Isolation Levels in Popular Commercial DBMSs

Keqiang Li
East China Normal Unversity
kqli@stu.ecnu.edu.cn

In this file, we summarize the classic mechanisms that implement isolation levels (ILs) widely used in popular commercial DBMSs. In Sec. 1, we first discuss the definition of ILs. Then, we summarize the classic mechanisms of eliminating IL anomalies in Sec. 2. Finally, in Sec. 3, we analyze how to guarantee ILs provided in popular DBMSs by combing our four mechanisms.

Contents

1 Isolation Level Definitions

The concept of isolation level (IL) was first introduced in [1] with the name "degrees of consistency". It serves as a correctness contract between applications and DBMSs. The strongest IL is serializable, but it usually exhibits a relative poor performance. A weak IL, on the other hand, offers better performance but sacrifices the guarantees of a perfect isolation. For example, Snapshot Isolation allows some data corruptions to facilitate a high concurrency [2]. Thus, commercial DBMSs provide various ILs for a trade-off between consistency and performance [3].

Given so many different isolation levels, how to precisely define them has become a challenging and critical problem in the database community. The early ANSI standard [4] defines different isolation levels as preventing different phenomena. However, the ANSI SQL definitions fail to characterize several popular isolation levels, including the standard locking implementations of the levels. Instead, the following work [5] defines isolation levels as different locking strategies. For example, *read committed* should hold write lock to the end of the transaction and hold read lock to the end of the operation; *serializable* should hold both write and read locks to the end of the transaction.

However, this definition is criticized by following works as well because it does not work with databases that do not use lock implementation. To achieve both precision and generality, Adya et. al. define different isolation levels as preventing different types of cycles in the dependency serialization graph [6]. For example, *serializable* should prevent any cycles and *read committed* should prevent cycles consisting of *ww* and *wr* dependencies. Feketa et. al. extend this definition to *snapshot isolation*, showing that *snapshot isolation* should only allow cycles with at least two consecutive *rw* dependencies [2]. Generally speaking, there is no unified definition of isolation levels, and each DBMS has its own definition, as described in Section 3.

2 Isolation Anomalies Elimination Mechanisms

Data accesses by concurrent transactions potentially suffer from isolation anomalies. In this section, we first define the notations used in our paper. Then, we introduce the popular isolation anomalies. Next, we discuss how to eliminate these anomalies with concurrency control protocols. Finally, we summarize four classic mechanisms implementing concurrency control protocols in popular commercial DBMSs.

2.1 Notations

A database $\mathbb{D}$ has a set of data items, i.e., $x \in \mathbb{D}$. A transaction t consists of several operations typed read or write, ended with either commit or abort as a terminal operation. A write operation creates a new version for a record while a read operation queries a specific database snapshot. A database snapshot is consistent with versions created at the time of the snapshot creation. A commit installs all versions created by a transaction while an abort discards them. For a transaction t, we denote $r_t(rs)$ as a read in t with its read set rs, and denote $w_t(ws)$ as a write in t with its write set ws. Each element in rs (resp. ws) is an accessed version by the read operation (resp. the write operation). We denote x^i as the i^{th} version of record x, and x^{i+1} as its direct successor version. Table. 1 summarizes the notations used in this paper.

Table 1: Notations

Notations	**Description**
t	a transaction
x and x^i	a data item and the i^{th}(new) version of x
$r_t(rs)/w_t(ws)$	a read/write in t with read/write set rs/ws
c_t, a_t	a commit or abort in t
$ww/wr/rw$	a direct write-/read-/anti-dependency

ILs define the degree to which a transaction must be isolated from the modifications made by any other transaction. An isolation anomaly can be indicated by a specific transaction dependency pattern [6]. In general, there are three kinds transaction dependencies between any two committed transactions (denoted as t_m and t_n): 1) If t_m installs a version x^i and t_n installs the direct successor of x^i, i.e., x^{i+1}, t_n has a **direct write-dependency** (ww) on t_m. 2) If t_m installs a version x^i and t_n reads x^i, t_n has a **direct read-dependency** (wr) on t_m. 3) If t_m reads a version x^i and t_n installs the direct successor version of x^i, i.e., x^{i+1}, t_n has a **direct anti-dependency** (rw) on t_m.

2.2 Popular Isolation Anomalies

We introduce several isolation anomalies that are usually prohibited by ILs [5].

Dirty Write: Transaction t_0 modifies a data item. Another transaction t_1 then further modifies that data item before t_0 performs a commit or an abort. If t_0 or t_1 performs an abort, it is unclear what the correct data value should be.

Dirty Read: Transaction t_0 modifies a data item. Another transaction t_1 then reads that data item before t_0 performs a commit or an abort. If t_0 then performs an abort, t_1 has read a data item that is never committed and so never really exist.

Non-repeatable Read: Transaction t_0 reads a data item. Another transaction t_1 then modifies or deletes that data item and commits. If t_0 then attempts to reread the data item, it receives a modified value or discovers that the data item has been deleted.

Phantom: Transaction t_0 reads a set of data items satisfying some <search condition>. Transaction t_1 then creates data items that satisfy <search condition> of t_0 and commits. If t_0 repeats its read with the same <search condition>, it gets a set of data items different from its first read.

Read Skew: Suppose there are two data items x_0 and x_1. Transaction t_0 reads x_0, and then a second transaction t_1 updates x_0 and x_1 to new versions and commits. If now t_0 reads the version of x_1 created by t_1, it sees an inconsistent state of the database.

Lost Update: The lost update anomaly occurs when transaction t_0 reads a data item and then t_1 updates the data item (possibly based on a previous read), then t_0 (based on its earlier read value) updates the data item and commits.

