Precisely Serializable Snapshot Isolation

Stephen Revilak

Ph.D. Dissertation Supervised by Patrick O'Neil

University of Massachusetts Boston

Nov. 8, 2011

What is Snapshot Isolation?

Snapshot Isolation (SI) is a form of multiversion concurrency control.

- The technique was first published in a 1995 paper, *A Critique of ANSI Isolation Levels*
- SI is used in a variety of DBMS Systems: Oracle, Postgres, Microsoft SQL Server, BerkeleyDB, and others.

SI's advantages and disadvantages:

- **Advantages**: Good throughput. Reads never wait for writes. Writes never wait for reads.
- **Disadvantage**: SI is not serializable; it permits anomalies that locking avoids.

What is PSSI?

PSSI = *Precisely Serializable Snapshot Isolation*. PSSI is a set of extensions to SI that provide serializability.

- PSSI detects dependency cycles; PSSI aborts transactions to break these cycles.
- PSSI retains much of SI's throughput advantage. (Reads and writes do not block each other.)
- PSSI is precise; PSSI minimizes the number of unnecessary aborts by waiting for complete cycles to form.

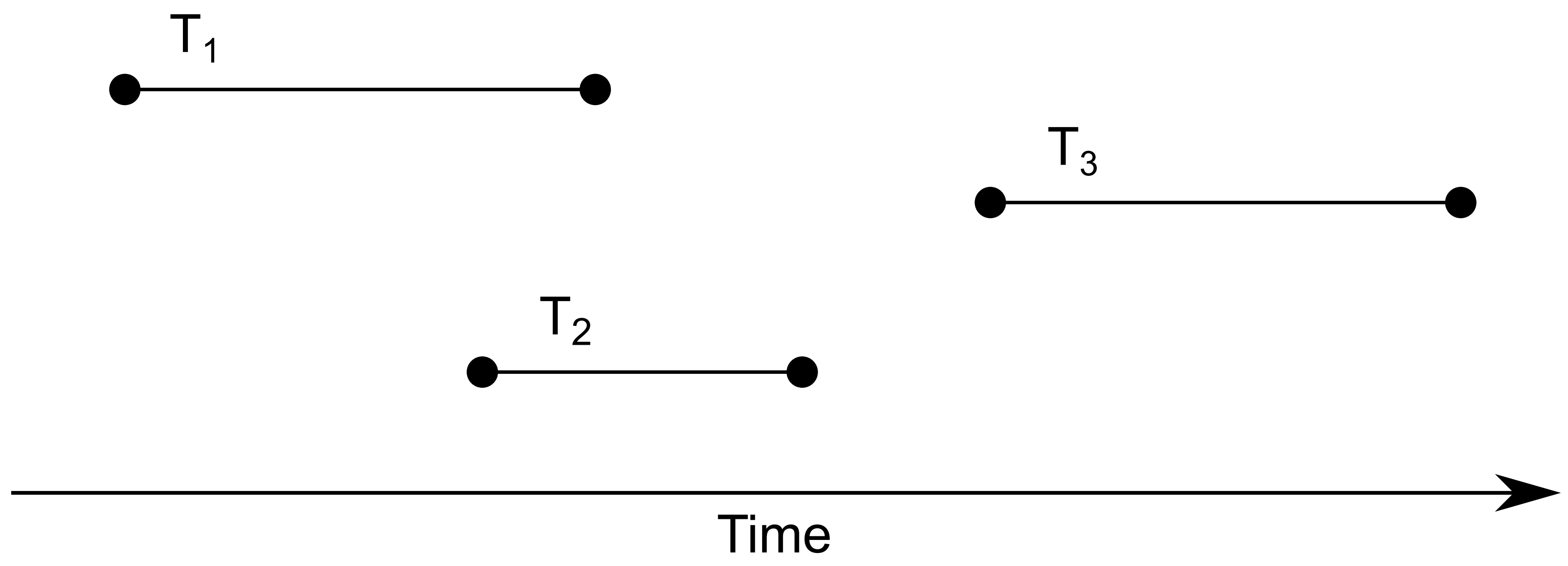
Talk Outline

- Snapshot Isolation, SI anomalies, dependency theory
- PSSI design
- Testing and Experimental results

Mechanics of Snapshot Isolation

- Each transaction has two timestamps: a *start timestamp* start(T_i), and a *commit timestamp*, commit(T_i).
- All data is versioned (labeled with the writer's transaction id). Different transactions may read different versions of a single data item x.
- When T_i reads x, T_i reads the last version of x committed prior to start(T_i)
- When T_i writes x, x_i becomes visible to T_j that start after commit(T_i).
- T_i does not see changes made by a concurrent transaction T_j.

SI Mechanics Illustrated



- T_1, T_2 can't see each other's changes
- T_3 can read changes from both T_1, T_2
- As we'll see shortly, T_1, T_2 can't write the same data.

First Committer Wins (FCW)

FCW is SI's way of preventing lost updates.

- **First Committer Wins Rule**: if two concurrent transactions (overlapping lifetimes) write the same data, then only one of those transactions can commit.
- If this condition is tested at update time (rather than at commit time), then we call this First Updater Wins (FUW)

FCW and FUW accomplish the same goal, but through different means.

First Updater Wins (FUW)

Let T$_i$ and T$_j$ be two concurrent transactions.

