# Postgres Locking

By: Chun Li

Postgres is pretty good at supporting concurrent transactions, and being ACID compliant. However, it still needs to use locks to properly function sometimes.

In general, transactions will run concurrently until encountering a lock, at which point it will either acquire it, or queue in line for the lock.

# Table Locks

There is a table wide lock, mostly important for DDL changes.

| Runs concurrently with | SELECT | INSERT UPDATE DELETE | CREATE INDEX CONC VACUUM ANALYZE | CREATE INDEX | CREATE TRIGGER |
|---:|---|---|---|---|---|
| SELECT | | | | | |
| INSERT UPDATE DELETE | | | | | |
| CREATE INDEX CONC VACUUM ANALYZE | | | | | |

| | | | | | |
|---|---|---|---|---|---|
| CREATE INDEX | | | | | |
| CREATE TRIGGER | | | | | |
| ALTER TABLE DROP TABLE TRUNCATE VACUUM FULL | | | | | |

# Row Locks

There is also a lock for each row, which maintains Postgres’ consistency. There are two types for each row, `share` and `exclusive`. Multiple transactions can hold onto a `share` lock at a time, but only one can hold onto an `exclusive` lock at a time.

| **Can access row concurrently with:** | **SELECT** | **SELECT FOR SHARE** | **SELECT FOR UPDATE** | **UPDATE DELETE** |
|---|---|---|---|---|
| SELECT | | | | |
| SELECT FOR SHARE | | | | |
| SELECT FOR UPDATE | | | | |
| UPDATE | | | | |

For completeness, there are also Page-level locks and Advisory-level (application) locks, but they are usually not important.

# Diagnostics

`pg_locks` shows what locks are granted and what processes are waiting for locks to be acquired.

Who is waiting on a lock?

```
SELECT relation::regclass, * FROM pg_locks WHERE NOT GRANTED;
```

```
SELECT a.datname,
         c.relname,
         l.transactionid,
         l.mode,
         l.GRANTED,
         a.usename,
         a.query,
         a.query_start,
         age(now(), a.query_start) AS "age",
         a.pid
    FROM  pg_stat_activity a
     JOIN pg_locks         l ON l.pid = a.pid
     JOIN pg_class         c ON c.oid = l.relation
    ORDER BY a.query_start;
```

You can `SET application_name='%your_logical_name%'` at the beginning of transactions to make tracking the transactions easier. This can then show what is blocking row-level locks:

```
SELECT blocked_locks.pid     AS blocked_pid,
         blocked_activity.usename  AS blocked_user,
         blocking_locks.pid     AS blocking_pid,
         blocking_activity.usename AS blocking_user,
         blocked_activity.query    AS blocked_statement,
         blocking_activity.query   AS current_statement_in_blocking_process,
         blocked_activity.application_name AS blocked_application,
         blocking_activity.application_name AS blocking_application
   FROM  pg_catalog.pg_locks         blocked_locks
    JOIN pg_catalog.pg_stat_activity blocked_activity  ON blocked_activity.pid = blocked_locks.pid
    JOIN pg_catalog.pg_locks         blocking_locks
        ON blocking_locks.locktype = blocked_locks.locktype
        AND blocking_locks.DATABASE IS NOT DISTINCT FROM blocked_locks.DATABASE
        AND blocking_locks.relation IS NOT DISTINCT FROM blocked_locks.relation
        AND blocking_locks.page IS NOT DISTINCT FROM blocked_locks.page
        AND blocking_locks.tuple IS NOT DISTINCT FROM blocked
```

```
_locks.tuple
        AND blocking_locks.virtualxid IS NOT DISTINCT FROM bl
ocked_locks.virtualxid
        AND blocking_locks.transactionid IS NOT DISTINCT FROM
 blocked_locks.transactionid
        AND blocking_locks.classid IS NOT DISTINCT FROM block
ed_locks.classid
        AND blocking_locks.objid IS NOT DISTINCT FROM blocked
_locks.objid
        AND blocking_locks.objsubid IS NOT DISTINCT FROM bloc
ked_locks.objsubid
        AND blocking_locks.pid != blocked_locks.pid
    JOIN pg_catalog.pg_stat_activity blocking_activity ON blo
cking_activity.pid = blocking_locks.pid
   WHERE NOT blocked_locks.GRANTED;
```

## Logging

Also, you may set `log_lock_waits (boolean)` to log a message whenever a session takes longer than `deadlock_timeout` to acquire a lock.

The setting `deadlock_timeout` determines how long to wait before checking for a potential deadlock (which is expensive).

# Some examples

Or, *How one guy found 7 easy ways to make your Postgres faster!!!*

*(Engineers hate him!!)*

## 1. Don’t specify default values when adding columns

When specifying a default value for a large table, Postgres needs to write it in for every row, locking the whole table. Instead, you should create the column and add in the default value afterwards.

BAD:

```
ALTER TABLE friends ADD COLUMN salary int DEFAULT 100000;
```

BETTER:

```
ALTER TABLE friends ADD COLUMN salary; -- pretty fast
-- now update
UPDATE friends SET salary = 100000;
```

EVEN BETTER:

Batch it, for example:

```
do {
   rowsUpdated = sql.execute("UPDATE friends SET salary=10000
0" +
   " WHERE id IN " +
```

```
    "(SELECT id FROM friends WHERE salary IS NULL LIMIT ?)", batchSize)
} while(rowsUpdated > 0);
```

<!-- this is also why we batch big table updates like Karen’s from last night -->

## 2. Beware of lock queues, use lock timeouts

When a transaction attempts to acquire a lock, it must check for conflicting locks for *every transaction in the lock queue*. This means that if you must hold an exclusive lock, like with `ALTER TABLE`, any transactions that come after will also block on it, **and** `ALTER TABLE` will block until the table is free.

