Serializable Snapshot Isolation in PostgreSQL

Dan Ports
University of Washington
MIT

Kevin Grittner
Wisconsin Supreme Court
For years, PostgreSQL’s “SERIALIZABLE” mode did not provide true serializability

• instead: snapshot isolation – allows anomalies

PostgreSQL 9.1: **Serializable Snapshot Isolation**

• based on recent research [Cahill, SIGMOD ’08]

• first implementation in a production DB release & first in a purely-snapshot DB
This talk....

- Motivation: Why serializability?
  Why did we choose SSI?
- Review of snapshot isolation and SSI
- Implementation challenges & optimizations
- Performance
Serializability vs. Performance

Two perspectives:

• Serializability is important for correctness
  • simplifies development; don’t need to worry about race conditions

• Serializability is too expensive to use
  • locking restricts concurrency; use weaker isolation levels instead
Serializability vs. Performance
(in PostgreSQL)

PostgreSQL offered *snapshot isolation* instead

- better performance than 2-phase locking
  “readers don’t block writers, writers don’t block readers”
- but doesn’t guarantee serializability!

Snapshot isolation isn’t enough for some users

- complex databases with strict integrity requirements,
  e.g. Wisconsin Court System
Serializability vs. Performance (in PostgreSQL)

PostgreSQL offered *snapshot isolation* instead

• better performance than 2-phase locking
  “readers don’t block writers, writers don’t block readers”

• but doesn’t guarantee serializability!

Snapshot isolation isn’t enough for some users

• complex databases with strict integrity requirements,
  e.g. Wisconsin Court System

**Serializable Snapshot Isolation**

offered true serializability with

performance benefits of snapshot isolation!
Serializable Snapshot Isolation

SSI approach:

- run transactions using snapshot isolation
- detect conflicts between transactions at runtime; abort transactions to prevent anomalies

Appealing for performance reasons

- aborts less common than blocking under 2PL
- readers still don’t block writers!

[Cahill et al. Serializable Isolation for Snapshot Databases, SIGMOD ’08]
SSI in PostgreSQL

Available in PostgreSQL 9.1; first production implementation

Contributions: new implementation techniques

- Detecting conflicts in a purely-snapshot DB
- Limiting memory usage
- Read-only transaction optimizations
- Integration with other PostgreSQL features
Outline

- Motivation
- Review of snapshot isolation and SSI
- Implementation challenges & optimizations
- Performance
- Conclusions
Goal:
ensure at least one guard always on-duty

<table>
<thead>
<tr>
<th>guard</th>
<th>on-duty?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>y</td>
</tr>
<tr>
<td>Bob</td>
<td>y</td>
</tr>
</tbody>
</table>
Goal:
ensure at least one guard always on-duty

<table>
<thead>
<tr>
<th>guard</th>
<th>on-duty?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>y</td>
</tr>
<tr>
<td>Bob</td>
<td>y</td>
</tr>
</tbody>
</table>

BEGIN

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

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

COMMIT
<table>
<thead>
<tr>
<th>guard</th>
<th>on-duty?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>y</td>
</tr>
<tr>
<td>Bob</td>
<td>y</td>
</tr>
</tbody>
</table>
BEGIN

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

[result = 2]

<table>
<thead>
<tr>
<th>guard</th>
<th>on-duty?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>y</td>
</tr>
<tr>
<td>Bob</td>
<td>y</td>
</tr>
</tbody>
</table>
BEGIN

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

BEGIN

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

<table>
<thead>
<tr>
<th>guard</th>
<th>on-duty?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>y</td>
</tr>
<tr>
<td>Bob</td>
<td>y</td>
</tr>
</tbody>
</table>
BEGIN

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

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

<table>
<thead>
<tr>
<th>guard</th>
<th>on-duty?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>y</td>
</tr>
<tr>
<td>Bob</td>
<td>y</td>
</tr>
</tbody>
</table>
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]

BEGIN

<table>
<thead>
<tr>
<th>guard</th>
<th>on-duty?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>y</td>
</tr>
<tr>
<td>Bob</td>
<td>y</td>
</tr>
</tbody>
</table>