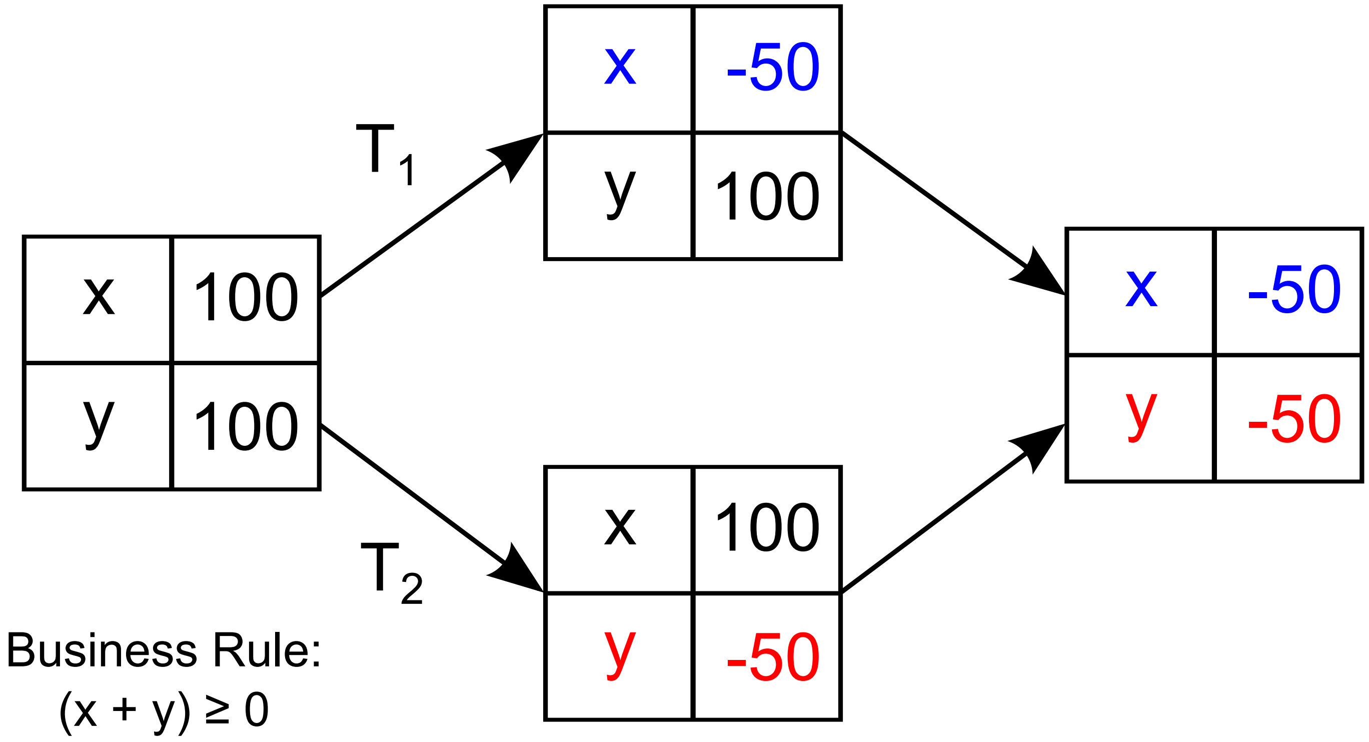
- T$_i$ updates x. T$_i$'s update takes a $w_i(x)$ lock.
- T$_j$ tries to update x. There are two possible outcomes:
  - If T$_i$ is still active, then T$_j$ goes into lock wait:
    - If T$_i$ commits, then T$_j$ aborts
    - If T$_i$ aborts, then T$_j$ acquires $w_j(x)$ lock and continues.
  - If T$_j$ tries to update x after commit(T$_i$), then T$_j$ aborts immediately (no lock wait).

First Updater Wins was first implemented by Oracle.
We used FUW in our PSSI prototype.

SI Anomalies – Write Skew

Example of an SI write skew anomaly:

H$_1$: $r_1(x_0, 100)$,$r_1(y_0, 100)$, $r_2(x_0, 100)$, $r_2(y_0, 100)$,
$w_1(x_1, -50)$, c_1, $w_2(y_2, -50)$, c_2



SI Anomalies – Predicate Write Skew

T_1 select sum(hours) from assignments where empid = 12 and date = '2011-09-01'; -- *sees zero hours*

T_2 select sum(hours) from assignments where empid = 12 and date = '2011-09-01'; -- *sees zero hours*

T_1 insert into assignments (empid, project, date, hours) values (12, 'p101', '2011-09-01', 6); -- *adds six-hour assignment*

T_2 insert into assignments (empid, project, date, hours) values (12, 'p102', '2011-09-01', 5); -- *adds five-hour assignment*

Interpretation: business rules limit employees to eight scheduled hours per day. Employee 12 is now scheduled for eleven hours.

Dependency Theory

Dependencies are typed, ordered conflicts between pairs of transactions.

- Write-write dependency: T$_i$--ww→T$_j$
  - T$_i$ writes x, T$_j$ writes successor version of x.
  - commit(T$_i$) $<$ start(T$_j$)
- Write-read dependency: T$_i$--wr→T$_j$
  - T$_i$ writes x. T$_j$ reads what T$_i$ wrote.
  - Data item or predicate read, conflicting with data item write.
  - commit(T$_i$) $<$ start(T$_j$)
- Read-write anti-dependency: T$_i$--rw→T$_j$
  - T$_i$ reads x, T$_j$ writes the successor version of x.
  - start(T$_i$) $<$ commit(T$_j$). T$_i$, T$_j$ may or may not be concurrent.

Note the ordering: given T$_i$ $\rightarrow$ T$_j$, T$_i$ must serialize before T$_j$.

Dependency Serialization Graph

We can model dependencies using a *Dependency Serialization Graph* (DSG). A DSG over a history H is a directed graph where:

- Each node represents a transaction T_i that committed in H.
- Each edge $\mathrm{T}_i \rightarrow \mathrm{T}_j$ represents a dependency between T_i and T_j in H.

An SI history H is serializable iff DSG(H) is acyclic (*Making Snapshot Isolation Serializable*, TODS '05).

This is provable via the Serialization Theorem. DSG(H) and SG(H) represent the same set of paths.

Therefore, we can achieve serializability by preventing dependency cycles.

SI-RW Diagrams

SI-RW diagrams present a time-oriented view of an SI history H.

