Extreme PostgreSQL

Christophe Pettus, CEO, PGX Inc.

PGX
pgexperts.com

How extreme can PostgreSQL get?

- PostgreSQL has almost no hard limits on anything.
  - Number of databases: 4,294,950,911.
  - Relations per database: 1,431,650,303.
  - Relation size: 32TB (bigger if you compile with a larger block size).

But how about speed?

- GitLab has sustained 360,000 transactions per second.
- We’ve seen individual servers at >500,000 INSERTs per second.

How about lots of tables?

- Schema-based multi-tenancy has introduced a new world of big schemas.
- 250,000+ tables are not uncommon.

BUT HOW?

This is how!

- Very large databases (with lots of read activity).
- Databases with very high write rates.
- Databases with very large schemas.

BIG DATA.

How big is big?

- A terabyte here, a terabyte there, and soon you’re talking about real data.
- Our view is:
  - Under 100GB is small.
  - 100GB - 4TB is medium-sized.
  - 4TB - 20TB is large-ish.
  - Above 20TB, now you’re talking!
  - Above a petabyte? OK, that’s a *real* database.

You can’t be too rich, or…

- … have too much RAM.
- Databases of this scale will not fit into memory.
  - Although you can get a 4TB RDS instance!
- Get as much RAM as you can possibly afford. You’ll need all of it.
- More cores are nice, but max out RAM before spending money on cores.
  - Of course, on certain cloud providers, you don’t get a choice there.

Partition early and often.

- Partitioned tables are your friend.
- Recent (14+) versions of PostgreSQL handle large (1000-ish) number of partitions reasonably well.
- Be sure you have a good partition key!
- This will take you into the “big schema” territory, so see that section too.
- Also helps with retention policies for time-based data.

Enjoy the parallelism.

- Very large tables and indexes benefit (a lot) from parallel operations.
- Be sure to give the system enough parallel workers.
  - max_worker_processes = cores * 4
  - max_parallel_workers = cores * 3
  - max_parallel_workers_per_gather = cores
- As of version 14, PostgreSQL can scan multiple partitions in parallel!

Partial Indexes.

- Make use of partial indexes to “precalculate” queries.
- Be sure the predicate reduces the index size by 50%+.
- Be sure the indexes are actually being used (pg_stat_user_indexes).

BRIN indexes.

- If you have time series or other data with monotonically increasing keys.
- BRIN indexes are much smaller than B-tree indexes (even with the new B-tree optimizations).
- BRIN indexes + time-based partitions is a great combination.
- Data that is heavily updated can break this, though.
  - Consider setting a low fill_factor to help with updates landing on the same page.

Autovacuum.

- Big tables can take a long time to scan.
- Reduce autovacuum_vacuum_cost_delay (even to 0) to get autovacuum to work faster.
- More workers don’t help for big tables (but they do help for bigger schemas).

Shared Buffers.

- Be generous with shared_buffers in a database like this.
- … assuming you don’t need the memory for big sort etc. operations’ work_mem.
- 40-50% of total system memory is not unreasonable here.

Planner Settings.

- If you have joins with a large number of tables, increase join_collapse_limit (be aware that plan time will go up).
- Push the planner away from sequential scans:
  - seq_page_cost = 0.1
  - random_page_cost = 0.1
  - cpu_tuple_cost = 0.03
- Make sure that effective_cache_size is big enough (really, you can lie to the planner here, the planner police won't arrest you).

Backups.

- Consider doing incremental disk snapshots instead of copy-based backups.
  - This will also help keep the restore time under control.
- Or use a tool that does block-level incremental backups.
  - pgBackRest
- Even though expensive, you definitely need a secondary or two.
  - Restores from backup will be slow no matter what system you use, and you don’t want to be down for days while they are happening.

Xtra for Xperts.

- Recompile PostgreSQL with a larger block size.
- Can reduce the overhead of large shared_buffers, make more efficient use of I/O operations.
- 32KB is an attractive point.
- **Warning:** This may break some tools that assume an 8KB block size without checking.
  - If you are the author of one of those tools, please fix it.

Cache Everything That Moves.

- Roll-up tables.
- Front-end query caches.
- Application-specific caches.
- Try to hit the database as little as possible.

BIG WRITES.

Step #1: Fast I/O.

- You can’t write to secondary storage faster than secondary storage can be written to.
- NVMe local storage is the way to go here.
  - RAID-0 striped!
  - Be sure you have a good recovery plan (see later for that).
- Move the pg_wal directory onto a separate volume.
- ~~NAS~~

sysctl.conf

- The default configuration on most Linux distros isn’t great for very fast I/O.
- You want the file system cache flush to happen early and often.
    - vm.dirty_background_ratio = 5
    - vm.dirty_ratio = 90 # The usual recommendation on this is wrong.
    - vm.swappiness = 1

Bandwidth is not the whole story.

- Every storage vendor loves to talk about their incredibly high bandwidth.
- No storage vendor loves to talk about their incredibly bad latency.
- Especially on the WAL, latency is very very important.
  - Especially latency after a sync.
- NAS storage (which includes EBS) tends to have bad latency characteristics.

Schema design, or, no you don’t need that index.

