IN2267 – Transaction Systems

Week 4: Snapshot Isolation Concurrency Control

Dr. Uwe Röhm
The University of Sydney

Learning Objectives

- **Background**
  - Reprise: Concurrency Control Approaches
  - Synchronization Problems & ANSI SQL Isolation Levels
- Optimistic Concurrency Control
  - Snapshot Isolation
  - SI Implementation Details
  - Serialisable Snapshot Isolation
- Outlook
  - SI on multi-core CPUs

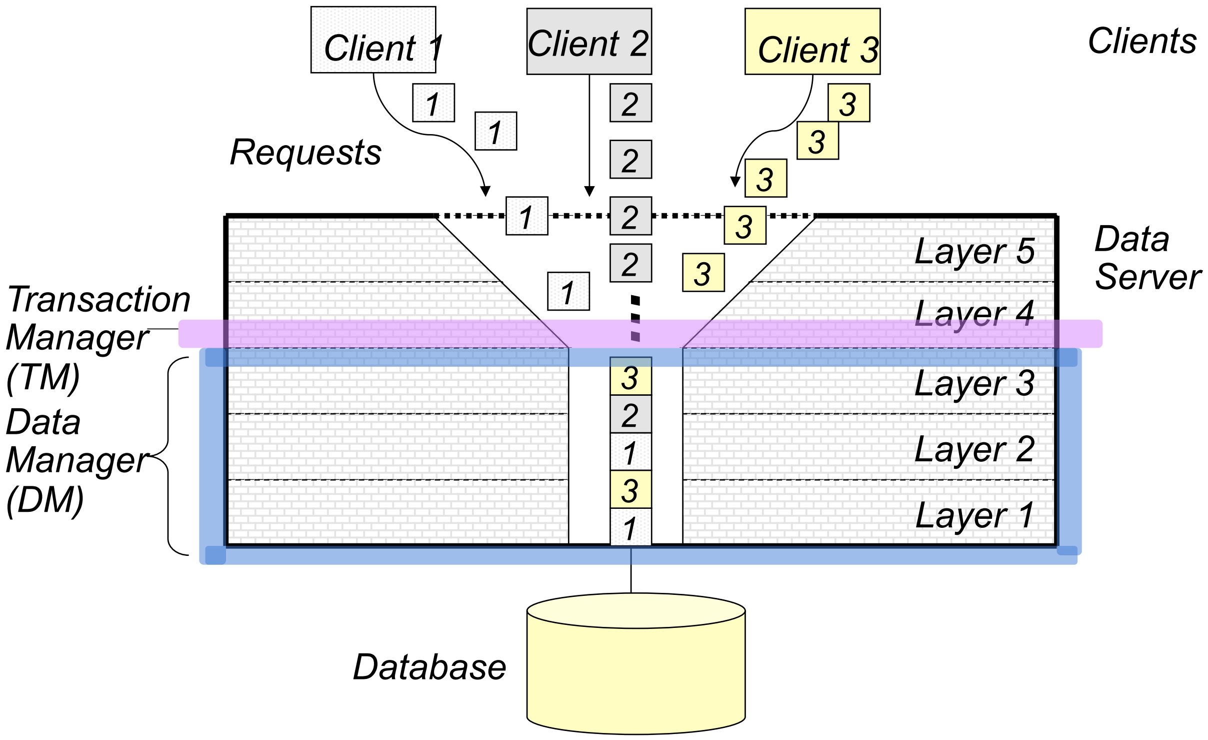
Based on USydney slides from U. Roehm and M. Cahill,
and Weikum/Vossen (2002) "Transactional Information Systems"

Concurrency Control

- The **concurrency control** of a DBMS is responsible for enforcing serializability among concurrent transactions
  - Two important techniques: *Locking* and *Versioning*
- Note: In addition to serializable, DBMSs implement less stringent isolation levels
  - Serializable schedules correct for *all* applications
  - Less stringent levels *do not* guarantee correctness for all applications, but are correct for *some*
  - Application programmer is responsible for choosing appropriate level

---

Transaction Scheduler



Potential Anomalies

- **lost update** ('dirty write'):
  updating a value that was already updated by a concurrent, uncommitted transaction.
- **dirty read**:
  reading a value that was updated by a concurrent, uncommitted transaction
- **non-repeatable read** ('fuzzy read'):
  reading a value twice gives different results because of a concurrent update by a different transaction in between
- **phantom read**:
  using the same selection criteria on a table twice gives different result sets, because a concurrent updater deleted or inserted elements satisfying the selection criteria

---

ANSI SQL Isolation Levels

- Defined in terms of anomalies
  - Anomaly prohibited at one level is also prohibited at all higher levels
  - READ UNCOMMITTED: all anomalies possible
  - READ COMMITTED: dirty read prohibited
  - REPEATABLE READ: reads of individual tuples are repeatable (but phantoms are possible)
  - SERIALIZABLE: phantoms prohibited; transaction execution is serializable
- Serializable is according to SQL standard the default…
  - In practice, most systems have weaker default level! (Oracle!)
- Lower degrees of consistency useful for gathering approximate information about the database, e.g., statistics for query optimizer.

Comparison of SQL Isolation Levels

Isolation Level	Lost Update	Dirty Read	Unrepeatable Read	Phantom
READ UNCOMMITTED	not possible	*possible*	*possible*	*possible*
READ COMITTED	not possible	not possible	*possible*	*possible*
REPEATABLE READ	not possible	not possible	not possible	*possible*
SERIALIZABLE	not possible	not possible	not possible	not possible

Note: ANSI SQL Isolation Level *SERIALIZABLE*
!=
Definition in serialisability theory
(such as conflict serialisability)

Locks in Relational Databases

- DBMS guarantees that each SQL statement is isolated
- Early (non-strict) lock release used to implement levels
  - Short-term locks - held for duration of single statement
  - Long-term locks - held until transaction completes (strict)
- At all levels, transactions obtain long-term write locks
- This means for isolation levels:
  - READ UNCOMMITTED - no read locks (dirty reads possible since transaction can read a write-locked item)
  - READ COMMITTED - short-term read locks on rows (non-repeatable reads possible since transaction releases read lock after reading)
  - REPEATABLE READ - long-term read locks on rows (phantoms possible)
  - SERIALIZABLE - combination of table, row, and index locks

Agenda

- Background
  - Reprise: Concurrency Control Approaches
  - Synchronization Problems & ANSI SQL Isolation Levels
- **Optimistic Concurrency Control**
  - Snapshot Isolation
  - SI Implementation Details
  - Serialisable Snapshot Isolation
- Outlook
  - SI on multi-core CPUs

Optimistic Concurrency Control

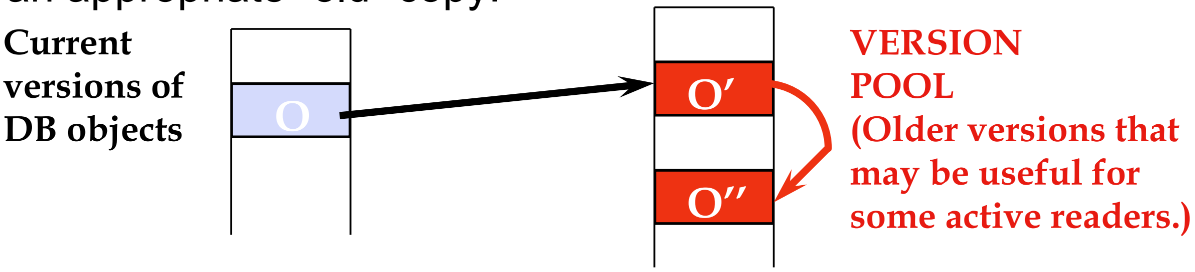
- Locking is a conservative approach in which conflicts are prevented. Disadvantages:
  - Lock management overhead.
  - Deadlock detection/resolution.
  - Lock contention for heavily used objects.
- If conflicts are rare, we might be able to gain concurrency by not locking, and instead checking for conflicts before transactions commit.
  - Optimistic, validating CC
  - Multiversion CC

