Serializable Snapshot Isolation

Making ISOLATION LEVEL SERIALIZABLE Provide Serializable Isolation

Dan Ports
MIT

Kevin Grittner
Wisconsin Court System

Overview

Serializable isolation makes it easier to reason about concurrent transactions

In 9.0 and before, SERIALIZABLE was really *snapshot isolation* – allows anomalies

in 9.1: *Serializable Snapshot Isolation (SSI)*

- a new way to ensure true serializability (first implementation in a production DBMS!)

Agenda

- **What is serializability? Why do we want it?**
- Snapshot isolation vs. serializability
- Serializable Snapshot Isolation
- SSI implementation overview
- Using SSI
- Performance results

Transactions

Transactions group related operations:

shouldn't see one operation without the others

- ...even if the system crashes (recoverability)
- ...even if other transactions are executing concurrently (**isolation**)

Isolation

Serializable isolation:
each transaction is guaranteed to behave as though it's the only one running

- makes it easy to reason about each transaction's behavior in isolation

Weaker isolation levels:

- concurrent transactions can cause anomalous behavior

Isolation Levels

SQL Standard

SERIALIZABLE

REPEATABLE
READ

READ
COMMITTED

READ
UNCOMMITTED

Isolation Levels

9.0	**SQL Standard**
snapshot isolation	SERIALIZABLE
	REPEATABLE READ
per-statement snapshots	READ COMMITTED
	READ UNCOMMITTED

Isolation Levels

9.0

SQL Standard

9.1

SERIALIZABLE → ***SSI***

snapshot isolation

REPEATABLE READ → snapshot isolation

per-statement snapshots ← READ COMMITTED → per-statement snapshots

READ UNCOMMITTED

Snapshot Isolation

Each transaction sees a “snapshot” of DB taken at its first query

- implemented using MVCC
- tuple-level write locks prevent concurrent modifications

Still a weaker isolation level than true serializability!

Agenda

- What is serializability? Why do we want it?
- **Snapshot isolation vs. serializability**
- Serializable Snapshot Isolation
- SSI implementation overview
- Using SSI
- Performance results

Goal:

ensure at least one guard always on-duty

guard	on-duty?
Alice	y
Bob	y

Goal:

ensure at least one guard always on-duty

```
BEGIN

SELECT count(*)
FROM guards
WHERE on-duty = y

if > 1 {
  UPDATE guards
  SET on-duty = n
  WHERE guard = x
}

COMMIT
```

guard	on-duty?
Alice	y
Bob	y

guard	on-duty?
Alice	y
Bob	y

BEGIN

SELECT count(*)
FROM guards
WHERE on-duty = y
[result = 2]

guard	on-duty?
Alice	y
Bob	y

BEGIN

SELECT count(*)
FROM guards
WHERE on-duty = y
[result = 2]

BEGIN

SELECT count(*)
FROM guard
WHERE on-duty = y
[result = 2]

guard	on-duty?
Alice	y
Bob	y

```
BEGIN

SELECT count(*)
FROM guards
WHERE on-duty = y
      [result = 2]

if > 1 {
  UPDATE guards
  SET on-duty = n
  WHERE guard = 'Alice'
}
COMMIT
```

```
BEGIN

SELECT count(*)
FROM guard
WHERE on-duty = y
      [result = 2]
```

guard	on-duty?
Alice	y
Bob	y

```
BEGIN

SELECT count(*)
FROM guards
WHERE on-duty = y
        [result = 2]

if > 1 {
   UPDATE guards
   SET on-duty = n
   WHERE guard = 'Alice'
}
COMMIT
```

```
BEGIN

SELECT count(*)
FROM guard
WHERE on-duty = y
        [result = 2]
```

guard	on-duty?
Alice	~~y~~ n
Bob	y

```
BEGIN

SELECT count(*)
FROM guards
WHERE on-duty = y
      [result = 2]

if > 1 {
  UPDATE guards
  SET on-duty = n
  WHERE guard = ‘Alice’
}
COMMIT
```

guard	on-duty?
Alice	~~y~~ n
Bob	y

```
BEGIN

SELECT count(*)
FROM guard
WHERE on-duty = y
      [result = 2]

if > 1 {
  UPDATE guards
  SET on-duty = n
  WHERE guards = ‘Bob’
}
COMMIT
```

```
BEGIN

SELECT count(*)
FROM guards
WHERE on-duty = y
      [result = 2]

if > 1 {
  UPDATE guards
  SET on-duty = n
  WHERE guard = 'Alice'
}
COMMIT
```

guard	on-duty?
Alice	~~y~~ n
Bob	~~y~~ n

```
BEGIN

SELECT count(*)
FROM guard
WHERE on-duty = y
      [result = 2]

if > 1 {
  UPDATE guards
  SET on-duty = n
  WHERE guards = 'Bob'
}
COMMIT
```

