Breaking PostgreSQL at Scale.

Christophe Pettus
PostgreSQL Experts
FOSDEM 2019
Christophe Pettus

CEO, PostgreSQL Experts, Inc.

christophe.pettus@pgexperts.com

thebuild.com

twitter @xof
So, what is this?

- PostgreSQL can handle databases of any size.
  - Largest community-PostgreSQL DB I’ve worked on was multiple petabytes.
- But how you handle PostgreSQL changes as databases get larger.
- What works for a 1GB database doesn’t for a 10TB database.
- Let’s talk about that!
Ten Gigabytes.
Your New Database!

• It’s very hard to go wrong with small databases on PostgreSQL.

• Nearly everything will run fast…
  
  • … even “pathological” joins, unless then are fully N^2.

• The stock postgresql.conf will work.
How much memory?

- If you can’t fit your database in memory…
  - … reconsider your life choices.

- Even small “micro” instances can handle a database this size.

- The entire database can probably fit in memory.

- Even sequential scans will zip right along.
Backups.

• Just use pg_dump.

• A 5GB pg_dump takes 90 seconds on my laptop.

• No need for anything more sophisticated.

• Stick the backup files in cloud storage (S3, B2), and you’re done.
High Availability.

- A primary and a secondary.
- Direct streaming, or basic WAL archiving.
Tuning.

- If you insist.
- The usual memory-related parameters.
- A couple of specialized parameters for all-in-memory databases.
- But at this stage, just keep it simple.
Tuning.

seq_page_cost = 0.1
random_page_cost = 0.1
cpu_tuple_cost = 0.03
shared_buffers = 25% of memory
work_mem = 16MB
maintenance_work_mem = 128MB
Tuning.

log_destination = 'csvlog'
logging_collector = on
log_directory = '/var/log/postgresql'
log_filename = 'postgresql-%Y%m%d-%H%M%S.log'
log_rotation_size = 1GB
log_rotation_age = 1d
log_min_duration_statement = 250ms
log_checkpoints = on
log_connections = on
log_disconnections = on
log_lock_waits = on
log_statement = 'ddl'
log_temp_files = 0
log_autovacuum_min_duration = 1000
Upgrades.

• `pg_dump/pg_restore`.

• You’re done.

• But do it!

• The farther you fall behind on major versions, the harder it becomes.

• Get into the habit of planning your upgrade strategy.
100 Gigabytes.
Not huge, but…

• … the database is starting to get bigger than will fit in memory.

• Queries might starting performing poorly.

• pg_dump backups take too long to take or restore.
How much memory?

- How much memory does a PostgreSQL database need?
- If you can fit the whole thing in memory, great.
- Otherwise, try to fit at least the top 1-3 largest indexes.
  - Ideally, effective_cache_size > largest index.
- If not, more memory is always better, but…
- … more memory does not help write performance.
Backups.

- pg_dump won’t cut it anymore.
- Time for PITR backups!
- pgBackRest is the new hotness.
- WAL-E is the old warhorse.
- Can roll your own (if you must).
PITAR

- Takes an entire filesystem copy, plus WAL archiving.
- More frequent filesystem copies means faster restore…
- … at the cost of doing the large copy.
- Other benefits: Can restore to a point in time, can use backup to prime secondary instances.
Tuning.

seq_page_cost = 0.5-1.0
random_page_cost = 0.5-2.0
shared_buffers = 25% of memory
maintenance_work_mem = 512MB-2GB
work_mem

• Base work_mem on actual temporary files being created in the logs.

• Set to 2-3x the largest temporary file.

• If those are huge? Ideally, fix the query that is creating them.

• If you can’t, accept it for low-frequency queries, or…

• … start thinking about more memory.
Load balancing.

- Consider moving read traffic to streaming secondaries.
- Be aware that replication lag is non-zero.
- Handle the traffic balancing in the app if you can.
- If you can’t, pgpool is there for you (although it’s quirky).
Monitoring.

• Time for real monitoring!

• At a minimum, process logs through pgbadger.

• `pg_stat_statements` is very valuable.
  • `pganalyze` is a handy external tool.

• New Relic, Datadog, etc., etc. all have PostgreSQL plugins.
Queries.

• Check pgbadger / pg_stat_statements regularly for slower queries.

• Missing indexes will start becoming very apparent here.

• Create as required, but…

• … don’t just start slapping indexes on everything!

• Base index creation on specific query needs.
High Availability.

• Probably don’t want to fix it manually anymore.

• Look at tooling for failover:
  
  • pgpool2
  
  • Patroni
  
  • Hosted solutions (Amazon RDS, etc.)
Upgrades.

• pgupgrade.

• In-place, low downtime.

• Very reliable and well-tested.

• Some extensions are not a comfortable fit, especially for large major version jumps.

• We’re looking at you, PostGIS.
One Terabyte.
Things Get Real.

- Just can’t get enough memory anymore.
- Queries are starting to fall apart more regularly.
- Running out of read capacity.
- Doing full PITR backups is taking too long.
Resources

- As much memory as you can afford.
- Data warehouses need much more than transactional databases.
- I/O throughput becomes much more important.
- Consider moving to fast local storage from slower SAN-based solutions (such as EBS, etc.).
Backups

- Start doing incremental backups.
- pgBackRest does them out of the box.
- You can roll your own with rsync, but…
  - … this is very much extra for experts!
Checkpoints/WAL.

min_wal_size = 2GB+
max_wal_size = 8GB+
checkpoint_timeout = 15min
checkpoint_completion_target = 0.9
wal_compression = on
Restrain yourself.