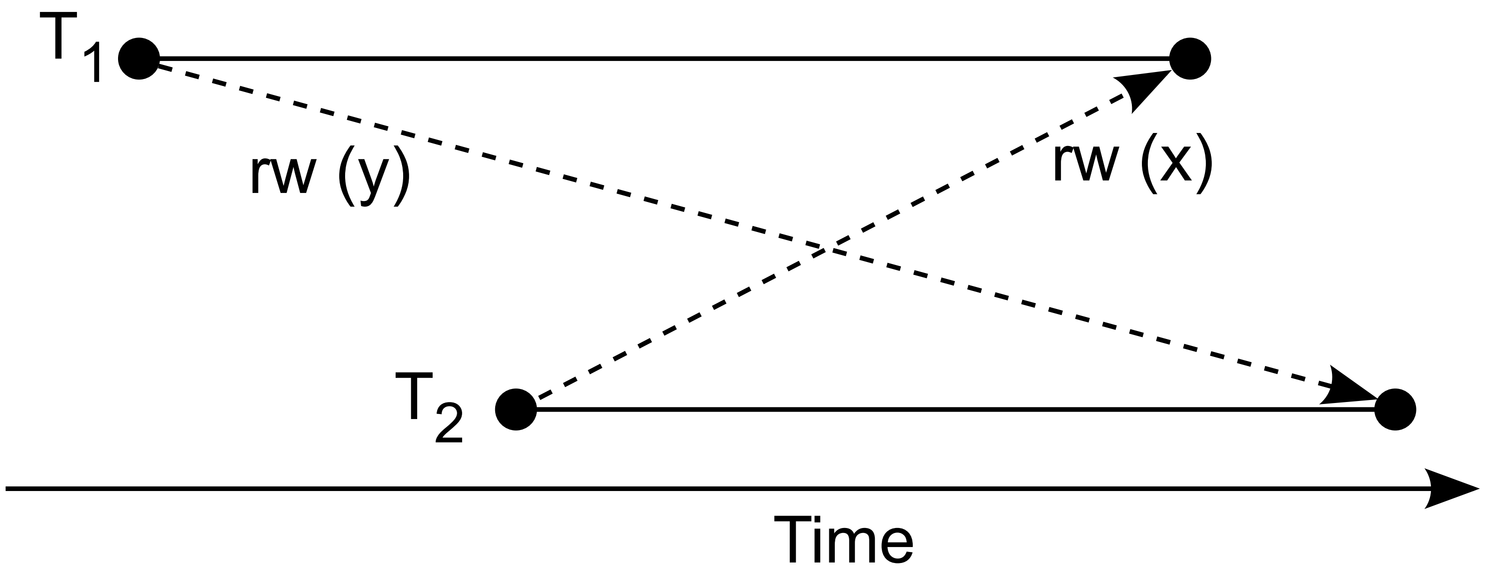
- Each Transaction T_i is represented by two vertices, joined by a solid line.
  - All reads occur at start(T_i), the left vertex.
  - All writes occur at commit(T_i), the right vertex.

- $\mathrm{T}_i \rightarrow \mathrm{T}_j$ dependencies are represented by arrows.
  - T_i--ww$\rightarrow$$\mathrm{T}_j$ - a solid arrow from commit(T_i) to commit(T_j)
  - T_i--wr$\rightarrow$$\mathrm{T}_j$ - a solid arrow from commit(T_i) to start(T_j)
  - T_i--rw$\rightarrow$$\mathrm{T}_j$ - a dashed arrow from start(T_i) to commit(T_j).

SI-RW Diagram For Write Skew

The SI-RW diagram for history H_1:

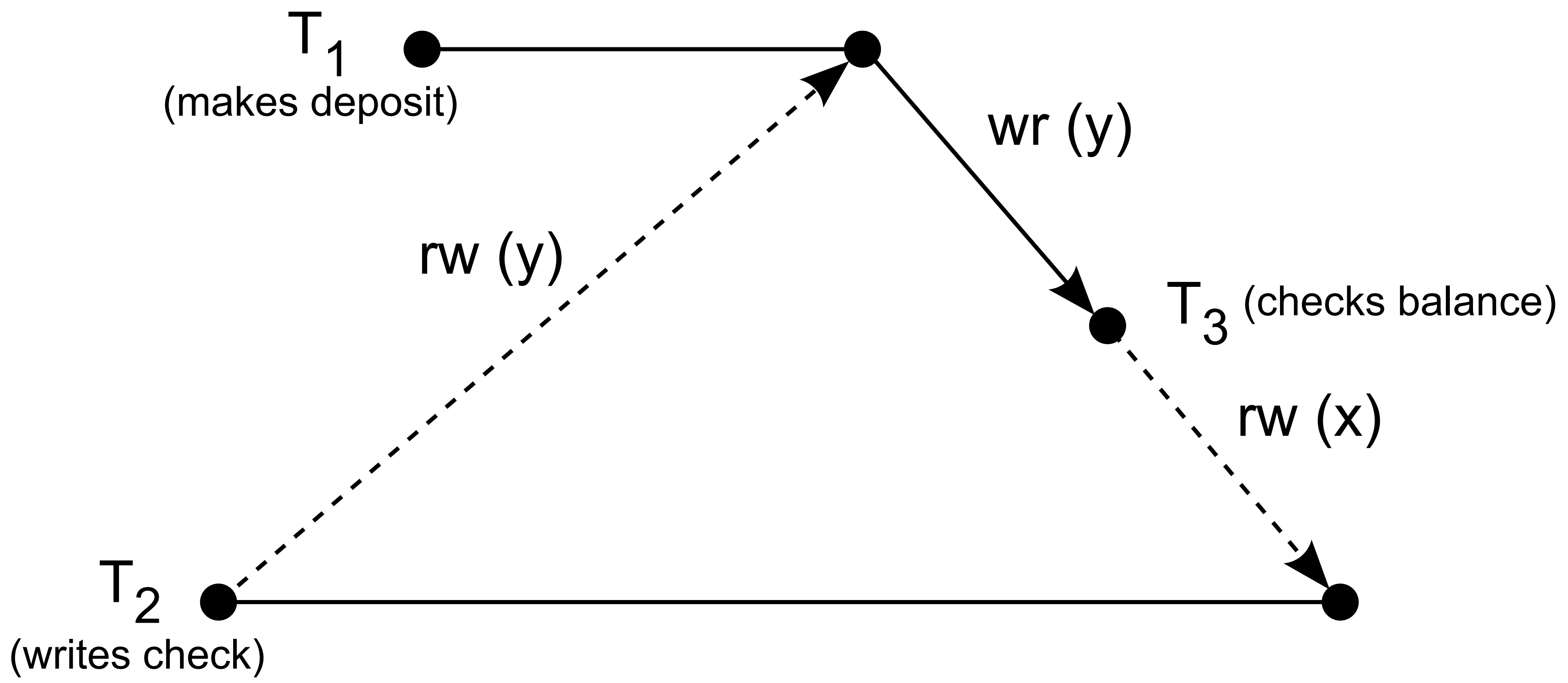
H_1: $r_1(x_0, 100)$,$r_1(y_0, 100)$, $r_2(x_0, 100)$, $r_2(y_0, 100)$,
$w_1(x_1, -50)$, c_1, $w_2(y_2, -50)$, c_2



Note: the cycle T$_1$--rw$\rightarrow$T$_2$--rw$\rightarrow$T$_1$ wasn't formed until *after* commit(T$_1$).

