Why is PostgreSQL Terrible?

Christophe Pettus
PostgreSQL Experts
Nordic PGDay 2018

Christophe Pettus

CEO, PostgreSQL Experts, Inc.

christophe.pettus@pgexperts.com

thebuild.com

twitter @xof

"It's more of a comment..."


"It's more of a comment..."


EXIT

Why is PostgreSQL terrible?

Spoiler Alert!

It’s not.

Thanks for coming!

So, what is this?

- The most common pain points we hear from customers.
- A field report.
- Not edited for ease of solving the problem.
- Freely concede that some of these will be very hard to fix.
- But the customer is always right!

A bit about PGX

- San Francisco (area)-based consultancy, since 2009.
- Hundreds of clients over the life of the company.
- Dozens of active clients.
- Primarily focused, ad hoc engagements.
- Companies from one-person startups to Fortune 5; databases from 10MB to multiple petabytes.

A bit about me.

- A DBA with a lot of developer experience.
  - (wrote an entire SQL RDBMS in C++ once.)
- PostgreSQL user since ~~7.1~~ **6.5!**
  - That’s before foreign key constraints.
- PostgreSQL consultant for 10+ years.
- PostgreSQL is my life!


TREK


TREK

No bikeshedding!

- “This is solved by third-party tool x.”
  - Worthy tools, but we’re focusing on community PostgreSQL here.
- “This would be very hard and you can’t do it because of architecture / resources / malignant influence of the moon.”
  - We said that about SERIALIZABLE.
- “This is being worked on!”
  - Great!

Query Planning

PostgreSQL's query planning is great.

- But to most application developers, it's a random number generator.

- Query optimization is an experimental art, and requires a lot of trial-and-error.

- There's very little visibility into exactly why the query planner chose the precise execution plan it did.

- Optimizing complex queries can be very difficult, especially in a test or staging environment.

Things that are hard to explain.

- “Why isn’t it using an index?”
- “Why can’t I explain to it that the query plan is wrong?”
- “Why didn’t it collect statistics on the sequential scan it just did?”
- “How can I tell how this query is going to be planned on a much larger database?”

Query “tuning” parameters.

- No longer connected to anything real.
- Spinning disk parameters in an SSD/SAN world.
- Requires tribal knowledge, back-engineering, and just fooling around until something works…
- … and hope it doesn’t break as the database grows.

What I'd like.

- Ability to trace the query planning process.
- Ability to produce plans for theoretical database sizes and distributions.
- Experiential evidence gathered during query execution fed back into the query planning process.
- Whisper it: ***Hints***.
  - But hints in a PostgreSQL-type way.
  - More insight into the data, not "walk this way."

VACUUM

Everyone blames VACUUM for everything all the time.

- The two most common initial requests are “adding indexes” and “tuning VACUUM.”

- It’s almost never VACUUM.

- That being said, VACUUM is very hard to tune.

- The current set of tuning parameters are pure guesswork, and far too focused on the underlying implementation.

- When VACUUM goes wrong, it goes really wrong.

What we want.

- Flexible resource caps based on real-world parameters (“no more than x% of system resources”), with emergency fallbacks.

- Better solutions for high-update situations.
    - More work done at update time, to reduce the amount of bulk VACUUM activity.

- Parallel VACUUM on large tables.

- Better (=any) autovacuum prioritization.

- Or no VACUUM at all, but one step at a time.

Database Functionality

Database

Functionality

Everyone Loves PostgreSQL.

- Everyone is very happy with core database functionality.
- New features are appreciated and adopted…
  - … but the lack of them is almost never a source of complaints.
- No one ever rejects or moves away from PostgreSQL because of core engine functionality.
- It’s always an operations problem.

Upgrades


PostgreSQL
upgrades are terrible.

The options are:

- Dump/restore — Impractical with real-life databases.
- pg_upgrade — Does the job, but weird gotchas and unexpected behaviors, not great interaction with secondaries.
- Slony/Bucardo/etc. — Weird and fiddly, imposes requirements on the schema, and load on the source.
- pglogical — Requires 9.4+, imposes requirements on the schema, and is not bug-free.

So many problems.

- Entire streaming replica clusters must be upgraded together.

- PITR backups are not restorable into the new version, making them of reduced value.

- Every PostgreSQL extension creates a new place for upgrade to fail.

    - “Oh, an upgrade across those versions is easy, we’ll just use pg_upgrade oh you’re running PostGIS?”

What we want.

- Major version upgrades that are just like minor version upgrades.
    - Version-aware disk format.
- For many customers, even the “bounce” for a minor version upgrade requires extensive planning.
- This is why large sites often stay on EOL’d versions.
    - 15TB database on 8.1 (and Solaris).

Connection Management

Let’s be honest.

- Most application stacks have terrible connection management.

- The reaction to running out of connections is to kick up max_connections until the problem goes away.

  - “Just set max_connections to 25000 we’ll sort it out later.”

- And then watch as the database server melts down when too many connections become active at once.

PgBouncer, right?

- Yeah sure fine.
- One more moving piece…
- … with its own high availability (and monitoring, and deployment, etc.) story.
- Single-threaded, and can easily become the bottleneck under high connection churn and query rates.
- Problematic for environments with many different databases, or database roles.

What we want.

- PgBouncer-like multithreaded event based connection management in core.
    - And a pony, while you're at it.
- A (possibly optional) layer on top of the existing backend model.
- Ability to switch database and role without re-forking a back end.

High Availability

The PostgreSQL HA Story.

- “Add a streaming replica.”
- “Add um some up-front component that handles routing to the active server mumble mumble PGPool.”
- “Write a bunch of scripts to handle instance replacement and provisioning.”
- “Problem solved!”

