

Paper Reading: Serializable Snapshot Isolation in PostgreSQL

Presented by Xuwei FU



Part I – Before step into the paper



SERIALIZABLE and NONREPEATABLE READ in PostgreSQL

In PostgreSQL 11:

Table 13.1. Transaction Isolation Levels

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

Questions before reading:

1. What are anomalies in Snapshot Isolation?
2. How to prevent the system from the anomalies above?
3. How SSI optimize read-only transaction?
4. How do PostgreSQL implement SSI?

Part II – Snapshot Isolation and Serializable



Serializable anomaly: Simple Write Skew

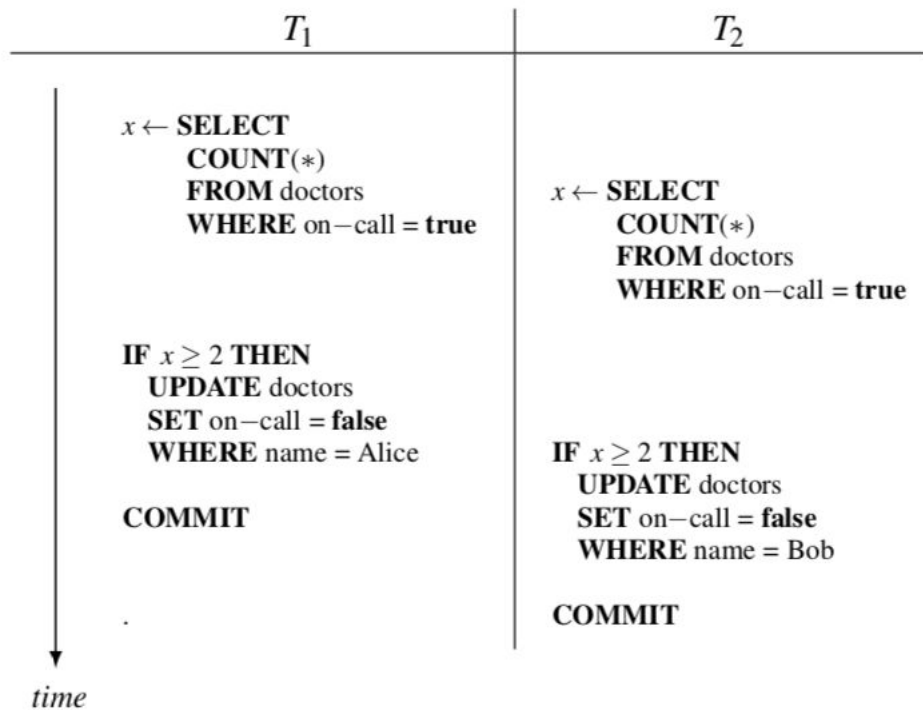


Figure 1: A simple write-skew anomaly

Serializable anomaly: Batch Processing

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

Figure 2: An anomaly involving three transactions

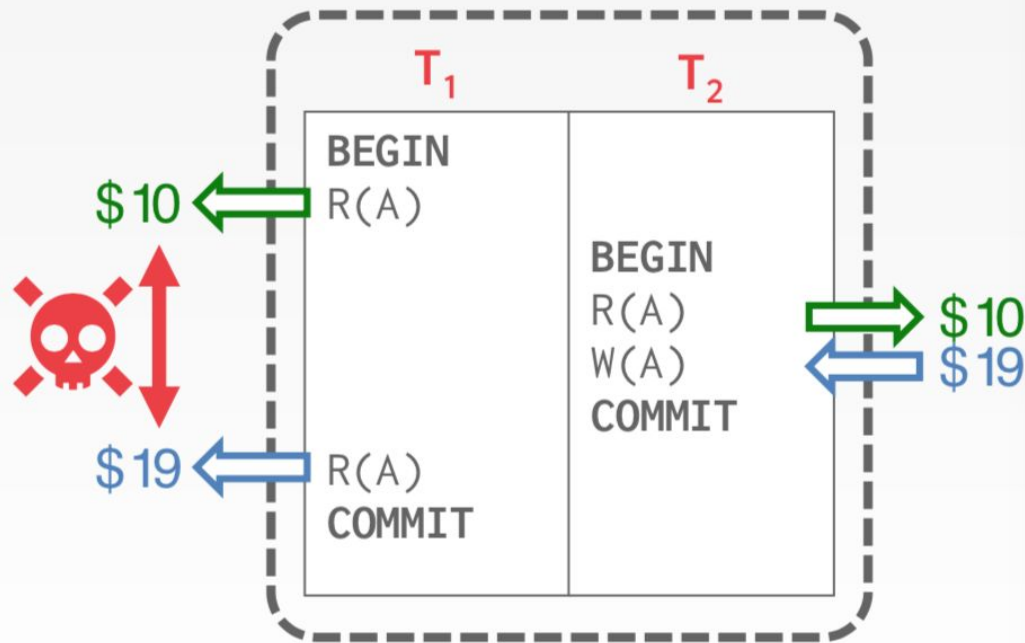


Snapshot Isolation IS NOT Serializable!!!



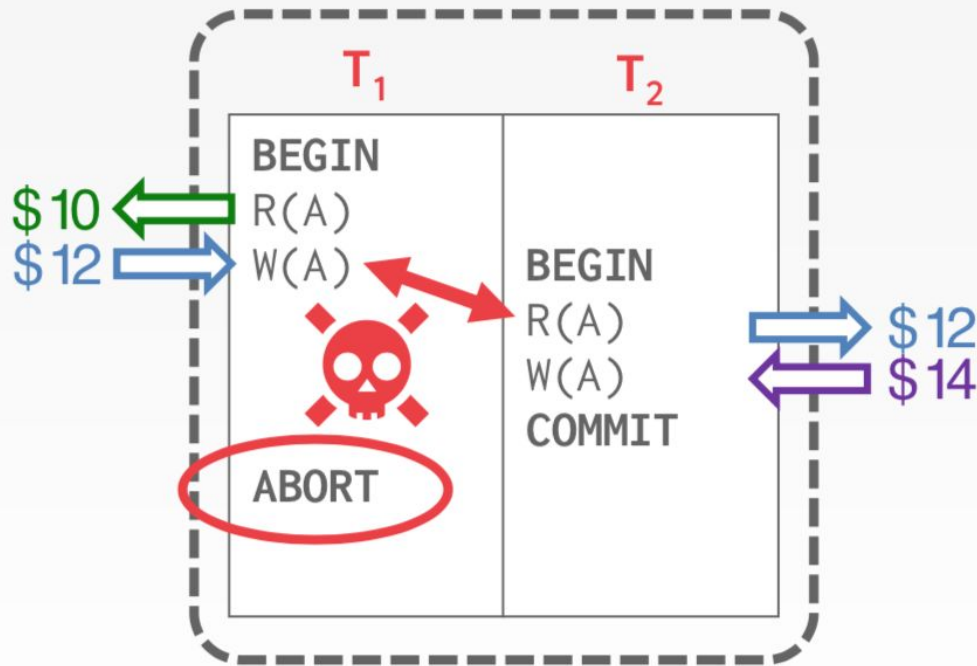
Transaction Conflict: Read-Write Conflicts

