Isolation Levels in Popular Commercial DBMSs

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

In this paper, we summarize the classic mechanisms that implement isolation levels widely used in popular commercial DBMSs. In Sec. 1, we first discuss the definition of isolation levels. Then, we summarize the classic mechanisms of eliminating isolation anomalies in Sec. 2. Finally, in Sec. 3, we analyze that what classic mechanism combination does a DBMS use to implementing isolation levels.

1 Isolation Level Definitions

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

Various isolation levels are wildly used in DBMSs, including *read committed* (*RC*), *repeatable read* (*RR*), *snapshot isolation* (*SI*), *serializable* (*SR*), and so on. However, how to precisely define isolation levels is still a challenging and critical problem. The early ANSI standard [4] defines different isolation levels as preventing different phenomena. Due to its inaccuracy, the following work [5] takes the most popular implementation, i.e., *two-phase locking*, to define isolation levels. However, this definition is criticized by work [6] because it does not work for DBMSs that do not use lock implementations. To achieve both precision and generality, Adya et. al. [6] define different isolation levels as preventing different types of cycles in the dependency graph. Several recent work [2, 7, 8] attempt to make such a definition more developer-friendly by defining isolation levels from the external view. Generally speaking, there is no unified definition of isolation levels, and each DBMS has its own definition.

2 Isolation Anomalies Elimination Mechanisms

The data accesses performed by concurrently executing transactions have a potential of suffering from isolation anomalies. In this section, we first define the notations used in our paper. Then, we introduce several isolation anomalies. Next, we discuss how to eliminate isolation anomalies with concurrency control protocols. Finally, we summarize the 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

Isolation levels 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 x^i's direct successor 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 x^i's direct successor version x^{i+1}, t_n has a **direct anti-dependency** (rw) on t_m.

2.2 Isolation Anomalies

We introduce several isolation anomalies that are usually prohibited by the definition of isolation levels.

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 abort. If t_0 or t_1 then performs a 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 abort. If t_1 then performs a abort, t_1 has read a data item that was never committed and so never really existed.

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 t_0's <search condition> and commits. If t_0 then repeats its read with the same <search condition>, it gets a set of data items different from the 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 x_1's version 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 T1 (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 conflicts between concurrent transactions is not equivalent to the conflicts between serial transactions.

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 rules that locks are acquired and released in two phases, that is, growing phase and shrinking phase. Growing phase rules that locks are acquired and no locks are released, while shrinking phase rules that locks are released and no locks are acquired. Note that, commercial DBMSs usually takes a variant of two-phase locking, called strict two-phase locking. Specifically, strict two-phase locking rules 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 sees the newest versions that existed when the transaction started.

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 rules 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 item 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 $ts(t)$. Every data item x is tagged with timestamp of the last transaction that successfully did read/write, denoted as $r-ts(x)/w-ts(x)$. If $ts(t) < r-ts(x)$ or $ts(t) < w-ts(x)$, this violates timestamp order of transaction t with regard to the reader or writer of data item x, then abort t.

2.4 Classic 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*). In Fig. 2, we summarize the relationship among isolation anomalies, concurrency control protocols and the four classic mechanisms. Note that, there still exist some other mechanisms [9, 10, 11, 12, 13, 14], almost all of which stay only in the academic papers instead of in practical products.

Table 2: Four Classic Mechanisms

CCP	**Mechanism**	**Anomaly**
MVCC	CR	Read Skew, Dirty Read
2PL	ME	Dirty Write, Dirty Read, Non-repeatable Read, Phantom
OCC	SC	Serialization Anomaly
TO	SC	Serialization Anomaly
SSI	FUW+SC	Read Skew, Lost Update, Write Skew

2.4.1 Consistent read

CR provides a consistent view of the database at a specific time point. To provide different isolation levels, there are transaction-level CR and statement-level CR. Transaction-level consistent read provides a consistent view of the database at the beginning of a transaction, while statement-level consistent read at the beginning of an operation. Fig. 1 shows an example of two types of CR. There are four transactions, i.e., t_{0-3}, operating on record x, among which t_{0-1} present two different types of consistent reads. The two reads in t_0 see the same snapshot generated by t_3 for the requirement of transaction-level

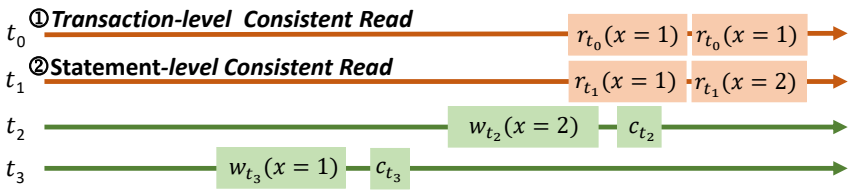

Figure 1: Example for Consistent Read

consistent read; the two reads in t_1 see different values, i.e., $x = 1$ and $x = 2$, constrained by statement-level consistent read.

