple. The label for a non-leaf node, n, with children n_l and n_r is computed as:

$$\Phi(n) = h(\Phi(n_l)||\Phi(n_r)) \quad (1)$$

where $||$ is concatenation and h is a one-way hash function.

To provide correctness guarantees, a Merkle Hash Tree is created on top of the database table. The root label (called $Proof$) of the tree is made public. $Proof$ alone is enough to ensure that if the database server falsifies any part of a query result, it will be caught upon verification. To verify the correctness of the query results, the user asks the server for a Verification Object (VO). Using the Verification Object and the query results, the user recomputes the $Proof$. If the computed $Proof$ is the same as the one declared publicly, the user can be assured that the query results were not fabricated. The use of one-way hash functions ensures that it is computationally impossible to cheat the verification process.

Completeness requires that no data item which satisfies the query is omitted from the query result. Instead of using binary trees for Merkle Hash Tree, a B+-tree can be used to provide both correctness and completeness guarantees [5]. Using Merkle B+ Trees (MBT), a user can verify that the data items just preceding and just following the query results (in sorted order) were indeed outside the query range and all data items in between were part of the query result. Figure 2 shows a sample MB-tree structure built on the attribute key of Table II. Consider a query $\sigma_{15<key\leq 25}$. The query result should be $\{t_{13}, t_{14}, t_{15}\}$. To verify the trustworthiness of the query result, the server sends a verification object to the user. The verification object will include the following: (1) the tuples just preceding and just following the query results. In this case, it will be t_{12} and t_{16}. (2) all node labels required by the user to compute the $Proof$. In this case, it will be $h(t_1)$, H_7, H_8, and H_9. Using this verification object and the query result, the user can independently compute node labels and the $Proof$. If the computed $Proof$ matches the publicly declared $Proof$, the user is satisfied with the correctness and completeness of the query results. For more details, please refer to [1], [5].

[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.

tupleID	key
..	..
..	..
11	10
12	14
13	17
14	18
15	25
16	27
17	30
18	35
..	..
..	..

Table II: Sample Data Table

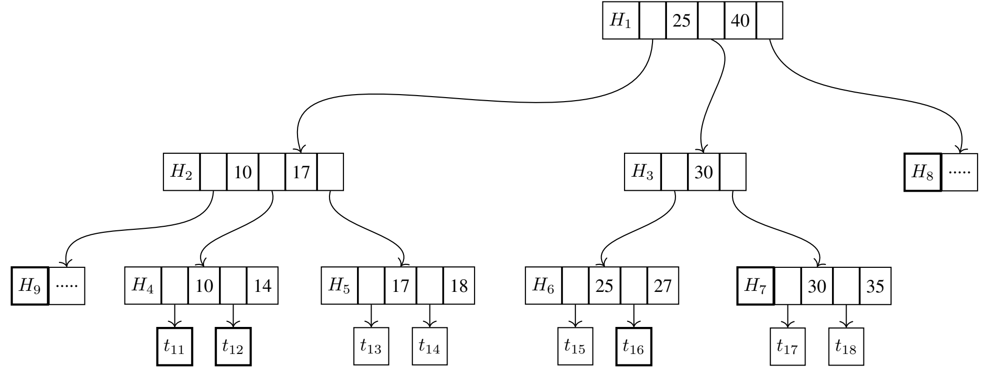


Figure 2: An MB-tree on attribute A of Table II. The highighted values will form the Verification Object for query $\sigma_{15<key\leq 25}$ with the query result $\{t_{13}, t_{14}, t_{15}\}$.

IV. 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 reexamine the semantics of different types of isolation levels without the tacit assumption of trust on the database server, and the implications for verifying faithful execution under these isolation levels.

As in [1], 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, or dependence on anything except the input parameters and values read from the database.

The ANSI/ISO SQL standard [26] 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 is seen by the transactions. As mentioned before, we also do not allow *Dirty Writes* as that can lead to a inconsistent state.