Write Skew: Suppose there are two data items x_0 and x_1. Transaction t_0 reads x_0 and x_1, and then a transaction t_1 reads x_0 and x_1, writes x_0, and commits. Then t_0 writes x_1. If there were a constraint between x_0 and x_1, it might be violated.

Serialization Anomaly: The result of successfully committing a group of transactions is inconsistent with all possible orderings of running those transactions one at a time.

2.3 Concurrency Control Protocols

We describe how the commonly used concurrency control protocols eliminate the above isolation anomalies.

Two-phase Locking(2PL): In a transaction, 2PL specifies that locks are acquired and released in two phases, that is, growing phase and shrinking phase. Growing phase specifies that locks are acquired and no locks are released, while shrinking phase specifies that locks are released and no locks are acquired. Note that, commercial DBMSs usually take a variant of 2PL, called strict two-phase locking. Specifically, strict two-phase locking requires that all locks that a transaction has acquired are held until the transaction terminates.

Multi-version Concurrency Control (MVCC): MVCC requires that the DBMS maintains multiple physical versions of each logical data item as an ordered version chain. Specifically, in a transaction, write operations append a new version into the version chain of a data item, and read operations see the newest versions that exist when the transaction starts.

Serializable Snapshot Isolation (SSI): SSI provides serializability using snapshot isolation, by detecting potential anomalies at runtime and aborting transactions as necessary.

Optimistic Concurrency Control (OCC): OCC specifies that the DBMS executes a transaction in three phases: *read*, *validation*, and *write*. In the *read* phase, the transaction performs read and write operations to data items without blocking. When the transaction finishes execution, it enters the *validation* phase where the DBMS checks whether the transaction conflicts with any other active transaction. If there are no conflicts, the transaction enters the *write* phase where the DBMS propagates the changes in the transaction's write set to the database and makes them visible to other transactions.

Timestamp Ordering (TO): TO uses timestamps to determine the serializability order of transactions. Specifically, each transaction t is assigned a unique fixed timestamp that is monotonically increasing, denoted as $ts(t)$. Every data item x is tagged with timestamp of the last transaction that successfully does the read/write, denoted as $r\text{-}ts(x)/w\text{-}ts(x)$. If $ts(t) < r\text{-}ts(x)$ or $ts(t) < w\text{-}ts(x)$, this violates timestamp order of transaction t with regard to the reader or writer of data item x, then abort t.

Percolator [1]: *Percolator* [7] provides transactions with snapshot-isolation semantics. Snapshot isolation provides multiple versions to see a consistent view of database as of

[1]Strictly speaking, *percolator* is a transaction processing framework, not just a concurrency control protocol. This file mainly focuses on the implementation details of the isolation level of *percolator*.

transactions starting. Specifically, each version is tagged with a installation timestamp, and each transaction is tagged with a start timestamp and a commit timestamp. To see a consistent view of database, each transaction queries the latest versions as of the start timestamp. Additionally, snapshot isolation protects against write-write conflicts: if two transactions, running concurrently, write to the same data item, at most one will commit. Before a transaction committing, *percolator* checks whether a newer version has been installed as of the start timestamp of the transaction.

2.4 Classic Implementation Mechanisms

Although there are various concurrency control protocols (*CCP*) as shown in the above section, we discover that almost all CCPs in commercial DBMSs can be implemented by assembling the following four classic mechanisms: *consistent read* (*CR*), *first updater wins* (*FUW*), *serialization certifier* (*SC*) and *mutual exclusion* (*ME*), as we summarized in Tab. 2. In Tab. 3, we summarize the correlations between isolation anomalies and the four classic mechanisms. Note that, there still exist some other mechanisms [8, 9, 10, 11, 12, 13], almost all of which stay only in the academic papers instead of in practical products.

Table 2: Concurrency Control Protocols Using Classic Mechanisms

CCP	**Mechanism**
MVCC	CR
2PL	ME
OCC	SC
TO	SC
SSI	CR+ME+FUW+SC
Percolator	SC

Table 3: Four Classic Mechanisms Eliminating Isolation Anomalies

Mechanism	**Anomaly**
CR	Read Skew, Dirty Read
ME	Dirty Write, Dirty Read, Non-repeatable Read, Phantom
FUW	Lost Update, Dirty Write
SC	Serialization Anomaly

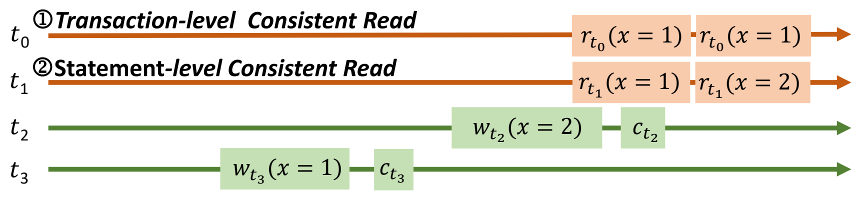

Figure 1: Example for Consistent Read

2.4.1 Consistent read

Consistent Read (CR) provides a consistent view of the database at a specific time point. To provide different ILs, there are transaction-level CR and statement-level CR. Transaction-level CR (resp. Statement-level CR) provides a consistent view of the database at the beginning of a transaction (resp. an operation). Fig. 1 shows an example of the two types of CR. There are four transactions, i.e., t_{0-3}, operating on record x, among which t_0(resp. t_1) takes transaction-level (resp. statement-level) CR. The two reads in t_0 see the same snapshot generated by t_3 for the requirement of transaction-level CR; the two reads in t_1 see different values, i.e., $x = 1$ and $x = 2$, constrained by statement-level CR.

