

EECS 262a
Advanced Topics in Computer Systems
Lecture 9

Transactions and Isolation Levels
February 22nd, 2016

John Kubiataowicz

Based on slides by Alan Fekete, Uwe Roehm and
Michael Cahill (University of Sydney),
updated by John Kubiataowicz and Anthony D. Joseph
Electrical Engineering and Computer Sciences
University of California, Berkeley
<http://www.eecs.berkeley.edu/~kubitron/cs262>

Today's Papers

- Granularity of Locks and Degrees of Consistency in a Shared Database (2-up version)
J.N. Gray, R.A. Lorie, G.R. Putzolu, I.L. Traiger. Appears In IFIP Working Conference on Modeling of Data Base Management Systems. 1975
- Serializable Isolation for Snapshot Databases
Michael J. Cahill, Uwe Röhm, Alan D. Fekete. Appears in ACM SIGMOD'08, June 9–12, 2008

- Thoughts?

2/22/2016

cs262a-S16 Lecture-09

2

Overview

- **Transactions**
 - ACID properties
 - Isolation Examples and counter-examples
- Classic Implementation Techniques
- Weak isolation issues

2/22/2016

cs262a-S16 Lecture-09

3

Definition

- A transaction is a collection of one or more operations on one or more databases, which reflects a single real-world transition
 - In the real world, this happened (completely) or it didn't happen at all (**Atomicity**)
 - Once it has happened, it isn't forgotten (**Durability**)
- Commerce examples
 - Transfer money between accounts
 - Purchase a group of products
- Student record system
 - Register for a class (either waitlist or allocated)

2/22/2016


cs262a-S16 Lecture-09

4

Consistency

- Each transaction can be written on the assumption that all integrity constraints hold in the data, before the transaction runs
- It must make sure that its changes leave the integrity constraints still holding
 - However, there are allowed to be intermediate states where the constraints do not hold
- A transaction that does this, is called **consistent**
- This is an obligation on the programmer
 - Usually the organization has a testing/checking and sign-off mechanism before an application program is allowed to get installed in the production system

Threats to data integrity

- Need for application rollback
- System crash  **Provided by the log**
- Concurrent activity
 - Today's lecture is about this

Concurrency

- When operations of concurrent threads are interleaved, the effect on shared state can be unexpected
- Well known issue in operating systems, thread programming
 - see OS textbooks on critical section
 - Java use of synchronized keyword

Famous anomalies

- Dirty data
 - One task T reads data written by T' while T' is running, then T' aborts (so its data was not appropriate)
- Lost update
 - Two tasks T and T' both modify the same data
 - T and T' both commit
 - Final state shows effects of only T, but not of T'
- Inconsistent read
 - One task T sees some but not all changes made by T'
 - The values observed may not satisfy integrity constraints
 - This was not considered by the programmer, so code moves into absurd path

Serializability

- To make isolation precise, we say that an execution is serializable when
- There exists some serial (ie batch, no overlap at all) execution of the same transactions which has the same final state
 - Hopefully, the real execution runs faster than the serial one!
- NB: different serial txn orders may behave differently; we ask that *some* serial order produces the given state
 - Other serial orders may give different final states

Serializability Theory

- There is a beautiful mathematical theory, based on formal languages
 - Model an execution as a sequence of operations on data items
 - » eg $r1[x] \ w1[x] \ r2[y] \ r2[x] \ c1 \ c2$
 - Serializability of an execution can be defined by equivalence to a rearranged sequence ("view serializability")
 - Treat the set of all serializable executions as an object of interest (called SR)
 - Thm: SR is in NP-Hard, i.e. the task of testing whether an execution is serializable seems unreasonably slow
- Does it matter?
 - The goal of practical importance is to design a system that produces some subset of the collection of serializable executions
 - It's not clear that we care about testing arbitrary executions that don't arise in our system

Conflict serializability

- There is a nice sufficient condition (ie a conservative approximation) called **conflict serializable**, which can be efficiently tested
 - Draw a **precedes graph** whose nodes are the transactions
 - Edge from T_i to T_j when T_i accesses x , then later T_j accesses x , and the accesses conflict (not both reads)
 - The execution is conflict serializable iff the graph is acyclic
- Thm: if an execution is conflict serializable then it is serializable
 - Pf: the serial order with same final state is any topological sort of the precedes graph
- Most people and books use the approximation, usually without mentioning it!