Unrepeatable Reads



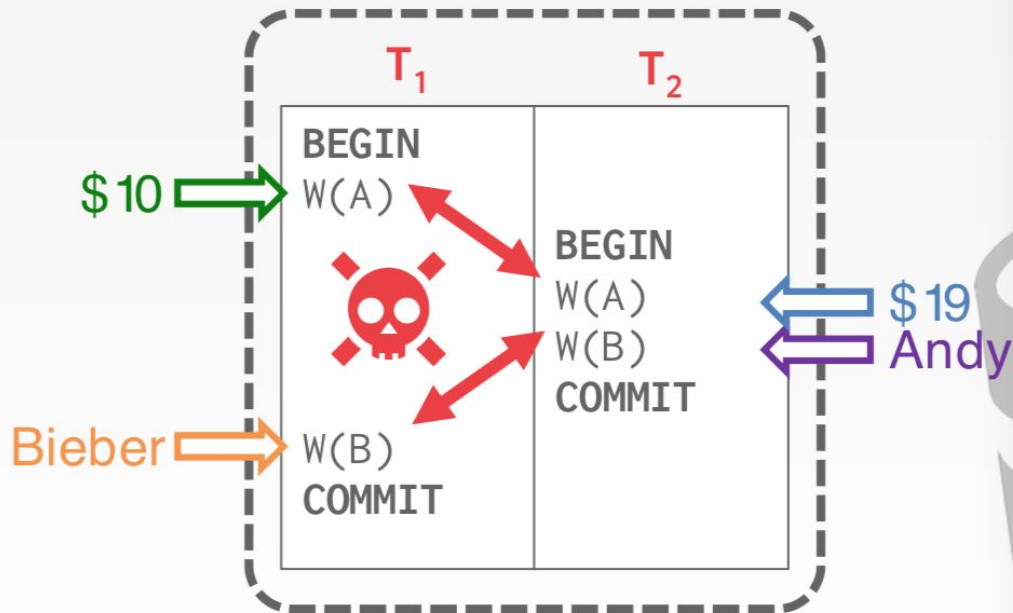
Transaction Conflict: Write-Read Conflicts

Reading Uncommitted Data ("Dirty Reads")



Transaction Conflict: Write-Write Conflicts

Overwriting Uncommitted Data

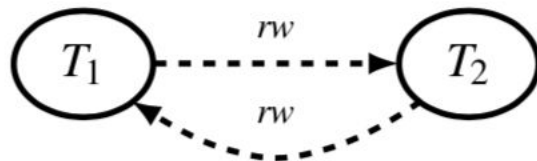


Multi-Version Serialization History Graph

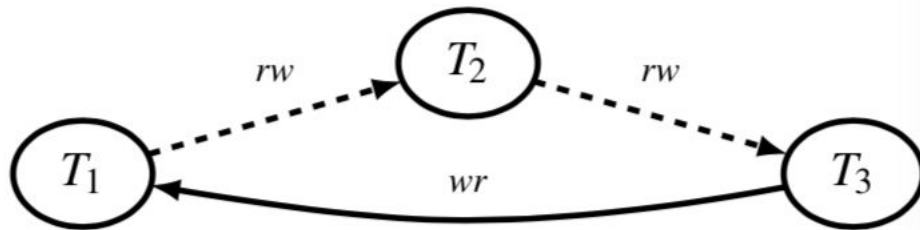
- ***wr-dependencies***: if $T1$ writes a version of an object, and $T2$ reads that version, then $T1$ appears to have executed before $T2$
- ***ww-dependencies***: if $T1$ writes a version of some object, and $T2$ replaces that version with the next version, then $T1$ appears to have executed before $T2$
- ***rw-antidependencies***: if $T1$ writes a version of some object, and $T2$ reads the previous version of that object, then $T1$ appears to have executed after $T2$, because $T2$ did not see its update. As we will see, these dependencies are central to SSI; we sometimes also refer to them as *rw-conflicts*.



Multi-Version Serialization History Graph



(a) Example 1: Simple Write Skew



(b) Example 2: Batch Processing

Figure 3: Serialization graphs for Examples 1 and 2

Serializability Theory: Theorem

Theorem 1 (Fekete et al. [10]). *Every cycle 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-antidependency. Furthermore, T_3 must be the first transaction in the cycle to commit.*

Corollary 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.*



Serializability Theory: How to understand

- ***rw-antidependencies*** can be concurrent in Snapshot Isolation
- T1 -rw-> T2 -rw □ T3 is dangerous!
- Rw-antidependencies before all transaction commits!



SSI

- If any transaction has both an incoming rw-antidependency and an outgoing one, SSI aborts **one of** the transactions involved. (which one is better will be introduced later)
- May have false positives because not every dangerous structure is part of a cycle.
- Must keep rw-antidependency even after transaction committed (Like T3 in example2).