As described above, *read skew* indicates a kind of anomalies where a transaction sees an inconsistent state of the database. To eliminate *read skew*, most popular commercial DBMSs take the mechanism of CR to see the committed version of a record of a specific time point. Specifically, by maintaining multiple physical versions of each logical record, a read operation sees the changes made by other transactions committed before the specific time point and the changes made by earlier operations within the same transaction. Note that, as described above, *dirty read* indicates a read operation sees a temporary result of a concurrent write operation, which means that there is no *dirty read* when eliminating *read skew*.

2.4.2 Mutual exclusion

Mutual exclusion (ME) uses the locking strategy to provide a kind of exclusive access to a shared resource. There are two modes of locking, i.e., shared and exclusive. The exclusive mode is imcompatible with others and the shared mode is mutually compatible. According to the locking object, there are two granularities of locking, i.e., range-level lock and record-level lock. We discuss how popular commercial DBMSs takes the mechanism of ME to eliminate *dirty write*, *dirty read*, *non-repeatable read* and *phantom* as follows:

1. As described above, *dirty write* indicates two concurrent operations modify the same record. To eliminate *dirty write*, most popular commercial DBMSs take ME to exclusively modify a record. Specifically, in a transaction, each write operation should hold an exclusive record-level lock on the modified record until the transaction terminated.

2. As described above, *dirty read* indicates a read operation sees a temporary result of a concurrent write operation. To eliminate *dirty read*, some commercial DBMSs take ME to exclusively see a committed result. Specifically, in a transaction, each

read operation should hold a shared record-level lock on the access record until the read operation terminated.

3. As described above, *non-repeatable read* indicates there is a concurrent write operation between two read operations in a transaction. To eliminate *non-repeatable read*, most popular commercial DBMSs take ME to exclusively access a record. Specifically, in a transaction, each read operation should hold a shared record-level lock on the access record until the transaction terminated.
4. As described above, *phantom* indicates there is a write operation inserting a new record that falls into the <search condition> of a concurrent read operation. To eliminate *phantom*, most popular commercial DBMSs take ME to exclusively access the <search condition> of a read operation. Specifically, in a transaction, each read operation should hold a shared range-level lock on the the <search condition> until the transaction terminated.

2.4.3 First updater wins

First updater wins (FUW) prevents transactions which modify the same record from concurrent execution. As described above, *lost update* indicates a transaction does not see the update of another transaction before updating the same record itself. First updater wins *lost update*, most popular commercial DBMSs take FUW to ensure that all transactions modifying the same record execute in a serial order. Specifically, the target record of an update operation might have already been modified by another concurrent transaction by the time it is found. In this case, the second updater will wait for the first updater to commit or abort (if it is still in progress). If the updater aborts, then its effects are negated and the second updater can proceed with modifying the originally found record. But if the first updater commits, then the second updater will abort.

2.4.4 Serialization certifier

The concurrency control protocols (CCP) in popular DBMSs often take advantage of the *serialization certifier* (SC) mechanism to guarantee that the executed transactions are conflict serializable [14]. However, each concurrency control protocol has its specific "certifier". Table 4 lists the the certifiers used by popular DBMSs. Next, we discuss how each certifier is used.

The *serializable snapshot isolation* (*SSI*) protocol implements the *SC* mechanism based on *snapshot isolation*, and uses the detection of the *write skew* anomalies as its certifier. Specifically, the *write skew* anomaly can be efficiently detected by checking whether there exist two consecutive *rw* dependencies [15]. Thus, the certifier would abort one of the three transactions if there exist two *rw* dependencies between them.

The *timestamp ordering* (*TO*) protocol uses the transaction timestamp ordering as its certifier when implementing its *SC* mechanism. Specifically, the certifier checks whether a transaction with an older timestamp has a dependency on another transaction with a newer timestamp. If so, it would abort the old transaction.

Table 4: Serialization Certifier

DBMS	CCP	Certifier
PostgreSQL OpenGauss yugabyteDB	SSI	detecting write skew anomalies based on snapshot isolation
CockroachDB	TO	ordering transactions as a monotonically increase of timestamps
RocksDB FoundationDB	OCC	checking the transaction conflicts with other active ones
TiDB	Percolator	detecting write-write conflicts before committing a transaction

The *optimistic concurrency control* (*OCC*) protocol specifies that a transaction in the DBMS is executed in three phases: *read*, *validation*, and *write*. In the *read* phase, the transaction performs read and write operations on records without blocking. When the transaction prepares to commit, the *validation* phase uses the conflicts checking as a certifier to guarantee the conflict serializability. If there exist conflicts, the certifier would abort the transaction; otherwise, it commits the transaction in *write* phase.

The *percolator* [7] protocol[2] also implements the idea of *serialization certifier* mechanism. Similar to the three concurrency control protocols mentioned above, *percolator* use its certifier to eliminate some specific anomalies when the transaction starts to commit. However, the certifier of *percolator* only guarantees snapshot isolation, and could not guarantee the conflict serializability. Specifically, *percolator* uses the check of concurrent versions as a certifier to eliminate the lost update anomaly. That is, when a transaction starts to commit, the certifier checks whether a newer version has been installed since the transaction begins. If there exist concurrent versions, the certifier would abort the transaction; otherwise, it commits the transaction. We consider *percolator* as a special case of implementing the *serialization certifier* mechanism.

3 Isolation Level Implementation Mechanisms

A DBMS usually mixes up multiple concurrency control protocols (CCP) to eliminate isolation anomalies that should be prevented according to the definition of isolation levels (ILs). The widely used ILs in DBMSs include *read committed* (RC), *repeatable read* (RR), *snapshot isolation* (SI) and *serializable* (SR). In this section, we mainly discuss how to use the four implementation mechanisms, i.e., *consistent read* (CR), *mutual exclusion* (ME), *first updater wins* (FUW) and *serialization certifier* (SC), to guarantee ILs in the popular commercial DBMSs, as summarized in Table. 5.