ACID

- **Atomic**
 - State shows either all the effects of txn, or none of them
- **Consistent**
 - Txn moves from a state where integrity holds, to another where integrity holds
- **Isolated**
 - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- **Durable**
 - Once a txn has committed, its effects remain in the database

Big Picture

- If programmer writes applications so each txn is consistent
- And DBMS provides atomic, isolated, durable execution
 - i.e. actual execution has same effect as some serial execution of those txns that committed (but not those that aborted)
- Then the final state will satisfy all the integrity constraints

NB true even though system does not know all integrity constraints!

Overview

- Transactions
- Classic Implementation Techniques
 - Locking
 - Lock Manager
 - Granularity of locks
- Weak isolation issues

Automatic lock management

- DBMS requests the appropriate lock whenever the app program submits a request to read or write a data item
- If lock is available, the access is performed
- If lock is not available, the whole txn is **blocked** until the lock is obtained
 - After a conflicting lock has been released by the other txn that held it

Lock modes

- Locks can be for writing (X), reading (S)
 - Refinements have extra modes
- Standard conflict rules: two X locks on the same data item conflict, so do one X and one S lock on the same data
 - However, two S locks do not conflict
- X=exclusive
 - S=shared

Held/Requested	X	S
X	Block	Block
S	Block	OK

Strict two-phase locking

- Locks that a txn obtains are kept until the txn completes
 - Once the txn commits or aborts, then all its locks are released (as part of the commit or rollback processing) ← NB. This is different from when locks are released in O/S or threaded code
- Two phases:
 - Locks are being obtained (while txn runs)
 - Locks are released (when txn finished)

Serializability

- If each transaction does strict two-phase locking (requesting all appropriate locks), then executions are serializable
- However, performance does suffer, as txns can be blocked for considerable periods
 - Deadlocks can arise, requiring system-initiated aborts

Proof sketch

- Suppose all txns do strict 2PL
- If T_i has an edge to T_j in the precedes graph
 - That is, T_i accesses x before T_j has conflicting access to x
 - T_i has lock at time of its access, T_j has lock at time of its access
 - Since locks conflict, T_i must release its lock before T_j 's access to x
 - T_i completes before T_j accesses x
 - T_i completes before T_j completes
- So the precedes graph is subset of the (acyclic) total order of txn commit
- Conclusion: *the execution has same final state as the serial execution where txns are arranged in commit order*

Lock manager

- A structure in (volatile memory) in the DBMS which remembers which txns have set locks on which data, in which modes
- It does not allow a request to get a new lock if a conflicting lock is already held by a different txn
- NB: a lock does not actually prevent access to the data, it only prevents getting a conflicting lock
 - So data protection only comes if the right lock is requested before every access to the data

Lock manager API

- Access mainly based on item's unique name
 - eg tupleid, or primary key, for records
- Lock(name, txn, mode)
 - Block until lock is available
- RemoveTxn(txn)
 - Unlock(name, txn)
 - LockUpgrade(name, txn, newmode)
 - ConditionalLock(name, txn, mode)
 - » Returns error immediately if unsuccessful

Granularity

- What is a data item (on which a lock is obtained)?
 - Most times, in most modern systems: item is one tuple in a table
 - Sometimes (especially in early 1970s): item is a page (with several tuples)
 - Sometimes: item is a whole table

Granularity trade-offs

- Larger granularity: fewer locks held, so less overhead; but less concurrency possible
 - “false conflicts” when txns deal with different parts of the same item
- Smaller “fine” granularity: more locks held, so more overhead; but more concurrency is possible
- System usually gets fine grain locks until there are too many of them; then it replaces them with larger granularity locks

Multigranular locking

- Care needed to manage conflicts properly among items of varying granularity
 - Note: conflicts only detectable among locks on a given itemname
- System gets “intention” mode locks on larger granules before getting actual S/X locks on smaller granules
 - Conflict rules arranged so that activities that do not commute must get conflicting locks on some item

