Slow things down to make them go faster

Jimmy Angelakos
Senior PostgreSQL Architect

FOSDEM 2022-02-06
We’ll be looking at PostgreSQL:

- High concurrency
- ACID & MVCC
- Locks
- High transaction rates
- Mitigation strategies
High concurrency in PostgreSQL
What is high concurrency?

- RDBMS context: The ability to have transactions execute concurrently, not serially
- Practically: Serve multiple sessions/users “simultaneously”
  - How to avoid conflicts? (dirty reads, lost updates, etc.)
  - Made possible via concurrency control methods
- Postgres designed to be able to provide high concurrency safely
  - Hundreds of activities at the same time
PostgreSQL is multi-process

CLIENT/SERVER IMPLEMENTATION

- “Process per user” model
- Every client process connects to exactly one backend process
- Co-ordinated by postmaster supervisor process
- IPC via semaphores & shared memory
- Risk: CPU context switching
The I in ACID

Atomicity, Consistency, **Isolation**, Durability

- Isolation: how transaction integrity is visible by other sessions
- Anomalies: dirty/non-repeatable/phantom read, lost update, write/read-only transaction skew
- Lower isolation level: more sessions can access same data (at some risk)
- Higher isolation level: safer but increases resource usage & blocking
- Default PG isolation level: **READ COMMITTED**, highest isolation level: **SERIALIZABLE**
  - Each query sees only transactions committed before it started
MVCC

Multi-Version Concurrency Control

- MVCC rather than locking for high concurrency and high performance
- Reading never waits
  - Writing doesn’t block reading, reading doesn’t block writing
- Each write creates a new version of tuple
- Snapshot isolation: Timestamps & Transaction IDs (XIDs)
Transaction Snapshot

Provided by Transaction Manager

• Obtained by each transaction
• Contents:
  – Earliest transaction still active
  – First as-yet-unassigned transaction
  – List of active transactions
• Function: `pg_current_snapshot()`
SSI – Serializable Snapshot Isolation

The performance of MVCC with the safety of Serializable

• Checks for anti-dependency cycles and forbids them
  – Error instead of hazardous operation (serialization anomaly)

• Performance
  – Reduced concurrency, but:
  – No blocking, no explicit locks needed (SIReadLocks, rw-conflicts)
  – Just application-side retry after error
  – Best performance choice for some application types
Locks in PostgreSQL
Explicit locking

a.k.a. heavyweight locks – not what we’re talking about here

• Table-level (e.g. **SHARE**) or row-level (e.g. **FOR UPDATE**)
• Conflict with other lock modes (e.g. **ACCESS EXCLUSIVE** with **ROW EXCLUSIVE**)
• Block read/write access totally leading to waits
• Disastrous for performance
  - Unless your application is exquisitely crafted
  - Hint: it isn’t
Lightweight Locks (LWLocks) – i

a.k.a. “latches” in other DBs

• Protect data in shared memory
  – Remember? Multi-process
  – Ensure consistent reads/writes
  – Shared, Exclusive modes

• Enable fast MVCC
  – Generally held briefly
  – Sometimes protect I/O
Lightweight Locks (LWLocks) – ii

Under high concurrency

• Possible problem: a lock becomes heavily contended
  – Lots of lockers slow each other down
  – Throughput is reduced
  – No queuing, more or less random
  – May indicate existence of hot data

• Monitoring: `pg_stat_activity` (look for `wait_event_type: LWLock`)
Snapshot contention

Waiting for connections that are idle!

• Extremely high `max_connections` settings allow this
• Many idle open connections
  − Means: many snapshots
  − Can halve your performance in TPS
  − Even with simple R/O workload
• Improvement in PG14: snapshot caching (transaction completion counter)
Many connections...