Read under SSI

- A special “SIREAD” Lock will be applied on data being read.
- Write will not be blocked by “SIREAD” Lock.
- SIREAD locks must persist after a transaction commits, because conflicts rw can occur even after the reader has committed



Variants on SSI

Commit Ordering Optimization:

- Reduce false positives
- In Theorem1 T_3 need to commit first. Thus, even if a dangerous structure is found, no aborts are necessary if either T_1 or T_2 commits before T_3 .
- Cannot not eliminate all false positives

PSSI(Precisely Serializable Snapshot Isolation)

- eliminate all false positives
- building the full serialization history graph and testing it for cycles
- PSSI can reduce the abort rate by up to 40%



SSI Read-Only Optimization: Theory

Our read-only optimizations are based on the following extension of Theorem 1:

Theorem 3. *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.*

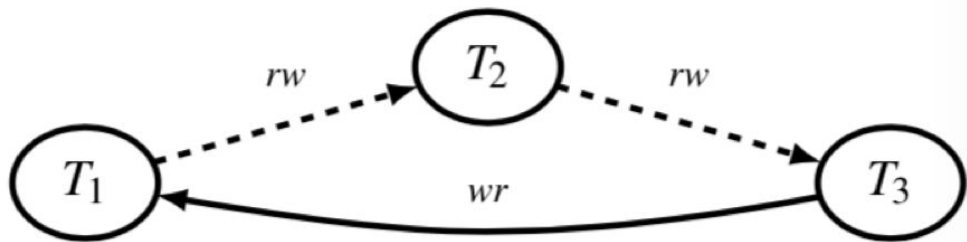


SSI Read-Only Optimization: Theory

If $T_1 \text{--}rw \square T_2 \text{--}rw \square T_3$ exists and serializable history graph has a cycle, and T_1 is an read-only transaction. And $T_0 \square T_1$ is part of the cycle:

$T_0 \square T_1$ must be wr-dependency. T_0 must be committed before T_1 takes snapshot.

Conclusion: 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.



(b) Example 2: Batch Processing

SSI Read-Only Optimization: Theory

Back to the example2:

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

Figure 2: An anomaly involving three transactions



SSI Read-Only Optimization: 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.



SSI Read-Only Optimization: Deferable Transactions

If you start a long-running read-only transactions, you may need deferable transactions.

Transaction T1 waits for all active transactions commit or abort, then it can get safe snapshot and not generate SIREAD lock.



Part III – Implementation In PostgreSQL



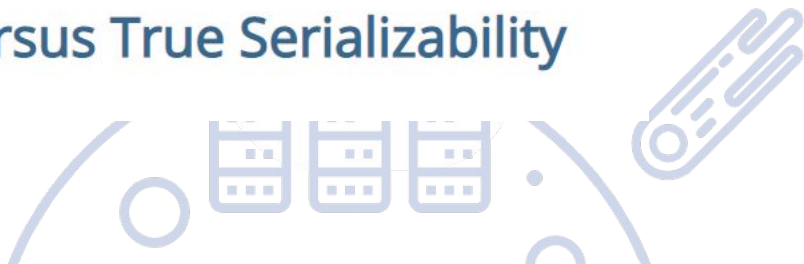
PostgreSQL before 9.1

Table 12-1. SQL Transaction Isolation Levels

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

PostgreSQL offers the Read Committed and Serializable isolation levels.

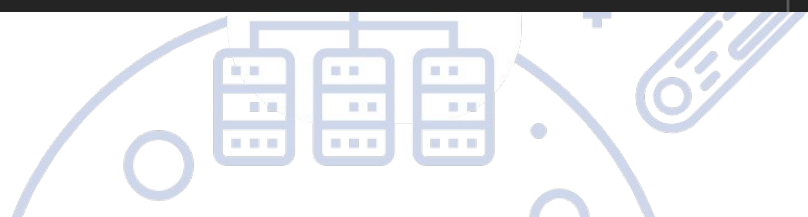
13.2.2.1. Serializable Isolation versus True Serializability



Mvcc in PostgreSQL

```
typedef struct HeapTupleFields
{
    TransactionId t_xmin;          /* inserting xact ID */
    TransactionId t_xmax;          /* deleting or locking xact ID */

    union
    {
        CommandId t_cid;          /* inserting or deleting command ID, or both */
        TransactionId t_xvac;      /* old-style VACUUM FULL xact ID */
    } t_field3;
} HeapTupleFields;
```



Mvcc in PostgreSQL: Challenges

- BerkeleyDB and MySQL InnoDB implement the SSI, but they all support Strict 2PL.
- Lock like predicate locking need to be implemented.
- PostgreSQL has to implement a new Lock Manager



SSI Lock Manager

- Only “SIREAD” lock
- SIREAD locks must be kept up to date when concurrent transactions modify the schema with data-definition language (DDL) statements.

