PostgreSQL Performance… when it’s not your job.

Christophe Pettus
PostgreSQL Experts, Inc.

PgDay  SCALE 10x
20 January 2012
Hi.

- Christophe Pettus
- Consultant with PostgreSQL Experts, Inc.
- http://thebuild.com/
- Got into being a DBA because no one else where I was working wanted to be.
The situation.

• You are a developer.
• Or system administrator.
• Or random passer-by.
• “Oh, we need PostgreSQL installed on that box.”
*This machine was bought in 1997.*

*It is running PostgreSQL 9.1.2.*

*Your argument is invalid.*
The problem.

- “Sorry, no hardware upgrade budget.”
- “But that machine should be plenty, right?”
- “Oh, make sure the database runs really fast.”
- “You know, tweak some settings and add some indexes or something.”
Courage!

- We’ll assume your hardware can’t be changed.
- We’ll assume you don’t really want to be a full-time DBA.
- We’ll do the best we can under trying circumstances.
- What else can one do in life, really?
Le menu.

- Settings: Memory, I/O, CPU, others.
- Application-Level Stuff.
- Monitoring and add-ons.
OK, three slides on hardware.

- More (ECC) RAM is better, and...
- More cores are better, but consider your real concurrency, so...
- Spend the real money on I/O.
  - RAID 1 for the transaction logs.
  - RAID 1+0 for the data.
- Hardware RAID >> software RAID.
Considered harmful.

• Parity-disk RAID (RAID 5/6, Drobo, etc.).
• iSCSI, especially for transaction logs.
  • The latency will kill you.
• SANs, unless you can afford multichannel fibre.
• Long network hauls between the app server and database server.
Special cases.

• If you can get RAM 2-3x your database size, do it.

• Largely read-only databases can use parity-disk RAID effectively.

• If you are on a VM, remember the hypervisor takes both memory and CPU; allow an extra 15% RAM and 1 core for it.
Let’s be reasonable.

- PostgreSQL 9.0 or higher.
- If you are still on 8.x, upgrade already.
- At least 4GB of main memory.
- At least one big RAID 1 drive pair.
  - “Big” means “30% or less full when everything’s on it.”
Only PostgreSQL on the box.

- This is the single easiest thing you can do to improve PostgreSQL performance.
- No mail server, web server, app server, LDAP server, massive JBOSS install...
- Messes up caching, steals CPU, eats memory and I/O… it’s just bad news.
- This includes host machines for VMs!
A note on parameter settings.

- Will try to be as quantitative as possible.
- Remember, different workloads will have different requirements.
- These are general guidelines and first cuts.
- YMWACV.
The PostgreSQL memory dance.

- Most of your memory should be available for file system cache.
- Don’t try to give PostgreSQL every last byte. It won’t appreciate it.
- “Somewhat more than just enough” is the right setting.
Memory types.

- File system cache
- Shared buffers
- Working memory
- Maintenance working memory
- Ancillary memory
File system cache.

- 30-50% of system memory should be available for file system cache.
- This is the level-1 disk cache for PostgreSQL.
- Don’t starve it! If you see it drop below 30% of total, you need to free up memory.
Memory parameters.

- shared_buffers
- work_mem
- maintenance_work_mem
- effective_cache_size
Shared buffers.

- PostgreSQL’s private cache of disk pages.
- In a perfect world, the current working set of database disk pages would fit in here.
- Shared across all currently running PostgreSQL processes.
- Allocated in full as soon as PostgreSQL starts up.
Working memory.

- Memory used for sorts, hash joins, etc.
- If you see lots of temp files being created (check the logs), increase it.
- The default is almost certainly too low, BUT:
- It applies *per planner node*; you can use *many times* this much memory at once.
That’s not much use. How big, already?

Big enough to cover 95% of temp files being created.

Note that on-disk operations are smaller (byte for byte) than in-memory ones.

So work_mem needs to be 2-3x the temp file size you are seeing.
work_mem, part 2

• Big, slow queries might be helped by more work_mem.

• Use EXPLAIN ANALYZE <query>, and look for on-disk operations.

• But remember! If you increase it system-wide, any query might eat it up.

• Consider setting it per-session or per-operation.
Quick show of hands.

• Everyone know about VACUUM, and why you do it?
• How about VACUUM FREEZE?
• And when should you do a VACUUM FULL?
• That’s right, never (well, almost never).
Maintenance working memory.

- Used mainly for VACUUM and related operations.
- Are your VACUUMs and autovacuums taking forever, or not finishing?
  - Bump it up, but…
  - 1GB is pretty much as high as you want to go.
Well, that didn’t work.