A. 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 3 shows this graphically. $t(x)$ represents a tuple with value x. Notice that this is different han strict serializability. Strict serializability requires that it should appear as if one transaction committed after the other, but also that the transaction read 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. In Figure 3, the ith column represents the data ($t(x)$) present in the consistent state DB_i. The initial state of the database (DB_0) is considered to be a consistent state. The database can go through multiple consistent states during the execution of a transaction (as other concurrent transactions commit). 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

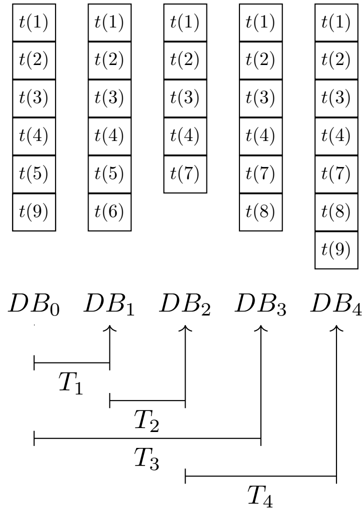


Figure 3: A conceptual view of database consistent stats

- 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.

1) 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 the 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 commit. Thus, each new MB-tree represents a consistent state produced by a single transaction.

2) 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, in *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 client 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.

B. *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 4 shows this graphically. 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 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 the previous consistent state (and not from older states)
- 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

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

C. *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. Figure 5 shows this graphically. As before $t(x)$ represent a tuple with key value x. $t(x, v)$ represents v^{th} version of tuple with key value x. At this isolation level, a transaction reads from a single consistent snapshot of the database. Two transactions can share the same snapshot. However, when the transaction is allowed to commit, it is ensured that no two concurrent transactions write to the same data item (*i.e.*, *Dirty Writes* are not allowed). Formally, a transaction T_i is executed against a

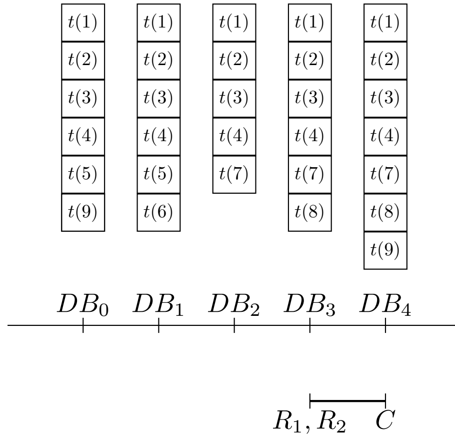


Figure 4: An example of strict serializable isolation level

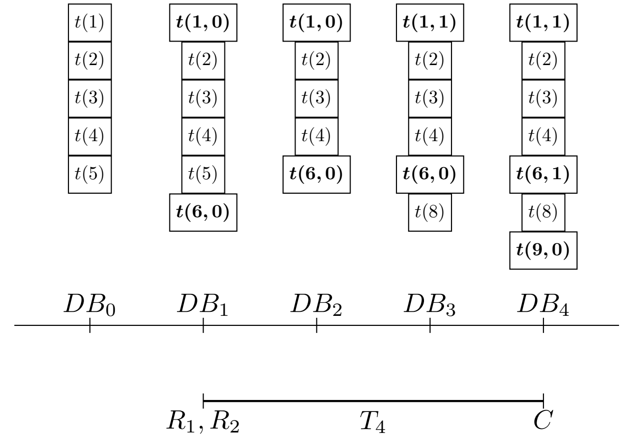


Figure 5: An example for snapshot isolation level

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 5, T_4 starts at DB_1, and modifies $T(6, 0)$ to $T(6, 1)$ and creates $t(9)$. Snapshot isolation level requires that T_2 and T_4 do not modify $t(6)$, however they are allowed to modify other tuples (for example, $t(1)$).