- Keep shared_buffers to 16-32GB.
  - Larger will increase checkpoint activity without much actual performance benefit.
- Don’t go crazy with maintenance_work_mem.
  - If most indexes are larger than 2GB, it is often better to decrease it to 256-512MB.
Load balancing.

- Read replicas become very important.

- Distinguish between the failover candidate (that stays close to the primary) and read replicas (that can accept delays due to queries).

- Have scripted / config-as-code ways of spinning up new secondaries.
Off-Load Services.

- Move analytic queries off of the primary database.
  - Consider creating a logical replica for analytics and data warehousing.
- Move job queues and similar high-update-rate, low-retention-period data items out of the database and into other datastores (Redis, etc.).
VACUUM.

• Vacuum can start taking a long time here.

• Only increase autovacuum_workers if you have a large number of database tables (500+).

• Let vacuum jobs complete!
  
  • Be careful with long-running transactions.

• Consider automated “manual” vacuums for tables that are very high update rate.
VACUUM.

• If autovacuum is taking too long, consider making it more “aggressive” by reducing `autovacuum_vacuum_cost_delay`.

• If autovacuum is causing capacity issues, consider increasing `autovacuum_vacuum_cost_delay`.

• But let autovacuum run! You can get yourself into serious (like, database-shutdown-serious) trouble without it.
Indexes

- Indexes are getting pretty huge now.
- Consider partial indexes for specific queries.
- Analyze which indexes are really being used, and drop those that aren’t necessary (pg_stat_user_indexes is your friend here).
Queries.

• Queries can start becoming problematic here.

• Even the “best” query can take a long time to run against the much larger dataset.

• “Index Scan” queries turning into “Bitmap Index Scan / Bitmap Heap Scan” queries, and taking much longer.
Partitioning.

• Look for tables that can benefit from partitioning.

• Time-based, hash-based, etc.

• PostgreSQL 10+ has greatly improved partitioning functionality.

• Just be sure that the table has a strong partitioning key.
Parallel Query Execution.

- Increase the number of query workers, and the per-query parallelism.
- Very powerful for queries that handle large result sets.
- Make sure your I/O capacity can keep up!
Statistics Targets.

- For fields with a large number of values, the default statistic target can be too low.

- Especially for longer values.
  - Strings, UUIDs, etc.

- Look for queries where a highly specific query is planned to return a large number of rows.

- Don’t go crazy! Increasing statistics targets slows ANALYZE time.
Alternative Indexes.

- Some fields are not good matches for B-tree indexes.
- Long strings, range types, etc.
- Use indexes appropriate for the type.
- Hash indexes are very good for strings, especially those with most of the entropy later in the string (URLs, etc.).
Upgrades.

- pgupgrade still works fine.

- Time is proportional to the number of database objects, not database size.

- If downtime is unacceptable, logical replication / rehoming works as well.

- Be sure to plan for major version upgrades…
  - … lest you be the 1PB database still on 8.3.
Ten Terabytes.
Big.

- Congratulations! You’re definitely in the big leagues now.
- Some hard decisions will need to be made.
Backups

- Anything involving copying is going to start being very slow and impractical.

- Consider moving to file system snapshots for the base backup in PITR.

  - ZFS, SAN-based snapshots, etc.
Tablespaces.

- Tablespaces are a pain.
- Only use them if you have a specific reason.
- Fast/slow storage, reaching limits of a single volume, etc.
- Understand that they will complicated backups and replication.
Index Bloat.

- Index bloat can be a significant problem at this size.
- Space in indexes is harder to reclaim that space in the heap.
- Reindex / replace scripts can be helpful here.
Index Contention.

- High write rates against a UNIQUE index can create locking issues.

- Especially with closely clustered keys, such as SERIAL, Snowflake-generated primary keys, etc.

- If generated keys are guaranteed to be unique, consider dropping the UNIQUE constraint.

- Consider using non-sequential keys, such as UUIDs.
Write Capacity.

• Write capacity might start being constrained.

• Time to consider sharding.

• Many options: Citus, Postgres-XL, custom application-based sharding.

• Also can significantly accelerate large-dataset reads.

• Be prepared for the increase in administration complexity.
Huge.
• PostgreSQL can handle really huge databases.

• But you need to be prepared to make some complex choices.

• Each large installation is unique, but…
What’s the working set?

• If most of the data is just archival, performance will be more manageable.

• But if it’s archival, why not archive it?

• Separate the system into a transactional system and a data warehouse.

• Logical replication is great for this.
Large-Scale Sharding.

• Instead of one gigantic database, or closely connected nodes.

• Geographic, enterprise, etc.

• Multi-master tools, if necessary, to handle synchronization.
  • Bucardo, 2nd Quadrant’s BDR.
Data Federation.

• Move archival data to alternative datastores.
  • Or even into cold storage if it’s not required for analytics.

• Use Foreign Data Wrappers to federate multiple databases.

• Or just run big/small databases on the same PostgreSQL instance.
In Sum.
PostgreSQL is amazing.

- It can handle everything from your laptop to world-spanning database environments.
- It will grow with you.
- Don’t over-tool your installation at each phase, but…
- … keep one eye out for how to handle the next step.
Thank you!
Questions?
Christophe Pettus
CEO, PostgreSQL Experts, Inc.

christophe.pettus@pgexperts.com

thebuild.com

twitter @xof