Serializable means: results equivalent to *some* serial ordering of the transactions

Serialization history graph shows dependencies between transactions

- A → B ("wr-dependency")
  if B sees a change made by A
- A → B ("ww-dependency")
  if B overwrites a change by A
- B → A ("rw-conflict")
  if B *doesn't* see a change made by A

Serializable if *no cycle in graph*

```
BEGIN

SELECT count(*)
FROM guards
WHERE on-duty = y
      [result = 2]

if > 1 {
  UPDATE guards
  SET on-duty = n
  WHERE guard = ‘Alice’
}
COMMIT
```

```
BEGIN

SELECT count(*)
FROM guard
WHERE on-duty = y
      [result = 2]

if > 1 {
  UPDATE guards
  SET on-duty = n
  WHERE guards = ‘Bob’
}
COMMIT
```

guard	on-duty?	
Alice	~~y~~ n	
Bob	~~y~~ n	

```
BEGIN

SELECT count(*)
FROM guards
WHERE on-duty = y
      [result = 2]

if > 1 {
  UPDATE guards
  SET on-duty = n
  WHERE guard = ‘Alice’
}
COMMIT
```

rw-conflict:
T1 didn’t see
T2’s UPDATE

```
BEGIN

SELECT count(*)
FROM guard
WHERE on-duty = y
      [result = 2]

if > 1 {
  UPDATE guards
  SET on-duty = n
  WHERE guards = ‘Bob’
}
COMMIT
```

guard	on-duty?
Alice	~~y~~ n
Bob	~~y~~ n

BEGIN

SELECT count(*)
FROM guards
WHERE on-duty = y
[result = 2]