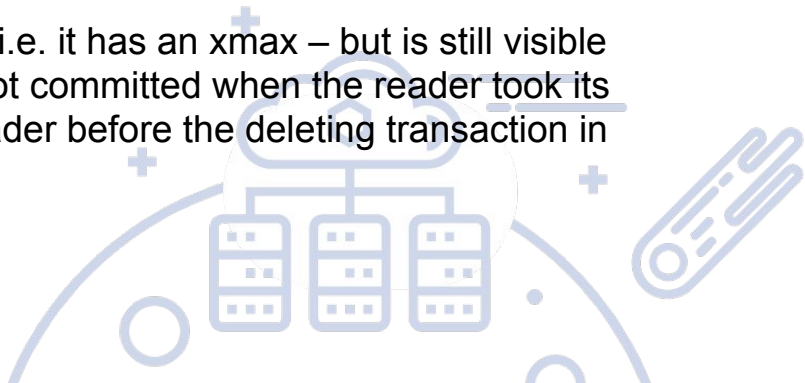


Detecting Conflicts: Read

PostgreSQL's SSI implementation uses existing MVCC data as well as a new lock manager to detect conflicts.

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.

- (rw-conflict) 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.
- (rw-conflict) Similarly, 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.



Detecting Conflicts: Write

(rw-conflict) 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.
- The SIREAD locks must support predicate reads.



Resolving Conflicts: Safe Retry

When dangerous structure detected, abort a transaction and retry it.

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

In $T1-(rw) \square T2-(rw) \square T3$:

1. Abort after T3 commits: Support Commit Ordering Optimization.
2. If possible, rollback T2.
3. If T2 and T3 committed, it's safe to retry T1.



Resolving Conflicts: Safe Retry

When dangerous structure detected, abort a transaction and retry it.

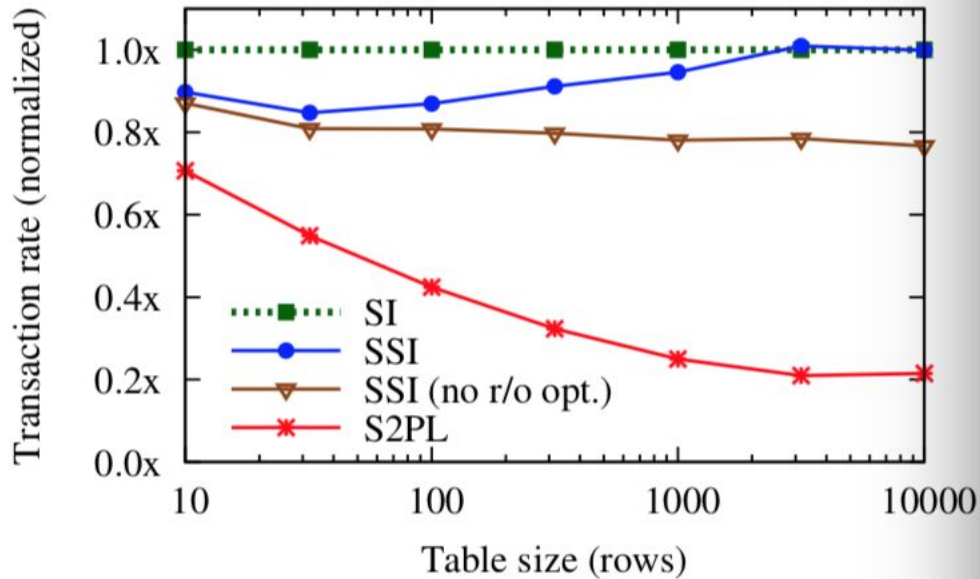
Safe retry: if a transaction is aborted, immediately retrying the same transaction will not cause it to fail again with the same serialization failure.

