

Serializable Snapshot Isolation

Heikki Linnakangas / EnterpriseDB

Serializable in PostgreSQL

`BEGIN ISOLATION LEVEL SERIALIZABLE;`

- In <= 9.0, what you actually got was *Snapshot Isolation*
- In 9.1, you get the real thing!

SQL-92 Isolation levels

BEGIN ISOLATION LEVEL

- READ UNCOMMITTED
 - READ COMMITTED
 - REPEATABLE READ
 - SERIALIZABLE
- ;
- Serializable means: results equivalent to *some* serial ordering of the transactions

PostgreSQL Isolation levels, in 9.0 and earlier

- READ COMMITTED
- Snapshot Isolation
- Snapshot Isolation level falls somewhere between ANSI Repeatable Read and Serializable.

Read Committed

```
create table t (id int not null primary key);  
insert into t select generate_series(1, 10);
```

```
delete from t where id = (select min(id) from  
t);
```

Q: How many rows are deleted by the delete statement if there are 10 rows in the table?

Read Committed

```
create table t (id int not null primary key);  
insert into t select generate_series(1, 10);
```

```
    # begin;  
    BEGIN  
    # update t set id = id - 1;  
    UPDATE 10
```

```
# delete from t where id = (select min(id) from  
t);  
DELETE 0  
    # commit;  
    COMMIT
```

A: It depends.

Snapshot Isolation (pre-9.1)

- At the beginning of transaction, the system takes a snapshot of the database
- All queries in the transaction return results from that snapshot
 - Any later changes are not visible
- On conflict, the transaction is aborted:

```
# delete from t where id = (select min(id) from t);  
ERROR: could not serialize access due to  
concurrent update
```

Goal:

ensure at least one
guard always on-duty

| guard | on-duty? |
|-------|----------|
| Alice | y |
| Bob | y |

```
BEGIN
```

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



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

```
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
```

DIY referential integrity

- CREATE TRIGGER BEFORE INSERT ON childtable ...
IF NOT EXISTS (
 SELECT 1 FROM parent WHERE id =
 NEW.parentid) THEN
 RAISE ERROR 'parent not found'
- Not safe without real Serializability!

How do I know if my application is affected?

- Carefully inspect **every** transaction in the application
 - Difficult.
 - Not feasible in large applications
- Bugs arising from insufficient isolation are difficult to debug

Summary this far

- Isolation Levels:
 - Read Committed
 - Snapshot Isolation
 - Still not good enough
- Serializable Snapshot Isolation

SSI to the rescue!

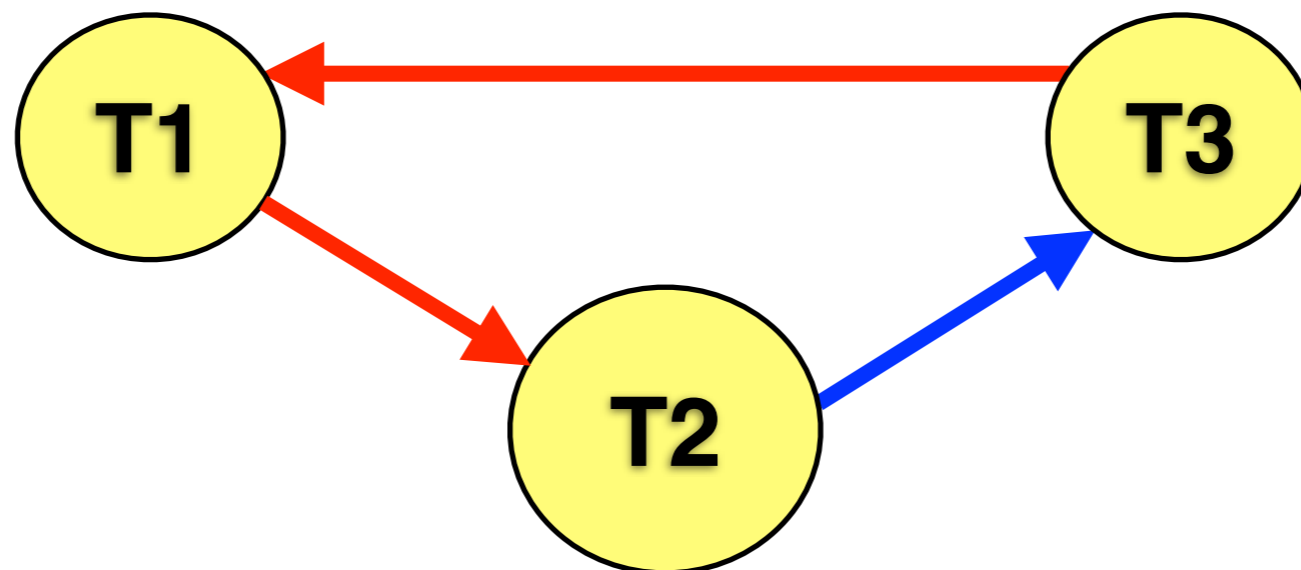
In 9.1, SERIALIZABLE gives you
Serializable Snapshot Isolation