if > 1 {
UPDATE guards
SET on-duty = n
WHERE guard = 'Alice'
}
COMMIT

rw-conflict:
T1 didn't see
T2's UPDATE

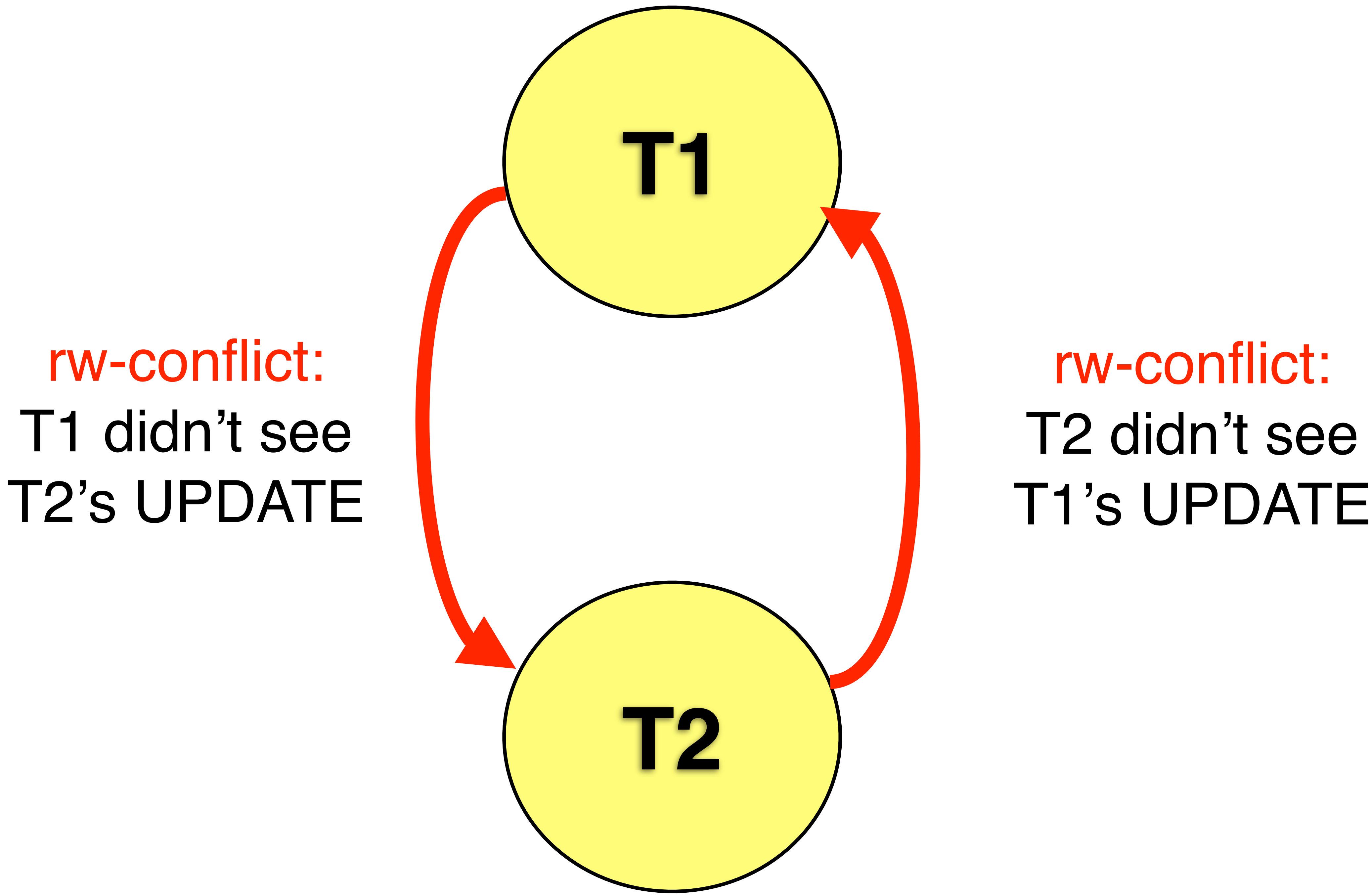
BEGIN

SELECT count(*)
FROM guard
WHERE on-duty = y
[result = 2]

rw-conflict:
T2 didn't see
T1's UPDATE

if > 1 {
UPDATE guards
SET on-duty = n
WHERE guards = 'Bob'
}
COMMIT

guard	on-duty?
Alice	~~y~~ n
Bob	~~y~~ n



cycle means no serial order exists!
T1 before T2 before T1...

Batch Processing Example

- control table just holds current batch #
- receipts table entries tagged w/ batch #

Three transactions:

- read current batch, insert receipt tagged w/ it
- increment current batch #
- read batch, get all receipts for previous batch

Invariant: after we read yesterday’s report, no new receipts for yesterday should appear

T1 **T2**

SELECT batch
FROM control
[result = 5/19]