- Minimize indexes.
- Minimize unique indexes especially.
  - Try to have uniqueness guaranteed at generation time.
- Don’t have foreign key constraints unless you ***really*** need them.
- Don’t use highly-random primary keys.
  - No UUIDs, unless they are sequential in the high bits.
- Don’t even think about having GIN indexes.

More schema thoughts.

- Avoid large datatypes.
  - Large being >2,000 bytes-ish.
  - Writing to the TOAST table significantly slows down write performance.
- Avoid triggers firing on writes.
- fillfactor = 100 unless you are also going to be updating rows frequently.

Shared Buffers and Checkpoints.

- The following applies to nearly append-only databases.
- Unlearn everything you’ve been taught about setting these.
- Reduce shared_buffers (10% is reasonable).
- Checkpoint every five minutes (experiment with even lower).
- Keep max_wal_size high enough so that the timeout fires first.
- For databases that are a mix of read, insert, and update, normal checkpointing (15 minutes) and shared_buffers is a better idea.

Background Writer etc.

- The default background writer configuration is not aggressive.
- bgwriter_delay = 5ms
- bgwriter_lru_maxpages = 1600 # or more!
- bgwriter_lru_multiplier = 3.0
- wal_compression = on

Autovacuum.

- Be prepared to do manual vacuums and analyzes.
  - On high write loads, autovacuum may not keep up by itself.
  - Planner statistics can get out of whack very quickly.
- Do manual VACUUM FREEZE operations to stay way ahead of a wraparound autovacuum.
- If you are updating frequently, you’re going to get index bloat.
  - Plan to rebuild indexes to squeeze it out.

Other settings.

- Increase wal_buffers; 128MB is not a bad place to start.
- synchronous_commit = off unless you *must* have it on.
  - Consider having it off, but turning it on locally for “important” transactions.

Client interactions.

- Use COPY instead of INSERT if you can.
- Use multi-valued INSERT statements.
- Use a prepared INSERT statement.
- Combine multiple INSERTs into a single transaction.
- And do not combine a high write workload with a read workload!
- Do not use subtransactions no matter how much they beg you.

Replicas.

- Recovery is single-threaded.
- Writing to the database is not.
- pgprefaulter can help speed up recovery; by all means use it:
  - https://github.com/TritonDataCenter/pg_prefaulter
- Consider using pgBackRest or another backup tool that can parallelize archive_command.

If we can’t do reads, how do we use this data?

- Move read load to replicas.
  - Understanding that there could be significant replication lag.
- If the write load is bursty, consider logical replication.
  - Replica can have indexes, triggers, summary tables, all sorts of good things.
  - It won’t be able to keep up with a very high continuous write load, though.
- If you can work with hours-old data, take snapshots periodically and bring them on-line.

Extend locks.

- Extend locks can be painful in a high-write situation.
  - PostgreSQL’s adaptive algorithm is better than nothing, but not perfect.
  - Version 16 has significant improvements here, so upgrade.
- Use a file system where extending a file is (relatively) fast.
- In our testing:
  - XFS > EXT4 > ZFS > anything NAS-based.

BIG SCHEMAS.

How big?

- Under 1,000 tables is pretty normal.
- 1,000 - 10,000… “Getting big.”
- 10,000 - 100,000… “Getting bigger!”
- 100,000+ … “OK, that’s big.”

Schema-based sharding

- Most common reason for large schemas.
- 1,000 tables times 250 tenants in a single database.
- Boom!

Generally, works out of the box!

- Most of the bad PostgreSQL issues with large schemas are in the past.
    - Remember O(N^N) pg_dump behavior? Ow.
- Plan to do upgrades using logical replication rather than pg_upgrade, unless you can take downtime in the hour range.
    - If you are doing tenancy, migrate tenant by tenant.
- Be careful about your monitoring tools! A lot of them do system catalog queries that can really grind to a halt on large schemas.

Settings.

- Many more autovacuum workers (25+).
- Be sure that autovacuum_work_mem is lower, to avoid memory pressure.
- Monitor xid age, and how long it is taking to get to individual tables, carefully.

Query Things.

- Do as much as you can with prepared statements.
- Planning queries system catalogs, and system catalogs get very big here.

File System Things.

- Increase max_files_per_process.
  - Be sure to increase any system-level limits on number of open files.
- Increase max_locks_per_transaction.
  - This is especially true if you are using partitioned tables.
- Use a file system that handles a large number of files gracefully.
  - ZFS > EXT4 > XFS (although the differences are not large).

Databases or schemas?

- Using databases for sharding instead of schemas can reduce the overhead of large system catalogs.
- It defeats pooling and makes the connection model more complicated, though.

WHEN IN DOUBT…

Shard.

- Sharding across multiple PostgreSQL instances helps with all of these scenarios.
  - Less data per node.
  - Distributes write activity.
  - Fewer tables per node.
- Can be application-specific or an automatic-sharding solution (Citus).
- Often easier than trying to wring maximum performance out of a single node.
- Often the only real way to get very very high performance in a DBaaS environment.

THANK YOU!

thebuild.com

QUESTIONS?

PGX

pgexperts.com