Concurrency & PostgreSQL

Marko Tiikkaja

The European PostgreSQL Day 2010

SQL is easy

Not Easy!

SQL is not easy

- Multi-threaded
- MVCC
- Different isolation levels

Basics

What are snapshots?

A `snapshot` is the state of a system at a particular point in time.

Hard to do:

- Ignore changes of `aborted` transactions
- Ignore changes of `uncommitted` transactions
- Performance

What is MVCC?

`MVCC` stands for Multiversion Concurrency Control

- Multiple row versions
- No read locks!
- Consistent view of the data
- Readers don't block writers and vice versa

Two Levels of Isolation

There are two different `isolation levels`:

- `READ COMMITTED`: new snapshot for every query.
- `SERIALIZABLE`: single snapshot for the entire transaction.

The default can be controlled using `default_transaction_isolation`, which defaults to `READ COMMITTED`.

Locks

`Locks` are used for synchronization between sessions.

You can lock different objects:

- Tables
- Rows
- Advisory locks

With the exception of advisory locks, locks are released at the end of the transaction.

Table Locks

Rows and advisory locks can be locked in either `SHARED` or `EXCLUSIVE` mode.

Table locks have a finer granularity:

Requested Lock Mode	Current Lock Mode							
	ACCESS SHARE	ROW SHARE	ROW EXCL	SH UPD EXCL	SHARE	SH ROW EXCL	EXCL	ACCESS EXCL
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCL					X	X	X	X
SH UPD EXCL				X	X	X	X	X
SHARE			X	X		X	X	X
SH ROW EXCL			X	X	X	X	X	X
EXCL		X	X	X	X	X	X	X
ACCESS EXCL	X	X	X	X	X	X	X	X

Deadlocks

A deadlock is a situation where two are each waiting for each other to release a resource, or more than two processes are waiting for resources in a circular chain.

To avoid deadlocks:

- Lock objects in the same order in every transaction.
- Take the most restrictive lock first.

Order Matters

Lock objects in the same order in every transaction.

```
BEGIN;                                  BEGIN;
LOCK TABLE foo;                         LOCK TABLE bar;

LOCK TABLE bar;  -- waits

                                        LOCK TABLE foo;
```

Order Matters

ERROR: deadlock detected
DETAIL: Process 9509 waits for AccessExclusiveLock on relation 31235 of database 16386; blocked by process 9504. Process 9504 waits for AccessExclusiveLock on relation 28674 of database 16386; blocked by process 9509.
HINT: See server log for query details.

Order Matters

Take the most restrictive lock first.

```
BEGIN;                                  BEGIN;
SELECT * FROM foo                       SELECT * FROM foo
  FOR SHARE;                              FOR SHARE;

-- both succeed

SELECT * FROM foo
  FOR UPDATE;          -- waits

                                        SELECT * FROM foo
                                          FOR UPDATE;
```

Order Matters

ERROR: deadlock detected
DETAIL: Process 9521 waits for ExclusiveLock on tuple (0,1) of relation 31235 of database 16386; blocked by process 9519. Process 9519 waits for ShareLock on transaction 14468334; blocked by process 9521.
HINT: See server log for query details.

SELECT and DML behaviour

SELECT

`SELECT`s only see data visible to their snapshot.

.. except when `FOR SHARE` or `FOR UPDATE` is present.

SELECT

Consider the following:

```
CREATE TABLE foo(a int);
INSERT INTO foo VALUES(0);
```

```
BEGIN;
UPDATE foo SET
   a = 1;
```

```
BEGIN;
```

```
SELECT a FROM foo;
-- sees a=0
```

```
SELECT a FROM foo
   FOR UPDATE;
-- waits
```

```
COMMIT;
```

```
-- and now sees a=1
```

But when we're in `SERIALIZABLE` isolation:

```
BEGIN;
UPDATE foo SET
  a = 1;
```

```
BEGIN SERIALIZABLE;

SELECT a FROM foo;
-- sees a=0

SELECT a FROM foo
  FOR UPDATE;
-- waits
```

```
COMMIT;
```

ERROR: could not serialize access due to concurrent update

However, the `WHERE` clause works differently:

```
BEGIN;
UPDATE foo SET
   a = 1;
```

```
BEGIN SERIALIZABLE;


SELECT a FROM foo;
-- sees a=0

SELECT a FROM foo
   WHERE a = 1
   FOR UPDATE;
-- does NOT wait
```

```
COMMIT;
```

```
SELECT a FROM foo
   WHERE a = 1
   FOR UPDATE;
-- does not wait or
-- see the row
```

.. with a small exception:

```
BEGIN;
UPDATE foo SET
   a = 1;
```

```
BEGIN;

SELECT a FROM foo
   WHERE a = 0;
-- sees a=0

SELECT a FROM foo
   WHERE a = 0
   FOR UPDATE;
-- waits
```

```
COMMIT;
```

```
-- does not see the row!
```

In `SERIALIZABLE`, this results in a serialization error.

Keep in mind that only direct references see the latest version:

```
BEGIN;
UPDATE foo SET
   a = 1;



COMMIT;
```

```
BEGIN;


SELECT a FROM foo;
-- sees a=0

SELECT (SELECT a FROM foo)
  FROM foo FOR UPDATE;
-- waits


-- also sees a=0
```

UPDATE and DELETE

`UPDATE` and `DELETE` work very similarly to `SELECT .. FOR UPDATE`.