SI-RW Diagram For a Read-Only Anomaly

H_2: $r_2(x_0, 0)$, $r_2(y_0, 0)$, $r_1(y_0, 0)$, $w_1(y_1, 20)$, c_1, $r_3(x_0, 0)$, $r_3(y_1, 20)$, c_3, $w_2(x_2, -11)$, c_2

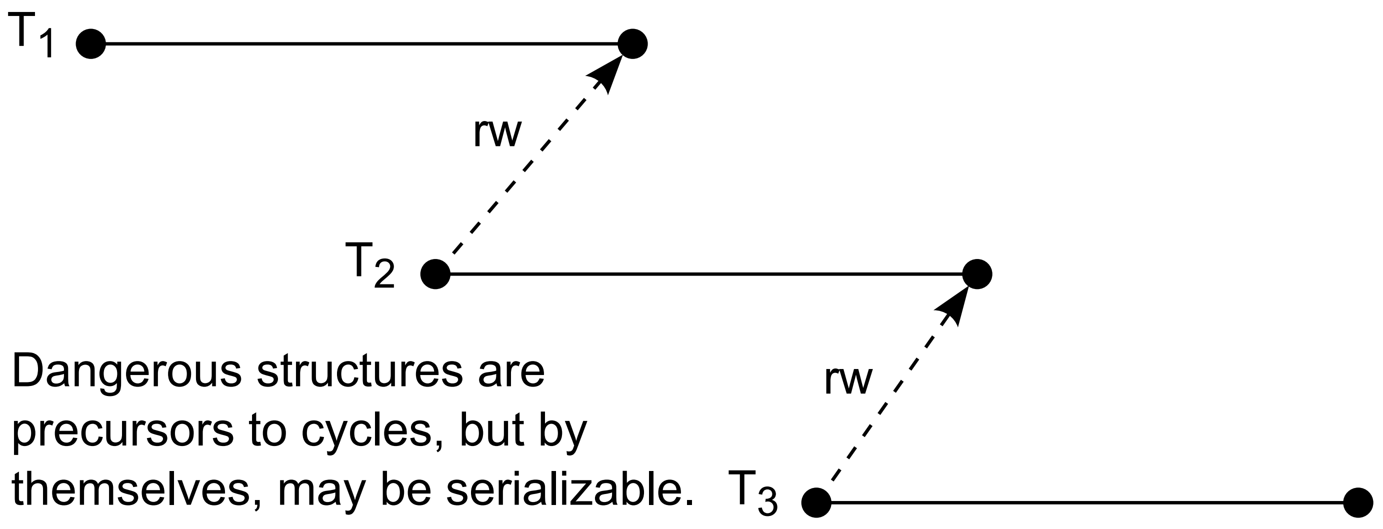


x = checking account, y = savings account.

Dangerous Structures

- Three transactions T_1, T_2, T_3 (perhaps with $T_1 = T_3$).
- T_1 and T_2 are concurrent with T_2--rw$\rightarrow$$T_1$.
- T_2 and T_3 are concurrent with T_3--rw$\rightarrow$$T_2$.

Every non-serializable SI history contains a dangerous structure (*Making Snapshot Isolation Serializable*, TODS '05).



T_1
rw
T_2
rw
T_3
Dangerous structures are precursors to cycles, but by themselves, may be serializable.

An *Essential Dangerous Structure* is a dangerous structure where T_1 commits first.

Two Strategies for Serializable SI

- (Essential) Dangerous Structure Testing
  - Published by Cahill, Röhm, and Fekete in SIGMOD '08 (SSI) and TODS '09 (ESSI). Since implemented in Postgres 9.1.
  - Aborts T_i when T_i creates an (essential) dangerous structure.
  - Can produce false-positive aborts; dangerous structures are only precursors to cycles.
- Cycle Testing (PSSI)
  - Published by Revilak, O'Neil and O'Neil in ICDE '11.
  - Aborts T_i when commit(T_i) would create a dependency cycle.
  - A more precise test, which results in fewer aborts.

PSSI's Design and our InnoDB Prototype

PSSI ensures serializability be detecting dependency cycles.

- Use a lock table to find all $T_i \rightarrow T_j$ dependencies.
  Note: PSSI's "locks" behave differently than in traditional two-phased locking (discussed shortly).
- Use a cycle testing graph (CTG) to test for cycles.
- Use a modified version of index-specific ARIES/IM to prevent phantom anomalies.

We implemented PSSI in MySQL 5.1.31's InnoDB storage engine.

- InnoDB supports multiversioning (MVCC), but not SI.
  InnoDB provides serializability through S2PL.
- InnoDB has a good implementation of ARIES range locking.

PSSI Lock Table

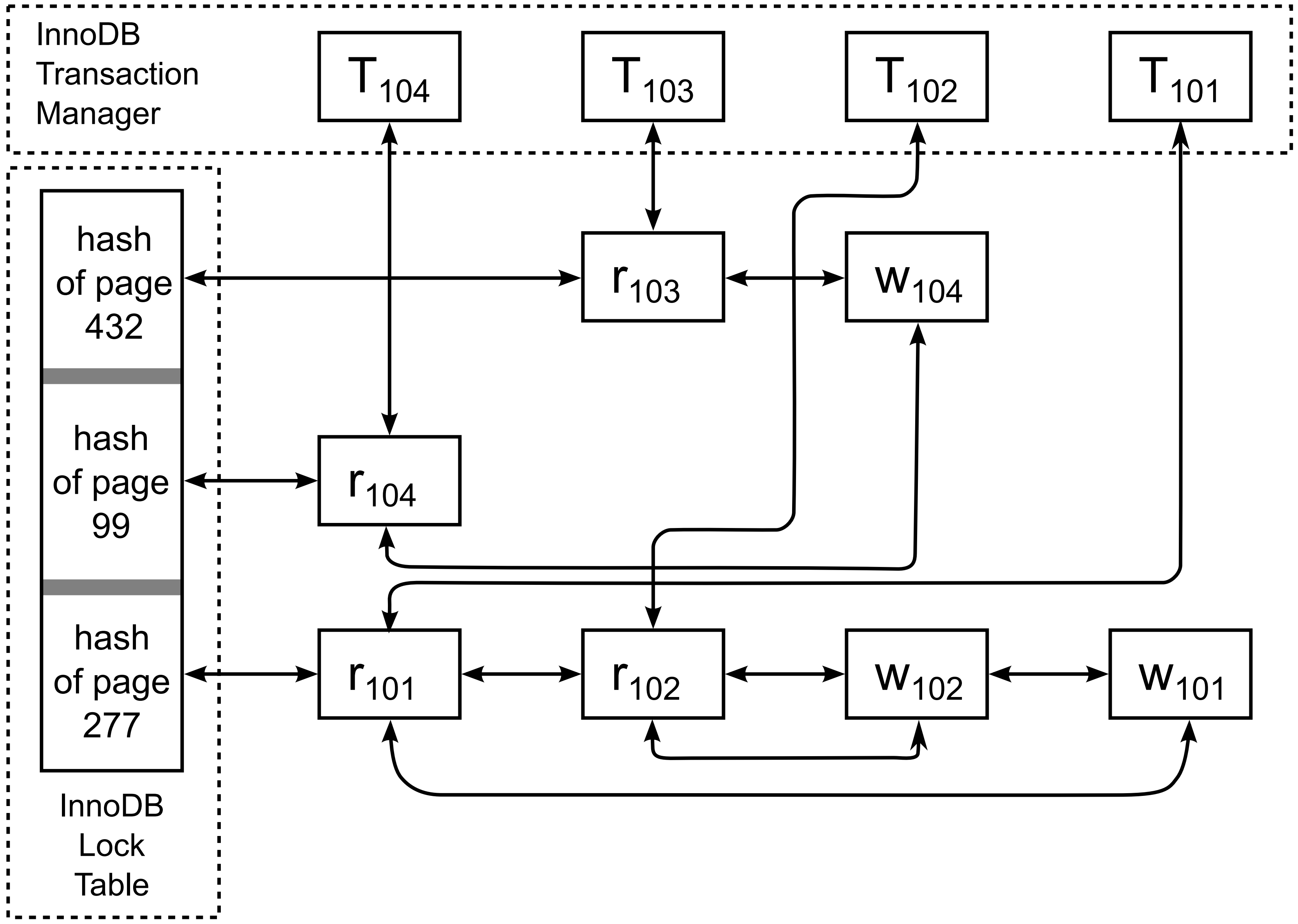
PSSI uses the lock table to detect conflicts, and to find dependencies between transactions.