- Can’t give it enough memory for VACUUMs to finish?
- Consider doing a manual VACUUM (via cron) at low-demand periods.
  - maintenance_work_mem can be set per session or per role.
- Create a VACUUM-specific superuser.
Ancillary memory.

- Per-connection, lock tables, prepared transactions, etc., etc.
- These days, these tend not be huge memory sinks.
- But if you feel like cranking max_connections above 200?
- Use pooling instead (hold that thought).
**effective_cache_size**

- One of these things is not like the other, one of these things, doesn’t belong.
- Does not allocate any memory.
- It’s just a planner hint.
- \(=\) file system cache + shared_buffers
I/O.

- This is generally where databases fall apart.
- Proper ACID compliance comes at a cost.
- That cost is paid in the currency of I/O operations per second.
- Ultimately, it’s up to the speed of the underlying storage subsystem.
- But you can help.
I/O parameters.

- wal_segments
- checkpoint_completion_target
- checkpoint_segments
- checkpoint_interval
- effective_io_concurrency
The write-ahead log.

- Every committed change to the database is written to the write-ahead log.
- Exceptions: Temporary and unlogged tables.
- Constant stream of write activity, broken into segments of 16MB each.
- Ideally, put it on its own set of disks (HD or SSD).
WAL parameters.

- Not much, really.
- `wal_buffers` can be bumped up to 8-16MB.
- But the big-deal activity is in checkpoints.
About checkpoints.

- A complete flush of dirty buffers to disk.
- Potentially a lot of I/O.
- Done when the first of two thresholds are hit:
  - A particular number of WAL segments have been written.
  - A timeout occurs.
But why?

• Well, the stuff has to get flushed sometime.
• Otherwise, it would have to replay the WAL segments from the beginning in case of a crash…
• … and that means you’d have to keep them all.
• … and it can take as long to replay as it did to create them.
The goal.

- Keep checkpoints from flooding the I/O subsystem.
- Background writer trickles out changes all the time.
- The various parameters interact in a complex way.
- That sounds too much like work.
Checkpoint parameters.

- checkpoint_segments will take 16MB x setting in disk space. Set it as high as you can afford (in disk space).

- Set checkpoint_timeout so that you will almost always hit checkpoint_segments first at high-load times. (30min? Measure.)

- Set checkpoint_completion_target to 0.9.
What this does.

• Spreads the write activity out over 90% of checkpoint_timeout.

• You can’t actually reduce the amount of stuff it has to write…

• … you can just tell it to write it out more smoothly.
Caveat checkpoint.

- Assume a dirty restart of PostgreSQL will take as long as checkpoint_timeout to replay the pending logs.
- So, cap the values based on your pain tolerance there.
To reiterate.

- The checkpoint parameters do not change how much data PostgreSQL has to write.
- That’s purely a result of how much write activity your applications are creating.
- They do change the pattern of how it is written, however.
effective_io_concurrency

• Another planner hint.
• Set to the number of disks in a RAID array.
• Set to the number of channels in an SSD or SSD array.
• Sit back and watch it... well, it can help hash joins.
synchronous_commit

- PostgreSQL guarantees that data has been hardened on disk when COMMIT returns.
- Turning this off disables that guarantee.
- The database will not get corrupted.
- But transactions that were COMMITed might disappear on a dirty restart.
- Turn “off” if you can tolerate that.
fsync = on
CPU.

- PostgreSQL spawns one process per connection (plus some supporting utilities).
- More cores are better.
- Roughly, 2 concurrent queries per core.
- Boring! Let’s talk about the planner instead.
ANALYZE

• Collects statistics on the data to help the planner choose a good plan.

• Done automatically as part of autovacuum.

• Always do it manually after substantial database changes (loads, etc.).

• Remember to do it as part of any manual VACUUM process.
default_statistics_target

- PostgreSQL keeps a histogram of statistics for most columns in the table.
- This is the number of bins in that histogram. Default is 100.
- Crank it up a bit if you have lots of big tables.
- You can set it per column... and you should for key columns in big tables.
seq_page_cost vs random_page_cost

• A vague guestimate of how much more expensive it is to fetch a random page than a sequential page.

• Default is 4x (random_page_cost=4.0).

• Lower it for big RAID arrays, SSDs.
Logging.

• This one’s easy:
  • Use the built-in logging collector.
  • Use CSV logs.
  • Keep at least 30 days of logs.
  • They gzip down nice.
Logging.

- `log_temp_files = 0`
- `log_checkpoints = on`
- `log_lock_waits = on`
- `log_min_statement_duration = …`
  - 100-250ms for interactive usage, 1-100s+ for data warehousing.
Special situations.