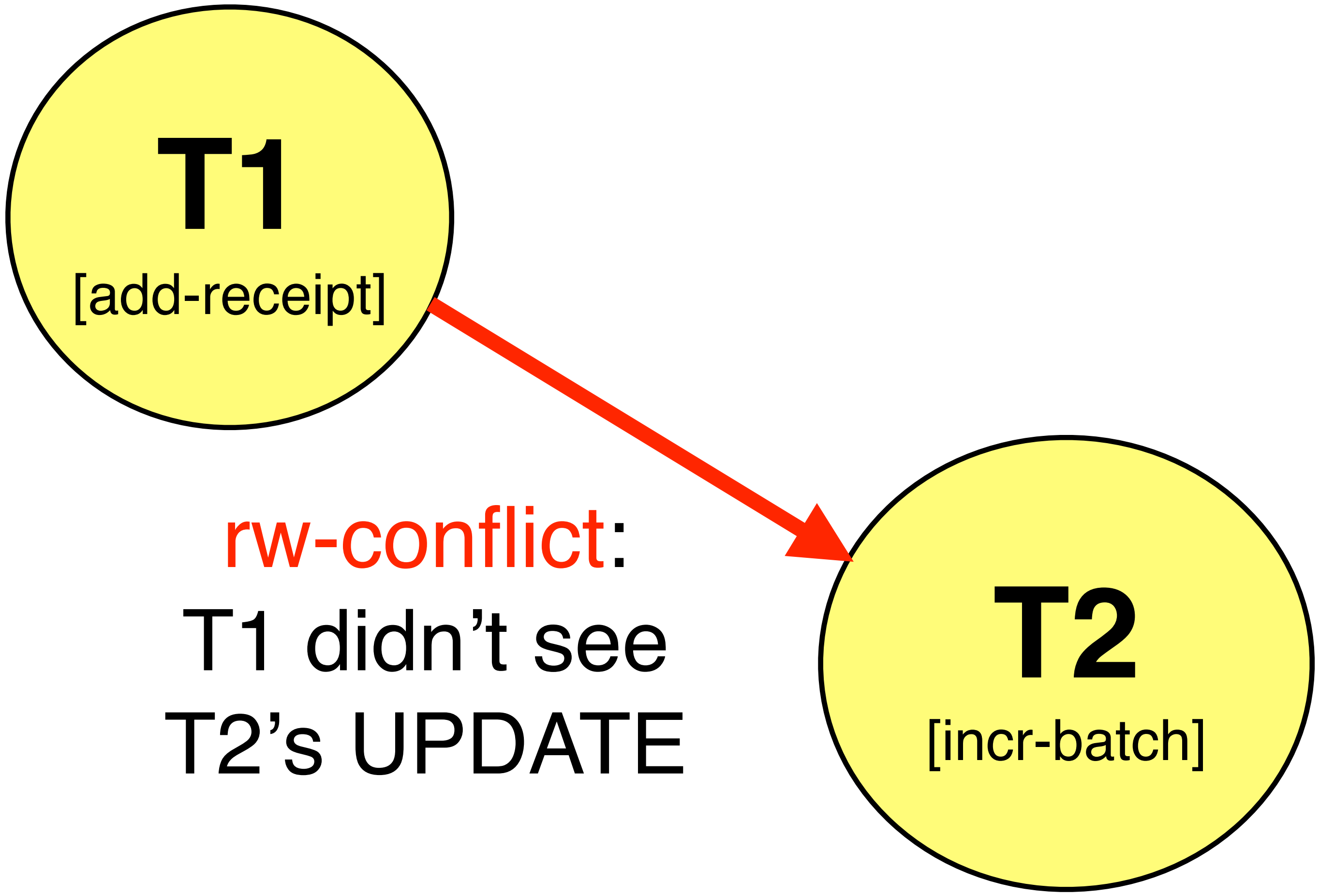
rw-conflict: T1 didn't see T2's UPDATE

UPDATE control
SET batch = 5/20

COMMIT

INSERT receipt
(5/19, …)

COMMIT



Serializable!

Apparent order of execution: T1 before T2

...but T2 committed before T1. That's OK!

T1	**T2**	**T3**
SELECT batch FROM control *[result = 5/19]*		
	UPDATE control SET batch = 5/20	
	COMMIT	
INSERT receipt (5/19, …)		
COMMIT		

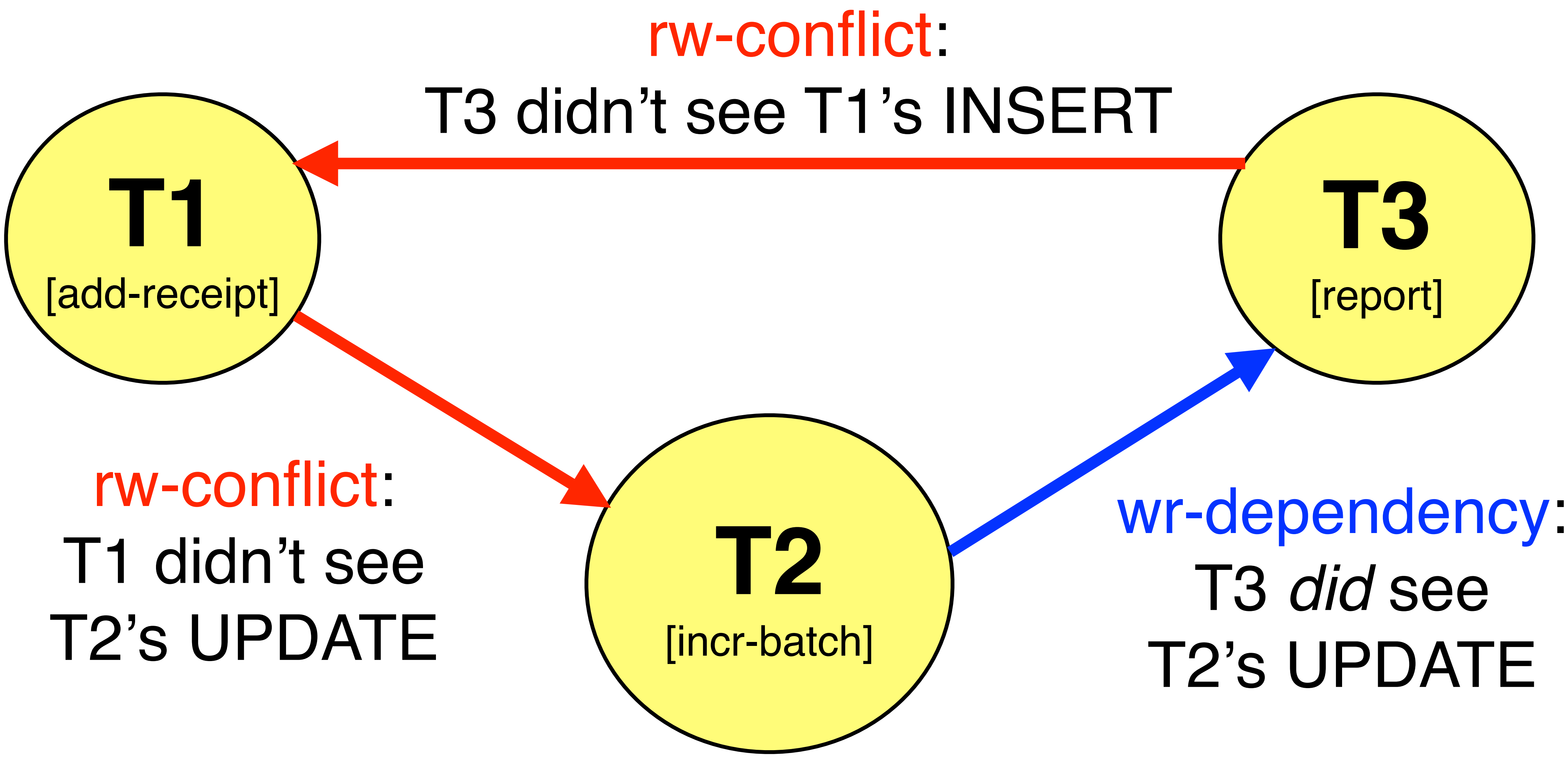
T1	**T2**	**T3**
SELECT batch FROM control *[result = 5/19]*		
	UPDATE control SET batch = 5/20	
	COMMIT	
		SELECT batch... *[result = 5/20]*
		SELECT 5/19 receipts [...]
INSERT receipt (5/19, …)		
COMMIT		