As described above, *read skew* indicates a kind of anomalies where a transaction sees an inconsistent state of the database. Aiming at eliminating *read skew*, most popular commercial DBMSs take the mechanism of CR to see the committed version of a record as 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 a specific time point and the changes made by earlier operations within the same transaction and the changes. Note that, as described above, *dirty read* indicates a read operation sees a temporary results of a concurrent write operation, which means that there is no *dirty read* when eliminating *read skew*.

2.4.2 Mutual exclusion

ME uses the locking strategy to provide a kind of exclusive access to a shared resource. There are two mode of locking, i.e., shared and exclusive, where 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 lock. We discuss how 2PL 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. Aiming at eliminating *dirty write*, most popular commercial DBMSs take ME to exclusively modify a record. Specifically, in a transaction, each write operation should hold a 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 results of a concurrent write operation. Aiming at eliminating *dirty read*, a part of popular 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 operation in a transaction. Aiming at eliminating *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 fall into the <search condition> of a concurrent read operation. Aiming

Table 3: Serialization Certifier

CCP	Certifier
OCC	aborting the transaction conflicting with other active ones
TO	ordering transactions as a monotonically increasing timestamp
SSI	detecting write skew anomalies

at eliminating *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

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. Aiming at eliminating *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

SC guarantees transaction execution is conflict serializability [15]. Conflict serializability means that the conflicts between concurrent transactions is equivalent to the conflicts between serial transactions, which eliminates all serialization anomalies. Each concurrency control protocol has its own "certifier" to guarantee conflict serializability. Here, we discuss three concurrency control protocols widely used in commercial DBMSs, and we have summarized them in Table. 3.

Before committing a transaction, OCC takes the mechanism of SC to check whether the transaction conflicting with other active ones. TO takes the mechanism of SC to generate serializable transaction execution by ordering transactions as a monotonically increasing timestamp. SSI is based on snapshot isolation to provide conflict serializability. As the definition of snapshot isolation, all reads within a transaction see a consistent view of the database, concurrent transactions are prohibited from modifying the same record. However, snapshot isolation stills suffers from *write skew*, as discussed in [5]. SSI takes the mechanism of SC to detect *write skew* and abort transactions as necessary.

Table 4: Isolation Level Implementations in DBMSs

DBMS	CCP	IL	CR	ME	FUW	SC
PostgreSQL [16], OpenGauss [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	SR	✓	✓		
	OCC+MVCC	SR	✓			✓
SQLite [25]	2PL	SR		✓		
FoundationDB [26]	OCC+MVCC	SR	✓			✓
SingleStore [27]	2PL+MVCC	RC	✓	✓		
CockroachDB [28]	TO+MVCC	SR	✓			✓
Spanner [29]	2PL+MVCC	SR	✓	✓		
yugabyteDB [30]	2PL+MVCC	SR,RR,RC	✓	✓	✓	✓
Oracle [31], NuoDB [32], SAP HANA [33]	2PL+MVCC	SI	✓	✓	✓	
		RC	✓	✓		

3 Isolation Level Implementation Mechanisms

A DBMS usually mixes up multiple concurrency control protocols (*CCP*) to eliminate isolation anomalies that should be prevented as the definition of isolation levels. There are various isolation levels widely used in DBMS community, including *read committed* (*RC*), *repeatable read* (*RR*), *snapshot isolation* (*SI*) and *serializable* (*SR*). In this section, we discuss the mechanisms of popular commercial DBMSs implementing isolation levels, including *consistent read* (*CR*), *mutual exclusion* (*ME*), *first updater wins* (*FUW*) and *serialization certifier* (*SC*). We summarize the mechanisms of implementing isolation levels in Table. 4.

3.1 PostgreSQL

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

1. **Eliminating Read Skew** 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 are 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.

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[1].

Serializable are 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 Fig. 2, we summarize the relationship between isolation anomalies and the four classic mechanisms. According to the above definitions, we conclude that the classic mechanisms of implementing isolation levels of PostgreSQL as follows:

1. *Read committed* in PostgreSQL takes *consistent read* to eliminate *read skew*, while takes *mutual exclusion* to eliminating *dirty write*.

2. *Snapshot isolation* in PostgreSQL takes *consistent read* to eliminate *read skew*, while takes *first updater wins* to eliminating *lost update*.

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

4 Conclusion

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.

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

[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.

[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] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. Seeing is believing: A client-centric specification of database isolation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 73–82, 2017.

[8] Adriana Szekeres and Irene Zhang. Making consistency more consistent: A unified model for coherence, consistency and isolation. In *Proceedings of the 5th Workshop on the Principles and Practice of Consistency for Distributed Data*, pages 1–8, 2018.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

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

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

[17] 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.

[18] Innodb. 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] yugabyteDB. https://www.yugabyte.com/.

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

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

[33] 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.

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