PostgreSQL Concurrency Issues

Tom Lane
Red Hat Database Group
Red Hat, Inc.

Introduction

What I want to tell you about today:

- **How PostgreSQL handles concurrent operations**
  - Snapshots
  - Conflict resolution rules
- **How to write application queries to get the right results without sacrificing performance**
  - Read-only transactions are pretty simple
  - For read-write transactions, there are several approaches depending on complexity of the transaction

What problem are we talking about?

We want to run transactions concurrently with:

- correct behavior
- good performance

“Correct behavior” basically means *serializability*. The transactions should appear to execute one at a time, in isolation. When two transactions physically overlap in time, we don’t care which one appears to go first; but we don’t want to see any behavior that reveals that they were concurrent. Any such behavior would look like a bug to one or both users.

The easy way to do this would be to force transactions to actually go one at a time . . . but for high-volume sites that kills performance. Ideally, no transaction should block another from proceeding.

Snapshots isolate transactions

Here’s the key idea in PostgreSQL’s implementation of concurrency:

> Snapshots isolate transactions from the effects of concurrently running transactions.

Each transaction runs against a notional “snapshot” of the committed state of the database at a particular instant. It cannot see the effects of uncommitted transactions (except itself), nor of transactions that commit after the snapshot time.

Every update of a database row generates a new version of the row.

Snapshot information determines which version of a given row is visible to a given transaction. Old versions of rows can be reclaimed (by `VACUUM`) once they are no longer visible to any running transaction.

PostgreSQL has two “isolation levels”

You can run PostgreSQL transactions in either of two modes:

SERIALIZABLE: a snapshot is taken at start of a transaction (actually, at start of first query within transaction). A reader’s view of the database is thus fixed throughout a transaction.

READ COMMITTED: a new snapshot is taken at start of each interactive query. So, view of the database is stable within an interactive query, but can change across queries within a transaction.

(The two modes also differ in update behavior, as we’ll discuss later.)

These modes are a subset of the four isolation levels defined by SQL 92. The other two SQL 92 modes correspond to behaviors that are useful in a locking-based implementation, but are not interesting in a snapshot-based implementation.

Easy cases

We can cover these “easy” cases pretty quickly:

- All read-only transactions
- Many read-only transactions, only one writer (at a time)

The all-read-only case is obviously trivial. The one-writer case is nearly trivial: snapshotting isolates the readers from the effects of the writer until the writer commits.

“Interesting” cases have multiple writers to the same or related tables. Then we have to worry about conflicting updates.

But let's look more closely at the one-writer case

The writer's updates will all become visible when it commits, and will not be visible before that.

Readers running in SERIALIZABLE mode will not see the changes from writers that commit after they (the readers) start. So, they have a stable view of the database throughout each transaction.

Readers running in READ COMMITTED mode will be able to see changes beginning at first SQL statement issued after writer commits. So, the view is stable within a statement but not across statements. This is messier, but necessary if (for example) you have procedures that want to observe concurrent changes.

Either way, readers do not block the writer, and the writer does not block readers, so concurrent performance is excellent.

Reader correctness: a simple example

Suppose we want to cross check deposit totals against bank branch totals:

```
BEGIN;
SELECT SUM(balance) FROM accounts;
SELECT SUM(branch_balance) FROM branches;
-- check to see that we got the same result
COMMIT;
```

This is correct in serializable mode, but **wrong** in read committed mode. To select serializable mode, either add

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

to each transaction block (just after `BEGIN`), or do

```
SET DEFAULT_TRANSACTION_ISOLATION TO SERIALIZABLE;
```

(this can be set in the `postgresql.conf` file, or per-connection).

An important coding rule

Be sure that reader activity is broken into transactions of appropriate length.

Pure reader transactions should typically be run in SERIALIZABLE mode, so that they see a consistent database view throughout the transaction; then you set the transaction boundaries to define the times when you want to recognize updates committed by other clients.

If you start one serializable transaction and hold it for the entire run of an application, you'll never observe any changes in database state, other than those you make yourself. This is what you want in a few cases (for instance, the backup utility `pg_dump` does it to make a self-consistent dump) but usually it's not what you want.

Beware of really long transactions

Holding a transaction open means that rows potentially visible to that transaction can't be reclaimed during `VACUUM`, even if they're long dead to everyone else.

Unlike Oracle, we won't bomb out when a limited "rollback" area is filled; but once your whole disk fills up, you've got a problem anyway.

So, avoid coding an application in a way that allows it to issue `BEGIN` and then go to sleep for hours or days. Even though it's not doing anything, it may be blocking `VACUUM` from recovering space.

The rules for concurrent writing

Two concurrent transactions cannot write (`UPDATE`, `DELETE`, or `SELECT FOR UPDATE`) the same row. They can proceed as long as they write nonoverlapping sets of rows.