T1	**T2**	**T3**
SELECT batch FROM control *[result = 5/19]*		
	UPDATE control SET batch = 5/20	
	COMMIT	
		SELECT batch... *[result = 5/20]*
		SELECT 5/19 receipts [...]
INSERT receipt (5/19, …)		
COMMIT		

rw-conflict

T1	**T2**	**T3**
SELECT batch FROM control *[result = 5/19]*		
	rw-conflict	
	UPDATE control SET batch = 5/20	
	COMMIT	
		wr-dependency
		SELECT batch... *[result = 5/20]*
		SELECT 5/19 receipts [...]
INSERT receipt (5/19, …)		
COMMIT		

T1	**T2**	**T3**
SELECT batch FROM control *[result = 5/19]*		
	rw-conflict	
	UPDATE control SET batch = 5/20	
		wr-dependency
	COMMIT	
		SELECT batch... *[result = 5/20]*
	rw-conflict	SELECT 5/19 receipts [...]
INSERT receipt (5/19, …)		
COMMIT		



Not serializable!

Adding the read-only transaction creates a cycle.

Agenda

- What is serializability? Why do we want it?
- Snapshot isolation vs. serializability
- **Serializable Snapshot Isolation**
- SSI implementation overview
- Using SSI
- Performance results

Existing Approaches to Serializability

- ignore the problem, make the user deal
  - use SELECT FOR UPDATE, LOCK TABLE
  - can be hard to figure out where to put these!
- run one transaction at a time [not practical]
- strict two-phase locking
  - acquire lock on every object read or written
  - causes readers to block writers & vice versa

T1 **T2**

SELECT batch
FROM control
[result = 5/19]

T1 **T2**

SELECT batch
FROM control 
[result = 5/19]

T1

T2

SELECT batch
FROM control 
[result = 5/19]

UPDATE control
SET batch = 5/20
[blocked!]

T1 **T2**

SELECT batch
FROM control 
[result = 5/19]

UPDATE control
SET batch = 5/20
[blocked!]

INSERT receipt
(5/19, …)

COMMIT

SSI Approach (Almost.)

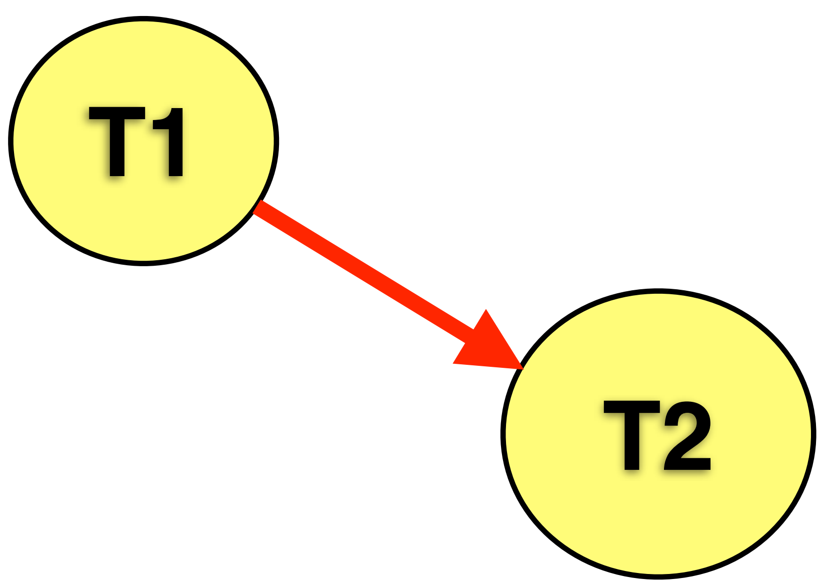
Actually build the dependency graph!

- If a cycle is created,
  abort some transaction to break it

SSI Approach (Almost.)

Actually build the dependency graph!