Snapshot Isolation – Conceptual Idea

- Every transaction reads from a consistent ***snapshot*** (copy) of the database (the db state of when tx started)
- Writes are collected into a transaction's ***writeset***
  - Writeset is not visible to concurrent transactions
- At commit time, the writeset is compared to the writesets of all concurrent transactions.
  - If they are disjoint (no overlap), then they are applied to the actual database => **commit**
  - If there's an overlap with the writeset of a concurrent, but already committed transaction, the later transaction must **abort**
    - **=> "First Committer Wins" rule**

In Practice: Snapshot Isolation (SI)

- A **multiversion concurrency control** mechanism which was described in SIGMOD '95 by H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil
  - Incremental implementation of an optimistic concurrency control scheme
- **Core Idea:** Let writers create a "new" copy while readers use an appropriate "old" copy.



❖ **Readers are always allowed to proceed.**
  – But may be blocked until writer commits.

Reads with Snapshot Isolation

- **Multiversion database**: The old value of an item is *not* overwritten when it is updated (no ‘in-place updates’). Instead, a new version is created
- Read of an item does not necessarily give latest value
- Instead, use old versions (kept with timestamps) to find value that had been most recently committed *at the time the transaction started*
  - Exception: if the txn has modified the item, use the value it wrote itself
- The transaction sees a “snapshot” of the database, at an earlier time
  - Intuition: this should be consistent, if database was consistent before
  - No read locks necessary: a transaction reads all values from latest snapshot at time it started. Thus, read/only transactions do not wait.

Writes with Snapshot Isolation

- A transaction *T* that has updated *x* can commit if no other transaction that concurrrently updated *x* has committed
  - **“First-committer-wins” rule**:
  - Updater *T* will not be allowed to commit if any other transaction has committed and *installed a changed* value for that item, between T’s start (snapshot) and T’s commit
  - Similar to optimistic validation-based cc, but only write-sets are checked
- T must hold X-lock on modified items at time of commit, to install them. In practice, commit-duration X-locks may be set when write executes. These help to allow conflicting modifications to be detected (and T aborted) when T tries to write the item, instead of waiting till T tries to commit.

Benefits of SI

- Reading is *never* blocked, and also doesn't block other transactions' activities
  - Fast performance similar to Read Committed
- Avoids the usual anomalies
  - No dirty read
  - No lost update
  - No inconsistent read
  - Set-based selects are repeatable (no phantoms)

  - Note: not Write-Skews – cf. later slides

Who does this?

- Oracle: used for "Isolation Level Serializable"
  - But does *not* guarantee serializable execution as defined in standard transaction management theory!
- PostgreSQL: used for "Isolation Level Serializable"
  - As of version 9.1 guarantees serializable execution, but not earlier
- Available in Microsoft SQL Server 2005 and above as "Isolation Level Snapshot"
  - If mssql db is configured to provide snapshots
- Berkeley DB
- MySQL / InnoDB (sort of)

Agenda

- Background
  - Reprise: Concurrency Control Approaches
  - Synchronization Problems & ANSI SQL Isolation Levels

- **Optimistic Concurrency Control**
  - Snapshot Isolation
  - SI Implementation Details
  - Serialisable Snapshot Isolation

- Outlook
  - SI on multi-core CPUs

SI Design Choices

- Tuple Versions
  - Store old versions or generate as required?

- Granularity
  - should individual records be versioned, or pages?
  - (or even tables?)

- How is a snapshot represented?
  (“what is time?”)

SI Common Themes

- Almost every implementation takes locks for updates
  - This blocks other updates until commit / abort
  - Guarantees forward progress
  - Reduces conflict-abort-retry thrashing

- First-committer-wins implemented as “has a version been committed since I started?”

PostgreSQL: Intro

- Full RDBMS, long history
- Provides SI when you ask for REPEADABLE READ or SERIALIZABLE
- Stores old versions of rows in the database
  - Needs regular VACUUMing

pgsql: SnapshotData

```
typedef struct SnapshotData
{
  SnapshotSatisfiesFunc satisfies; /* tuple test function */

 /*
  * The remaining fields are used only for MVCC snapshots, and are normally
  * just zeroes in special snapshots. (But xmin and xmax are used
  * specially by HeapTupleSatisfiesDirty.)
  *
  * An MVCC snapshot can never see the effects of XIDs >= xmax. It can see
  * the effects of all older XIDs except those listed in the snapshot. Xmin
  * is stored as an optimization to avoid needing to search the XID arrays
  * for most tuples.
  */
  TransactionId xmin;     /* all XID <  xmin are visible to me   */
  TransactionId xmax;     /* all XID >= xmax are invisible to me */
  TransactionId *xip;     /* array of xact IDs in progress */
  uint32 xcnt;            /* # of xact ids in xip[]        */
  /* note: all ids in xip[] satisfy xmin <= xip[i] < xmax */
  int32 subxcnt;          /* # of xact ids in subxip[], -1 if overflow */
  TransactionId *subxip; /* array of subxact IDs in progress */

…
} SnapshotData;
```

[src/include/utils/snapshot.h]

Pgsql: Tuple Visibility

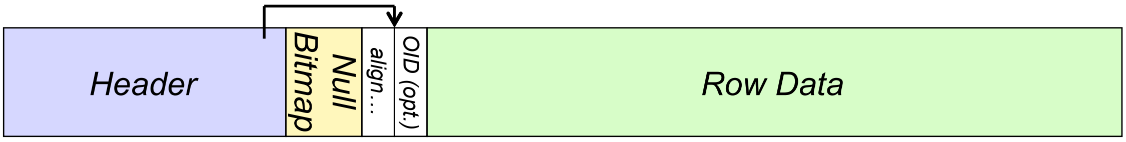
[src/backend/utils/time/tqual.c]

```
 *      mao says 17 march 1993:  the tests in this routine are correct;
 *      if you think they're not, you're wrong, and you should think
 *      about it again.  i know, it happened to me.  we don't need to
 *      check commit time against the start time of this transaction
 *      because 2ph locking protects us from doing the wrong thing.
 *      if you mess around here, you'll break serializability.  the only
 *      problem with this code is that it does the wrong thing for system
 *      catalog updates, because the catalogs aren't subject to 2ph, so
 *      the serializability guarantees we provide don't extend to xacts
 *      that do catalog accesses.  this is unfortunate, but not critical
 */
bool
HeapTupleSatisfiesNow(HeapTupleHeader tuple, Snapshot snapshot, Buffer buffer)
{ … }
```

- Tuple header defines a closed-open transaction-time interval
  - Basic Idea: A Tuple is visible iff
    - xmin is a committed transaction ID < own transaction ID and not in-progress at transaction start.
    - xmax is either blank, or greater than the start transaction ID and in-progress at transaction start

