Serializable Snapshot Isolation in PostgreSQL

February 6, 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	

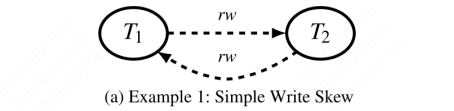
Figure 2: An anomaly involving three transactions

2 Serializable Snapshot Isolation

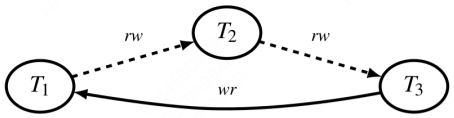
2.1 Snapshot Isolation Anomalies

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* $T_1 \xrightarrow{rw} T_2 \xrightarrow{rw} T_3$*, 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 $T_1 \xrightarrow{rw} T_2 \xrightarrow{rw} T_3$ 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. (*commit ordering optimization*)

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 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. Here, PostgreSQL differs from other MVCC implementations (e.g. Oracle's) that update tuples in-place and keep a separate rollback log.

Internally, PostgreSQL uses three distinct lock mechanisms:

- **lightweight locks** are standard reader-writer locks for synchronizing access to shared memory structures and buffer cache pages; these are typically referred to as latches elsewhere in the literature

- **heavyweight locks** are used for long-duration (e.g. transaction-scope) locks, and support deadlock detection. A variety of lock modes are available, but normal-case operations such as SELECT and UPDATE acquire locks in non-conflicting modes. Their main purpose is to prevent schema-changing operations, such as DROP TABLE or REINDEX, from being run concurrently with other operations on the same table. These locks can also be explicitly acquired using LOCK TABLE.

- **tuple locks** prevent concurrent modifications to the same tuple. Because a transaction might acquire many such locks, they are not stored in the heavyweight lock table; instead, they are stored in the tuple header itself, reusing the $xmax$ field to identify the lock holder. SELECT FOR UPDATE also acquires these locks. Conflicts are resolved by calling the heavyweight lock manager, to take advantage of its deadlock detection.

4.2 Detecting Conflicts

One of the main requirements of SSI is to be able to detect rw-conflicts as they happen.

Earlier work suggested modifying the lock manager to acquire read locks in a new SIREAD mode, and flagging a rw-antidependency when a conflicting lock is acquired. Unfortunately, this technique cannot be directly applied to PostgreSQL because the lock managers described above do not have the necessary information.

To begin with, PostgreSQL did not previously acquire read locks on data accessed in any isolation level, unlike the databases used in prior SSI implementations, so SIREAD locks cannot simply be acquired by repurposing existing hooks for read locks. Worse, even with these locks, there is no easy way to match them to conflicting write locks because PostgreSQL's tuple-level write locks are stored in tuple headers on disk, rather than an in-memory table.

Instead, PostgreSQL's SSI implementation uses existing MVCC data as well as a new lock manager to detect conflicts. Which one is needed depends on whether the write happens chronologically before the read, or vice versa.

1. If the write happens first, then the conflict can be inferred from the

MVCC data, without using locks. Whenever a transaction reads a tuple, it performs a visibility check, inspecting the tuple's $xmin$ and $xmax$ to determine whether the tuple is visible in the transaction's snapshot.

2. If the tuple is not visible because the transaction that created it had not committed when the reader took its snapshot, that indicates a rw-conflict: the reader must appear before the writer in the serial order.

3. if the tuple has been deleted – i.e. it has an $xmax$ – but is still visible to the reader because the deleting transaction had not committed when the reader took its snapshot, that is also a rw-conflict that places the reader before the deleting transaction in the serial order.

We also need to handle the case where the read happens before the write. This cannot be done using MVCC data alone; it requires tracking read dependencies using SIREAD locks. Moreover, the SIREAD locks must support predicate reads. As discussed earlier, none of PostgreSQL's existing lock mechanisms were suitable for this task, so we developed a new SSI lock manager. The SSI lock manager stores only SIREAD locks. It does not support any other lock modes, and hence cannot block. The two main operations it supports are to obtain a SIREAD lock on a relation, page, or tuple, and to check for conflicting SIREAD locks when writing a tuple.

4.2.1 Implementation of the SSI Lock Manager

The PostgreSQL SSI lock manager, like most lock managers used for S2PL-based serializability, handles predicate reads using index-range locks.

Reads acquire SIREAD locks on all tuples they access, and index access methods acquire SIREAD locks on the "gaps" to detect phantoms. Currently, locks on B+-tree indexes are acquired at page granularity; we intend to refine this to next-key locking in a future release.

Both heap and index locks can be promoted to coarser granularities to save space in the lock table, e.g. replacing multiple tuple locks with a single page lock.