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 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 on DB_{i-1} is same as its version on DB_s. In figure 5, since the version number of $t(6)$ at DB_3 is same as that in DB_1, we can be assured that $t(6)$ was not modified by other transactions during the execution of T_4.

D. 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 is allowed. Consider Figure 6. Lets say R_1 reads data items with values greater than 5 from the consistent state DB_1. The results would be $t(6)$. When the transaction runs

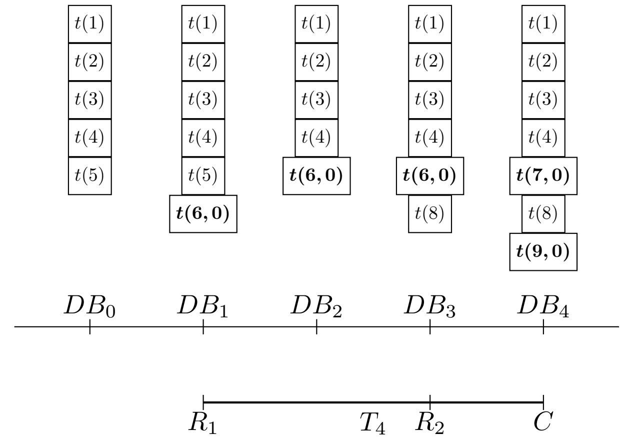


Figure 6: An example for repeatable read isolation level

R_2 which also reads values greater than 5 against DB_3, then output would be $t(6)$ and $t(8)$. It is not possible that $t(6)$ was modified between DB_1 and DB_3. 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 history 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, expect 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.

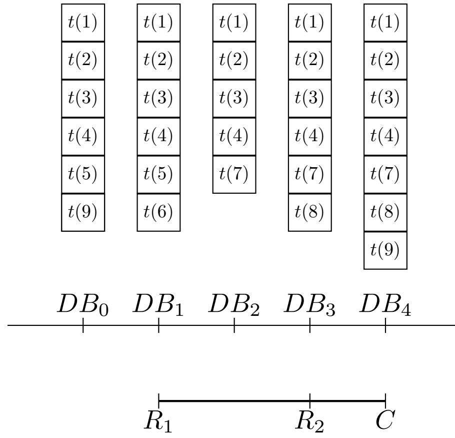


Figure 7: An example of stricter read committed isolation level

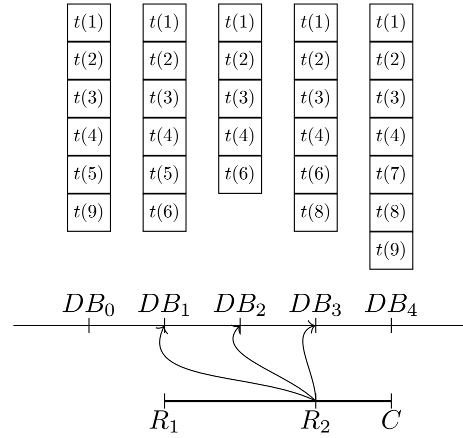


Figure 8: An example of weaker read committed isolation level

E. Read Committed

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

1) 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. Figure 7 describes it graphically. Lets say statement R_i reads data from consistent state DB_{r_i}. Then, for statements R_{i1} and R_{i2} where R_{i1} is executed before R_{i2}, we should ensure that $r_{i1} <= r_{i2}$. That is, consistent state $DB_{r_{i2}}$ should be either same as $DB_{r_{i1}}$ or newer. In Figure 7, lets say R_1 reads data items greater than 5, then R_1 reads $t(6)$. When R_2 is executed, which also reads data items greater than 5, it must execute at the consistent state DB_1 or newer. Thus, it will not read $t(9)$ from DB_0.