Row Format in PostgreSQL

Row = RowHeader + NullBitmap + *alignment padding* + *(OID)* + RowData



- RowHeader:
  - 23 Bytes; cf. next slide
- NullBitmap
  - variable len: either 0 (if all NOT NULL) or ((|*Columns*| + 7) / 8) Bytes
  - one bit per attribute; 1 if NULL, 0 otherwise
- *OID (PostgreSQL speciality as it supports objects)*
  - *fix 4 Bytes (optional,depending whether table WITH OIDs or not)*
- RowData = FixedColumns + VarColumns
  - FixedColumns: directly stored & aligned!
  - VarColumns = varattrib + userdata + aligned
    - varattrib = 4 bytes length words including 2 bits for compression/TOAST flags

---

Row Format in PostgreSQL (cont’d)

- Row Header structure
  - 23 bytes (plus bitmap plus padding; cf. *t_hoff* value as ‘pointer’)
  - Cf. **src/include/access/htup.h**:
    `typedef struct HeapTupleHeaderData`
  - Some information on visibility of a tuple for current transaction snapshot or newer version (needed for snapshot isolation algorithm)
    - **t_xmin** TransactionId 4 bytes insert XID stamp
    - **t_xmax** TransactionId 4 bytes delete XID stamp
    - **t_cid** CommandId 4 bytes insert CID stamp *(actual a UNION struct)*
    - **t_ctid** ItemPointerData 6 bytes current TID of this or newer row version
  - How long is this row? Is it variable length? Does it have NULLs?
    - **t_natts** int16 2 bytes number of attributes
    - **t_infomask** uint16 2 bytes various flag bits
      - e.g. HAS_NULL | HASVARWIDTH | HASOID | locks(!)
    - **t_hoff** uint8 1 byte /* sizeof header incl. bitmap, padding */

pgSQL SI: Tuple Visibility Example

Current Transaction ID: 100

In-Progress Transactions: 25
50
75

Cre 30 Exp		Visible
Cre 30 Exp 80		Skip
Cre 30 Exp 110		Visible
Cre 30 Exp 75		Visible
Cre 50 Exp		Skip
Cre 110 Exp		Skip

InnoDB: Intro

- Transactional backend for MySQL
  - MySQL supports different storage engines!
- Only needs to deal with read / writes of rows
  - MySQL looks after SQL and query processing
- Generates old values on demand
  - Uses “undo” records from the log

InnoDB: Concurrency Control

- MVCC but not SI
  - Read-only Transactions (pure queries) read from a snapshot
  - Locking reads (including updates) read most recently committed value
  - No first-committer-wins rule

InnoDB: read_view

```
/* Read view lists the trx ids of those transactions for which a consistent
   read should not see the modifications to the database. */

struct read_view_t{
…
        trx_id_t  low_limit_no;   /* The view does not need to see the undo
                                   logs for transactions whose transaction number
                                   is strictly smaller (<) than this value: they
                                  can be removed in purge if not needed by other
                                  views */
        trx_id_t  low_limit_id;   /* The read should not see any transaction
                                  with trx id >= this value; in other words, this is the “high water mark” */
        trx_id_t  up_limit_id;    /* The read should see all trx ids which
                                  are strictly smaller (<) than this value; this is the “low water mark”  */
        ulint   n_trx_ids;        /* Number of cells in the trx_ids array */
        trx_id_t* trx_ids;       /* Additional trx ids which the read should
                                 not see: typically, these are the active
                                 transactions at the time when the read is
                                 serialized, except the reading transaction
                                 itself; the trx ids in this array are in a
                                descending order */
        trx_id_t  creator_trx_id; /* trx id of creating transaction, or
                                    (0, 0) used in purge */
        UT_LIST_NODE_T(read_view_t) view_list;
                             /* List of read views in trx_sys */
};
```

[storage/innobase/include/read0read.h]

InnoDB: read old versions

```
/* Constructs the version of a clustered index record which a consistent
read should see. We assume that the trx id stored in rec is such that
the consistent read should not see rec in its present version. */
ulint
row_vers_build_for_consistent_read(
/*==============================*/
            /* out: DB_SUCCESS or DB_MISSING_HISTORY */
  const rec_t*   rec,     /* in: record in a clustered index; */
  read_view_t*    view,   /* in: the consistent read view */
  rec_t**     old_vers)   /* out, own: old version, or NULL if the
            record does not exist in the view, that is,
            it was freshly inserted afterwards */
{
…
  for (;;) {
…
    err = trx_undo_prev_version_build(rec, mtr, version, index, *offsets, heap, &prev_version);
…
    if (prev_version == NULL) {
      /* It was a freshly inserted version */
      *old_vers = NULL;
      err = DB_SUCCESS;
      break;
    }
…
    trx_id = row_get_rec_trx_id(prev_version, index, *offsets);

    if (read_view_sees_trx_id(view, trx_id)) {
…
      err = DB_SUCCESS;
      break;
    }

    version = prev_version;
  }/* for (;;) */
```

---

pgsql and InnoDB: Lessons

- MVCC is used by default
  - Has to work well under most conditions
- Old versions need space somewhere
  - pgsql: in the database file
  - InnoDB: undo records in the log buffer

Summary: SI Design Space

	BDB	pgsql	InnoDB
old versions	store in cache	store on disk	generate on demand
granularity	page	record	record
transaction time	LSNs	snapshot of active txnIDs	snapshot of active txnIDs

---

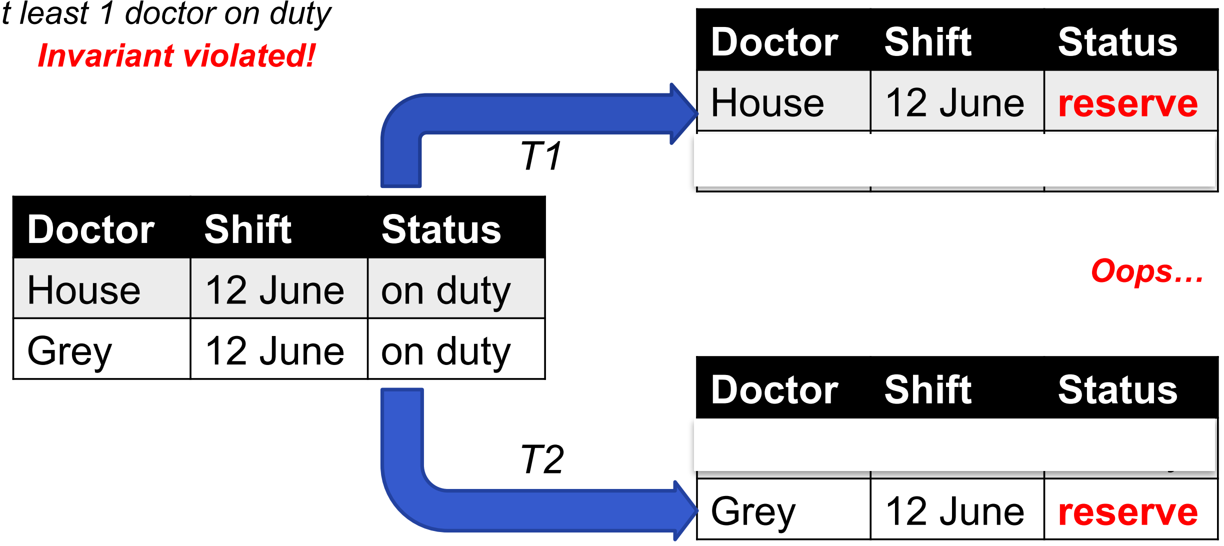
Agenda

- Background
  - Reprise: Concurrency Control Approaches
  - Synchronization Problems & ANSI SQL Isolation Levels
- **Optimistic Concurrency Control**
  - Snapshot Isolation
  - SI Implementation Details
  - Serialisable Snapshot Isolation
- Outlook
  - SI on multi-core CPUs

Problem with SI: Snapshot Isolation ≠ Serializable

Inherent constraint: for every date, there is at least 1 doctor on duty

Invariant violated!