- The lock table keeps track of all reads and writes.
- Looks like a traditional lock manager, but behaves differently.
  - Requires separate notions of *lock wait* and *lock conflict*.
  - Write-write conflicts cause lock waits (as needed to support FUW).
  - Read-Write conflicts never cause lock waits.
- PSSI assumes that all dependencies can be found via conflicts on data items.
  - ARIES/IM makes this possible.

Integrating the Lock Table with InnoDB

- InnoDB uses one lock control block (LCB) per transaction, per lock mode, per page.
- Each LCB has a bitmap: bit i is on if record i (on page p) is locked.
- Bitmaps intersection allows us to quickly find conflicts on a given page p
- The lock table is a multilist
  - Vertical links group LCBs for a single transaction T_i. Reads appear at the top, writes appear at the bottom.
  - Horizontal links are *lock chains* - all locks that fall on a single page p. Reads appear at start(T_i), writes appear at commit(T_i).

PSSI Lock Table Diagram



PSSI's Cycle Testing Graph (CTG)

The CTG is a directed graph.

- Each node is a transaction T_i.
- Each edge is a $T_i \rightarrow T_j$ dependency.
- CTG is a suffix of a complete history, containing only:
  - Committed transactions with potential to become part of a cycle ("zombie transactions").
  - The currently committing transaction T_k.
- T_k commits if CTG $\cup$ T_k is acyclic.

PSSI tests for cycles by performing a depth-first-search, starting from the committing transaction T_k. (One cycle test per transaction, thus very fast.)

CTG Pruning

If T_i can't become part of a future cycle, then we'd like to prune (remove) T_i from the CTG.

Pruning Theorem: We can prune T_i if:

1. T_i has no in-edges, and
2. T_i committed before the oldest active transaction started.

The pruning theorem comes from the following observations:

- To be part of a cycle, T_i needs an in-edge and an out-edge
- Condition (1) guarantees that T_i has no in-edges
- Condition (2) guarantees that any future edges are out-edges.

Pruning happens each time a transaction commits or aborts.
When T_i is pruned, T_i's locks are removed from the lock table.
This is similar to SGT pruning (described in BHG '87).

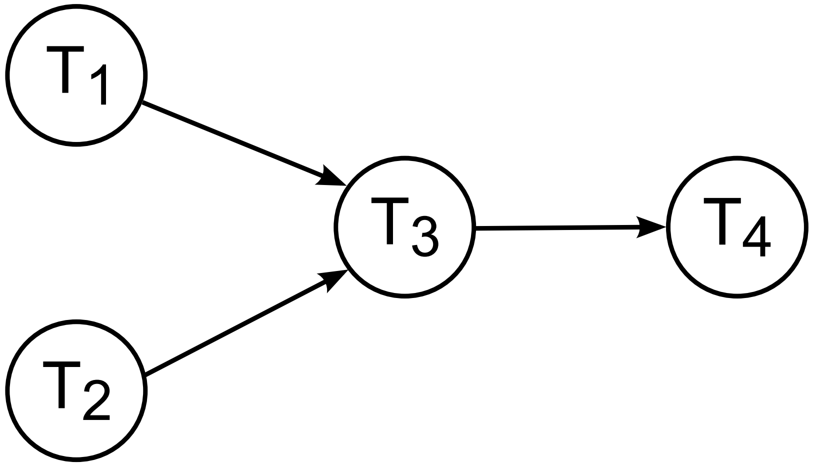
CTG Pruning Algorithm

```
CTG-Prune():
    let $S$ = the set of prunable transactions
    for each $\mathrm{T}_i \in S$:
        CTG-Prune-Recursive($\mathrm{T}_i$)

CTG-Prune-Recursive($\mathrm{T}_i$):
    if $\mathrm{T}_i$ does not satisfy the pruning theorem:
        return
    let $R = \{\ \mathrm{T}_j \mid \text{there is an edge } \mathrm{T}_i \to \mathrm{T}_j\ \}$
    remove $\mathrm{T}_i$ from the CTG
    for each $\mathrm{T}_j \in R$:
        CTG-Prune-Recursive($\mathrm{T}_j$)
```

CTG Pruning Example

Assume T_1 is the first transaction to start, and the last transaction to commit.



- $S = \{T_1, T_2\}$
- Pruning Order: T_1, T_2, T_3, T_4

Pruning is a topological sort, and pruning order gives an equivalent serial history.

Range Locking and Phantom Avoidance

InnoDB range-locking is a variant of index-specific ARIES/IM.

To prevent predicate write skew anomalies, range locking must handle two cases:

- Insert before range scan
- Range scan before insert

We'll cover these cases one at a time.

Insert Before Range Scan

- T_1 inserts x_1. Assume no conflict is found at the time of insert.
    - x_1 is labeled with T_1's transaction id. This transaction id acts as an *implicit* lock on x. (No LCB in lock table).
- T_2 (concurrent with T_1) does a range scan across x.
    - T_2 examines x_1's transaction id. x_1's transaction it tells T_2 that a conflict is occurring.
    - T_2 adds $w_1(x)$ to the lock table, on behalf of T_1.
    - T_2 adds $r_2(x)$ to the lock table, on behalf of itself.
- These two locks – $w_1(x)$ and $r_2(x)$ – allow PSSI to detect a T_2--rw$\rightarrow$$\text{T}_1$ dependency.