- Based on Snapshot Isolation
- Detects the cases where Snapshot Isolation goes wrong, and aborts

SSI Approach (Almost.)

Actually build the dependency graph!

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



SSI behavior

- Conservative
 - Transactions are sometimes aborted unnecessarily
- Prefers to abort "pivot" transaction, so that when the aborted transaction is retried, you make progress
- Introduces predicate locking
 - Also locks "gaps" between rows

SSI Predicate locks

```
SELECT * FROM mytable  
WHERE id BETWEEN 5 AND 10;
```

- Locks not only the matched rows, but the range where any matches might've been
- Detects later INSERTs that match the WHERE-clause
- Lock granularity: index page or whole table



3
5
7
9
11
13

SSI Predicate locks

```
BEGIN ISOLATION LEVEL SERIALIZABLE;
```

```
SELECT * FROM mytable WHERE id = 10;
```

```
SELECT mode, locktype, relation::regclass, page, tuple  
FROM pg_locks WHERE mode = 'SIReadLock';
```

| mode | locktype | relation | page | tuple |
|-------------------|----------|--------------|------|-------|
| SIReadLock | tuple | mytable | 0 | 10 |
| SIReadLock | page | mytable_pkey | 1 | |

(2 rows)

Performance

- SSI has overhead
 - Predicate locking
 - Detecting conflict
 - Increased number of rollbacks due to conflicts

Performance

”

The only real answer is "it depends". At various times I ran different benchmarks where the overhead **ranged from "lost in the noise" to about 5% for one variety of "worst case"**. Dan ran DBT-2, following the instructions on how to measure performance quite rigorously, and came up with a **2% hit versus repeatable read for that workload**. I rarely found a benchmark where the hit exceeded 2%, but **I have a report of a workload where they hit was 20%** -- for constantly overlapping long-running transactions contending for the same table.

– Kevin Grittner on pgsql-hackers mailing list (Mon, 10 Oct 2011)

Performance

However, we seem to have a problem with scaling to many CPUs:

”