- Only see data visible to their `snapshot`
- See the latest versions of rows
- `Serialization errors` in `SERIALIZABLE`
- Same behaviour for `WHERE` clauses
- Same behaviour in the `scalar subquery` case

UPDATE and DELETE

`UPDATE` and `DELETE` work very similarly to `SELECT .. FOR UPDATE`.

- Only see data visible to their `snapshot`
- See the latest versions of rows
- `Serialization errors` in `SERIALIZABLE`
- Same behaviour for `WHERE` clauses
- Same behaviour in the `scalar subquery` case

However, two of these do not apply when the target is a `VIEW`.

Views Are Dangerous

Consider:

```
CREATE TABLE bar(id serial, a int);
INSERT INTO bar VALUES(DEFAULT, 0);

CREATE VIEW foo AS SELECT * FROM bar;

CREATE RULE foo_update_rule AS
  ON UPDATE TO foo DO INSTEAD
    UPDATE bar SET a = NEW.a
      WHERE id = OLD.id;

CREATE RULE foo_delete_rule AS
  ON DELETE TO foo DO INSTEAD
    DELETE FROM bar
      WHERE id = OLD.id;
```

Views Are Dangerous

```
BEGIN;
UPDATE foo SET
   a = a + 1;
```

```
BEGIN;
```

```
UPDATE foo SET
   a = a + 1;
-- waits
```

```
COMMIT;
```

```
-- wakes up
COMMIT;
```

Uh-oh! foo.a is now 1, not 2.

Views Are Dangerous

And the issue with the `WHERE` clause:

```
BEGIN;
UPDATE foo SET
  a = 1;
```

```
BEGIN;

DELETE FROM bar
  WHERE a = 0;
-- waits
```

```
COMMIT;
```

```
-- wakes up and deletes
-- the row!
COMMIT;
```

The row we removed did not match the `WHERE` clause.

Views Are Dangerous

Fortunately, both problems can be solved by using the `SERIALIZABLE` isolation.

Unfortunately, it has its problems too:

- Performance degradation because of transaction retries
- Can't be done transparently in a server-side function
- False positives

All in all, not a very good solution.

Constraints

Why Should You Use Constraints?

Why should you use constraints?

- Efficiency
- Correctness
- Ease of use

UNIQUE And PRIMARY KEY

`UNIQUE` (a,b,c) specifies that the combination of (a,b,c) must be unique across the whole table.

`PRIMARY KEY` just means `UNIQUE + NOT NULL`.

Another way to think about it is:

```
SELECT count(*) FROM tbl
  WHERE a = NEW.a AND b = NEW.b AND c = NEW.c;
```

must return 0 for the `INSERT` to succeed.

UNIQUE And PRIMARY KEY

But that's the easy part; the bigger problem is that it also must remain that way for the remainder of the transaction (ignoring, of course, the tuple we `INSERTed`).

Do not do this:

```
BEGIN;

SELECT count(*) FROM tbl
  WHERE a = NEW.a AND b = NEW.b AND c = NEW.c;

-- if there were no rows:
INSERT INTO tbl VALUES (NEW.a, NEW.b, NEW.b);

COMMIT;
```

Unless you have a UNIQUE constraint in place.

Exclusion Constraints

`Exclusion constraints` are a generalization of that idea; using operators other than = can be useful:

```
CREATE TABLE circles
(
    a int,
    b circle,
    EXCLUDE USING gist (a WITH =, b WITH &&)
);

SELECT count(*) FROM circles
  WHERE a = NEW.a AND b && NEW.b;
```

must return 0 for the `INSERT` to succeed.

FOREIGN KEY Constraints

`FOREIGN KEY` constraints are different:

```
FOREIGN KEY (a,b,c) REFERENCES other_tbl(a,b,c)
```

```
SELECT count(*) FROM other_tbl
  WHERE a = NEW.a AND b = NEW.b AND c = NEW.c;
```

must return at least one row for the `INSERT` to succeed.

The same problem occurs: that fact can not change before the transaction commits.

FOREIGN KEY Constraints

- Implemented using `row-level locks` (i.e. `SELECT .. FOR SHARE`) and `AFTER EACH ROW` triggers
- User-space implementations are possible in `READ COMMITTED` already, and possibly all `isolation levels` in 9.1

FOREIGN KEY Constraints

Assume the following schema:

```
CREATE TABLE products
(
    name text PRIMARY KEY,
    unsold int
);

CREATE TABLE orders
(
    product text REFERENCES products
);
```

FOREIGN KEY Constraints

```
BEGIN;

INSERT INTO orders
  VALUES ('car');

UPDATE products
  SET unsold = unsold - 1
  WHERE name = 'car';

-- danger!

COMMIT;
```

How do we fix this?

UPDATE first:

```
BEGIN;

UPDATE products
  SET unsold = unsold - 1
  WHERE name = 'car';

INSERT INTO orders
  VALUES ('car');

COMMIT;
```

Lock the row yourself:

```
BEGIN;

SELECT 1 FROM products
  WHERE name = 'car'
  FOR UPDATE;

INSERT INTO orders
  VALUES ('car');

UPDATE products
  SET unsold = unsold - 1
  WHERE name = 'car';

-- safe!

COMMIT;
```

Summary

Summary

- Be careful with `VIEWs`
- Pay attention to `isolation levels`
- Use the built-in `constraints`
- Be aware of implicit locking
- Be wary of if (SELECT ..)
- Think at least twice before using `INSERT/UPDATE/DELETE .. WHERE NOT EXISTS` and its variants
- Do not expect `RULEs` to be a good idea

SELECT * FROM questions;

Thank you!

Remember to give feedback:
http://2010.pgday.eu/feedback

marko.tiikkaja@cs.helsinki.fi