Note: T_2 only examines x_1's row header and x_1 does not affect the results of T_2's range scan query. T_2 merely notes that a conflict occurred.

Range Scan Before Insert

- T_1 performs a range scan. T_1 locks each record examined.
- T_2 (concurrent) wishes to insert x into the range that T_1 scanned.
  - T_2 determines the position where x will be inserted
  - T_2 checks for conflicting locks on the next key (call it z). T_2 sees $r_1(z)$ – conflicting operation.
  - T_2 inserts x
  - T_2 adds $w_2(x)$ to the lock table, on behalf of itself.
  - T_2 adds $r_1(x)$ to the lock table, on behalf of T_1.

Note that T_1 has $r_1(x)$, even though T_1 never examined x!

These two locks – $r_1(x)$, $w_2(x)$ – allow PSSI to detect the T_1--rw$\rightarrow$$T_2$ dependency.

Summary of Range Locking

- If there's no conflict at the time of insert, then it's not necessary to add an LCB to the lock table (implicit locking).
    - An LCB will be added later, if a conflict occurs.
- Transactions can take locks on each others behalf, to note conflicts that occur.
- Deletes are treated like updates. Deletes turn on a "deleted" bit in the affected record. (i.e., deletes create *dead versions*)

The SICycles Workload

SICycles is a parametrized workload, which allows dependency cycles of varying lengths to form in a variety of ways.

- 1MM row "bench" table, 100-bytes per row.
  - Important columns: **kseq**, krandseq, kval
- $sku n\text{-}h$ configuration. Each transaction runs:
  - k select statements (by krandseq),
  - n update statements (by krandseq).
  - All select and update statements choose rows from a hotspot of size h.
  - randomized 3 ms $\pm$ 50% delay between statements.
  - No think time between transactions
- Vary k, n, h, and multi-programming level (MPL).

Why not TPC-C? TPC-C doesn't create dependency cycles (TPC-C is serializable under ordinary SI).

Experimental Setup

Two-machine client/server setup:

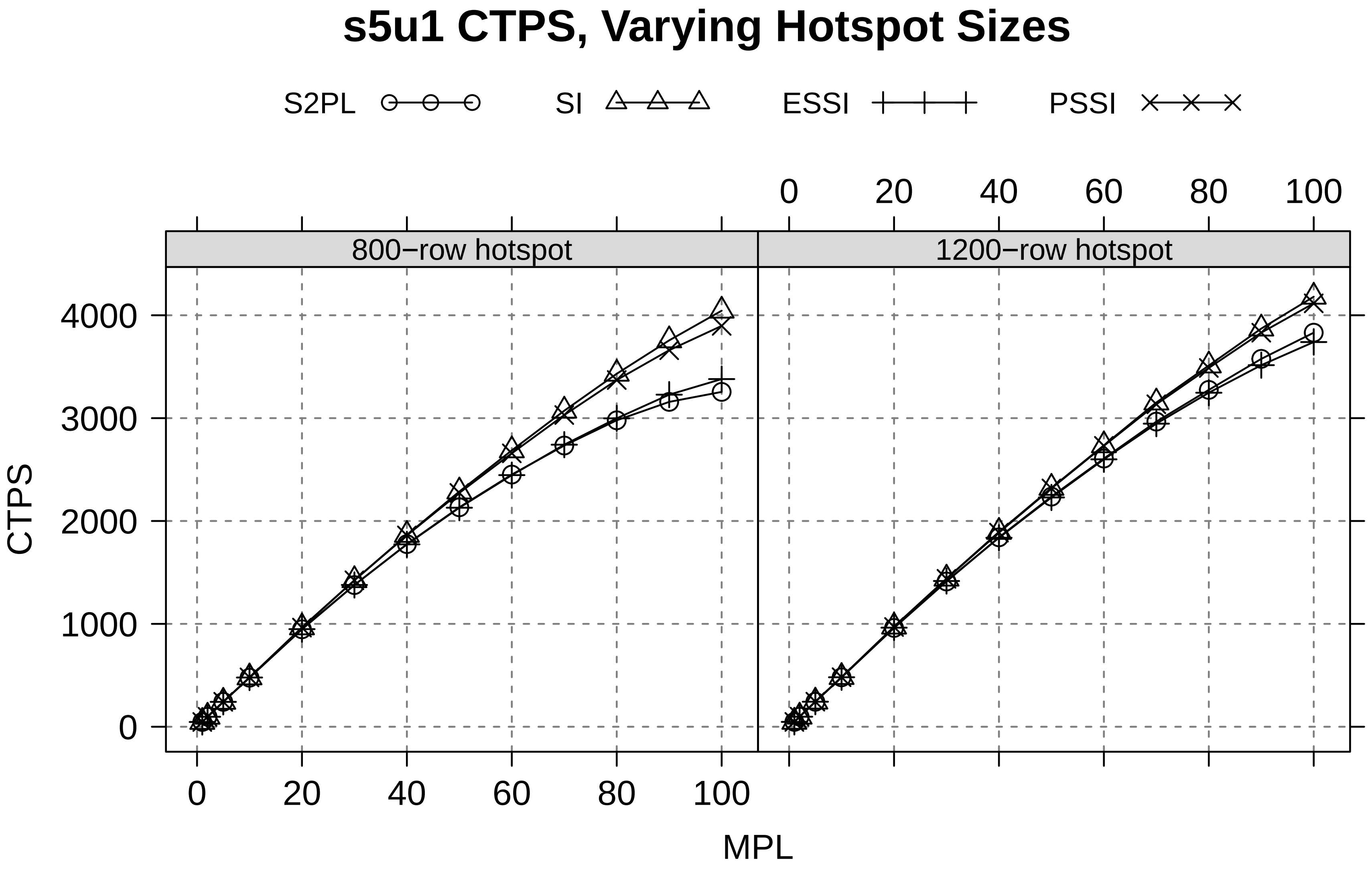
- SICycles client is java program, running on a dual-core GNU/Linux system
- mysqld server runs on a quad-core GNU/Linux system.
    - Table data stored on a 7200 RPM SATA disk (ext4).
    - Transaction logs written to an Intel X25-E SSD (also ext4).