Doctor	Shift	Status
House	12 June	on duty
Grey	12 June	on duty

T1

Doctor	Shift	Status
House	12 June	**reserve**

Oops…

T2

Doctor	Shift	Status
Grey	12 June	**reserve**

Write Skew

- Formally this is known as **Write Skew Problem**
  - SI breaks serializability when transactions modify *different* items, each based on a previous state of the item the other modified
  - This is fairly rare in practice
    - Eg the TPC-C benchmark runs correctly under SI: when transactions conflict due to modifying different data, there is also a shared item they both modify too (like a total quantity) so SI will abort one of them

Vendor Advice

- Oracle: *"Database inconsistencies can result unless such application-level consistency checks are coded with this in mind, even when using serializable transactions."*
- *"PostgreSQL's Serializable mode does not guarantee serializable execution..."*
  - **FIXED since PostgreSQL 9.1!!!**
- SQL Server: *only gives performance advices, but keeps quiet on the correctness issue...*

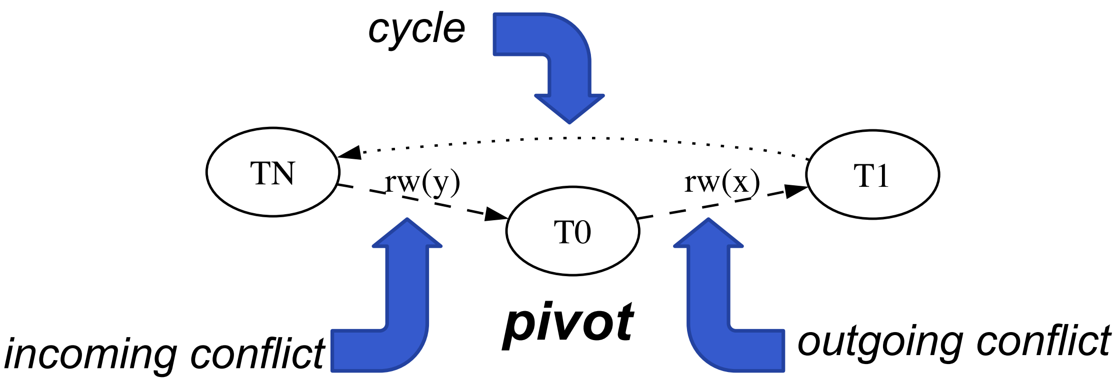
Serializable SI

- Theory exists about how write-skews can be detected
  - A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, D. Shasha in TODS2005: "*Making Snapshot Isolation Serializable*"
    - Analyze the graph of transaction conflicts
    - Conditions on the graph for application to be serializable at SI; def. of *dangerous structure*
- Solution: Two Approaches
  - Introduce artificial ww-conflicts to application to trigger first-committer-wins rule
    - Requires semantic program analysis before -> NP Hard
  - Modify SI CC to identify 'dangerous patterns' in concurrent snapshot transactions and abort one of them
    - false positives are possible
    - PhD thesis of Michael Cahill at U Sydney [SIGMOD2008]
    - Fortunately, we now have a system that implements this → PostgreSQL-9.1

Dangerous Structures => SI Anomalies

[Fekete et al., TODS2005]

SI Serialisability Test:
Build the static dependency graph,
then check for any “**dangerous structures**“



cycle

pivot

incoming conflict

outgoing conflict

Limitations of This Approach

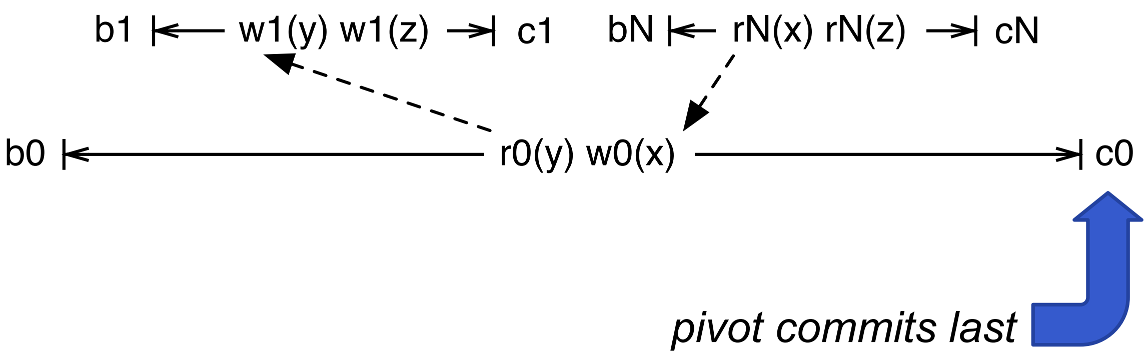
- Determining the conflict graph is non-trivial
- Repeat for every change to the application
- Ad hoc queries not supported
- Difficult to automate: reasoning required to avoid false positives
- What to do with the outcome?
  - Standard approach as of 2005 was that the applications needed to get changed, e.g. by introducing artificial writes to ‘promote’ a rw-dependency to a ww-dependency

“Online” SSI Approach

[Cahill, SIGMOD2008]

- New algorithm for serializable isolation
  - Online, dynamic
  - Modifications to standard Snapshot Isolation
- Core Idea:
  - Detect read-write conflicts at runtime
  - Abort transactions with consecutive rw-edges
  - Don’t do full cycle detection

