# Serializable Snapshot Isolation in PostgreSQL

January 7, 2025

The pr. The discussion. Discussion 2. The problem in mind

1. There is abstract, conceptual agreement that support for serializable transactions would be A Good Thing.
2. There is doubt that an acceptably performant implementation is possible in PostgreSQL.
3. Some, but not all, don't want to see an implementation which produces false positive serialization faults with some causes, but will accept them for other causes.
4. Nobody believes that an implementation with acceptable performance is possible without the disputed false positives mentioned in (3).
5. There is particular concern about how to handle repeated rollbacks gracefully if we use the non-blocking technique.
6. There is particular concern about how to protect long-running transactions from rollback. (I'm not sure those concerns are confined to the new technique.)
7. Some, but not all, feel that it would be beneficial to have a correct implementation (no false negatives) even if it had significant false positives, as it would allow iterative refinement of the locking techniques.
8. One or two people feel that there would be benefit to an implementation which reduces the false negatives, even if it doesn't eliminate them entirely. (Especially if this could be a step toward a full implementation.)

# 1 Snapshot Isolation

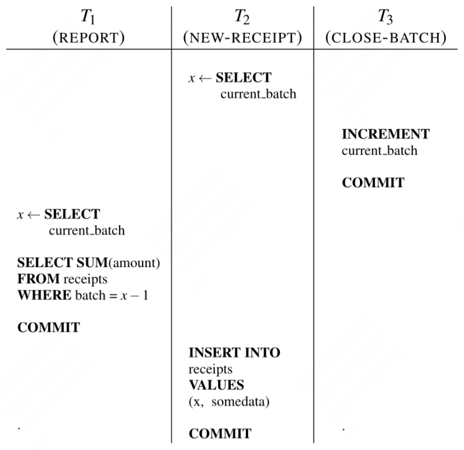
## 1.1 Example 1: Simple Write Skew

## 1.2 Example 2: Batch Processing

Consider a transaction-processing system that maintains two tables. A *receipts* table tracks the day's receipts, with each row tagged with the associated batch number. A separate *control* table simply holds the current batch number. There are three transaction types:

- NEW-RECEIPT: reads the current batch number from the control table, then inserts a new entry in the receipts table tagged with that batch number
- CLOSE-BATCH: increments the current batch number in the control table
- REPORT: reads the current batch number from the control table, then reads all entries from the receipts table with the previous batch number (i.e. to display a total of the previous day's receipts)

The following useful invariant holds under serializable executions: after a REPORT transaction has shown the total for a particular batch, subsequent transactions cannot change that total.



| $T_1$ (REPORT) | $T_2$ (NEW-RECEIPT) | $T_3$ (CLOSE-BATCH) |
|---|---|---|
| | $x \leftarrow$ **SELECT** current_batch | |
| | | **INCREMENT** current_batch |
| | | **COMMIT** |
| $x \leftarrow$ **SELECT** current_batch | | |
| **SELECT SUM**(amount) **FROM** receipts **WHERE** batch = $x - 1$ | | |
| **COMMIT** | | |
| | **INSERT INTO** receipts **VALUES** (x, somedata) | |
| | **COMMIT** | |

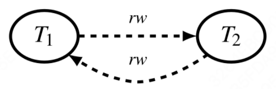
Figure 2: An anomaly involving three transactions

# 2 Serializable Snapshot Isolation

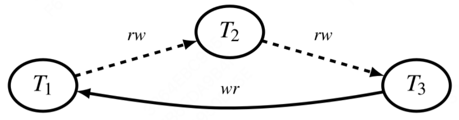
## 2.1 Snapshot Isolation Anomalies

Adya proposed representing an execution with a multi-version serialization history graph. This graph contains a node per transaction, and an edge from transaction $T_1$ to transaction $T_2$ if $T_1$ must have preceded $T_2$ in the apparent serial order of execution. Three types of dependencies can create these edges:

- **wr-dependencies**:
- **ww-dependencies**:
- **rw-dependencies**: if $T_1$ writes a version of some object, and $T_2$ reads the previous version of that object, then $T_1$ appears to have executed after $T_2$ , because $T_2$ did not see its update.



(a) Example 1: Simple Write Skew



(b) Example 2: Batch Processing

Figure 3: Serialization graphs for Examples 1 and 2

## 2.2 Serializability Theory

Note that a wr-dependency from $A$ to $B$ means that $A$ must have committed before $B$ began, as this is required for $A$'s changes to be visible to $B$'s snapshot. The same is true of ww-dependencies because of write locking. However, rw-antidependencies occur between concurrent transactions: one must start while the other was active. Therefore, they play an important role in SI anomalies.

**Theorem 2.1.** *Every cyle in the serialization history graph contains a sequence of edges* $T_1 \xrightarrow{rw} T_2 \xrightarrow{rw} T_3$ *where each edge is a rw-dependency. Furthermore,* $T_3$ *must be the first transaction in the cycle to commit.*

**Corollary 2.2.** *Transaction* $T_1$ *is concurrent with* $T_2$*, and* $T_2$ *is concurrent with* $T_3$*, because rw-antidependencies occur only between concurrent transactions.*

## 2.3 SSI

The SSI paper describes a method for identifying these dependencies by having transactions acquire locks in a special "SIREAD" mode on the data they read. These locks do not block conflicting writes (thus, "lock" is somewhat of a misnomer). Rather, a conflict between a SIREAD lock and a write lock flags an rw-antidependency, which might cause a transaction to be aborted. Furthermore, SIREAD locks must persist after a transaction commits, because conflicts can occur even after the reader has committed (e.g. the $T_1 \xrightarrow{rw} T_2$ conflict in Example 2). Corollary 2 implies that the locks must be retained until all concurrent transactions commit. Our PostgreSQL implementation uses SIREAD locks, but their implementation differs significantly because PostgreSQL was purely snapshot-based, as we describe in Section 5.2.