Two mysqld binaries used:

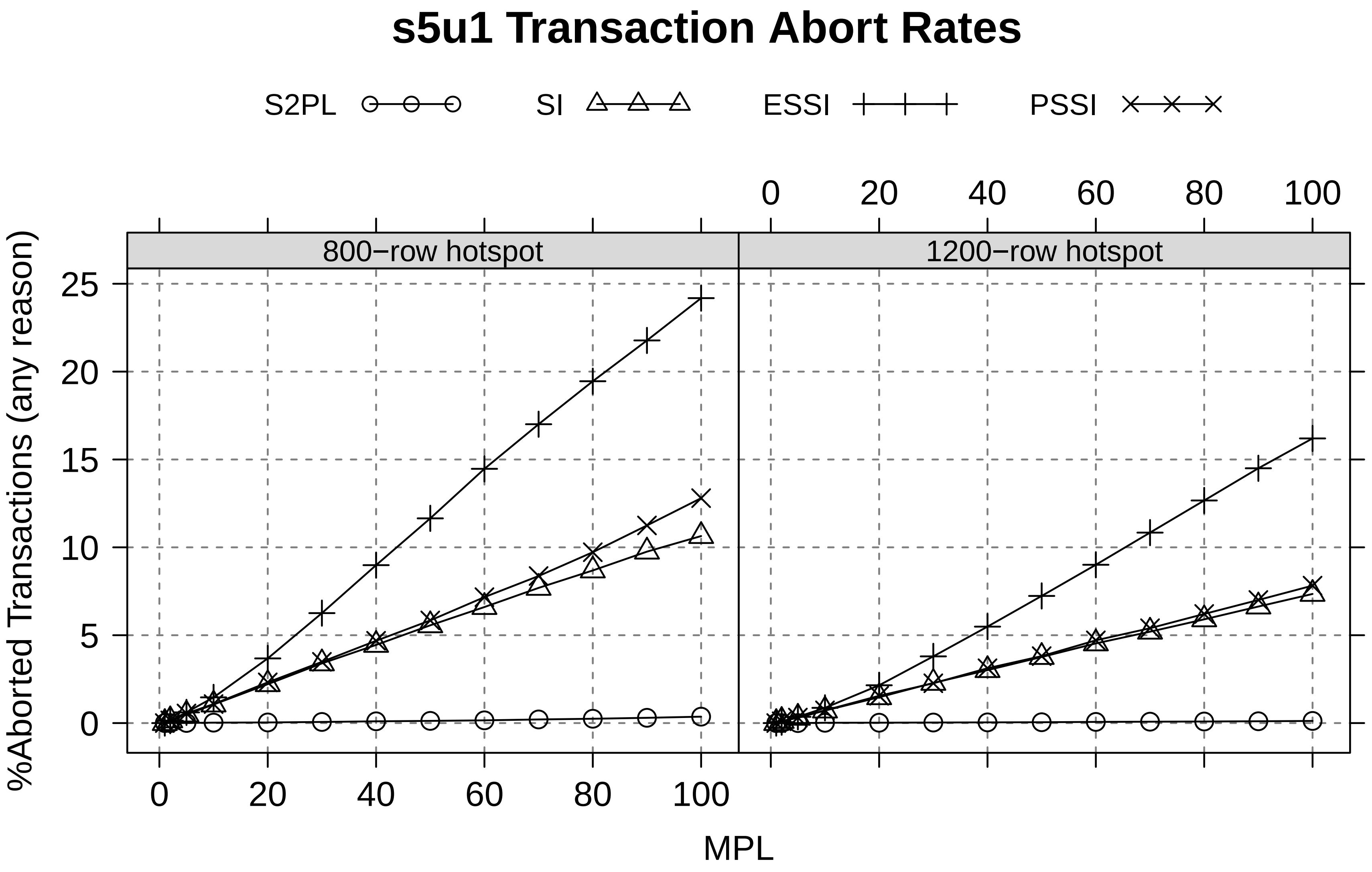
- Our MySQL/InnoDB prototype, for SI, PSSI, and (our implementation of) ESSI.
- A standard-distribution MySQL for S2PL.

All tests use durable group commit. Both binaries use late lock release (locks are held until after log flush).