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:
  - 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 the statements to produce updates
- Verify that all updates produced by this transaction were applied to produce the next consistent state

2) Weaker form: Some databases use a weaker semantics for Read Committed isolation leve. 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 8 as an example. R_1 reads all data items that are greater than 5. The query answer can include $t(6)$ and/or $t(9)$. Lets say that it reads $t(6)$. If the same query is run again (labelled as R_2 this time), R_2 must not read $t(9)$ even though it is a committed data item. This is because a previous statement (R_1) has read consistent state DB_1. So, all future statements must see either DB_1 or newer states. Also, R_2 must include $t(6)$ as it is present in all consistent states between DB_1 and DB_3. Formally, if DB_{i0} is the state of the database when the transaction started, DB_{in} is the consistent state it produces when allowed to commit, and DB_{ri} is the freshest state seen by the statement R_i, then $ri <= rj$ 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

F. The Protocol

We now discuss our solution for ensuring Transactional Integrity under a given isolation level. Our solutions allow

Algorithm 1 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.

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 the transaction was executed. However, we need to ensure that it is impossible for Bob to go back and change its 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 users and the servers are signed by the 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.

1) 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 database and start accepting transactions from the users. Algorithm 1 describes the initialization step.

2) Transaction Execution: When Carol wants to execute a transaction, she sends the transaction, along with the current timestamp (TS_s). The current timestamp should be unique to the user, *i.e.* user cannot submit two transactions with the timestamps. Bob must ensure that the timestamp is larger than all earlier timestamps used by Carol and that the timestamp is not a future timestamp. The timestamps are used for two purposes. Firstly, it prevents the *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, it is 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 state produced by the transaction at the time of commit. Algorithm 2 describes this step and Figure 9 shows the steps graphically.

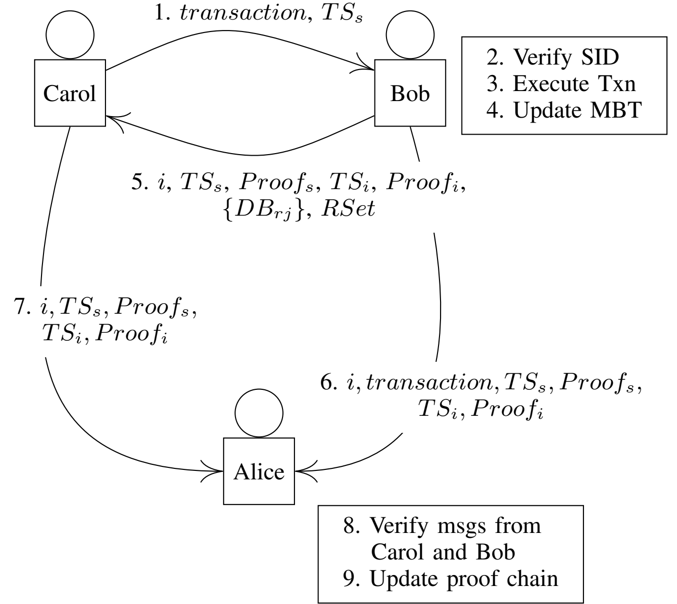


Figure 9: Transaction Execution

Say that for each statement in T_i, R_j, the freshest consistent state that it saw was DB_{rj}. 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 $< Proof_0, TS_0 >$, $< Proof_1, TS_1 >$, ..., $< Proof_k, TS_k >$, 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. Proof chain is used by the transaction verification protocol to establish the 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.

Algorithm 2 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.

3) 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 transactions 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 3 explains the verification protocol formally.

Firstly, the user needs to verify that the transaction read correct and complete data. 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 to computes 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. Since we allow only replayable transactions, this can be done given the data read by the transaction. Carol executes the transaction over the data read by the transaction and computes the updates.

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 it out using the proof chain), Alice ensures that the first consistent state seen by the transaction, DB_s, 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.

Algorithm 3 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.

G. Discussion

We now present some malicious scenarios, and show how our solution handle them. For further details, please refer to our previous paper [1].