SI Anomalies: a Simple Case

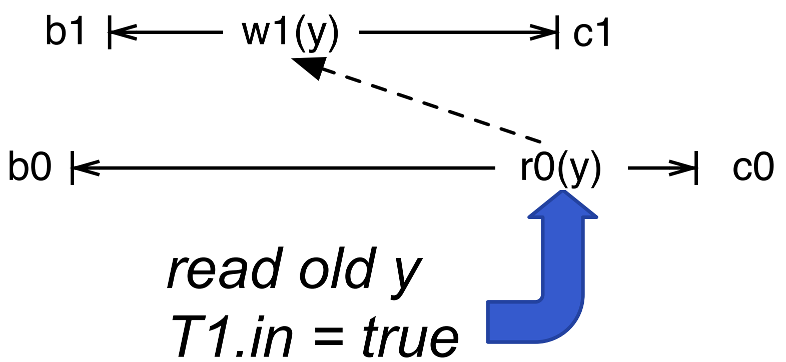


pivot commits last

The Algorithm in a Nutshell

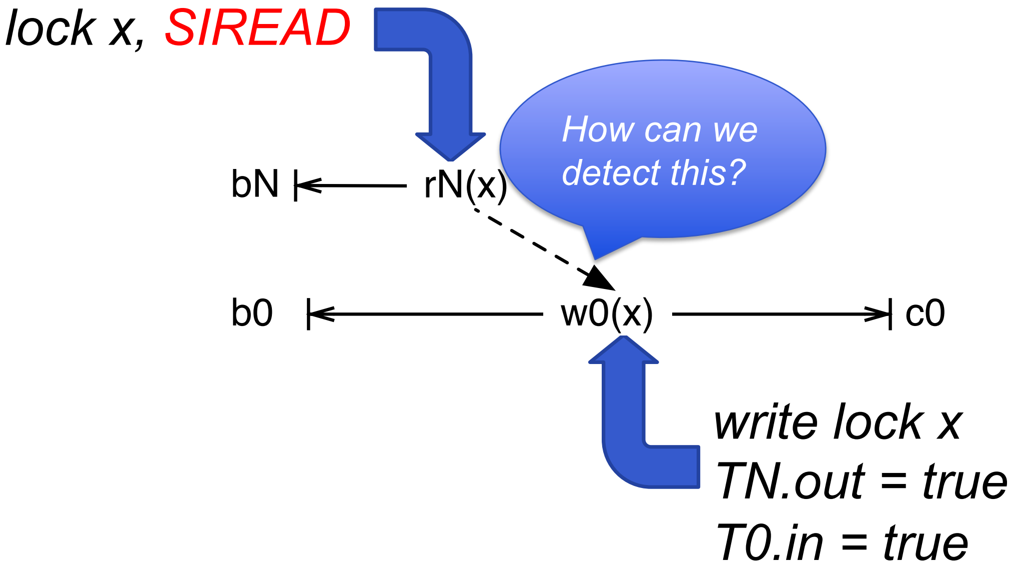
- Add two flags to each transaction (*in* & *out*)
- Set T0.out if *rw*-conflict T0 ➔ T1
- Set T0.in if *rw*-conflict TN ➔ T0
- Abort T0 (the pivot) if both T0.in and T0.out are set
  - If T0 has already committed, abort the conflicting transaction

Detection: Write before Read



read old y
T1.in = true
T0.out = true

Detection: Read before Write

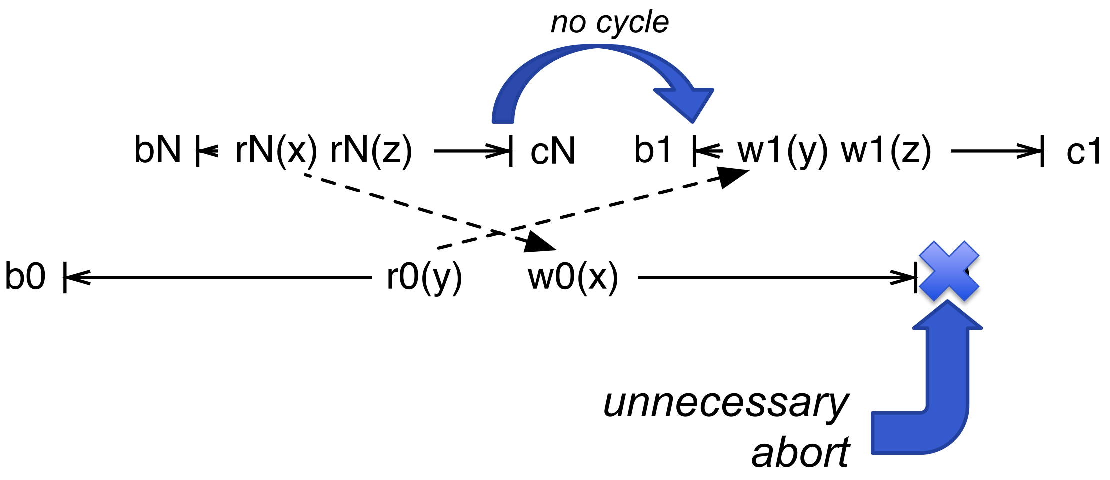


lock x, SIREAD

How can we detect this?

write lock x
TN.out = true
T0.in = true

Main Disadvantage: False Positives



no cycle

unnecessary
abort

SSI Variants

- SSI:
  Original SIGMOD08 paper (and more detailed in TODS2009)

- Precise Serialisable Snapshot Isolation (PSSI)
  - Revilak *et al* in ICDE 2011
  - In essence a full serialization graph test on top of Cahill's SSI

- Revilak's ICDE2011 paper also did their own implementation of TODS2009 algorithm in InnoDB -> referred to as ESSI

- PostgreSQL implementation of SSI by D. Ports (VLDB2012)
  - Including some optimisations for read-only transactions which do not need to take SIREAD locks on a *safe snapshot*

Design Decisions for SSI

- Local versus Global Dependency Tracking
  - anti-dependencies tracked per transaction or in a separate global data structure?

- Approximate versus accurate serialisability check
  - Check only for a *dangerous structure*, or perform a full cycle test?

- Ongoing checks versus commit-time check
  - check for potential abort with each update operation or at commit?

Previous Work on SSI

	SSI [Cahill,SIGMOD08]	**ESSI** [Revilak, ICDE11; Cahill, TODS09]	**PSSI** [Revilak, ICDE11]	**pgSSI** [Ports, VLDB12]
Tracking	local	local	global	local
Data Structure	two **Bits** per transact.	two **Pointers** per transact.	cycle testing **Graph** (CTG)	two **Lists** per transact.
Check	*dangerous structure*	*dangerous structure* (using CTG)	*cycle test*	*dangerous structure*
When	each update	at commit	at commit	each update

SSI in PostgreSQL 9.1

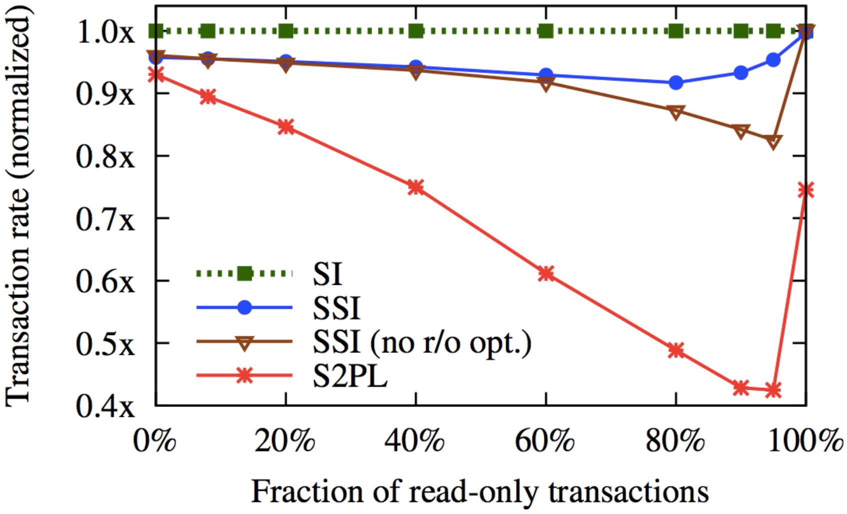
[Ports, VLDB2012]

- Basically follows Cahill’s approach
- Some additional optimisations:
  - Identify ‘safe snapshot’ situations for read-only transactions
  - If a read-only tx is found to run on a safe snapshot, will never abort and does not need to take and SIREAD locks (less overhead)
  - For long running transactions (e.g. backup):
    - Allow DBA to delay them until they are guaranteed to run on a safe snapshot

- Fully integrated into DBMS
  - Needed a bit more complex code than Michael’s prototype ;)
- ISOLATION LEVEL SERIALISABLE is now SSI
- ISOLATION LEVEL REPEADABLE READ is the former SI
- Default isolation level is still read committed…

Performance Penalty for Correctness?