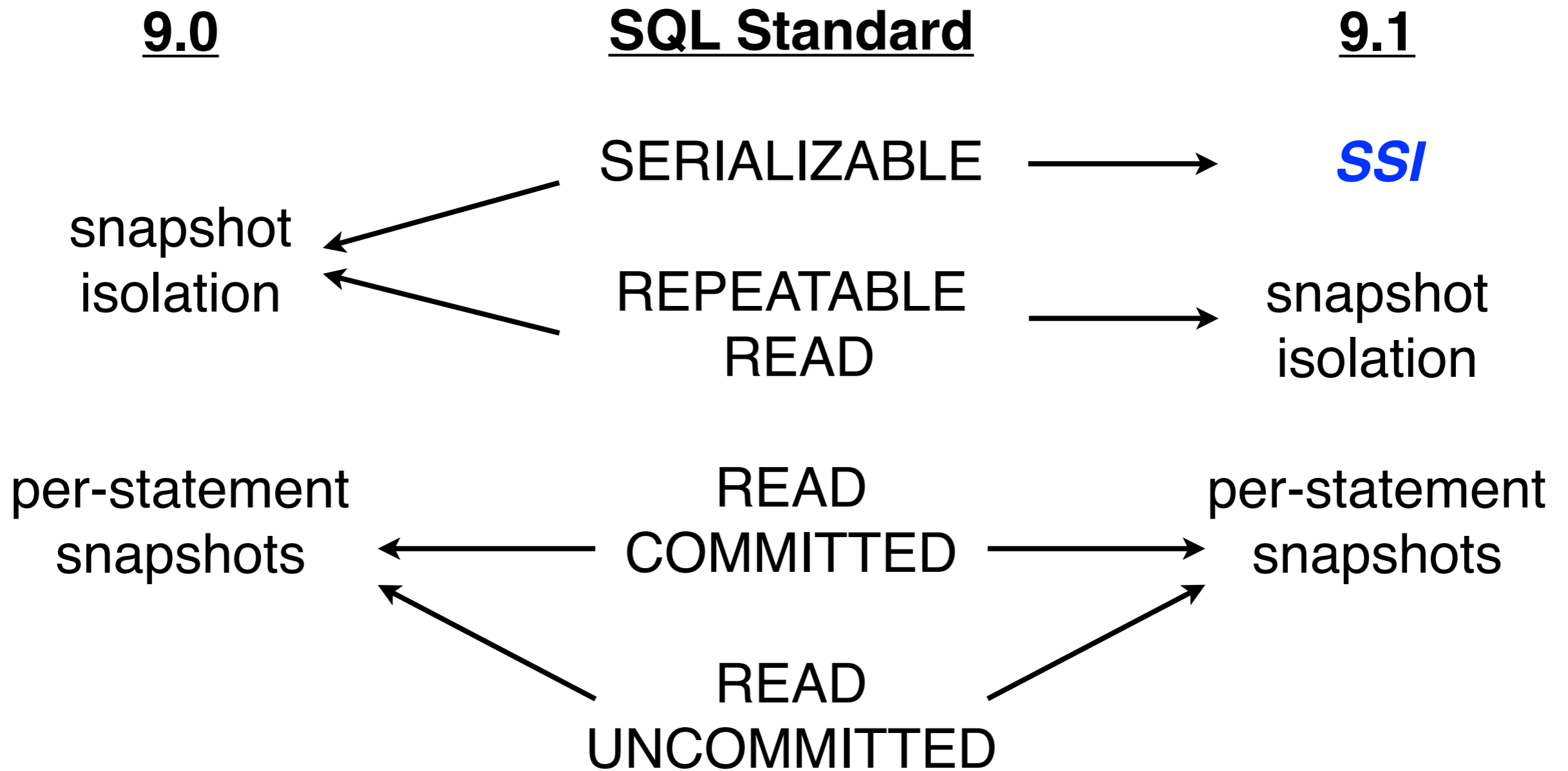
I ran my good old pgbench -S, scale factor 100, shared_buffers = 8GB test on Nate Boley's box.

... Serializable mode is much slower on this test, though. On REL9_1_STABLE, **it's about 8% slower with a single client**. At 8 clients, the difference rises to 43%, and at 32 clients, it's 51% slower. On 9.2devel, raw performance is somewhat higher (e.g. +51% at 8 clients) but the performance when not using SSI has improved so much that the performance gap between serializable and the other two isolation levels is now huge: with 32 clients, in serializable mode, the median result was 21114.577645 tps; in read committed, 218748.929692 tps - that is, **read committed is running more than ten times faster than serializable**.

– Robert Haas on pgsql-hackers mailing list (Tue, 11 Oct 2011)

- Hopefully that will be improved in 9.2 ...

Isolation Levels



Writing applications under SSI

- You can ignore concurrency issues
 - No need for SELECT FOR UPDATE/SHARE, let alone LOCK TABLE
- Be prepared to retry aborted transactions
- Declare read-only transactions as such
 - BEGIN READ ONLY;
- Avoid long-running transactions

Thank you!

- Michael J. Cahill et al
 - For inventing SSI
- Kevin Grittner and Dan Ports
 - For implementing SSI in PostgreSQL

Feedback:

<http://2011.pgconf.eu/feedback>