1) If Bob claim a different transaction execution environment: Each statement in the trandition 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 4). 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 once the transaction has been committed.

2) If 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 executed 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.

3) Delayed transaction execution: Our solution does not stop the server from maliciously delaying a particular transaction. We believe QoS agreement between the data owner and the service provider will address this issue.

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

Table III: Transactions

4) If untrusted users: 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 prioritized or delayed a particular user transaction on purpose. We leave that for future work.

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

V. 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.

A. 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 III lists different transactions used in the experiments. For example, transaction $T3$ performs 5 range queries, and inserts one tuple.

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.

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.

B. Results

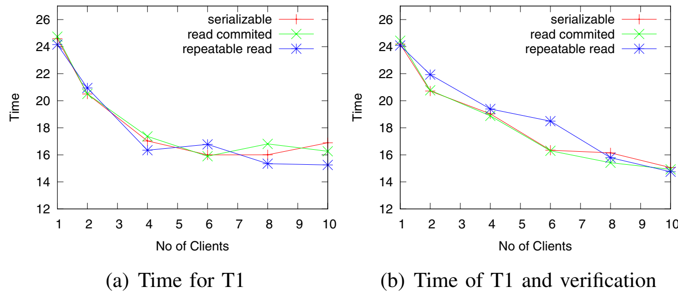
We now present the results of our experiments. As mentioned before, we created multiple sample transactions using our implemented API. We analyze the cost of running those transactions in terms of the isolation level, concurrency and transaction verification. We present cost in terms of execution time 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 can be achieved by locking the tuples read by the transaction for updates, *i.e.*, by replacing *SELECT* with *SELECT FOR UPDATE*.

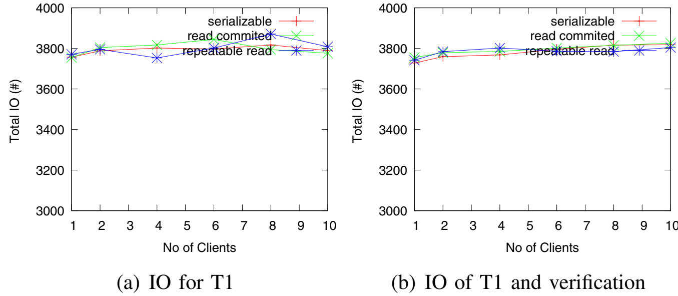
1) Effect of concurrency: First, we evaluate how our solutions scale with concurrency. For this, we first consider transactions that perform simple updates (insert, delete, and update). Since these operations show similar costs, we present the results for only inserts (transaction $T1$). As mentioned before, the decision to verify can be made independently per transaction. We present results for no verification, and complete verification of all transactions. Figure 10(a) and 10(b) show the transaction execution times as the number of concurrent users is changed. Figure 10(a) shows transaction execution times when the transactions are not verified, and Figure 10(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 11(a) and 11(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.

2) 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 number of write operations. Each range query picks a random range with approximately 10 tuples. For these experiments, we used 5 concurrent users. Figure 14 shows the execution times of these transactions for 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 15



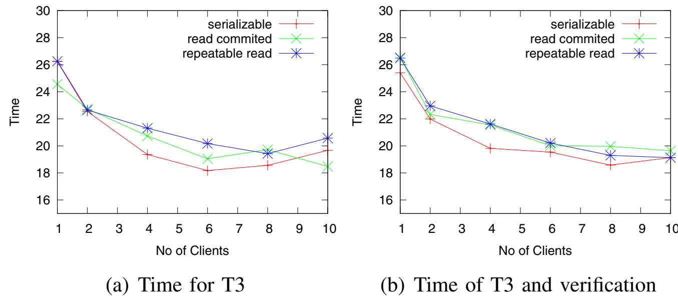
(a) Time for T1 (b) Time of T1 and verification