- Obviously, tracking dependencies and aborting transactions (some of them false aborts) doesn't come for free
- What are the costs for being correct?



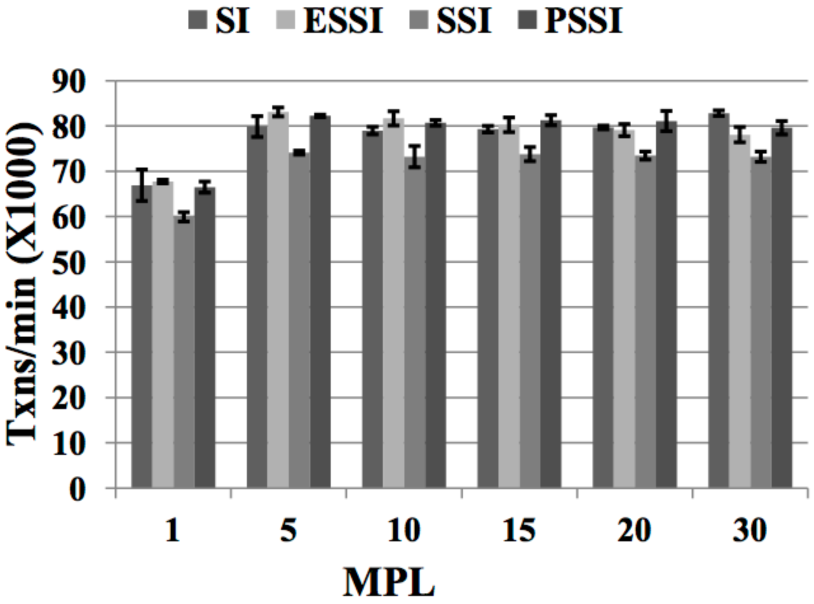
psql 9.1 with TPC-C, in-memory configuration (25 warehouses) [VLDB2012]

Research Question

Is SSI still as fast on a multicore server?

The Price of Serialisability

- MySQL 5.1.3 with InnoDB on 24 core Xeon Server
  - Implemented SSI, ESSI and PSSI in same engine (Linux)
  - quantified overhead at ca. 10%

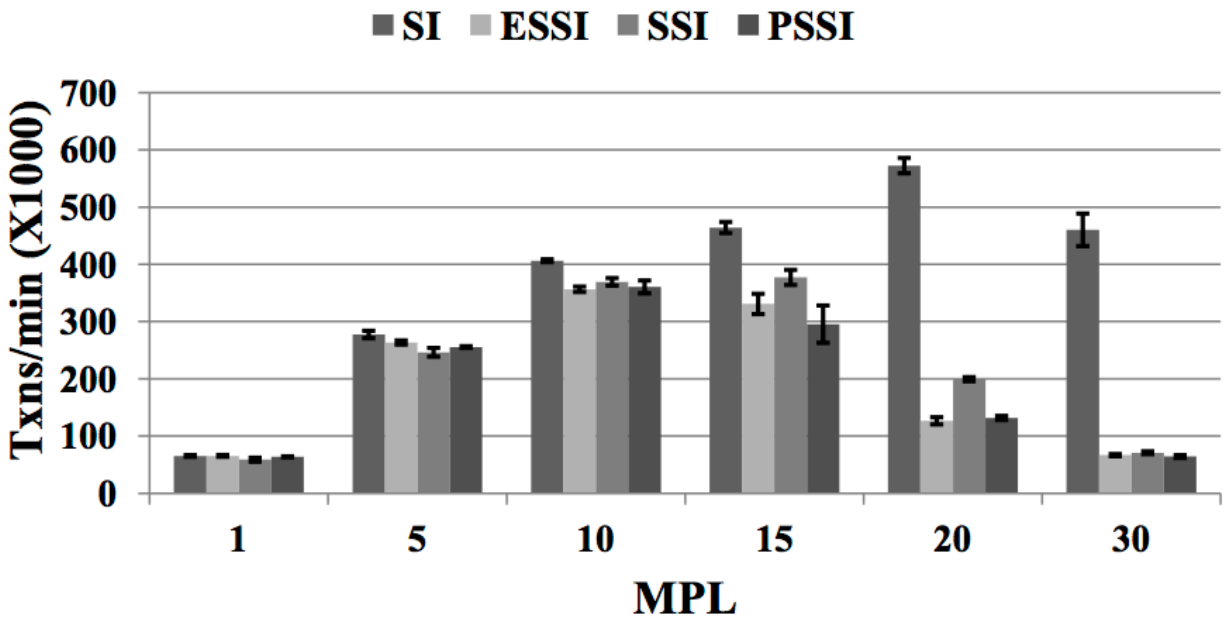


Scalability with Number of Cores?

- The previous figure was for just a single core server
  - with a (75%-RO-25%-RU) workload

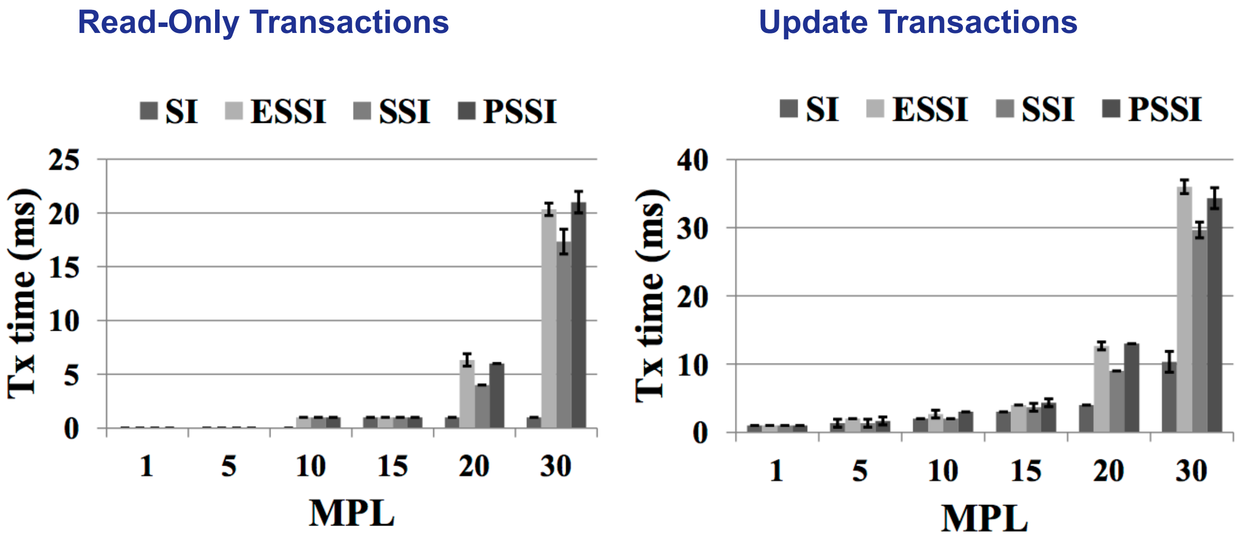
- What happens if we enable all 24 cores of the server?