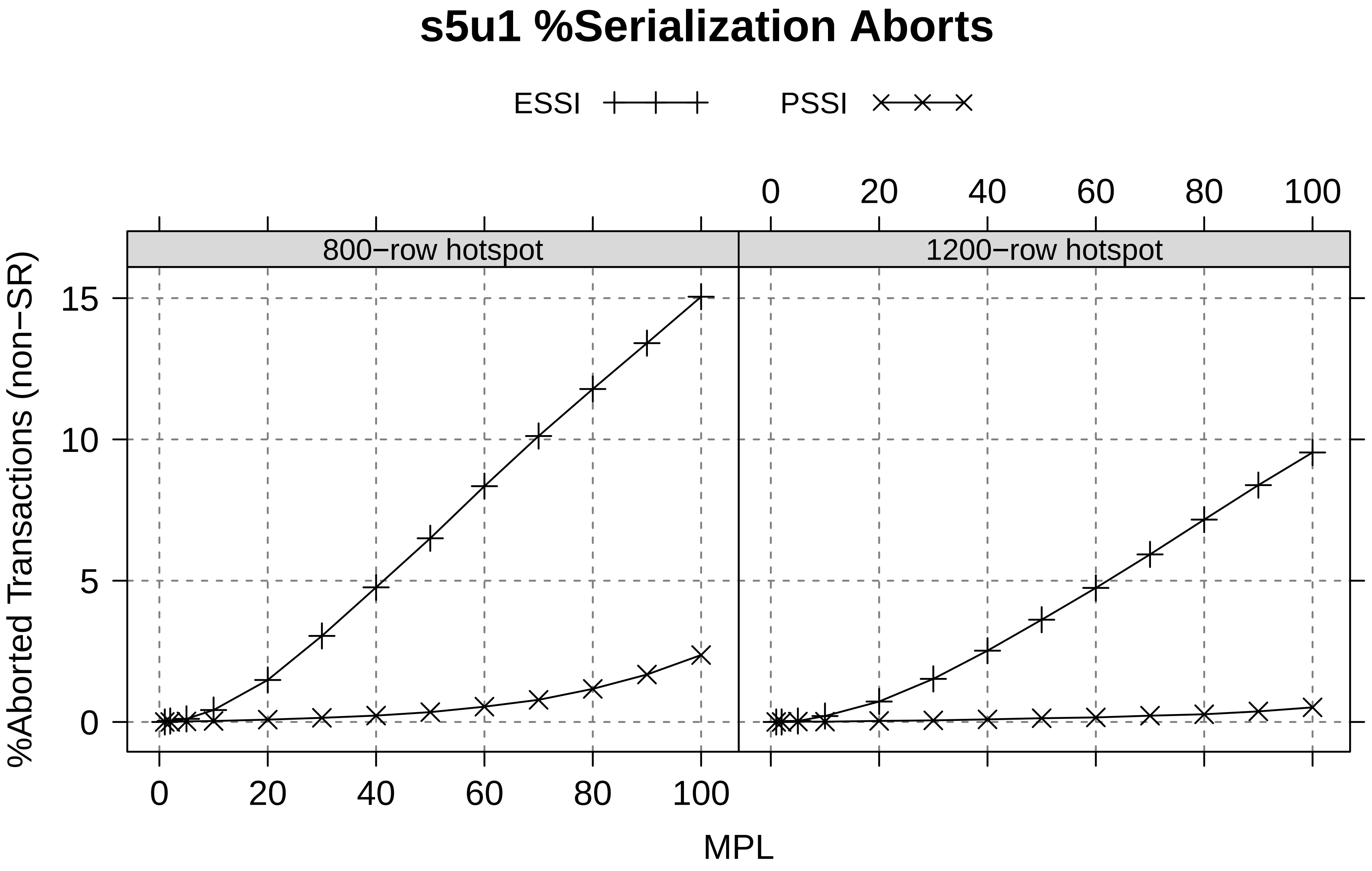
CTPS Measurements



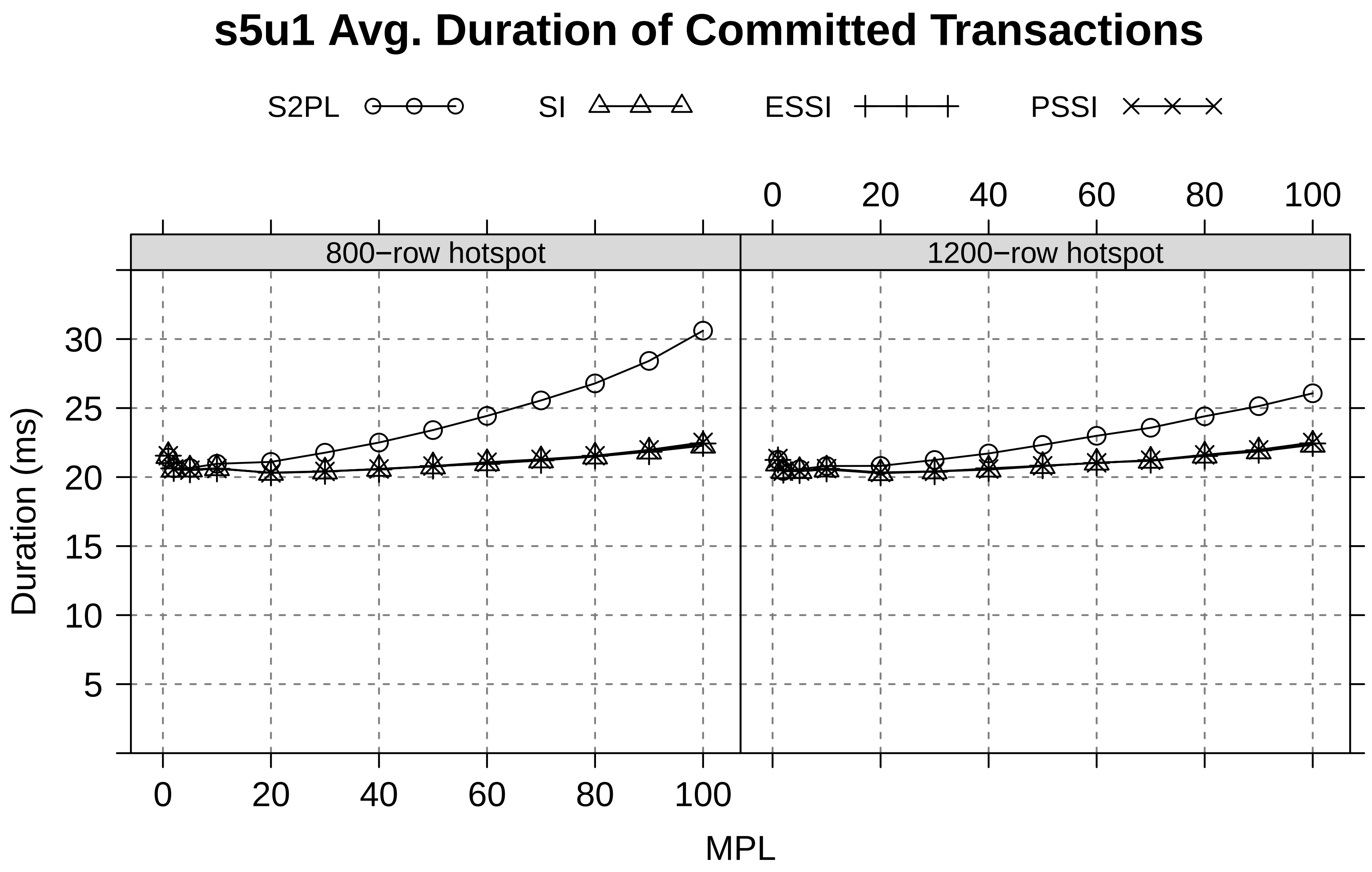
Abort Rate Measurements



Serialization Abort Rate Measurements



Transaction Duration Measurements



Conclusion

- Snapshot Isolation: good throughput, but doesn't provide serializability.
- PSSI: makes SI serializable, with a minimum of false-positive aborts.

Key elements of PSSI:

- Lock Table to find dependencies.
- Cycle Testing Graph to find cycles.
- Pruning to keep the CTG small.
- A variant of ARIES/IM to prevent predicate/phantom anomalies

References

Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil.
"A Critique of ANSI SQL Isolation Levels."
In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pp. 1–10, New York, NY, USA, 1995. ACM.

Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman.
Concurrency Control and Recovery in Database Systems.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

Michael J. Cahill, Uwe Röhm, and Alan D. Fekete.
"Serializable Isolation for Snapshot Databases."
In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 729–738, New York, NY, USA, 2008. ACM.

Michael J. Cahill, Uwe Röhm, and Alan D. Fekete.
"Serializable Isolation for Snapshot Databases."
ACM Trans. Database Syst., **34**(4):1–42, 2009.

Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha.
"Making Snapshot Isolation Serializable."
ACM Trans. Database Syst., **30**(2):492–528, 2005.

Stephen Revilak, Patrick O'Neil, and Elizabeth O'Neil.
"Precisely Serializable Snapshot Isolation."
In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, pp. 482–493, 2011.

Thank You