Figure 10: Insert (T1) time and verification overhead vs # of clients



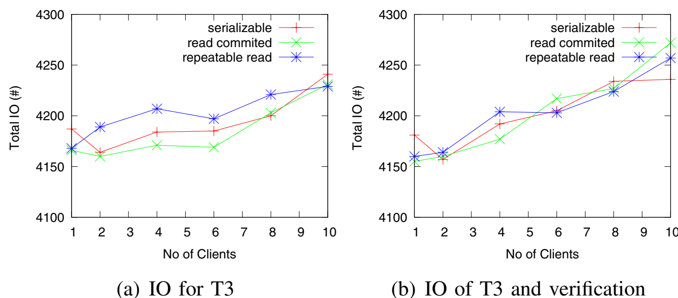
(a) IO for T1 (b) IO of T1 and verification

Figure 11: Insert (T1) IO and verification overhead vs # of clients



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

Figure 12: T3: Execution time and verification overhead vs # of clients



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

Figure 13: T3: IO and verification overhead vs # of clients

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.

VI. Conclusion

In this paper, we developed the first solutions for ensuring trustworthiness of a dynamic transactional database under different isolation levels without trusting the database server. With our solutions, the data owner can verify that the untrusted server executed the user transactions faithfully, under a given isolation level. Our solution allows multiple users to concurrently and independently execute transactions on the database. The isolation level for each transaction can be chosen independently of other transactions. 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.

References

[1] R. Jain and S. Prabhakar, "Trustworthy data from untrusted databases," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2013.

[2] R. Jain and S. 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.

[3] C. Papamanthou and R. Tamassia, "Update-optimal authenticated structures based on lattices," *IACR Cryptology ePrint Archive*, vol. 2010, p. 128, 2010.

[4] P. T. Devanbu, M. Gertz, C. Martel, and S. 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.

[5] F. Li, M. Hadjileftheriou, G. Kollios, and L. Reyzin, "Dynamic authenticated index structures for outsourced databases," in *Proceedings of ACM Special Interest Group on Management Of Data (SIGMOD)*, 2006.

[6] E. Mykletun, M. Narasimha, and G. Tsudik, "Authentication and integrity in outsourced databases," in *Network and Distributed System Security Symposium (NDSS)*, 2004.

[7] M. Narasimha and G. Tsudik, "Authentication of outsourced databases using signature aggregation and chaining," in *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*, 2006.

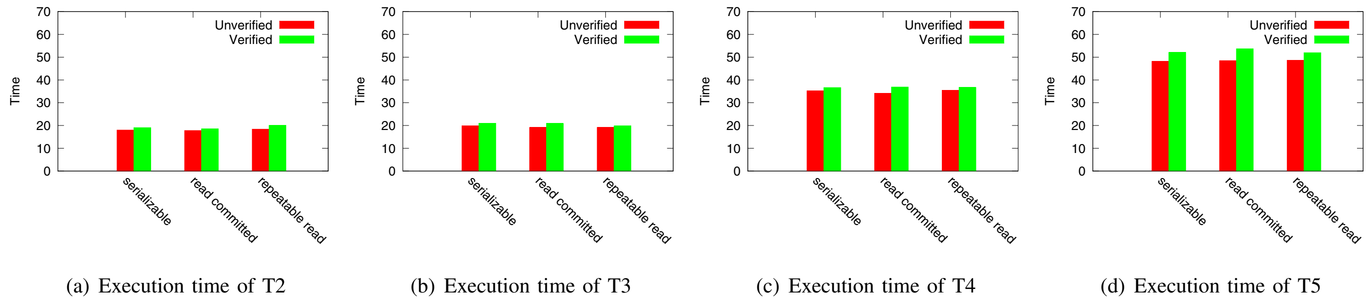
[8] H. Pang, A. Jain, K. Ramamritham, and K. lee Tan, "Verifying completeness of relational query results in data publishing," in *Proceedings of ACM Special Interest Group on Management Of Data (SIGMOD)*, 2005.