The Big Picture



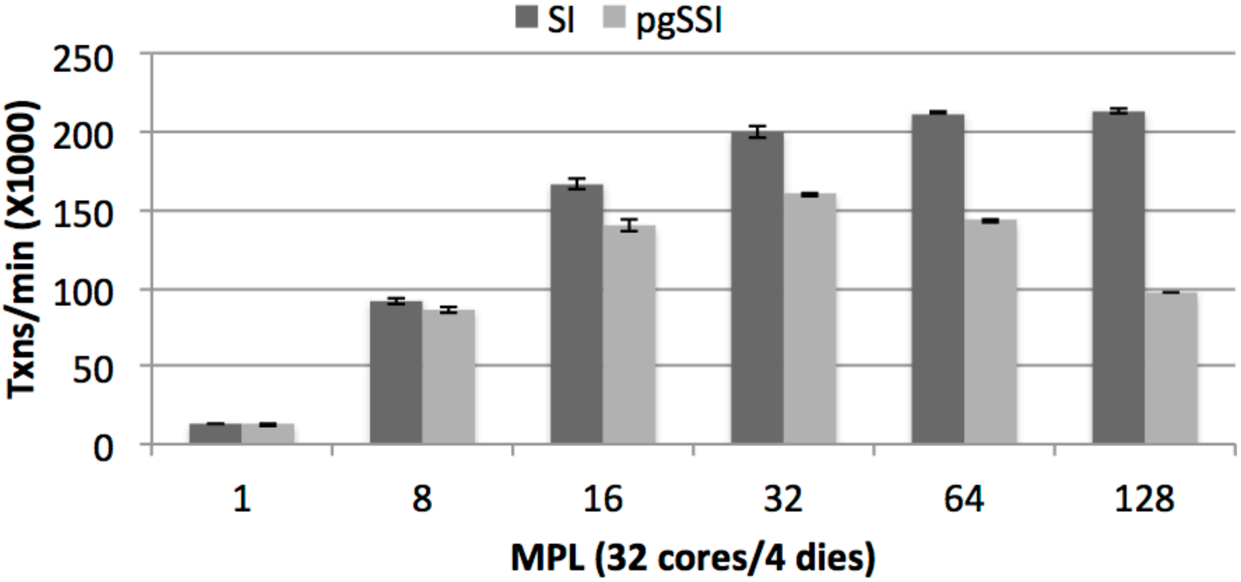
On 24 cores with a (75%-RO-25%-RU) workload

Transaction Runtimes



Transaction runtimes increase massively with MPL on 24 cores.

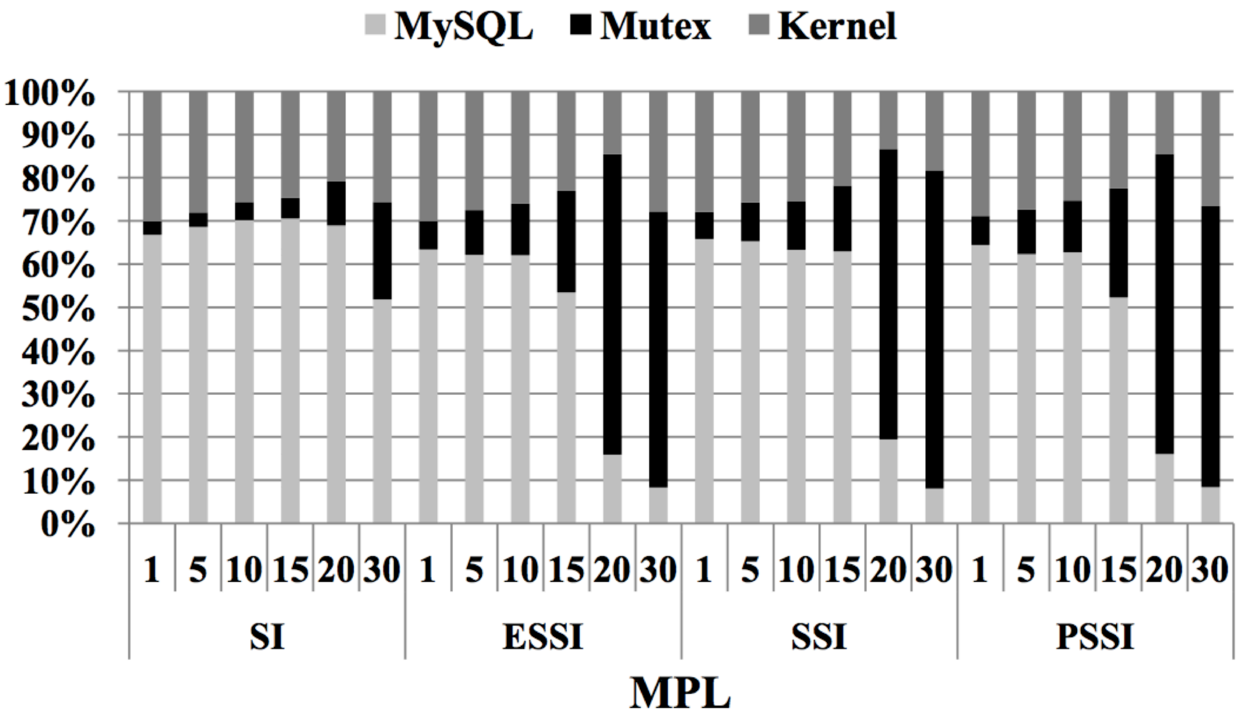
Same Picture with Latest Postgres 9.2



On 32 cores with a (75%-RO-25%-RU) workload

Why?