[2]Strictly speaking, *percolator* is a transaction processing framework, not only a concurrency control protocol.

Table 5: Isolation Level Implementations in DBMSs

DBMS	CCP	IL	CR	ME	FUW	SC
PostgreSQL [15], OpenGauss [16], yugabyteDB [17]	2PL+MVCC +SSI	SR	✓	✓	✓	✓
		SI	✓	✓	✓	
		RC	✓	✓		
InnoDB [18], Aurora [19], PolarDB [20], SQL server [21]	2PL+MVCC	SR,RR, RC	✓	✓		
TiDB [22]	2PL+MVCC	RR,RC	✓	✓		
	Percolator [23]	SI	✓			✓
RocksDB [24]	2PL+MVCC	SI	✓	✓	✓	
	OCC+MVCC	SI	✓			✓
SQLite [25]	2PL	SR		✓		
FoundationDB [26]	OCC+MVCC	SR	✓			✓
SingleStore [27]	2PL+MVCC	RC	✓	✓		
CockroachDB [28]	TO+MVCC	SR	✓			✓
Spanner [29]	2PL+MVCC	SR	✓	✓		
Oracle [30], SAP HANA [31], NuoDB [32], Oceanbase [33]	2PL+MVCC	SI	✓	✓	✓	
		RC	✓	✓		

3.1 PostgreSQL

In PostgreSQL document [34], *read committed* is defined as:

1. **Eliminating Dirty Read** A read operation sees a snapshot of the database as of the instant the read operation begins to run.

2. **Eliminating Dirty Write** The would-be writer will wait for the previous writing transaction to commit or abort (if it is still in progress). If the previous writer aborts, then its effects are negated and the would-be writer can proceed with modifying the originally found record. If the previous writer commits, the would-be writer will attempt to apply its operation to the modified version of the record.

Snapshot isolation[3] is defined as:

1. **Eliminating Read Skew** A read operation sees a snapshot as of the start of the first non-transaction-control statement in the transaction.

[3]It is called repeatable read on PostgreSQL official website, but it is essentially a snapshot isolation [15].

2. **Eliminating Lost Update&Dirty Write** The would-be writer will wait for the previous writing transaction to commit or abort (if it is still in progress). If the previous writer aborts, then its effects are negated and the would-be writer can proceed with modifying the originally found record. If the previous writer commits, the would-be writer will be aborted[4].

Serializable is defined as:

1. **Eliminating Serialization Anomaly** SR is implemented using a concurrency control protocol known in academic literature as *serializable snapshot isolation*, which builds on snapshot isolation by adding checks for serialization anomalies.

As discussed above, in Table 3, we summarize the corresponding relationships between isolation anomalies and the four classic implementation mechanisms. According to the above definitions, we map the classic mechanisms to guarantee ILs of PostgreSQL as follows:

1. *Read committed* in PostgreSQL takes *consistent read* to eliminate *dirty read*, and takes *mutual exclusion* to eliminate *dirty write.*
2. *Snapshot isolation* in PostgreSQL takes *consistent read* to eliminate *read skew*, takes *mutual exclusion* to eliminate *dirty write*, and takes *first updater wins* to eliminate *lost update.*
3. *Serialzable* in PostgreSQL is based on snapshot isolation except that it takes *serialization certifier* to eliminate *serialization anomalies.*

3.2 YugabyteDB

In YugabyteDB document [35], *read committed* is defined as:

1. **Eliminating Dirty Read** Every statement in the transaction will see all data that has been committed before it is issued.
2. **Eliminating Dirty Write** The would-be writer will wait for the previous writing transaction to commit or abort (if it is still in progress). If the previous writer aborts, then its effects are negated and the would-be writer can proceed with modifying the originally found record. If the previous writer commits, the would-be writer will attempt to apply its operation to the modified version of the record.

Snapshot isolation[5] is defined as:

1. **Eliminating Read Skew** The snapshot isolation level only sees data committed before the transaction begins. Transactions running under snapshot isolation do not see either uncommitted data or changes committed during transaction execution by

[4]The definition is more strict than *read committed*, so it also eliminates *dirty write.*

[5]It is called repeatable read on PostgreSQL official website, but it is essentially a snapshot isolation [15].

other concurrently running transactions. Note that the query does see the effects of previous updates executed within its own transaction, even though they are not yet committed.

2. **Eliminating Lost Update&Dirty Write** INSERT, UPDATE, and DELETE commands behave the same as SELECT in terms of searching for target rows. They will only find target rows that are committed as of the transaction start time. If such a target row might have already been updated (or deleted or locked) by another concurrent transaction by the time it is found. This scenario is called a transaction conflict, where the current transaction conflicts with the transaction that makes (or is attempting to make) an update. In such cases, one of the two transactions get aborted, depending on priority[6].

Serializable is defined as:

1. **Eliminating Serialization Anomaly** The serializable isolation is implemented using a concurrency control protocol known in academic literature as *serializable snapshot isolation*, which builds on snapshot isolation by adding checks for serialization anomalies. The serializable isolation provides the strictest transaction isolation. This level emulates serial transaction execution for all committed transactions; as if transactions have been executed one after another, serially, rather than concurrently. Serializable isolation can detect read-write conflicts in addition to write-write conflicts. This is accomplished by writing provisional records for read operations as well.

