PostgreSQL
when it’s not your job.

Christophe Pettus
PostgreSQL Experts, Inc.
DjangoCon Europe 2012
The DevOps World.

- “Integration between development and operations.”
- “Cross-functional skill sharing.”
- “Maximum automation of development and deployment processes.”
- “We’re way too cheap to hire real operations staff. Anyway: Cloud!”
Thus.

• No experienced DBA on staff.
  • Have you seen how much those people cost, anyway?
• Development staff pressed into duty as database administrators.
• But it’s OK… it’s **PostgreSQL**!
Everyone Loves PostgreSQL

- Robust, feature-rich, fully-ACID compliant database.
- Very high performance, can handle hundreds of terabytes or more.
- Well-supported by Python, Django and associated tools.
- Open-source under a permissive license.
But then you hear…

• “It’s hard to configure.”
• “It requires a lot of on-going maintenance.”
• “It requires powerful hardware to get good performance.”
• “It’s SQL, and everyone knows how old and boring that is. Also: It’s not WebScale™.”
• “Elephants scare me.”
We’re All Going To Die.
It Can Be Like This.
*This machine was bought in 1997.*

*It is running PostgreSQL 9.1.3.*

*Tell them: "Your argument is invalid."*
PostgreSQL when it is not your job.

- Basic configuration.
- Easy performance boosts (and avoiding pitfalls).
- On-going maintenance.
- Hardware selection.
Hi, I’m Christophe.

- PostgreSQL person since 1997.
- Django person since 2008.
- Consultant with PostgreSQL Experts, Inc. since 2009.
- Django guy.
No time to explain!
Just do this!
The philosophy of this talk.

- It’s hard to seriously misconfigure PostgreSQL.
- Almost all performance problems are application problems.
- Don’t obsess about tuning.
PostgreSQL configuration.

• Logging.
• Memory.
• Checkpoints.
• Planner.
• You’re done.
• No, really, you’re done!
Logging

- Be generous with logging; it’s very low-impact on the system.
- It’s your best source of information for finding performance problems.
Where to log?

• syslog — If you have a syslog infrastructure you like already.

• standard format to files — If you are using tools that need standard format.

• Otherwise, CSV format to files.
What to log?

```python
log_destination = 'csvlog'
log_directory = 'pg_log'
logging_collector = on
log_filename = 'postgres-%Y-%m-%d_%H%M%S'
log_rotation_age = 1d
log_rotation_size = 1GB
log_min_duration_statement = 250ms
log_checkpoints = on
log_connections = on
log_disconnections = on
log_lock_waits = on
log_temp_files = 0
```
Memory configuration

- shared_buffers
- work_mem
- maintenance_work_mem
shared_buffers

• Below 2GB (?), set to 20% of total system memory.
• Below 32GB, set to 25% of total system memory.
• Above 32GB (lucky you!), set to 8GB.
• Done.
work_mem

- Start low: 32-64MB.
- Look for ‘temporary file’ lines in logs.
- Set to 2-3x the largest temp file you see.
- Can cause a huge speed-up if set properly!
- But be careful: It can use that amount of memory per planner node.
• 10% of system memory, up to 1 GB.
• Maybe even higher if you are having VACUUM problems.
effective_cache_size

- Set to the amount of file system cache available.
- If you don’t know, set it to 50% of total system memory.
- And you’re done with memory settings.
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.
Checkpoint settings, part 1

wal_buffers = 16MB

checkpoint_completion_target = 0.9

checkpoint_timeout = 10m-30m # Depends on restart time

checkpoint_segments = 32 # To start.
Checkpoint settings, part 2

- Look for checkpoint entries in the logs.
- Happening more often than checkpoint_timeout?
  - Adjust checkpoint_segments so that checkpoints happen due to timeouts rather filling segments.
- And you’re done with checkpoint settings.
Checkpoint settings, part 3

- The WAL can take up to $3 \times 16\text{MB} \times \text{checkpoint\_segments}$ on disk.
- Restarting PostgreSQL can take up to \texttt{checkpoint\_timeout} (but usually less).
Planner settings.

- `effective_io_concurrency` — Set to the number of I/O channels; otherwise, ignore it.
- `random_page_cost` — 3.0 for a typical RAID10 array, 2.0 for a SAN, 1.1 for Amazon EBS.
- And you’re done with planner settings.
Easy performance boosts.

- General system stuff.
- Stupid database tricks.
- SQL pathologies.
- Indexes.
- Tuning VACUUM.
General system stuff.

- Do not run anything besides PostgreSQL on the host.
- If PostgreSQL is in a VM, remember all of the other VMs on the same host.
- Disable the Linux OOM killer.
Stupid database tricks, I

- 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.
• Using INSERT instead of COPY for bulk-loading data.

• psycopg2 has a very good COPY interface.