- Very write-intensive work loads.
- Very high-variability work loads.
- Repeatable bulk loads.
Write-intensive workloads.

- `shared_buffers` can be reduced.
- `checkpoint_segments` ++
- `checkpoint_timeout` ++
- `synchronous_commit` off, if you can handle the data-loss window.
Write-intensive workloads.

- If the load is has a large percentage of UPDATEs (vs INSERTs):
  - bgwriter_lru_maxpages = 0
  - autovacuum_cost_delay ++ (or consider disabling in favor of manual VACUUM, see later).
Highly-variable workloads.

- Periods of very high demand, periods of very slack demand.

- If the cycles are short:
  - bgwriter_delay ++
  - checkpoint_segments ++
  - checkpoint_timeout ++
Highly-variable workloads.

- If the periods are long (daily), in addition:
  - autovacuum = off
  - Do a manual VACUUM with large maintenance_work_mem during slack periods.
  - Remember that you must do the manual vacuum regularly!
Repeatable bulk loads.

- Database can reinitialized in case of a failure.
- Use COPY or a dedicated tool.
- Create indexes at the end.
- This is the one time it is recommended to turn off fsync.
- Remember to turn it back on, OK?
Application stuff.

- Indexing
- SQL Pathologies
- Connection Management
- Workload Distribution
- Stupid Database Tricks
Indexing.

• What is a good index?
• A good index:
  • … has high selectivity on commonly-performed queries.
  • … or, is required to enforce a constraint.
Indexing.

- What’s a bad index?
- Everything else.
- Non-selective / rarely used / expensive to maintain.
- Only the first column of a multi-column index can be used separately.
Indexing

- Don’t go randomly creating indexes on a hunch.
- That’s my job.
- `pg_stat_user_tables` — Shows sequential scans.
- `pg_stat_user_indexes` — Shows index usage.
SQL pathologies.

- Gigantic IN clauses.
- Expensive-to-process EXISTS clauses.
- Unanchored text queries like ‘%this%’; use the built-in full text search instead.
- Small, high-volume queries processed by the application.
Connection management.

- Opening a new connection is expensive.
- If you are getting more than 200 connections regularly, consider a pooler.
- If you are getting more than 500, run, don’t walk.
- pgbouncer.
Workload distribution.

- Transaction processing / web app workloads do not mix with data warehousing workloads.
- Use 9.x’s streaming replication to create a read-only secondary to do the data warehousing.
- And you get an emergency backup for free!
Stupid Database Tricks.

• Sessions in the database.
• Constantly-updated accumulator records.
• Task queues in the database.
• Using the database as a filesystem.
• Frequently-locked singleton records.
• Very long-running transactions.
INSERT storms.

- INSERTs are a terrible way to do a bulk load.
- Use COPY instead.
- Most language drivers have a good interface to it.
  - If it doesn’t, get a better driver.
Shiny on the outside.

- Prepared statements.
- Partitioning.
Prepared statements.

• Usually a “take it or leave it” situation with the particular language driver.

• They do not automatically improve performance.

• In fact, the most common situation is a total loss.

• Getting a benefit from them requires application participation.
Partitioning.

• Breaks a table up into a set of tables, based on a partitioning key.

• PostgreSQL can automatically direct (most) queries to the particular sub-tables for queries using the partitioning key.
Partitioning.

• Can be great IF:

  • Data has a (near-) uniform distributed key that never changes.
  • Data can be partitioned into equal(-ish)-sized bins on that key.
  • Queries always include that key.
  • Queries almost always hit 1-2 bins.
BONUS CONTENT!

PostgreSQL on AWS

• Remember that instances can restart at any time.

• Remember that EBS mounts can just disappear.

• EBS performance is... variable.

• Instance storage is limited, expensive, and can evaporate even more readily than EBS.
AWS Survival Guide

- As much memory as you can afford.
- Two 8-way RAID-0 EBS mounts: One for the data, one for the transaction log.
- Don’t use instance storage for any database data; config is OK if you keep a snapshot.
- `random_page_cost = 1.1`
- Set up streaming replication.
Monitoring.

- CPU usage.
- Memory usage.
- I/O usage.
- Query times.
- Table growth.
- Table behaviors (last vacuum, etc.).
Tools.

- `check_postgres`
- Lovely script, Nagios-friendly.
- `pg_stat_*`
- PostgreSQL has lots of built-in views.
Add-Ons.

- PostgreSQL extensions:
  - auto_explain
  - pg_stat_statements
- Plenty of others:
  - contrib/
  - http://pgxn.org/
Questions?
Thanks.

cpettus@pgexperts.com
xof@thebuild.com