In $T1-(rw) \square T2-(rw) \square T3$:

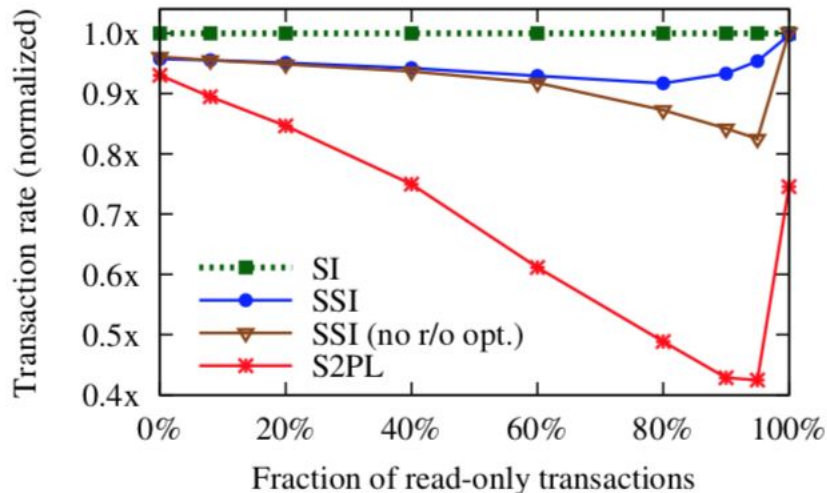
1. Abort after T3 commits: Support Commit Ordering Optimization.
2. If possible, rollback T2.
3. If T2 and T3 committed, it's safe to retry T1.



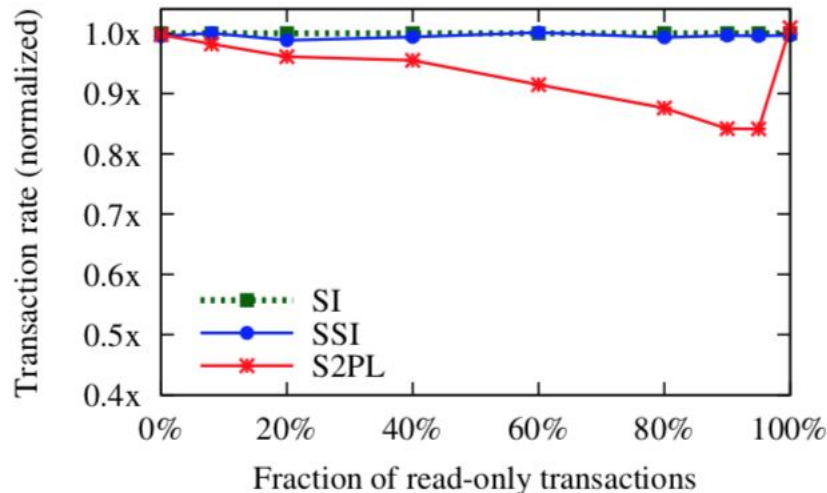
Evaluation



Evaluation

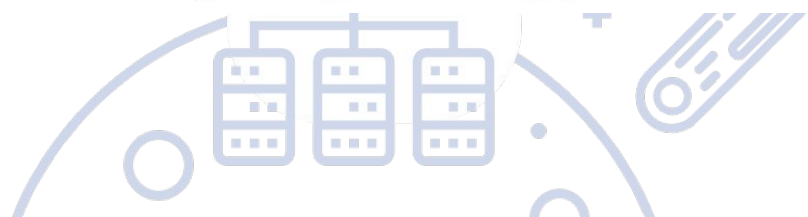


(a) in-memory configuration (25 warehouses)



(b) disk-bound configuration (150 warehouses)

Figure 5: DBT-2++ transaction throughput for SSI and S2PL as a percentage of SI throughput



The topic this reading not covered

- Memory Usage Limitation (6 in paper)
- Code and design for SSI Lock Manager in PostgreSQL (5.2 in paper)
- Feature Interactions (7 in paper)



Thank You !