Lock Mode Conflicts

Held\Request	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	Yes	Block
IX	Yes	Yes	Block	Block	Block
S	Yes	Block	Yes	Block	Block
SIX	Yes	Block	Block	Block	Block
X	Block	Block	Block	Block	Block

**Other modes also used for special purposes,
like select-for-later-update,
gaplocks for phantom prevention**

**Lock manager may allow higher level
to introduce its own conflict table**

Lock manager internals

- Hash table, keyed by hash of item name
 - Each item has a mode and holder (set)
 - Wait queue of requests
 - All requests and locks in linked list from transaction information
 - Transaction table
 - » To allow thread rescheduling when blocking is finished
 - Deadlock detection
 - » Either cycle in waits-for graph, or just timeouts

Explicit lock management

- With most DBMS, the application program can include statements to set or release locks on a table
 - Details vary
- e.g. LOCK TABLE InStore IN EXCLUSIVE MODE

Overview

- Transactions
- Classic Implementation Techniques
- Weak isolation issues
 - Especially Snapshot Isolation

Problems with serializability

- The performance reduction from isolation is high
 - Transactions are often blocked because they want to read data that another txn has changed
- For many applications, the accuracy of the data they read is not crucial
 - e.g. overbooking a plane is ok in practice
 - e.g. your banking decisions would not be very different if you saw yesterday's balance instead of the most up-to-date

A and D matter!

- Even when isolation isn't needed, no one is willing to give up atomicity and durability
 - These deal with modifications a txn makes
 - Writing is less frequent than reading, so log entries and write locks are considered worth the effort

Explicit isolation levels

- A transaction can be declared to have isolation properties that are less stringent than serializability
 - However SQL standard says that default should be serializable (Gray'75 called this "level 3 isolation")
 - In practice, most systems have weaker default level, and most txns run at weaker levels!

Browse

- SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
 - Do not set S-locks at all
 - » Of course, still set X-locks before updating data
 - » If fact, system forces the txn to be read-only unless you say otherwise
 - Allows txn to read dirty data (from a txn that will later abort)

Read Committed

Gray'75 called this “degree 2”

- SET TRANSACTION ISOLATION LEVEL READ COMMITTED **Most common in practice!**
 - Set S-locks but release them after the read has happened
 - » e.g. when cursor moves onto another element during scan of the results of a multirow query
 - i.e. do not hold S-locks till txn commits/aborts
 - Data is not dirty, but it can be inconsistent (between reads of different items, or even between one read and a later one of the same item)
 - » Especially, weird things happen between different rows returned by a cursor

Repeatable read

- SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
 - Set S-locks on data items, and hold them till txn finished, but release locks on indices as soon as index has been examined
 - Allows “phantoms”, rows that are not seen in a query that ought to have been (or vice versa)
 - Problems if one txn is changing the set of rows that meet a condition, while another txn is retrieving that set

Snapshot Isolation (SI)

- A multiversion concurrency control mechanism was described in SIGMOD '95 by H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil
 - Does not guarantee serializable execution!
- Supplied by Oracle DB for “Isolation Level Serializable” (also in PostgreSQL before rel 9.1)
- Available in Microsoft SQL Server 2005 as “Isolation Level Snapshot”, and in PostgreSQL (since rel 9.1) as “Isolation Level Repeatable Read”

Snapshot Isolation (SI)

- Read of an item may not give current value
- Instead, use old versions (kept with timestamps) to find value that had been most recently committed at the time the txn 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 the database was consistent before

First committer wins (FCW)

- T will not be allowed to commit a modification to an item if any other transaction has committed a changed value for that item since T's start (snapshot)
- T must hold write locks on modified items at time of commit, to install them.
 - In practice, commit-duration write locks may be set when writes execute.
 - These simplify detection of conflicting modifications when T tries to write the item, instead of waiting till T tries to commit.

Benefits of SI

- Reading is never blocked, and reads don't block writes
- Avoids common anomalies
 - No dirty read
 - No lost update
 - No inconsistent read
 - Set-based selects are repeatable (no phantoms)
- Matches common understanding of isolation: concurrent transactions are not aware of one another's changes

Is every execution serializable?