SIREAD locks must be kept up to date when concurrent transactions modify the schema with data-definition language (DDL) statements. Statements that rewrite a table, such as RECLUSTER or ALTER TABLE, cause the physical location of tuples to change. As a result, page- or tuple-granularity SIREAD locks, which are identified by phys- ical location, are no longer

valid; PostgreSQL therefore promotes them to relation-granularity. Similarly, if an index is removed, any index-gap locks on it can no longer be used to detect conflicts with a predicate read, so they are replaced with a relation-level lock on the associated heap relation.

4.3 Tracking Conflicts

We chose to keep a list of all rw-antidependencies in or out for each transaction, but not wr- and ww-dependencies. Keeping pointers to the other transaction involved in the rw-antidependency, rather than a simple flag, is necessary to implement the commit ordering optimization and read-only optimization.

4.4 Resolving Conflicts: Safe Retry

We want to choose the transaction to abort in a way that ensures the following property:

Safe retry: If a transaction is aborted, immediately retrying the same transaction will not cause it to fail again with the same serialization failure

Once we have identified a dangerous structure $T_1 \xrightarrow{rw} T_2 \xrightarrow{rw} T_3$, the key principle for ensuring safe retry is to abort a transaction that conflicts with a *committed* transaction. When the aborted transaction is retried, it will not be concurrent with the committed transaction, and cannot conflict with it. Specifically, the following rules are used to ensure safe retry:

1. Do not abort anything until T_3 commits. This rule is needed to support the commit ordering optimization, but it also serves the safe retry goal.

2. Always choose to abort T_2 if possible, i.e. if it has not already committed. T2 must have been concurrent with both T_1 and T_3 . Because T_3 is already committed, the retried T_2 will not be concurrent with it and so will not be able to have a rw-conflict out to it, preventing the same error from recurring.

3. If both T_2 and T_3 have committed when the dangerous structure is detected, then the only option is to abort T1 . But this is safe; T2 and T3 have already committed, so the retried transaction will not be concurrent with them, and cannot conflict with either.

One might worry that this delayed resolution could cause wasted work or additional conflicts, because a transaction continues to execute even after

a conflict that could force it to abort. However, aborting a transaction immediately would cause an equivalent amount of wasted work, if the transaction is immediately retried only to abort again.

5 Memory Usage Mitigation

After implementing the basic SSI functionality, one of the problems we were immediately confronted with was its potentially unbounded memory usage. The problem is not merely that one transaction can hold a large number of locks – a standard lock manager problem – but one unique to SSI: a transaction's locks cannot be released until that transaction and all concurrent transactions commit.

We were faced with two requirements related to memory usage.

1. The SSI implementation's memory usage must be *bounded*: the lock table and dependency graph must have a fixed size (specified by the configuration file).

2. The system must also be able to *gracefully degrade*. Even in the presence of long-running transactions, the system should not fail to process new transactions because it runs out of memory. Instead, it should be able to accept new transactions, albeit possibly with a higher false positive abort rate.

   Our PostgreSQL implementation uses four techniques to limit the memory usage of the SSI lock manager:

   (a) Safe snapshots and deferred transactions
   (b) granularity promotion
   (c) *Aggressive cleanup*
   (d) *Summarization*

5.1 Aggressive Cleanup

5.2 Summarizing Committed Transactions

Our SSI implementation reserves storage for a fixed number of committed transactions. If more committed transactions need to be tracked, we **summarize** the state of previously committed transactions. It is usually sufficient to discover that a transaction has a conflict with some previously committed transaction, but not which one. Summarization allows the database

to continue accepting new transactions, although the false positive abort rate may increasebecause some information is lost in the process.

Our summarization procedure is based on the observation that information about committed transactions is needed in two cases:

First, an active transaction modifying a tuple needs to know if some committed transaction read that tuple. This could create a dangerous structure $T_{committed} \xrightarrow{rw} T_{active} \xrightarrow{rw} T_3$. We need to keep a SIREAD lock to detect that such a transaction existed – but it does not matter what specific transaction it was, whether it had other rw-antidependencies in or out, etc. This motivates the first part of summarizing a committed transaction: the summarized transaction's SIREAD locks are *consolidated* with those of other summarized transactions, by reassigning them to a single dummy transaction. Each lock assigned to this dummy transaction also records the commit sequence number of the most recent transaction that held the lock, to determine when the lock can be cleaned up. The benefit of consolidation is that each lock only needs to be recorded once, even if it was held by multiple committed transactions. Combined with the ability to promote locks to a coarser granularity, this can make it unlikely that the SIREAD lock table will be exhausted.

6 Feature Interactions

6.1 Two-Phase Commit

6.2 Streaming Replication

6.3 Savepoints and Subtransactions

6.4 Index Types

7 Problems

8 References

References