As discussed above, in Table 3, we summarize the corresponding relationships between isolation anomalies and the four classic implementation mechanisms. According to the above definitions, we map the classic mechanisms to guarantee ILs of YugabyteDB as follows:

1. *Read committed* in YugabyteDB takes *consistent read* to eliminate *read skew*, and takes *mutual exclusion* to eliminate *dirty write.*

2. *Snapshot isolation* in YugabyteDB takes *consistent read* to eliminate *read skew*, takes *mutual exclusion* to eliminate *dirty write*, and takes *first updater wins* to eliminate *lost update.*

3. *Serialzable* in YugabyteDB is based on snapshot isolation except that it takes *serialization certifier* to eliminate *serialization anomalies.*

3.3 OpenGauss

As described OpenGauss document [36], OpenGauss is compatible with PostgreSQL. The IL definitions of OpenGauss are consistent with PostgreSQL except that *serializble* is not supported. If we set IL as *serializble*, it actually takes *repeatable read.*

[6]The definition is more strict than *read committed*, so it also eliminates *dirty write.*

3.4 MySQL InnoDB

In MySQL InnoDB document [18], *read committed* is defined as:

1. **Eliminating Dirty Read** Each read operation (SELECT statement), even within the same transaction, sets and reads its own fresh snapshot.
2. **Eliminating Dirty Write** Each write operation (UPDATE, DELETE, INSERT statement) locks the record accessed in the exclusive mode.

Repeatable read is defined as:

1. **Eliminating Read Skew** The read operations (SELECT statement) within the same transaction read the snapshot established by the first read operation.
2. **Eliminating Dirty Write** Each write operation (UPDATE, DELETE, INSERT statement) locks its <search condition> in the exclusive mode.

Serializable is defined as:

1. **Eliminating Phantom&Non-repeatable Read&Dirty Read** a read operation (SELECT statement) locks its <search condition> in the shared mode.
2. **Eliminating Dirty Write** Each write operation (UPDATE, DELETE, INSERT statement) locks its <search condition> in the exclusive mode.

As discussed above, in Table 3, we summarize the corresponding relationships between isolation anomalies and the four classic implementation mechanisms. According to the above definitions, we map the classic mechanisms to guarantee ILs of MySQL InnoDB as follows:

1. *Read committed* in MySQL InnoDB takes *consistent read* to eliminate *dirty read*, and takes *mutual exclusion* to eliminate *dirty write*.
2. *Repeatable read* in MySQL InnoDB takes *consistent read* to eliminate *read skew*, and takes *mutual exclusion* to eliminate *dirty write*.
3. *Serialzable* in MySQL InnoDB takes *mutual exclusion* to eliminate *dirty write*, *dirty read*, *non-repeatable read* and *phantom*.

3.5 Aurora&PolarDB

Since both Aurora [19] and PolarDB [20] are compatible with MySQL InnoDB, the definitions of their ILs are consistent with MySQL InnoDB.

3.6 SQL Server

In SQL Server document [37], *read committed* is defined as:

1. **Dirty Read** The DBMS uses row versioning to present each statement with a transactionally consistent snapshot of the data as it exists at the start of the statement.
2. **Dirty Write** The DBMS uses exclusive locks to prevent other transactions from modifying rows while the current transaction is running a write operation.

Repeatable read is defined as:

1. **Non-repeatable Read&Dirty Read** It specifies that statements cannot read data that has been modified but not yet committed by other transactions and that no other transactions can modify data that has been read by the current transaction until the current transaction completes.
2. **Dirty Write** The DBMS uses exclusive locks to prevent other transactions from modifying rows while the current transaction is running a write operation.

Snapshot[7] is defined as:

1. **Read Skew** It specifies that data read by any statement in a transaction will be the transactionally consistent version of the data that exists at the start of the transaction.
2. **Dirty Write** The DBMS uses exclusive locks to prevent other transactions from modifying rows while the current transaction is running a write operation.

Serialzable is defined as:

1. **Eliminating Phantom&Non-repeatable Read&Dirty Read** Statements cannot read data that has been modified but not yet committed by other transactions. No other transactions can modify data that has been read by the current transaction until the current transaction completes. Other transactions cannot insert new rows with key values that would fall in the range of keys read by any statements in the current transaction until the current transaction completes.
2. **Dirty Write** The DBMS uses exclusive locks to prevent other transactions from modifying rows while the current transaction is running a write operation.

As discussed above, in Table 3, we summarize the corresponding relationships between isolation anomalies and the four classic implementation mechanisms. According to the above definitions, we map the classic mechanisms to guarantee ILs of SQL Server as follows:

1. *Read committed* in SQL Server takes *mutual exclusion* to eliminate *dirty write* and *dirty read.*

[7]It is the same with *repeatable read* of MySQL. In Table. 5, we call it *repeatable read.*

2. *Repeatable read* in SQL Server takes *mutual exclusion* to eliminate *dirty write*, *dirty read* and *non-repeatable read.*

3. *Snapshot* in SQL Server takes *consistent read* to eliminate *read skew*, and takes *mutual exclusion* to eliminate *dirty write.*

4. *Serialzable* in SQL Server takes *mutual exclusion* to eliminate *dirty write*, *dirty read*, *non-repeatable read* and *phantom.*

3.7 TiDB

In TiDB paper [22], there are two transaction modes, including optimistic mode and pessimistic mode. They are adapted from Percolator [23], which selects one key as the primary key and uses it to stand for the status of a transaction, and bases on two-phase commit protocol to conduct transactions. With optimistic transactions, conflicting changes are detected as part of a transaction commit, while they are detected before operation execution with pessimistic transactions.

In optimistic mode, *snapshot isolation* is defined as:

1. **Eliminating Read Skew** Percolator stores multiple versions of each record. Each transaction reads from a stable snapshot at some timestamp.