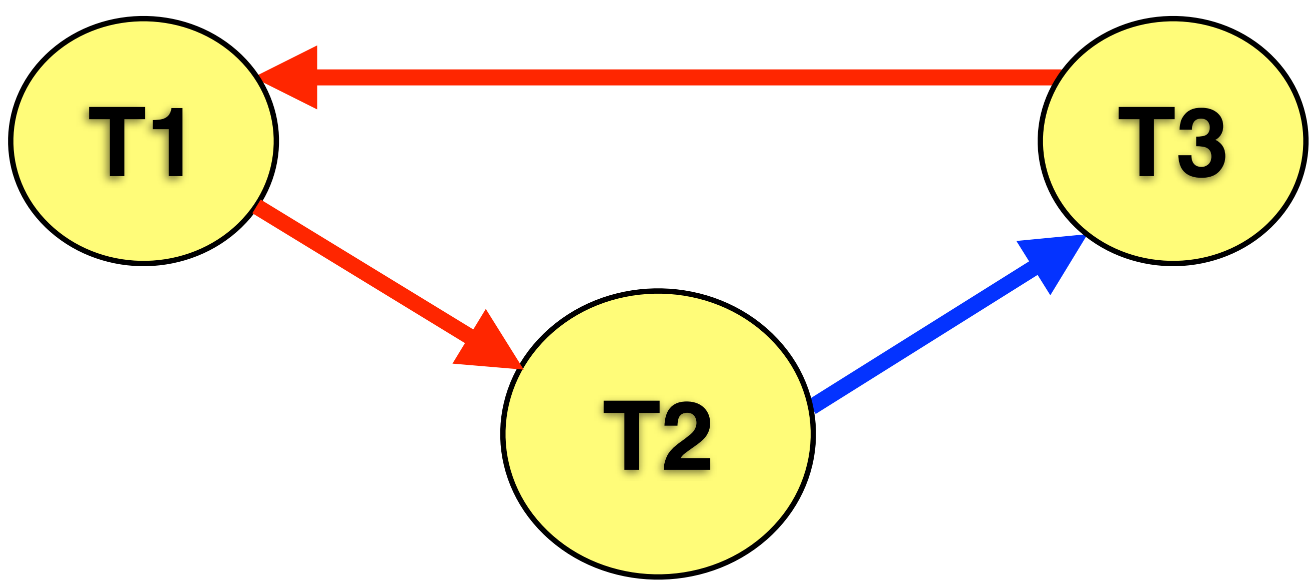
- If a cycle is created,
  abort some transaction to break it



SSI Approach (Almost.)

Actually build the dependency graph!

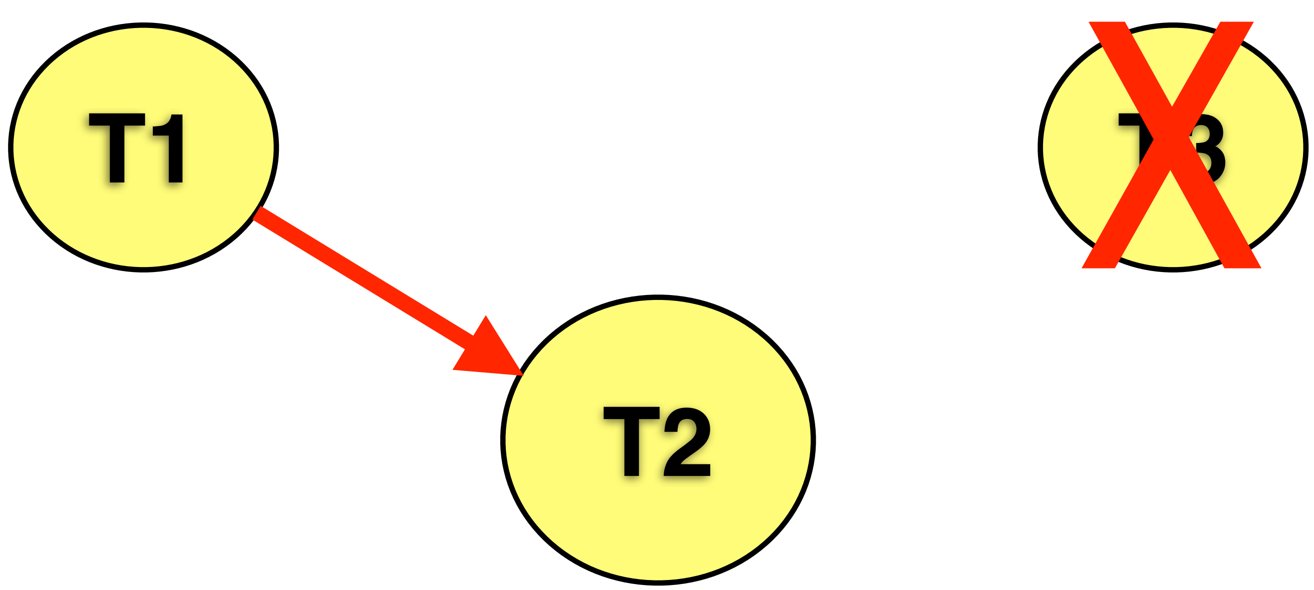
- If a cycle is created,
  abort some transaction to break it



SSI Approach (Almost.)