- For any set of txns, if they all run with Two Phase Locking, then every interleaved execution is serializable
- For some sets of txns, if they all run with SI, then every execution is serializable
 - Eg the txns making up TPC-C
- For some sets of txns, if they all run with SI, there can be non-serializable executions
 - Undeclared integrity constraints can be violated

Example

- Table Duties(Staff, Date, Status)
- Undeclared constraint: for every Date, there is at least 1 Staff with Status='Y'
- Transaction TakeBreak(S, D) **running at SI**

```
SELECT COUNT(*) INTO :tmp FROM Duties
WHERE Date=:D AND Status='Y';
IF tmp < 2 ROLLBACK;
UPDATE Duties
  SET Status = 'N'
  WHERE Staff =:S AND Date =:D;
COMMIT;
```

Example (continued)

- Possible execution, starting when two staff (S101, S103) are on duty for 2004-06-01
- *Concurrently* perform
TA: TakeBreak(S101, 2004-06-01)
TB: TakeBreak(S103, 2004-06-01)
 - Each succeeds, as each sees snapshot with 2 on duty
 - No problem committing, as they update different rows!
- End with no staff on duty for that date!
- RA(r1) RA(r3) RB(r1) RB(r3)
WA(r1) CA WB(r3) CB
 - Non-serializable execution

S101	2004-06-01	'Y'
S102	2004-06-01	'N'
S103	2004-06-01	'Y'
etc	etc	etc

Write Skew

- SI breaks serializability when txns modify different items in each other's read sets
 - Neither txn sees the other, but in a serial execution one would come later and so see the other's impact
 - This is fairly rare in practice
- Eg the TPC-C benchmark always runs correctly under SI
 - whenever its txns conflict (eg read/write same data), there is also a ww-conflict: a shared item they both modify (like a total quantity) so SI will abort one of them

Interaction effects

- You can't think about one program, and say "this program can use SI"
- The problems have to do with the set of application programs, not with each one by itself
- Example where T1, T2, T3 can all be run under SI, but when T4 is present, we need to fix things in T1
- Non-serializable execution can involve read-only transactions, not just updaters

Multiversion Serializability Theory

From Y. Raz in RIDE'93

- WW-conflict from T1 to T2
 - T1 writes a version of x, T2 writes a later version of x
 - » In our case, succession (version order) defined by commit times of writer txns
- WR-conflict from T1 to T2
 - T1 writes a version of x, T2 reads this version of x (or a later version of x)
- RW-conflict from T1 to T2
 - » Adya et al ICDE'00 called this "antidependency"
 - T1 reads a version of x, T2 writes a later version of x
- Serializability tested by acyclic conflict graph

Interference Theory

From Fekete et al, TODS 2005

- We produce the “static dependency graph”
 - Node for each application program
 - Draw directed edges each of which can be either
 - » Non-vulnerable interference edge, or
 - » Vulnerable interference edge
- Based on looking at program code, to see what sorts of conflict situations can arise
- More complicated with programs whose accesses are controlled by parameters
- A close superset of SDG can be calculated automatically in some cases

Edges in the SDG

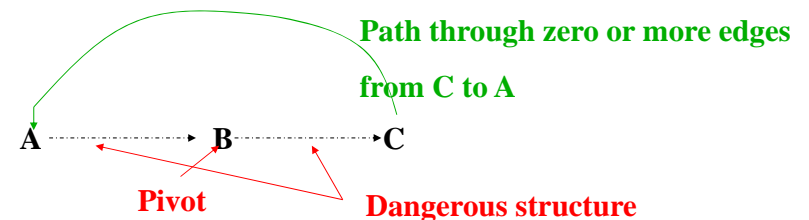
- Non-vulnerable interference edge from T1 to T2
- Conflict, but it can't arise transactions can run concurrently
 - Eg “ww” conflict
 - » Concurrent execution prevented by FCW
 - Or “wr” conflict
 - » conflict won't happen in concurrent execution due to reading old version
- Eg
 - T1 = R1(x) R1(y) W1(x)
 - T2 = R2(x) R2(y) W2(x) W2(y)
- Vulnerable interference edge from T1 to T2
- Conflict can occur when transactions run concurrently
 - Eg “rw without ww”: rset(T1) intersects wset(T2), and wset(T1) disjoint from wset(T2)
- Eg
 - T1 = R1(x) R1(y) W1(x)
 - T2 = R2(x) R2(y) W2(x)
- Shown as dashed edge on diagram