2. **Eliminating Lost Update** If two concurrent transactions A and B write to the same cell, at most one will commit[8].

In pessimistic mode, *read committed* is defined as:

1. **Eliminating Dirty Read** Each read operation, even within the same transaction, sets and reads its own fresh snapshot.

2. **Eliminating Dirty Write** Each write operation locks the record accessed in the exclusive mode.

In pessimistic mode, *repeatable read* is defined as:

1. **Eliminating Read Skew** The Repeatable Read isolation level only sees data committed before the transaction begins, and it never sees either uncommitted data or changes committed during transaction execution by concurrent transactions.

2. **Eliminating Dirty Write** Each write operation locks the record accessed in the exclusive mode.

As discussed above, in Table 3, we summarize the corresponding relationships between isolation anomalies and the four classic implementation mechanisms. According to the above definitions, we map the classic mechanisms to guarantee ILs of TiDB as follows:

[8]Percolator takes first committer wins to avoid lost update. First committer wins differs from first updater wins in that the time to check *lost update*, as described in [5].

1. *Read committed* in TiDB takes *consistent read* to eliminate *dirty read*, and takes *mutual exclusion* to eliminate *dirty write*.
2. *Repeatable read* in TiDB takes *consistent read* to eliminate *read skew*, and takes *mutual exclusion* to eliminate *dirty write*.
3. *Snapshot Isolation* in TiDB takes *consistent read* to eliminate *read skew*, and takes *serialization certifier* to eliminate *lost update*.

3.8 RocksDB

As described in RocksDB document [38], RocksDB supports two kinds of transactions, i.e., pessimistic transaction and optimistic transaction, both of which support *snapshot isolation*.

Under pessimistic transaction, *snapshot isolation* is defined as:

1. **Eliminating Read Skew** A snapshot captures a point-in-time view of the database at the time it's created.
2. **Eliminating Lost Update&Dirty Write** All keys that are written are locked internally by RocksDB to perform conflict detection.

Under optimistic transaction, *snapshot isolation* is defined as:

1. **Eliminating Read Skew** A snapshot captures a point-in-time view of the database at the time it's created.
2. **Eliminating Lost Update** Optimistic transactions do not take any locks when preparing writes. Instead, they rely on doing conflict-detection at commit time to validate that no other writers have modified the keys being written by the current transaction. If there is a conflict with another write (or it cannot be determined), the commit will return an error and no keys will be written.

As discussed above, in Table 3, we summarize the corresponding relationships between isolation anomalies and the four classic implementation mechanisms. According to the above definitions, we map the classic mechanisms to guarantee ILs of RocksDB as follows:

1. Under pessimistic transaction, *snapshot isolation* in RocksDB takes *consistent read* to eliminate *read skew*, takes *mutual exclusion* to eliminate *dirty write*, and takes *first updater wins* to eliminate *lost update*.
2. Under optimistic transaction, *snapshot isolation* in RocksDB takes *consistent read* to eliminate *read skew*, takes *mutual exclusion* to eliminate *dirty write*, and takes *first updater wins* to eliminate *lost update*.

3.9 SQLite

In SQLserver document [39], *read committed* is defined as:

1. **Eliminating Phantom&Non-repeatable Read&Dirty Read&Dirty Write** SQLite follows strict two phase locking, and implements a very simple database-level locking protocol which allows multiple readers but only one writer in one database at a time.

As discussed above, in Table 3, we summarize the corresponding relationships between isolation anomalies and the four classic implementation mechanisms. According to the above definitions, we map the classic mechanisms to guarantee ILs of Sqlite as follows:

1. *Serialzable* in MySQL takes *mutual exclusion* to eliminating *dirty write*, *dirty read*, *non-repeatable read* and *phantom.*

3.10 FoundationDB

In FoundationDB paper [37], *serializable* is defined as:

1. **Eliminating Read Skew&Dirty Read&Serialization Anomaly** FoundationDB implements serializable by combining optimistic concurrency control with multi-version concurrency control.

As discussed above, in Table 3, we summarize the corresponding relationships between isolation anomalies and the four classic implementation mechanisms. According to the above definitions, we map the classic mechanisms to guarantee ILs of FoundationDB as follows:

1. *Serialzable* in FoundationDB it takes *consistent read* to eliminate *read skew* and *dirty read*, and takes *serialization certifier* to eliminate *serialization anomalies.*

3.11 SingleStore

In SingleStore document [40], *read committed* is defined as:

1. **Dirty Read** It guarantees that no transaction will read any uncommitted data from another transaction. This does not guarantee that a row will remain the same for every read query in a given transaction.
2. **Dirty Write** It uses exclusive locks to prevent other transactions from modifying rows while the current transaction is running a write operation.

As discussed above, in Table 3, we summarize the corresponding relationships between isolation anomalies and the four classic implementation mechanisms. According to the above definitions, we map the classic mechanisms to guarantee ILs of SingleStore as follows:

1. *Read committed* in SingleStore takes *consistent read* to eliminate *dirty read*, and takes *mutual exclusion* to eliminate *dirty write.*

3.12 CockroachDB

In CockroachDB document [41], *read committed* is defined as:

1. **Eliminating Serialization Anomaly** A transaction behaves as though it has the entire database all to itself for the duration of its execution. This means that no concurrent writers can affect the transaction unless they commit before it starts, and no concurrent readers can be affected by the transaction until it has successfully committed.