Actually build the dependency graph!

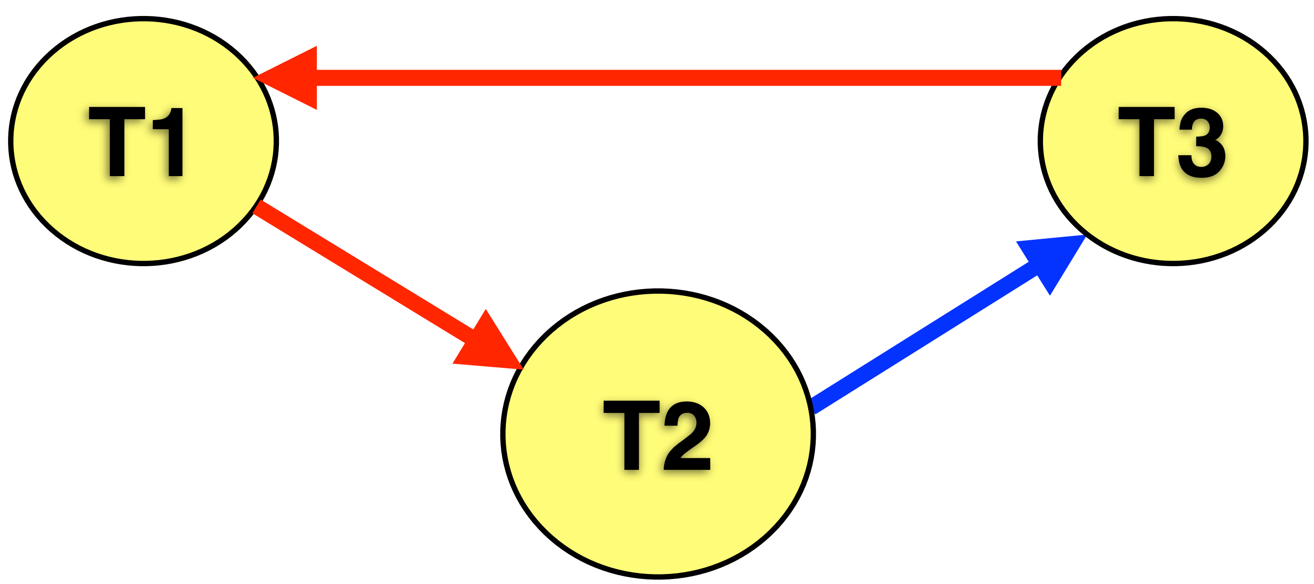
- If a cycle is created,
  abort some transaction to break it



SSI Approach (Almost.)

Actually build the dependency graph!

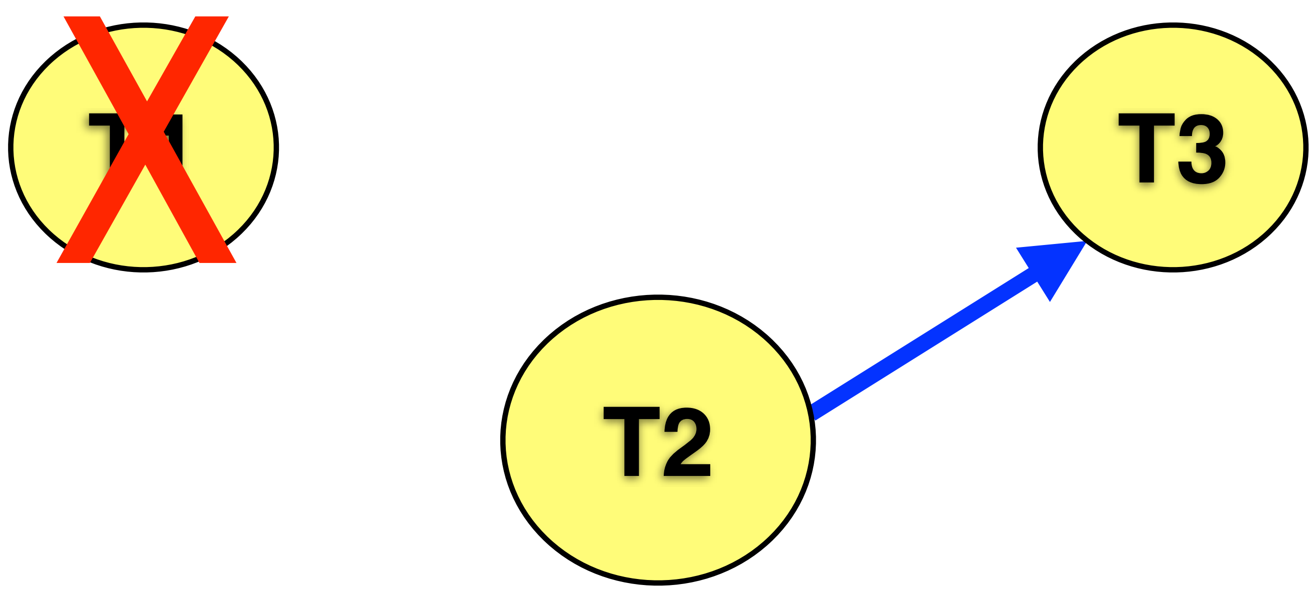
- If a cycle is created,
  abort some transaction to break it



SSI Approach (Almost.)