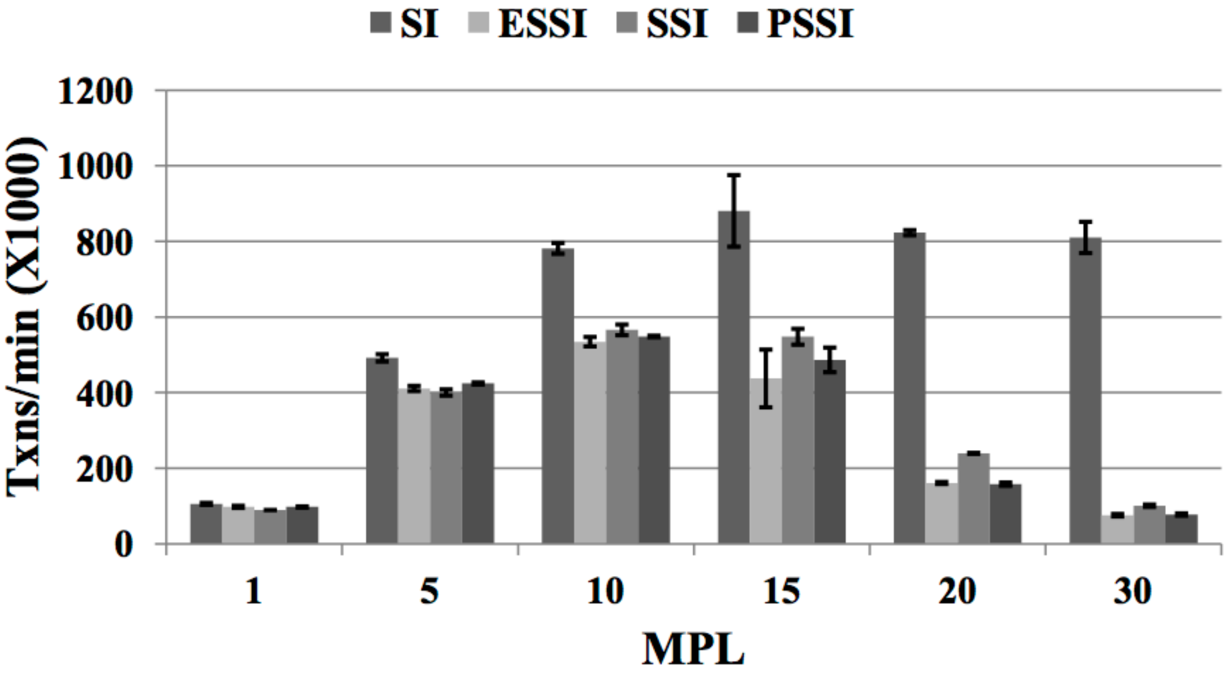
Profiled Execution Times



Reason: Mutex Contention

- More and more time is spend just waiting
- In order to avoid race conditions…
- …but which ’race’ ?
- And it can get even worse:

“Readers Never Block”?



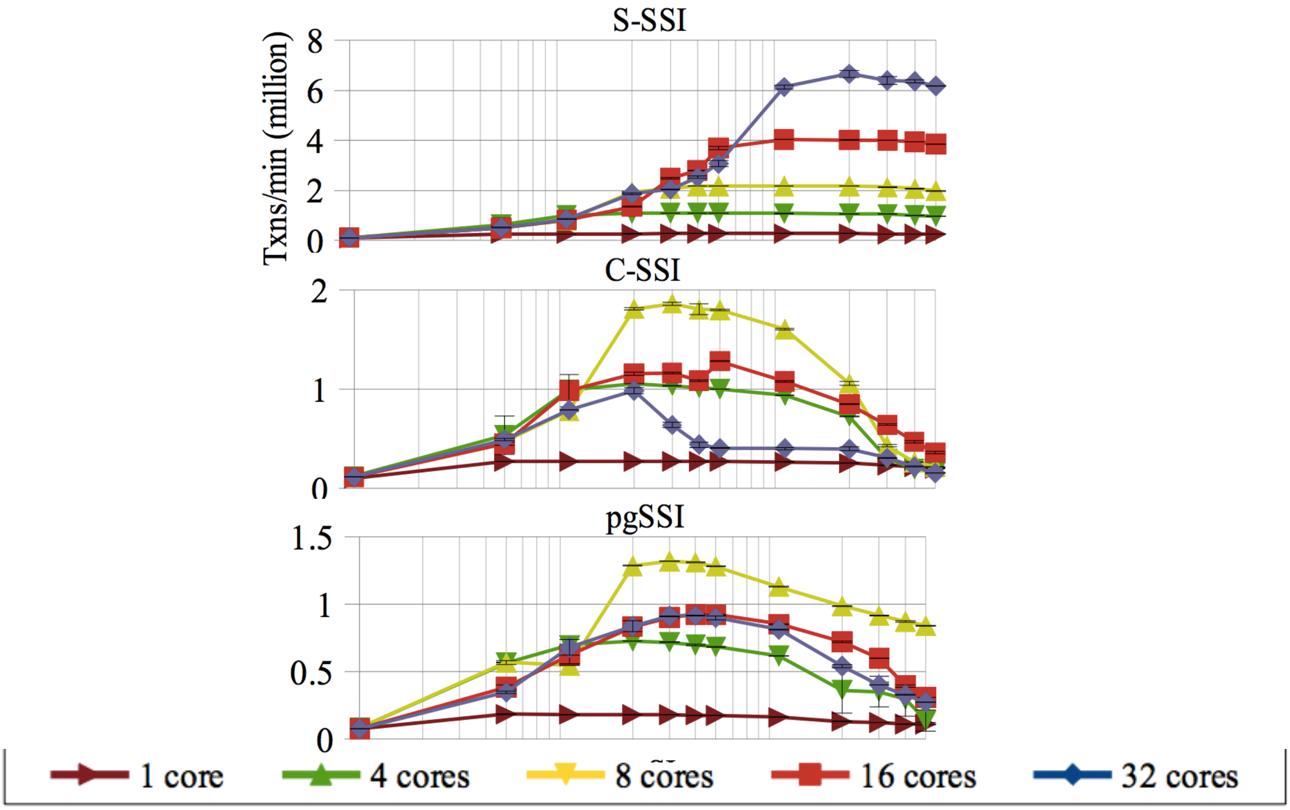
Read-Only Workload on 24 cores

Solution: Using Latch-free Data Structures for internal state of CC

- Latches are short-term locks (e.g. mutexes) that protect critical code sections from race conditions
  - E.g. only one thread is allowed to change the global transaction list
- Latch-Free Data Structures
  - Allow concurrent r/w access to in-memory data structures using atomic CPU operations such as Compare-And-Swap
  - Pro: Non-blocking, no latches needed anymore
  - Con: More complex; deletions require tombstones and some form of later garbage collection

- Proof of Concept: SSI with MySQL
  - Latch-free implementations of
    - read-write conflict checks, and
    - Consistent reads (read_view)

Performance Improvements



Summary

- Transaction Management is the backbone of DBMSs
- Pessimistic Concurrency Control
  - lock-based concurrency control schemes detect conflicts between concurrent transactions by incompatible locks on data items
  - Strict-2PL: Avoids cascading aborts, but deadlocks possible
  - Deadlocks can either be prevented or detected.
- Optimistic Concurrency Control
  - aims to minimize CC overheads in an ``optimistic'' environment where reads are common and writes are rare.
  - Multiversion Timestamp CC is a variant which ensures that read-only transactions are never restarted; they can always read a suitable older version. Additional overhead of version maintenance.
  - Snapshot Isolation as popular CC nowadays
    - But does not guarantee serializable executions!

References

- H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil in SIGMOD1995: “*A Critique of ANSI SQL Isolation Levels”*
- A. Bernstein, P. Lewis and S. Lu in ICDE2000: “*Semantic Conditions for Correctness at Different Isolation Levels”*
- A. Fekete, D. Liarokapis, E. O'Neil, P. O’Neil, D. Shasha in TODS2005: “*Making Snapshot Isolation Serializable”*
- M. Alomari, M. Cahill, A. Fekete, U. Röhm in ICDE2008: *“The Cost of Serializability on Platforms That Use Snapshot Isolation”*
- **M. Cahill, U. Röhm and A. Fekete in SIGMOD2008**: *“Serialisable Isolation for Snapshot Databases”*
- **D. Ports and K. Grittner in VLDB2012:** *“Serialisable Snapshot Isolation in PostgreSQL”*
- **H. Jung, H. Han, A. Fekete, U. Röhm and H. Y. Yeom, DASFAA 2013:** *“Performance of Serializable Snapshot Isolation on Multicore Servers”*
- **H. Han, S. Park, H. Jung, A. Fekete, U. Röhm and H. Y. Yeom, ICDE2014**: *“Scalable Serializable Snapshot Isolation for Multicore Systems”*