Tuesday, August 28, 2012
BEGIN

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

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

<table>
<thead>
<tr>
<th>guard</th>
<th>on-duty?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>y</td>
</tr>
<tr>
<td>Bob</td>
<td>y</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>guard</th>
<th>on-duty?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>y</td>
</tr>
<tr>
<td>Bob</td>
<td>y</td>
</tr>
</tbody>
</table>

BEGIN

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

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

COMMIT

Tuesday, August 28, 2012
BEGIN

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

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

<table>
<thead>
<tr>
<th>guard</th>
<th>on-duty?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>y</td>
</tr>
<tr>
<td>Bob</td>
<td>y</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>guard</th>
<th>on-duty?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>y</td>
</tr>
<tr>
<td></td>
<td>n</td>
</tr>
<tr>
<td>Bob</td>
<td>y</td>
</tr>
<tr>
<td></td>
<td>n</td>
</tr>
</tbody>
</table>

rw-conflict: T1 didn’t see T2’s UPDATE
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 guard = 'Alice'
}
COMMIT
SSI Approach

Detect these rw-conflicts and maintain a conflict graph

Serializability theory: each anomaly involves two adjacent rw-conflict edges

• if found, abort some involved transaction

• note: can have false positives
T2

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

T2

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

two adjacent edges:
T1 -> T2 and T2 -> T1
\textbf{rw-conflict:}

T1 didn’t see T2’s UPDATE

\textbf{rw-conflict:}

T2 didn’t see T1’s UPDATE

two adjacent edges:

T1 -> T2 and T2 -> T1
T1

ERROR: could not serialize access due to read/write dependencies among transactions
HINT: The transaction might succeed if retried.

T2

two adjacent edges: T1 -> T2 and T2 -> T1

Tuesday, August 28, 2012
Outline

• Motivation
• Review of snapshot isolation and SSI
• Implementation challenges & optimizations
• Performance
• Conclusions
SSI in PostgreSQL

Implementation challenges:

• Detecting conflicts in a purely-snapshot DB
  • requires new lock manager

• Reining in potentially-unbounded memory usage
Detecting Conflicts

How to detect when an update conflicts with a previous read?

Previous SSI implementations: reuse read locks from existing lock mgr

But...

- PostgreSQL didn’t have read locks!
- ...let alone predicate locks
SSI Lock Manager

Needed to build a new lock manager to track read dependencies

- Uses multigranularity locks, index-range locks
- Doesn’t block, just flags conflicts
  => no deadlocks
- Locks need to persist past transaction commit
Memory Usage

Need to keep track of transaction readsets + conflict graph

- not just active transactions; also committed ones that ran concurrently

- one long-running transaction can cause memory usage to grow without bound

Could exhaust shared memory space (esp. in PostgreSQL)
Read-Only Transactions

Many long-running transactions are read-only; optimize for these

Safe snapshots: cases where r/o transactions can never be a part of an anomaly

- can then run using regular SI w/o SSI overhead
- but: can only detect once all concurrent r/w transactions complete

Deferrable transactions: delay execution to ensure safe snapshot
Graceful Degradation

What if we still run out of memory?
Don’t want to refuse to accept new transactions
Instead: keep less information
(tradeoff: more false positives)

• keep less state about committed transactions
• deduplicate readsets: “read by some committed transaction”
Outline

• Motivation
• Review of snapshot isolation and SSI
• Implementation challenges & optimizations
• Performance
• Conclusions
Performance

TPC-C-derived benchmark; modified to have SI anomalies

Varied fraction of r/o and r/w transactions

Compared PostgreSQL 9.1’s SSI against SI, and an implementation of S2PL
Performance (in-memory)

25 warehouses (3 GB), tmpfs

![Graph showing performance metrics](image)

- **SI**
- **SSI**
- **SSI (no r/o opt.)**
- **S2PL**

**trans. rate (normalized)**

**Fraction of read-only transactions**

*Tuesday, August 28, 2012*
Performance (disk)

150 warehouses (19 GB)

Trans. rate (normalized)

Fraction of read-only transactions
Conclusion

SSI available now in PostgreSQL 9.1

• true serializability without blocking
• new lock manager to track read dependencies
• optimizations for read-only transactions

Performance close to that of SI

• outperforms S2PL on read-heavy workloads
• makes serializable mode a more practical option for some users