If a transaction tries to write a row already written by a concurrent transaction:

1. If other transaction is still in progress, wait for it to commit or abort.
2. If it aborted, proceed (using the non-updated version of the row).
3. If it committed:
   (a) In SERIALIZABLE mode: abort with “can’t serialize” error.
   (b) In READ COMMITTED mode: proceed, using the newly-committed (latest) version of the row as the starting point for my operation. (But first, recheck the new row to see if it still satisfies the `WHERE` clause of my query; if not, ignore it.)

Our first concurrent-writers example

We want to keep count of hits on different pages of a website. We have a table with one row per webpage, and we are going to have a lot of clients (webserver threads) concurrently executing something like

```
UPDATE webpages SET hits = hits + 1 WHERE url = '...';
```

Question: is this safe? In particular, can we ever “lose” a hit count because two clients do this statement in parallel on the same row?

Concurrent writers can fail without some interlock

It's easy to see that there could be a problem:

```
UPDATE webpages SET hits = hits + 1 WHERE url = '...';
```

CLIENT 1	CLIENT 2
reads row, sees hits = 531	
	reads row, sees hits = 531
computes hits+1 = 532	
	computes hits+1 = 532
writes hits = 532	
	writes hits = 532
commit	
	commit

Oops . . . final state of the row is 532, not 533 as we'd wish.

READ COMMITTED works perfectly for this

But this bug will not occur in PostgreSQL. In READ COMMITTED mode, the second client is forced to redo his computation:

CLIENT 1	CLIENT 2
reads row, sees hits = 531	
	reads row, sees hits = 531
computes hits+1 = 532	
	computes hits+1 = 532
writes hits = 532	
	tries to write row, must wait
commit	
	re-reads row, sees hits = 532
	re-computes hits+1 = 533
	writes hits = 533
	commit

READ COMMITTED vs. SERIALIZABLE mode

In SERIALIZABLE mode, the second client would instead get a “Can’t serialize access due to concurrent update” error, and would have to retry the transaction until he succeeded.

You always need to code a retry loop in the client when you are using SERIALIZABLE mode for a writing transaction.

Why would you want to use SERIALIZABLE mode, given that it requires extra client-side programming? Because READ COMMITTED doesn’t scale very well to more complex situations.

Example with separate read and write commands

Suppose we wanted to do our updates as a `SELECT` then `UPDATE`:

```
BEGIN;
SELECT hits FROM webpages WHERE url = '...';
-- client internally computes $newval = $hits + 1
UPDATE webpages SET hits = $newval WHERE url = '...';
COMMIT;
```

This might look silly, but it’s not so silly if the “internal computation” is complex, or if you need data from multiple database rows to make the computation, or if you’d like to know exactly what you just updated the hits count to.

READ COMMITTED case

In READ COMMITTED mode, this approach **does not work**:

CLIENT 1	CLIENT 2
reads row, sees hits = 531	
	reads row, sees hits = 531
computes $newval = 532	
	computes $newval = 532
writes hits = 532	
	tries to write row, must wait
commit	
	re-reads row, sees hits = 532
	re-computes $newval, still 532
	writes hits = 532
	commit

Solution #1: use SELECT FOR UPDATE, not just SELECT, to read the input data

We can make it work by doing this:

```
BEGIN;
SELECT hits FROM webpages WHERE url = '...'  FOR UPDATE;
-- client internally computes $newval = $hits + 1
UPDATE webpages SET hits = $newval WHERE url = '...';
COMMIT;
```

`FOR UPDATE` causes the initial `SELECT` to acquire a row lock on the target row, thus making it safe to do our internal computation and update the row.

Furthermore, `SELECT FOR UPDATE` returns the latest updated contents of the row, so we aren't working with stale data.

How SELECT FOR UPDATE solution works

CLIENT 1	CLIENT 2
locks row, reads hits = 531	
	tries to lock row, must wait
computes $newval = 532	
writes hits = 532	
commit	
	locks row, reads hits = 532
	computes $newval = 533
	writes hits = 533
	commit

We need both the delay till commit and the read of the updated value to make this work. Notice `SELECT FOR UPDATE` may return data committed after its start time, which is different from plain `SELECT`'s behavior in READ COMMITTED mode.

Solution #2: use SERIALIZABLE mode and retry on serialization error

```
loop
   BEGIN;
   SELECT hits FROM webpages WHERE url = '...';
   -- client internally computes $newval = $hits + 1
   UPDATE webpages SET hits = $newval WHERE url = '...';
   if (no error)
      break out of loop;
   else
      ROLLBACK;
end loop
COMMIT;
```

This works because client will re-read the new hits value during retry. The `UPDATE` will succeed only if the data read is still current.