PG13 Tests on AWS r5.8xlarge with `pgbench -j10 -C -c<clients> -T120 -b simple-update pgbenchdb`

- 100 connections
  - tps = 1560.134, latency average = 52.162 ms
- 300 connections
  - tps = 1307.431, latency average = 190.652 ms
- 1000 connections
  - tps = 1184.786, latency average = 668.470 ms
High transaction rates
Transaction ID

a.k.a. txid

• Postgres assigns an identifier to each transaction
  − Unsigned 32-bit integer (4.2B values)

• Circular space, with a visibility horizon
  − In transaction 10000: 9999 is the past (visible), 10001 the future (invisible)
  − 2.1B transactions into the past, 2.1B transactions in the future

• Basis for MVCC mechanism – just write into heap, each tuple has $\text{xmin, xmax}$

• Amazing write/rollback performance BUT requires maintenance operations
High transaction burn rate

Just because you can, doesn’t mean you should

• Very heavy OLTP workloads can go through 2.1B transactions in a short time

• **XID wraparound:** you try to read a very old tuple that is > 2.1B XIDs in the past
  
  – For you, that’s the future! (invisible)

• **Freezing:** Change tuple **xmin** to “frozen” txid 2 which is known to always be in the past

• Need to make sure **FREEZE** happens before XID wraparound

• Bloat

• Aborted transaction IDs remain
(Auto)VACUUM

The MVCC maintenance operation

- Removes dead tuples, freezes tuples
  - Among other things
- Has overhead
  - Scans tables & indices
  - Needs, obtains, and waits for locks
- Has limited capacity by default
Mitigation strategies
Lock contention

Waiting for explicit locks

• Avoid explicit locking!
• Use SSI (*SERIALIZABLE* isolation level)
• Make application tolerant
  – Allow it to fail and retry
LWLock contention

Too many connections to server

- Contention often caused by too much concurrency
- Insert a connection pooler between application and DB
- Allow fewer connections into the DB
- Make the rest queue for their turn
- Sounds counter-intuitive!
Connection pooling

PGBouncer is a pretty good solution

• “Throttle” application by reducing no. of connections reaching server
  - Leave max_client_conn to what app wants, only allow max_db_connections
  - Introduce latency on the application side, to save your server performance
  - Doesn’t necessarily slow anything down – queries may execute faster!

• transaction pool mode: PGBouncer reuses connection for user when transaction ends

• statement pool mode: PGBouncer reuses connection for next statement
  - No transaction control, for autocommit-type workloads
PgBouncer effect

Real world use case

• Misbehaving application with job parallelization, leaving jobs’ connections open

• `max_connections` set to 5000, 2500+ open connections observed

• After PgBouncer in Transaction mode:
  - 30 connections used in Postgres (!)
  - Queries executed much faster

• Other solutions: Application side, Pgpool-II, Odyssey
Split your workload

With streaming/logical replication

• Adapt your application: Read Only and Read/Write connections
• Send write operations to primary server
• Send read operations to standby servers
  - Horizontal read scalability
• Set up R/O and R/W PgBouncer endpoints
• Use logical replication if partial dataset required
XID wraparound

• Can batching help?
  - Batch size 1000 will have 1/1000th the burn rate
• Increase effectiveness of autovacuum
  - More efficient FREEZE
Autovacuum

Make it work harder to avoid problems

• People are concerned about overhead
  - Alternative is worse! You can’t avoid VACUUM in Postgres (yet).
  - You can outrun it (and then you’ll need VACUUM FULL)

• Increase potency via:
  - maintenance_work_mem (1GB is good)
  - autovacuum_max_workers
  - autovacuum_vacuum_cost_delay / autovacuum_vacuum_cost_limit
Monitoring tools

Detect contention, wait events

- psql 😊
- UNIX tools: ps, top, iostat, vmstat
- pgAdmin
- pg_view, pgstats, pgmetrics, …
- check_postgres, check_pgactivity
- Proprietary
Disclaimer

Every workload is different!

• OLTP
  - R/W, shorter queries, high contention, sustained rate, fewer idle connections (?)

• Web server
  - R/O bias, shorter queries, less contention, more idle connections (?)

• Spark/batch/analytics
  - R/W, longer queries, high contention, more idle connections (?)
To conclude...

• Know your workload & application behaviour!
• Monitor for signs of high contention
• Not overwhelming Postgres is the key
• Split your workload
• Use connection pooling & autovacuum
Thank you!

Find me on Twitter: @vyruss