Paired edges

- In SDG, an edge from X to Y implies an edge from Y to X
- But the type of edge is not necessarily the same
 - Both vulnerable, or
 - Both non-vulnerable, or
 - One vulnerable and one non-vulnerable

Dangerous Structures

- A **dangerous structure** is two edges linking three application programs, A, B, C such that
 - There are successive vulnerable edges (A,B) and (B,C)
 - (A, B, C) can be completed to a cycle in SDG
 - » Call B a **pivot**
 - Special case: pair A, B with vulnerable edges in both directions



The main result

- Theorem: If the SDG does not contain a dangerous cycle, then every execution is serializable (with all transactions using SI for concurrency control)
 - Applies to TPC-C benchmark suite

Serializable Isolation for Snapshot Databases [Sigmod'08 “Best paper”, then ACM TODS 2009]

Michael Cahill, Alan Fekete, Uwe Röhm

University of Sydney

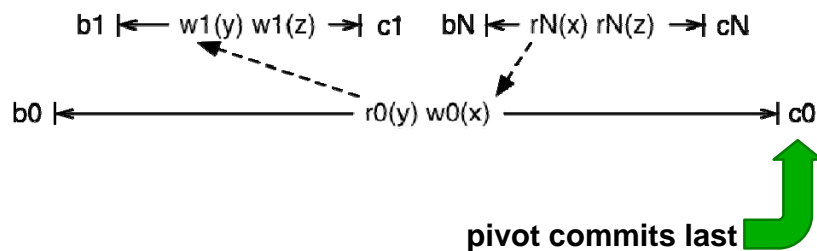
Serializable SI

- If we can alter the DBMS, we could provide a new algorithm for serializable isolation
 - Online, dynamic
 - Modifications to standard Snapshot Isolation
 - » Keep versions, read from snapshot, FCW (like SI)
 - Detect read-write conflicts at runtime
 - Abort transactions with consecutive rw-edges
 - » Much less often than traditional optimistic CC
 - » Don't do full cycle detection

Challenges

- During runtime, rw-pairs can interleave arbitrarily
- Have to consider begin and commit timestamps:
 - which snapshot is a transaction reading?
 - can conflict with committed transactions
- Want to use existing engines as much as possible
- Low runtime overhead
- But minimize unnecessary aborts