Actually build the dependency graph!

- If a cycle is created,
  abort some transaction to break it



Serializability theory tells us:

- every cycle contains two adjacent rw-conflict edges (where A didn't see B's update)

So we can just look for those

- don't need to track other types of edges
- conservative (occasional false positives)

SSI Rule:

Don't let a transaction have *both* a rw-conflict in and a rw-conflict out!

[Cahill et al. Serializable Isolation For Snapshot Databases, SIGMOD '08]

Agenda

- What is serializability? Why do we want it?
- Snapshot isolation vs. serializability
- Serializable Snapshot Isolation
- **SSI implementation overview**
- Using SSI
- Performance results

Implementing SSI

Need to keep some extra transaction state

- mainly: list of rw-conflicts in and out
- if one transaction has both, abort something
- note: need to keep lists after xact commits, until *all concurrent transactions* commit

But how do we identify a rw-conflict?

Identifying rw-conflicts

Recall: T1 —> T2 if T2 makes a change, and T1's read doesn't see its effects

- If T2's write happens first:
  T1 will see tuple's MVCC data and ignore it

xmin	xmax	data
T2		…

- If T1's read happens first:
  use a "lock" to know that T2's write conflicts

Identifying rw-conflicts

Recall: T1 —> T2 if T2 makes a change, and T1's read doesn't see its effects

- If T2's write happens first:
  T1 will see tuple's MVCC data and ignore it

xmin	xmax	data
T2		…

T2 not in T1's snapshot
=> conflict w/ T1

- If T1's read happens first:
  use a "lock" to know that T2's write conflicts

Tracking Read Dependencies

Acquire a “SIREAD lock” on anything read

Check for SIREAD locks on write, flag conflict

New lock manager — unlike current locks:

- no blocking! (just flag a conflict instead)
- can persist beyond transaction commit
- multi-granularity (relation, page, tuple); promotion
- needs predicate locking

Predicate Locking

Not enough just to lock returned tuples:

SELECT FROM...	INSERT INTO…
WHERE x=42	VALUES (x=42)
[3 results]	*[should conflict; won't]*

Really want predicate locking:
"lock everything where x=42" (but not feasible)

Instead: lock corresponding index page

- if no index, lock entire relation

Other Features

Deferrable read-only transactions

- wait until xact can be executed safely without lock overhead or risk of abort

Dealing with shared memory exhaustion

- promote locks to coarser granularity
- reduce information about committed transactions and push to disk if necessary (SLRU)

Agenda

- What is serializability? Why do we want it?
- Snapshot isolation vs. serializability
- Serializable Snapshot Isolation
- SSI implementation overview
- **Using SSI**
- Performance results

Conflicts may cause transactions to abort

- source of conflict might not be obvious
- will usually succeed if retried
- middleware that automatically retries can help

Performance tips

- declare transactions READ ONLY if possible
- don’t put more into a single transaction than needed
- don’t leave connections dangling “idle in transaction”

Agenda

- What is serializability? Why do we want it?
- Snapshot isolation vs. serializability
- Serializable Snapshot Isolation
- SSI implementation overview
- **Performance results**

Performance

Two main sources of slowdown

- How much CPU overhead does the SIREAD lock manager add?
  - in-memory pgbench: not much slowdown
- How often are transactions rolled back because of conflicts?
  - depends heavily on workload

Measuring Abort Rate

DBT-2 benchmark (OLTP, like TPC-C)

- modified to retry transactions after serialization failure

Configuration:

- 16-core Xeon E7310, 1.60GHz, 8 GB RAM
- 3x 15K drives for data; 1 for log
- database size ~20 GB

DBT-2 Performance

Approach: use highest scale factor that gives 90% request latency < 5 seconds

REPEATABLE READ (snapshot isolation):

- 160 warehouses, 1941 new order transactions/minute
- 1.5% transactions retried due to serialization failure

SERIALIZABLE (SSI):

- 157 warehouses, 1923 NOTPM (< 2% slowdown)
- 3.1% transactions retried due to serialization failure
  - no aborts of read-only transactions
  - 15% abort rate for “delivery” xacts (4% of workload)

Summary

True serializable transactions are here!

- avoiding snapshot isolation anomalies can simplify applications
- implemented using a novel technique
- reuses existing snapshot isolation mechanisms
- performance cost is reasonable