### 2.3.1 Varaints on SSI

Subsequent work has suggested refinements to the basic SSI approach. Cahill's thesis suggests a commit ordering optimization that can reduce false positives. Theorem 1 actually shows that every cycle contains a dangerous structure $T_1 \xrightarrow{rw} T_2 \xrightarrow{rw} T_3$ , where $T_3$ is the first to commit. Thus, even if a dangerous structure is found, no aborts are necessary if either $T_1$ or $T_2$ commits before $T_3$ . Verifying this condition requires tracking some additional state, but avoids some false positive aborts. We use an extension of this optimization in PostgreSQL. It does not, however, eliminate all $T_1$ that closes the false positives: there may not be a path $T_3$ cycle. For example, in Example 2, if $T_1$'s REPORT accessed only the receipts table (not the current batch number), there would be no wr-dependency from T3 to T1 , and the execution would be serializable with order $\langle T_1, T_2, T_3 \rangle$. However, the dangerous structure of rw-antidependencies $T_1 \xrightarrow{rw} T_2 \xrightarrow{rw} T_3$ would force some transaction to be spuriously aborted.

# 3 Read-only Optimizations

We improve performance for read-only transactions in two ways. Both derive from a new serializability theory result that characterizes when read-only transactions can be involved in SI anomalies.

1. the theory enables a *read-only snapshot ordering optimization* to reduce the false-positive abort rate

2. we also identify certain safe snapshots on which read-only transactions can execute safely without any SSI overhead or abort risk, and introduce deferrable transactions, which delay their execution to ensure they run on safe snapshots.

## 3.1 Theory

**Theorem 3.1.** *Every serialization anomaly contains a dangerous structure $T_1 \overset{rw}{\longrightarrow} T_2 \overset{rw}{\longrightarrow} T_3$, where if $T_1$ is read-only, $T_3$ must have committed before $T_1$ took its snapshot.*

*Proof.* Consider a cycle in the serialization history graph. From Theorem 2.1, we know it must have a dangerous structure $T_1 \overset{rw}{\longrightarrow} T_2 \overset{rw}{\longrightarrow} T_3$ where $T_3$is the first transaction in the cycle to commit. Consider the case where $T_1$ is read-only.

Because there is a cycle, there must be some transaction $T_0$ that precedes $T_1$ in the cycle. The edge $T_0 \rightarrow T_1$ can't be a rw-antidependency or a ww-dependency, because $T_1$ was read-only, so it must be a wr-dependency. A wr-dependency means that $T_0$'s change were visible to $T_1$, so $T_0$ must have committed before $T_1$ took its snapshot. Because $T_3$ is the first transaction in the cycle to commit, it must commit before $T_0$ commits, and therefore before $T_1$ takes its snapshot. □

Therefore if a dangerous structure is detected where $T_1$ is read-only, it can be disregarded as a false positive unless $T_3$ committed before $T_1$'s snapshot.

## 3.2 Safe Snapshots

A read-only transaction $T_1$ cannot have a rw-conflict pointing in, as it did not perform any writes. The only way it can be part of a dangerous structure, therefore, is if it has a conflict out to a concurrent read/write transaction $T_2$ , and $T_2$ has a conflict out to a third transaction $T_3$ that committed before $T_1$'s snapshot. If no such $T_2$ exists, then $T_1$ will never cause a serialization failure. This depends only on the concurrent transactions, not on $T_1$'s behavior; therefore, we describe it as a property of the snapshot:

**Safe snapshots**: A read-only transaction $T$ has a **safe snapshot** if no concurrent read/write transaction has committed with a rw-antidependency out to a transaction that committed before $T$'s snapshot, or has the possibility to do so.

An unusual property of this definition is that we cannot determine whether a snapshot is safe at the time it is taken, only once all concurrent read/write transactions complete, as those transactions might subsequently develop conflicts. Therefore, when a `READ ONLY` transaction is started, PostgreSQL makes a list of concurrent transactions. The read-only transaction executes as normal, maintaining SIREAD locks and other SSI state, until those transactions commit. After they have committed, if the snapshot is deemed safe, the read-only transaction can drop its SIREAD locks, essentially becoming a REPEATABLE READ (snapshot isolation) transaction.

### 3.3 Deferrable Transactions

Some workloads contain long-running read-only transactions and take more SIREAD locks.

These transactions would especially benefit from running on safe snapshots: they could avoid taking SIREAD locks, they would be guaranteed not to abort, and they would not prevent concurrent transactions from releasing their locks. **Deferrable transactions**, a new feature, provide a way to ensure that complex read-only transactions will always run on a safe snapshot. Read-only serializable transactions can be marked as deferrable with a new keyword, e.g. BEGIN TRANSACTION READ ONLY, DEFERRABLE. Deferrable transactions always run on a safe snapshot, but may block before their first query.

## 4 Implementing SSI in PostgreSQL

### 4.1 PostgreSQL Background

All queries in PostgreSQL are performed with respect to a snapshot, which is represented as the set of transactions whose effects are visible in the snapshot. Each tuple is tagged with the transaction ID of the transaction that created it (xmin), and, if it has been deleted or replaced with a new version, the transaction that did so (xmax). Checking which of these transactions are included in a snapshot determines whether the tuple should be visible. Updating a tuple is, in most respects, identical to deleting the existing version and creating a new tuple. The new tuple has a separate location in the heap, and may have separate index entries.2 Here, PostgreSQL differs from other MVCC implementations (e.g. Oracle's) that update tuples in-place and keep a separate rollback log.

## 5 Problems

## 6 References

## References