The client could get confused, too

Consider this small variant of the example, running in READ COMMITTED mode:

```
BEGIN;
SELECT hits FROM webpages WHERE url = '...';
UPDATE webpages SET hits = hits + 1 WHERE url = '...';
COMMIT;
```

If someone else increments the hits count while the SELECT is running, the UPDATE will compute the correct updated value . . . but it will be two more than what we just read in the SELECT, not one more. (And that's what we'd see if we did another SELECT.) If your application logic could get confused by this sort of thing, you need to use either SELECT FOR UPDATE or SERIALIZABLE mode.

Performance comparison

The READ COMMITTED + SELECT FOR UPDATE approach is essentially pessimistic locking: get the lock first, then do your work.

The SERIALIZABLE + retry-loop approach is essentially optimistic locking with retry. It avoids the overhead of getting the row lock beforehand, at the cost of having to redo the whole transaction if there is a conflict.

SELECT FOR UPDATE wins if contention is heavy (since the serializable approach will be wasting work pretty often). SERIALIZABLE wins if contention for specific rows is light.

What about updating multiple rows?

Example: transfer $100.00 from Alice's account to Bob's account.

Let's try to do it in READ COMMITTED mode:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.0
       WHERE ownername = 'Alice';
UPDATE accounts SET balance = balance + 100.0
       WHERE ownername = 'Bob';
COMMIT;
```

(Or we could do the arithmetic on the client side, in which case we'd need `SELECT FOR UPDATE` then `UPDATE`.)

This is correct . . . but . . .

We have trouble if the same rows can be updated in different orders

Problem: suppose another transaction is concurrently transferring from Bob's account to Alice's.

CLIENT 1	CLIENT 2
locks and updates Alice's row	
	locks and updates Bob's row
tries to lock Bob's row, must wait	
	tries to lock Alice's row, must wait

We have a classic deadlock situation. PostgreSQL will detect the deadlock and get out of it by aborting one of the two transactions with a deadlock error. (The other one will then be able to complete.)

So you end up needing a retry loop anyway

We can recover from such failures by placing a retry loop around the whole transaction on the client side. The transaction that got killed to resolve the deadlock will be retried, and eventually will complete successfully.

In such cases READ COMMITTED isn't any simpler to program than SERIALIZABLE mode: you need the same retry loop either way. (The expected failures are slightly different, but the response to them is the same.)

Unless you're expecting lots of contention, you might as well use SERIALIZABLE mode and avoid the row-locking overhead.

Multi-row constraints, or why serializable mode isn't really serializable

Suppose we want to check that we always have at least $1000 total cash in our checking and savings accounts. A typical transaction will look like:

```
BEGIN;
UPDATE my_accounts SET balance = balance - $withdrawal
       WHERE accountid = 'checking';
SELECT SUM(balance) FROM my_accounts;
if (sum >= 1000.00)
       COMMIT;
else
       ROLLBACK;
```

Is this safe? **NO**, not even in serializable mode.

Multi-row constraints can fail

Suppose we have $600 in each account and we concurrently try to withdraw $200 from each.

CLIENT 1	CLIENT 2
read checking balance, get 600	
update checking balance to 400	
	read savings balance, get 600
	update savings balance to 400
run SUM, get $400 + 600 = 1000$	
	run SUM, get $600 + 400 = 1000$
commit	
	commit

Now the total is only $800; oops. Neither client saw the other's uncommitted change, so the constraint checks both succeeded.

Some observations

This example proves that SERIALIZABLE mode isn't really serializable, because if we'd done the two transactions serially (in either order) the second one would have seen the problem and aborted.

Technically, the difficulty here is that PostgreSQL doesn't do *predicate locking*, which is necessary for full serializability guarantees. (Predicate locking means write-locking all rows that satisfy the WHERE condition of any query executed by any open transaction. It's horridly expensive. . .)

Note that it doesn't matter whether the constraint check is done by a client command or inside a trigger. The same visibility rules apply either way.

Solution: use explicit locking

The only solution available in current PostgreSQL is to do explicit locking at the table level:

```
BEGIN;
LOCK TABLE my_accounts IN SHARE ROW EXCLUSIVE MODE;
UPDATE my_accounts SET balance = balance - $withdrawal
       WHERE accountid = 'checking';
SELECT SUM(balance) FROM my_accounts;
-- commit if sum >= 1000.00, else abort
```

Since only one transaction can hold `SHARE ROW EXCLUSIVE` lock at a time, we know that only our own transaction is writing the table, and so the `SUM` will not miss any uncommitted updates. Furthermore, any new writer will have to wait at his `LOCK` command until we've committed our own update, and then he'll see it when he does his `SUM`.

Explicit locking and SERIALIZABLE mode

By the way, this solution works in either READ COMMITTED or SERIALIZABLE mode:

```
BEGIN;
LOCK TABLE my_accounts IN SHARE ROW EXCLUSIVE MODE;
UPDATE my_accounts SET balance = balance - $withdrawal
       WHERE accountid = 'checking';
SELECT SUM(balance) FROM my_accounts;
-- commit if sum >= 1000.00, else abort
```

The reason it works in SERIALIZABLE mode is that the transaction snapshot is not taken until the first DML statement (the `UPDATE`). If we have to wait at the `LOCK` for other writers, their commits will be included in the snapshot. Note this means that any explicit locks taken in a SERIALIZABLE transaction had better be taken at the top, or they may not do what you want.

What about concurrent inserts?

You might have noticed that I defined concurrent writes as `UPDATE`, `DELETE`, or `SELECT FOR UPDATE` of the same existing row. What about concurrent insertion cases?

Actually, it's not possible for two transactions to try to insert "the same row" – any inserted row is a unique object by definition. So multiple insertions can happen in parallel as far as PostgreSQL is concerned.

The only way to get a conflict during insert is if you have a uniqueness constraint (a unique index) on a table. If two concurrent transactions try to insert rows having the same key value, then the second one will block until the first one finishes. If the first transaction commits, the second one must abort because of the uniqueness constraint; but if the first one aborts the second one can proceed.

How do I manufacture unique row numbers?

If you want to assign a serial number to each row in a table, you might try

```
INSERT INTO mytable (id, ...)
   VALUES( (SELECT MAX(id) + 1 FROM mytable), ...);
```

This will not work safely unless you take an explicit lock on the whole table, which will prevent concurrent insertions. (It'll also be quite slow, because `MAX` scans the whole table in PostgreSQL.) A variant is to use a single-row table to hold the next ID number to assign:

```
SELECT next FROM mytable_counter FOR UPDATE;
UPDATE mytable_counter SET next = $next + 1;
INSERT INTO mytable (id, ...)  VALUES($next, ...);
```

This works (as long as you use FOR UPDATE), but you still have the problem that only one insertion transaction can proceed at a time. The implicit write lock on the counter row is the bottleneck.

Solution: sequence objects

To get around this, PostgreSQL offers *sequence objects*, which are much like the single-row counter table of the previous example:

```
CREATE SEQUENCE mytable_counter;
...
INSERT INTO mytable (id, ...)
   VALUES( nextval('mytable_counter'), ...);
```

Sequences are special because `nextval` is a non-transactional operation: it is never rolled back, and so there's no need to hold a lock on the sequence. When you do a `nextval`, you only gain exclusive access to the sequence for long enough to compute the next value; then other people can access the sequence while the rest of your transaction proceeds.

This allows multiple transactions to proceed with inserts concurrently.

More about sequences

If you want to find out what ID value you just inserted, you can do

```
INSERT INTO mytable (id, ...)
   VALUES( nextval('mytable_counter'), ...);
SELECT currval('mytable_counter');
```

Many people think this might be unsafe, but it is perfectly safe because `currval` is local to your session: it tells you the value that `nextval` last returned for this sequence *in your session*, not the last value obtained by anyone else. So concurrent transactions can't affect the value you get.

A limitation when using sequences to assign serial numbers is that aborted insertion transactions may leave "holes" in the set of used serial numbers – when a transaction fails after having called `nextval`, it won't insert a row into the table, but the sequence doesn't back up.

Summary (the whole talk on one slide)

SERIALIZABLE mode:

- snapshot isolation; no read locks
- row locks on updated/deleted rows (block only writers, not readers)
- can't write row already written by concurrent transaction
- no predicate locks (hence, not truly serializable)
- simple to reason about, but you need a transaction retry loop

READ COMMITTED mode is similar except:

- snapshot is updated at start of each new SQL command
- can write row already written by concurrent transaction (update starts from newest version)
- fine for single-row updates, doesn't scale so well to complex cases

Use explicit locking to check multi-row consistency conditions.

Further reading

A Critique of ANSI SQL Isolation Levels
Berenson, Bernstein, Gray et al. (1995)
`http://citeseer.nj.nec.com/berenson95critique.html`

Good, fairly readable discussion of transaction isolation.

Generalized Isolation Level Definitions
Adya, Liskov and O'Neil (2000)
`http://citeseer.nj.nec.com/adya00generalized.html`

More recent, more complete, more difficult to follow.

Database Tuning: Principles, Experiments, and Troubleshooting Techniques
Shasha and Bonnet (Morgan Kaufmann, 2002)

Appendix B shows how to test whether a set of transactions is serializable (doesn't require explicit locking) when run under snapshot isolation