• Mixing transactional and data warehouse queries on the same database.
One schema trick

• If one model has a constantly-updated section and a rarely-updated section…
  • (Like a user record with a name and a “last seen on site” field)
  • … split those into two models (and thus two database tables).
• You’ll thank me later.
SQL pathologies

• Gigantic IN clauses (a typical Django anti-pattern).

• Unanchored text queries like ‘%this%’; use the built-in full text search instead.

• Small, high-volume queries processed by the application.
Indexing, part I

• What is a good index?

• A good index:
  • … has high selectivity on commonly-performed queries.
  • … or, is required to enforce a constraint.
Indexing, part 2

• 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, part 3

- 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.
VACUUM

• autovacuum slowing the system down?
  • Increase autovacuum_vacuum_cost_limit (default is 200).

• If load is periodic…
  • Do manual VACUUMing instead at low-low times.

• You **must** VACUUM regularly!
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.
On-going maintenance.

- Monitoring.
- Backups.
- Disaster recovery.
- Schema migrations.
Monitoring.

- Always monitor PostgreSQL.
- At least disk space and system load.
- Memory and I/O utilization is very handy.
- 1 minute bins.
- `check_postgres.pl` at bucardo.org.
**pg_dump**

- Easiest PostgreSQL backup tool.
- Very low impact on the database being backed up.
- Makes a copy of the database.
- Becomes impractical as the database gets big (in the tens of GB).
Streaming replication, 1.

- Best solution for large databases.
- Easy to set up.
- Maintains an exact logical copy of the database on a different host.
  - Make sure it really is a different host!
- Does not guard against application-level failures, however.
Streaming replication, 2.

- Replicas can be used for read-only queries.
- If you are getting query cancellations...
  - Increase max_standby_streaming_delay to 200% of the longest query execution time.
- You can pg_dump a streaming replica.
Streaming replication, 3.

- Streaming replication is all-or-nothing.
- If you need partial replication, you need trigger-based replication (Slony, Bucardo).
- These are not part-time jobs!
WAL archiving.

- Maintains a set of base backups and WAL segments on a (remote) server.
- Can be used for point-in-time recovery in case of an application (or DBA) failure.
- Slightly more complex to set up, but well worth the security.
- Can be used alongside streaming replication.
Pitfalls

- Encoding.
- Schema migrations.
- <IDLE IN TRANSACTION>
- VACUUM FREEZE
Encoding.

- Character encoding is fixed in a database when created.
- The defaults are probably not what you want.
- Use UTF-8 encoding (with appropriate locale).
  - C Locale sometimes makes sense.
Who has done this?

• Add a column to a large table.
• Push out to production using South or something.
• Watch production system fall over and go boom as PostgreSQL appears to freeze?
• I’ve… heard about that happening.
Schema migrations.

- All modifications to a table take an exclusive lock on that table while the modification is being done.
- If you add a column with a default value, the table will be rewritten.
- This can be very, very bad.
Schema migration solutions.

- Create columns as not NOT NULL.
- Then add constraint later once field is populated.
- Takes a lock, but a faster lock.
- Create new table, copy values into it (old table can be read but not written).
A session state when a transaction is in progress, but the session isn’t doing anything.

Common in Django applications.

Be careful about your transaction model.

Don’t accept Django’s default transaction behavior.
Once in a long while, PostgreSQL needs to scan (and sometimes write) every table.

This can be a big surprise.

Once every few months, pick a (very) slack period and do a VACUUM FREEZE.
Hardware selection, one year ago.

• “Here are the kind of I/O subsystems to avoid, and to get.”

• “You need blah about this much memory…”

• “And you should think about cores and this and that and this other thing blah blah blah…”
The Cloud.
Hardware in the cloud.

- Get as much memory as you can.
- Get one CPU core for each two active connections.
- Usually, few connections are active.
- Hope the I/O subsystem can keep up with your traffic.
- Eventually, it won’t.
Your own hardware...

- Get lots of (ECC) RAM.
- CPU is usually not as vital as RAM.
- First step is hardware RAID, with:
  - RAID10 for the main database.
  - RAID1 for the transaction logs.
  - RAID1 for the boot disk.
Considered harmful.

- Parity-disk RAID (RAID 5/6, Drobo, etc.).
- iSCSI, especially for transaction logs.
- SANs, unless you can afford multichannel fibre.
- Long network hauls between the app server and database server.
• Biggest instance you can afford.
• EBS for the data and transaction logs.
• Don’t use instance storage for any database data; OK for text logs.
• random_page_cost = 1.1

• Set up streaming replication.
Additional tools.

• www.repmgr.org
• WAL-E from Heroku.
• pgFouine (log analyzer).
• pgbouncer (part of SkypeTools).
Additional reading.

- thebuild.com
- pgexperts.com
Questions?
Thank you!