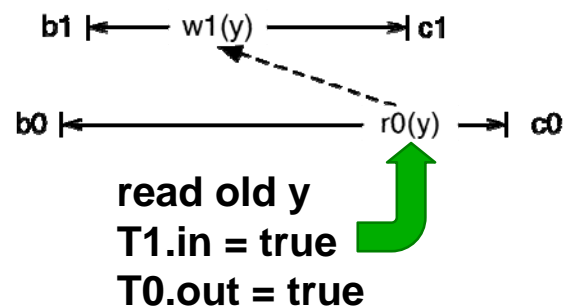
SI anomalies: a simple case



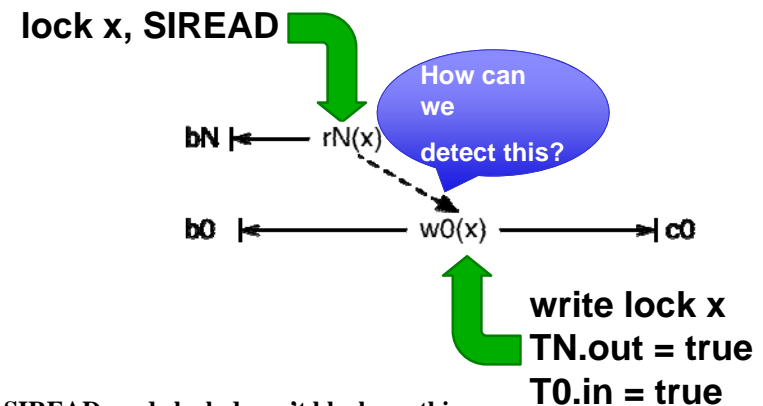
Algorithm in a nutshell

- 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



Detection: read before write

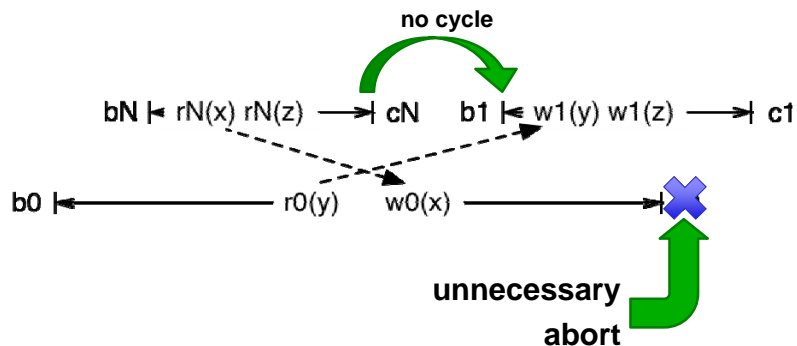


SIREAD mode lock doesn't block anything

Just for record keeping

Kept even after transaction commits

Main Disadvantage: False positives



Prototype in Oracle InnoDB

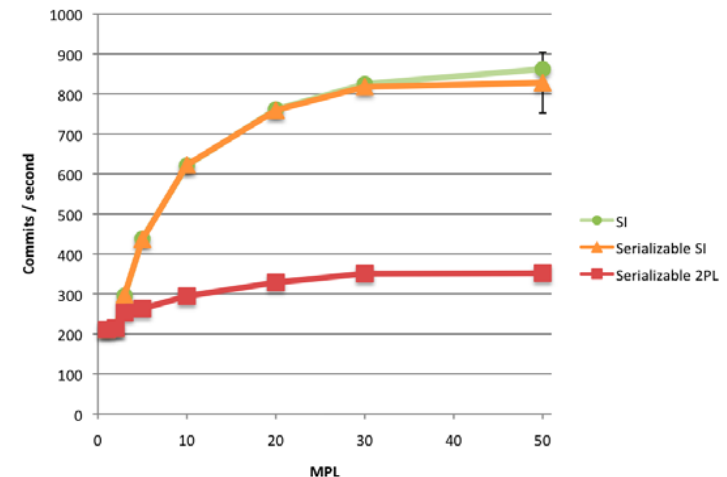
Not in SIGMOD'08; added work for TODS'09

- Implemented in Oracle InnoDB plugin 1.0.1
 - » Most popular transactional backend for MySQL
 - » Already includes multiversion concurrency control
- Serializable SI, including phantom detection (uses InnoDB's next-key locking)
 - » Also (for comparison) True Snapshot Isolation with first-committer-wins (InnoDB's "repeatable read" isolation has non-standard semantics)
- Added 230 lines of code to 130K lines in InnoDB
 - » Most changes related to transaction lifecycle management

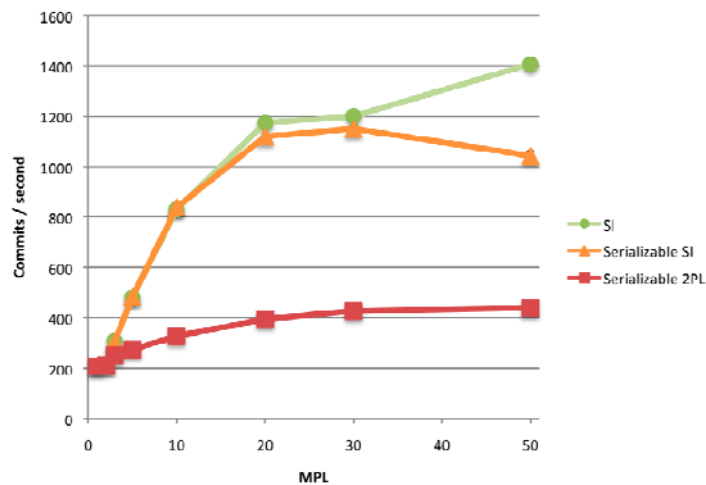
Experimental scenarios

- sibench – synthetic microbenchmark
 - conflict between sequential scan and updating a row
 - table size determines write-write conflict probability and CPU time required for scan
- TPC-C++ - modified TPC-C to introduce an SI anomaly
 - added a "credit check" transaction type to the mix
 - measured throughput under a variety of conditions
 - » most not sensitive to choice of isolation level, but we found a mix favoring "stock level" transactions that demonstrates the tradeoff