As discussed above, in Table 3, we summarize the corresponding relationships between isolation anomalies and the four classic implementation mechanisms. According to the above definitions, we map the classic mechanisms to guarantee ILs of CockroachDB as follows:

1. *Serialzable* in CockroachDB it takes *serialization certifier* to eliminate *serialization anomalies.*

3.13 Spanner

In Spanner document [42], *serializable* is defined as:

1. **Eliminating Read Skew** Spanner uses multi-versioned concurrency control. Spanner's MVCC implementation is unique in that it uses hardware devices (e.g., GPS, atomic clocks) for high-precision clock synchronization. Spanner uses these clocks to assign timestamps to transactions to enforce consistent views of its multi-version database over wide-area networks.
2. **Eliminating Phantom&Non-repeatable Read&Dirty Read&Dirty Write** Spanner uses a combination of shared locks and exclusive locks to control access to the data. When you perform a read as part of a transaction, Spanner acquires shared read locks, which allows other reads to still access the data until your transaction is ready to commit. When your transaction is committing and writes are being applied, the transaction attempts to upgrade to an exclusive lock. It blocks new shared read locks on the data, waits for existing shared read locks to clear, then places an exclusive lock for exclusive access to the data.

As discussed above, in Table 3, we summarize the corresponding relationships between isolation anomalies and the four classic implementation mechanisms. According to the above definitions, we map the classic mechanisms to guarantee ILs of Spanner as follows:

1. *Serializable* in Spanner takes *consistent read* to eliminate *read skew*, and takes *mutual exclusion* to eliminate *dirty write*, *dirty read*, *non-repeatable read* and *phantom.*

3.14 Oracle

In Oracle document [41], *read committed* is defined as:

1. **Eliminating Dirty Read** Every read operation executed by a transaction sees only data committed before the query—not the transaction—began.

2. **Eliminating Dirty Write** The transaction that prevents the row modification is sometimes called a blocking transaction. The read committed transaction waits for the blocking transaction to end and release its row lock. If the blocking transaction aborts, then the waiting transaction proceeds to change the previously locked record as if the other transaction never existed. If the blocking transaction commits and releases its locks, then the waiting transaction proceeds with its intended update to the newly changed record.

Serializable[9] is defined as:

1. **Eliminating Read Skew** A transaction sees only changes committed at the time the transaction—not the query—began and changes made by the transaction itself.

2. **Eliminating Lost Update&Dirty Write** Oracle Database permits a serializable transaction to modify a row only if changes to the row made by other transactions were already committed when the serializable transaction began[10].

As discussed above, in Table 3, we summarize the corresponding relationships between isolation anomalies and the four classic implementation mechanisms. According to the above definitions, we map the classic mechanisms to guarantee ILs of Oracle as follows:

1. *Read committed* in Oracle takes *consistent read* to eliminate *dirty read*, and takes *mutual exclusion* to eliminate *dirty write*.

2. *Serialzable* in Oracle takes *consistent read* to eliminate *read skew*, takes *mutual exclusion* to eliminate *dirty write*, and takes *first updater wins* to eliminate *lost update*.

3.15 NuoDB

In NuoDB document [43], *read committed* is defined as:

1. **Eliminating Dirty Read** Every read operation executed by a transaction sees only data committed before the query—not the transaction—began.

2. **Eliminating Dirty Write** The transaction that prevents the row modification is sometimes called a blocking transaction. The read committed transaction waits for the blocking transaction to end and release its row lock. If the blocking transaction

[9]It is essentially a snapshot isolation as defined in [5].

[10]The definition is more strict than *read committed*, so it also eliminates *dirty write*.

aborts, then the waiting transaction proceeds to change the previously locked record as if the other transaction never existed. If the blocking transaction commits and releases its locks, then the waiting transaction proceeds with its intended update to the newly changed record.

Serializable[11] is defined as:

1. **Eliminating Read Skew** A transaction reads a snapshot of the database at the start of the transaction.
2. **Eliminating Lost Update&Dirty Write** The transaction can perform updates, including deletes, successfully on those rows, providing no other concurrent transaction has updated those same rows. When a transaction running at this isolation level attempts to update or delete a record that has been changed by a concurrent transaction, it waits for the other transaction to complete. When the other transaction completes then the waiting transaction succeeds in its update only if the previous transaction rolls back. Otherwise, the waiting transaction gets an error.

As discussed above, in Table 3, we summarize the corresponding relationships between isolation anomalies and the four classic implementation mechanisms. According to the above definitions, we map the classic mechanisms to guarantee ILs of NuoDB as follows:

1. *Read committed* in NuoDB takes *consistent read* to eliminate *dirty read*, and takes *mutual exclusion* to eliminate *dirty write.*
2. *Serialzable* in NuoDB takes *consistent read* to eliminate *read skew*, takes *mutual exclusion* to eliminate *dirty write*, and takes *first updater wins* to eliminate *lost update.*

3.16 SAP HANA

In SAP HANA document [44], *read committed* is defined as:

1. **Eliminating Dirty Read** The boundary of the lifetime of the MVCC snapshot is the query itself.
2. **Eliminating Dirty Write** The transaction that prevents the row modification is sometimes called a blocking transaction. The read committed transaction waits for the blocking transaction to end and release its row lock. If the blocking transaction aborts, then the waiting transaction proceeds to change the previously locked record as if the other transaction never existed. If the blocking transaction commits and releases its locks, then the waiting transaction proceeds with its intended update to the newly changed record.

Serializable[12] is defined as:

[11]It is essentially a snapshot isolation as defined in [5].

[12]It is essentially a snapshot isolation as defined in [5].

1. **Eliminating Read Skew** The transaction token can be reused for the queries belonging to the same transaction.

2. **Eliminating Lost Update&Dirty Write** Database permits a serializable transaction to modify a row only if changes to the row made by other transactions were already committed when the serializable transaction began[13].