For example, let’s say there is a long running `SELECT` statement called transaction `A`. If you run an `ALTER TABLE` in transaction `B` at this point, it will block on `A`, but also any other transaction `C` that comes after will block, even if it wouldn’t block on the currently running transaction `A`.

One solution is to set `lock_timeout` as a safeguard like so:

```
SET lock_timeout TO '2s'
ALTER TABLE friends ADD COLUMN salary int;
```

This way, any DDL command you run can only block queries for up to 2s, and you can try again later if it fails. You can always query `pg_stat_activity` to see if there are any current long running transactions.

## 3. Create indexes `CONCURRENTLY`

For more information, see [https://www.postgresql.org/docs/9.5/static/sql-createindex.html#SQL-CREATEINDEX-CONCURRENTLY](https://www.postgresql.org/docs/9.5/static/sql-createindex.html#SQL-CREATEINDEX-CONCURRENTLY)

`CONCURRENTLY` allows you to create indexes without blocking writes on the table-level.

:information_source: Postgres supports this operation by scanning the table twice, which have to wait for existing transactions to terminate. `CONCURRENT` index creation is more expensive overall. Also, because Postgres creates the index in the system catalogs first, and then does the scan if the index creation fails due to deadlock or some constraints violation, there will be an “invalid” index left behind, which should be `DROPPED`. *#themoreyouknow* :rainbow:

## 4. Take aggressive locks as late as possible

A transaction only releases its locks at the end of the transaction. Thus, to allow as many concurrent transactions to run as possible, take aggressive locks as late as possible.

For example, if you are trying to replace the contents of a tables, you could do this:

BAD:

```
BEGIN;
-- reads and writes blocked from here:
TRUNCATE friends;
-- long-running operation:
\COPY friends FROM 'betterfriends.csv' WITH CSV
COMMIT;
```

BETTER:

```
BEGIN;
CREATE TABLE friends_new (LIKE friends INCLUDING ALL);
-- long-running operation:
\COPY friends_new FROM 'betterfriends.csv' WITH CSV
-- reads and writes blocked from here:
DROP TABLE friends;
ALTER TABLE friends_new RENAME TO friends;
COMMIT;
```

One concern is that since we didn't block writes from the beginning, any new friends that were added during the replacement will be gone forever. One solution is to block writes (but not reads) on the table at the beginning:

ALSO BETTER:

```
BEGIN;
LOCK friends IN EXCLUSIVE MODE;
-- continue as before
CREATE TABLE friends_new (LIKE friends INCLUDING ALL);
-- long-running operation:
\COPY friends_new FROM 'betterfriends.csv' WITH CSV
-- reads and writes blocked from here:
DROP TABLE friends;
ALTER TABLE friends_new RENAME TO friends;
COMMIT;
```

## 5. Adding a primary key with minimal locking

Adding a primary key using `ALTER TABLE` creates an index and sets that index as the primary key. However, since creating an index can be done with minimal locking (allowing writes), we can split this into two steps:

Instead of `ALTER TABLE friends ADD PRIMARY KEY (id);`
Do (in 2 transactions):

```
CREATE UNIQUE INDEX CONCURRENTLY friends_pk ON friends (id);
-- takes a long time if table is large
```

```
ALTER TABLE friends ADD CONSTRAINT friends_pk PRIMARY KEY USING INDEX friends_pk; -- blocks all queries, but only briefly
```

## 6. Never VACUUM FULL

- Simply put, `VACUUM FULL == PLEASE FREEZE MY DATABASE FOR HOURS`
- Rewrites whole table and puts a full lock on it
- The main reason why messing with `AUTOVACUUM` is in general, a bad idea

## 7. Avoid deadlocks by ordering commands

## Deadlock example:

Transaction 1:

```
BEGIN;
SELECT * FROM friends WHERE first_name = 'Dennis' AND last_name = 'Merino';
SELECT * FROM friends WHERE first_name = 'Michelle' AND last_name = 'Wong';
END;
```

Transaction 2:

```
BEGIN;
SELECT * FROM friends WHERE first_name = 'Michelle' AND last_name = 'Wong';
SELECT * FROM friends WHERE first_name = 'Dennis' AND last_name = 'Merino';
END;
```

Potentially will deadlock.

One solution to deadlocks is to try to always acquire your resources in a predetermined order (See `CS 343` or the UofT equivilent course)

# References

https://www.postgresql.org/docs/current/static/explicit-locking.html

https://wiki.postgresql.org/wiki/Lock_Monitoring

https://www.citusdata.com/blog/2018/02/15/when-postgresql-blocks/

https://www.citusdata.com/blog/2018/02/22/seven-tips-for-dealing-with-postgres-locks/

https://www.postgresql.org/docs/9.5/static/sql-createindex.html#SQL-CREATEINDEX-CONCURRENTLY

https://www.postgresql.org/docs/9.2/static/monitoring-stats.html#PG-STAT-ACTIVITY-VIEW