sibench: 10 reads per write



sibench: 100 reads per write

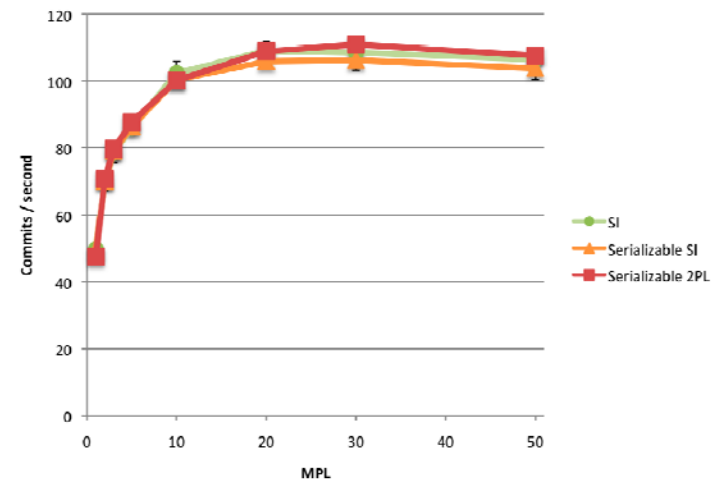


2/22/2016

cs262a-S16 Lecture-09

61

TPC-C++: 10 warehouses

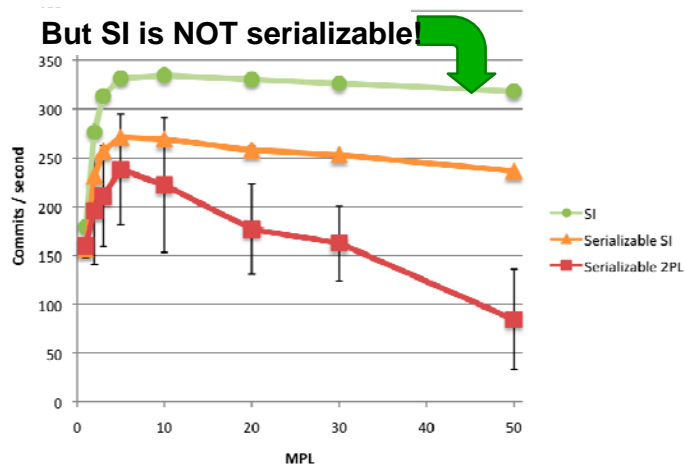


2/22/2016

cs262a-S16 Lecture-09

62

TPC-C++: special “stock level” mix



2/22/2016

cs262a-S16 Lecture-09

63

Serializable SI: Lessons

- New algorithm for serializable isolation
 - Online, dynamic, and general solution
 - Modification to standard Snapshot Isolation
 - Keeps the features that make SI attractive: Readers don't block writers, much better scalability than S2PL
- In most cases, performance is comparable with SI
- Never worse than locking serializable isolation
- Feasible to add to an RDBMS using Snapshot Isolation (such as Oracle) with modest changes
 - PostgreSQL release 9.1 did this – Isolation Level Serializable now executes serializably! See “Serializable Snapshot Isolation in PostgreSQL” by D. Ports and K. Gritter, PVLDB 5(12):1850-1861 (2012)

2/22/2016

cs262a-S16 Lecture-09

64

Were these good papers?

- What were the authors' goals?
- What about the evaluation / metrics?
- Did they convince you that this was a good system /approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?

Further Reading

- Big picture: "Principles of Transaction Processing" by P. Bernstein and E. Newcomer
- Theory: "Transactional Information Systems" by G. Weikum and G. Vossen
- The gory details: "Transaction Processing" by J. Gray and A. Reuter
 - Also, "Architecture of a Database System" by J. Hellerstein, M. Stonebraker, and J. Hamilton,