As discussed above, in Table 3, we summarize the corresponding relationships between isolation anomalies and the four classic implementation mechanisms. According to the above definitions, we map the classic mechanisms to guarantee ILs of SAP HANA as follows:

1. *Read committed* in SAP HANA takes *consistent read* to eliminate *dirty read*, and takes *mutual exclusion* to eliminate *dirty write*.

2. *Serialzable* in SAP HANA takes *consistent read* to eliminate *read skew*, takes *mutual exclusion* to eliminate *dirty write*, and takes *first updater wins* to eliminate *lost update*.

3.17 OceanBase

Since OceanBase [33] is compatible with Oracle, the definitions of their ILs is consistent with Oracle.

References

[1] R Lorie J Gray, GF Putzolu, and IL Traiger. Granularity of locks and degrees of consistency. *Modeling in Data Base Management Systems, GM Nijssen ed., North Holland Pub*, 1976.

[2] Alan D. Fekete, Dimitrios Liarokapis, Elizabeth J. O'Neil, Patrick E. O'Neil, and Dennis E. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[3] Andrew Pavlo. What are we doing with our lives? nobody cares about our concurrency control research. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 3–3, 2017.

[4] ANSI X3. American national standard for information systems-database language-sql, 1992.

[5] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

[13]The definition is more strict than *read committed*, so it also eliminates *dirty write*.

[6] Atul Adya and Barbara H Liskov. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions.* PhD thesis, Massachusetts Institute of Technology, 1999.

[7] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[8] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[9] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1629–1642, 2016.

[10] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: a fast and practical deterministic oltp database. *Proceedings of the VLDB Endowment*, 2020.

[11] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2012.

[12] Jiachen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. Polyjuice: High-performance transactions via learned concurrency control. In *OSDI*, pages 198–216, 2021.

[13] Dixin Tang and Aaron J Elmore. Toward coordination-free and reconfigurable mixed concurrency control. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 809–822, 2018.

[14] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery.* Elsevier, 2001.

[15] Dan R. K. Ports and Kevin Grittner. Serializable snapshot isolation in postgresql. *Proc. VLDB Endow.*, 5(12):1850–1861, 2012.

[16] Guoliang Li, Xuanhe Zhou, Ji Sun, Xiang Yu, Yue Han, Lianyuan Jin, Wenbo Li, Tianqing Wang, and Shifu Li. opengauss: An autonomous database system. *Proceedings of the VLDB Endowment*, 14(12):3028–3042, 2021.

[17] yugabyteDB. https://www.yugabyte.com/.

[18] InnoDB Isolation Levels. https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html.

[19] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052, 2017.

[20] Feifei Li. Cloud-native database systems at alibaba: Opportunities and challenges. *Proceedings of the VLDB Endowment*, 12(12):2263–2272, 2019.

[21] Per-Åke Larson, Adrian Birka, Eric N Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. Real-time analytical processing with sql server. *Proceedings of the VLDB Endowment*, 8(12):1740–1751, 2015.

[22] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020.

[23] Pramod Bhatotia, Alexander Wieder, İstemi Ekin Akkuş, Rodrigo Rodrigues, and Umut A Acar. Large-scale incremental data processing with change propagation. In *3rd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 11)*, 2011.

[24] Rocksdb. https://www.rocksdb.org/.

[25] SQLite. https://www.sqlite.org/index.html/.

[26] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, et al. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2653–2666, 2021.

[27] Singlestore. https://www.singlestore.com/.

[28] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. CockroachDB: The resilient geo-distributed SQL database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1493–1509, 2020.

[29] James C. Corbett r and et al. Spanner: Google's globally-distributed database. In *OSDI*, pages 251–264, 2012.

[30] Oracle database. https://www.oracle.com/hk/database/technologies/.

[31] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient transaction processing in sap hana database: the end of a column store myth. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 731–742, 2012.

[32] NuoDB. https://nuodb.com/.

[33] Oceanbase. https://www.oceanbase.com/.

[34] PostgreSQL Isolation Levels. https://www.postgresql.org/docs/10/transaction-iso.html#XACT-REPEATABLE-READ.

[35] YugabyteDB Isolation Levels. https://docs.yugabyte.com/preview/explore/transactions/isolation-levels/#serializable-isolation.

[36] OpenGauss Isolation Levels. https://support.huaweicloud.com/intl/en-us/devg-opengauss/opengauss_devg_0004.html.

[37] SQLserver Isolation Levels. https://learn.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql?view=sql-server-ver16.

[38] RocksDB Isolation Levels. https://github.com/facebook/rocksdb/wiki/Transactions.

[39] SQLite Isolation Levels. https://www.sqlite.org/isolation.html.

[40] SingleStore Isolation Levels. https://docs.singlestore.com/db/v7.8/en/introduction/faqs/durability/what-isolation-levels-does-singlestoredb-provide-.html.

[41] Oracle Isolation Levels. https://docs.oracle.com/en/database/oracle/oracle-database/19/cncpt/data-concurrency-and-consistency.html#GUID-2A0FDFF0-5F72-4476-BFD2-060A20EA1685.

[42] Spanner Isolation Levels. https://cloud.google.com/spanner/docs/transactions#rw_transaction_semantics.

[43] NuoDB Isolation Levels. https://doc.nuodb.com/nuodb/latest/sql-development/working-with-transactions/supported-transaction-isolation-levels/description-of-nuodb-transaction-isolation-levels/.

[44] Juchang Lee, Michael Muehle, Norman May, Franz Faerber, Vishal Sikka, Hasso Plattner, Jens Krueger, and Martin Grund. High-performance transaction processing in sap hana. *IEEE Data Eng. Bull.*, 36(2):28–33, 2013.