[9] H. Chen, X. Ma, W. Hsu, N. Li, and Q. Wang, "Access control friendly query verification for outsourced data publishing," in *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2008.

[10] A. Kundu and E. Bertino, "Structural signatures for tree data structures," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2008.

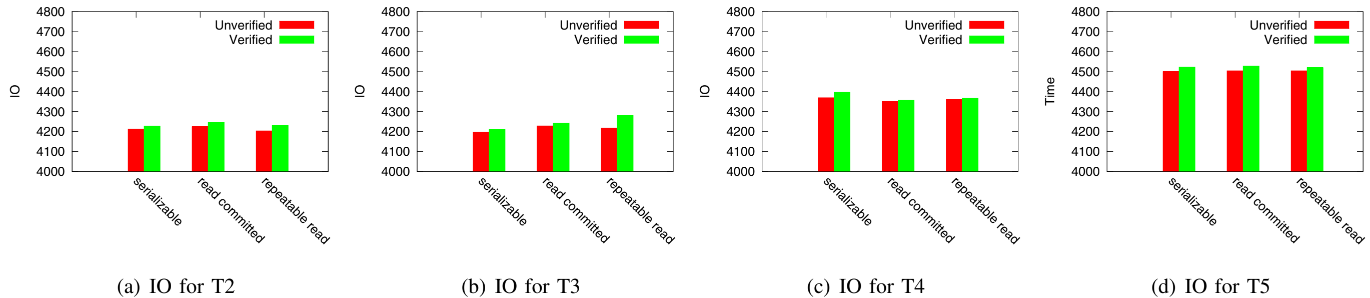
[11] P. Williams, R. Sion, and D. Shasha, "The blind stone tablet: Outsourcing durability," in *Network and Distributed System Security Symposium (NDSS)*, 2009.

[12] S. Bajaj and R. 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.



(a) Execution time of T2 (b) Execution time of T3 (c) Execution time of T4 (d) Execution time of T5

Figure 14: Execution time and verification overhead for different isolation levels



(a) IO for T2 (b) IO for T3 (c) IO for T4 (d) IO for T5

Figure 15: IO Cost and verification overhead for different isolation levels

[13] E. Mykletun and G. 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.

[14] D. Boneh, C. Gentry, B. Lynn, and H. 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.

[15] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Order preserving encryption for numeric data," in *Proceedings of ACM Special Interest Group on Management Of Data (SIGMOD)*, 2004.

[16] B. Hore, S. Mehrotra, and G. Tsudik, "A privacy-preserving index for range queries," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2004.

[17] F. Kerschbaum and J. Vayssiere, "Privacy-preserving data analytics as an outsourced service," in *Proceedings of the ACM International Computer and Communication Security Conference (CCS)*, 2008.

[18] H. Hacigümüş, B. Iyer, C. Li, and S. 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.

[19] B. Hore, S. Mehrotra, and G. Tsudik, "A privacy-preserving index for range queries," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2004.

[20] K. Fu, S. Kamara, and T. Kohno, "Key regression: Enabling efficient key distribution for secure distributed storage," in *Network and Distributed System Security Symposium (NDSS)*, 2006.

[21] M. J. Atallah, K. B. Frikken, and M. Blanton, "Dynamic and efficient key management for access hierarchies," in *Proceedings of the ACM International Computer and Communication Security Conference (CCS)*, 2005.

[22] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable secure file sharing on untrusted storage," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[23] G. Miklau and D. Suciu, "Controlling access to published data using cryptography," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2003.

[24] C. Cachin, A. Shelat, and A. Shraer, "Efficient fork-linearizable access to untrusted shared memory," in *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2007.

[25] D. Mazières and D. Shasha, "Building secure file systems out of byzantine storage," in *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2002.

[26] "American national standard for information systems - database language - sql," *ANSI X3.135-1992*, November 1992.