This is a terrible story.

- Requires third-party tools of varying degrees of quality, history, and community support.
  - Some are great, don’t get me wrong!
- Requires scripting and configuration.
- Often does not solve important problems (reprovisioning, etc.).
- Streaming replication has an inherent tradeoff between useful-for-load-balancing and useful-for-failover.

What we want.

AWS RDS

Amazon RDS solves this out of the box.

- RDS' high availability story is the strongest single driver towards RDS adoption.

- RDS has a ***lot*** of negatives as a product.
    - Black box performance, Amazon lock-in, version lag, upgrade strangeness, expense…

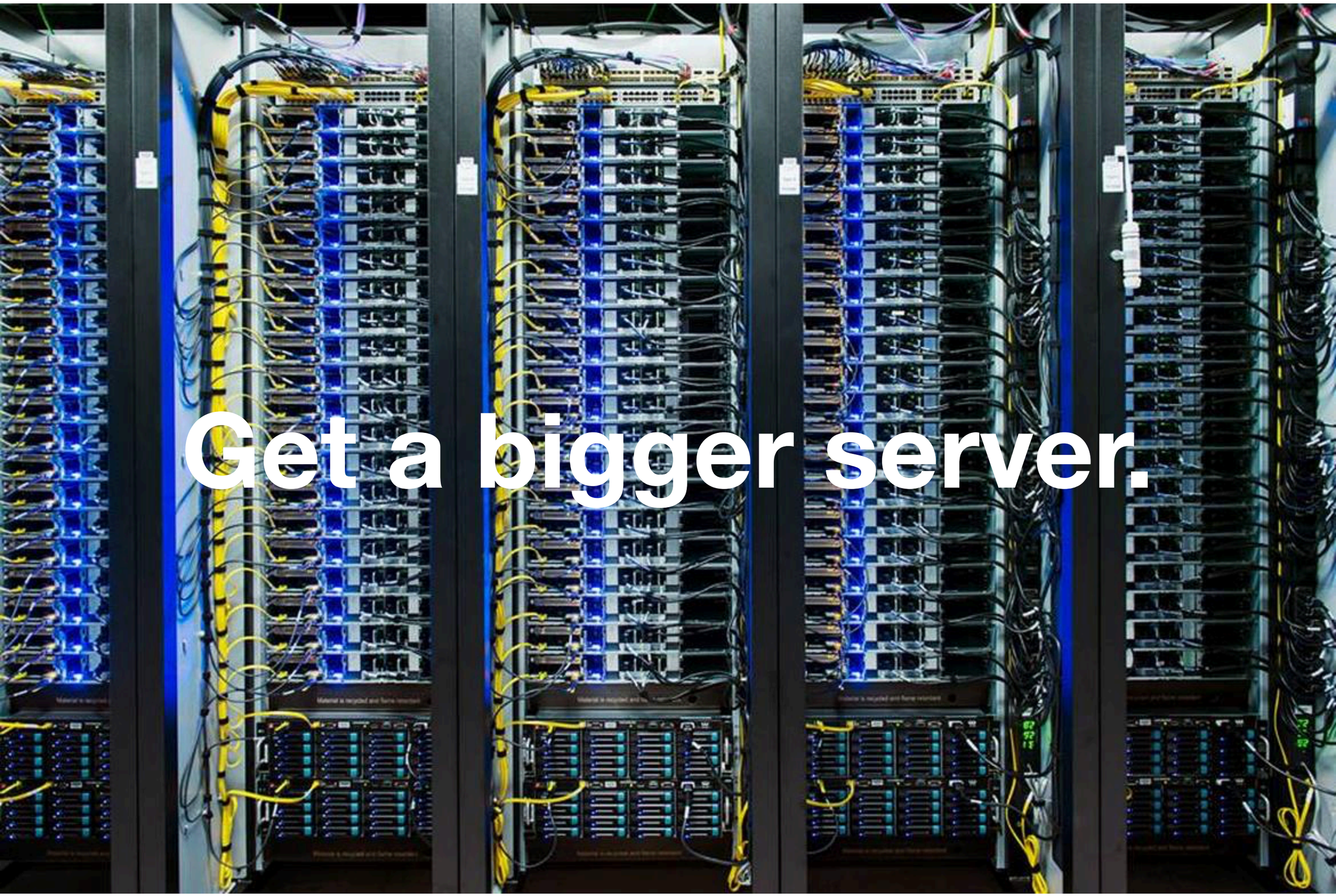
- But the HA story is complete, easy to understand, and easy to integrate into an application stack.

A proper solution needs to provide…

- A single consistent endpoint for the application to connect to.
- Load balancing to amortize the cost of the secondaries.
- Replacement of failed nodes.
- In a perfect world, aborted transactions rather than broken connections.
- A library of pre-packaged scripts for common environments (AWS, OpenStack…).
- A framework for scripting to handle bespoke environments.

Scaling

The out-of-the-box scaling solution.


Get a bigger server.

Party like it's 1997.

- PostgreSQL does not have an in-core distributed scaling solution.

- "Use read replicas" is fine as far as it goes, but requires front-end tooling and application awareness.

- Sharding requires a third-party solution or custom application development.

- Write scaling is **much** harder than read scaling.

- Sharding solutions often have uncomfortable integration with high availability.

Matching DB resources to load.

- Nearly all databases have a load profile that is not a straight line.
- Right now, the solution is to provision for the largest sustained spike.
  - Expensive for your own hardware, ruinous for cloud computing.
- “Spin up more read replicas” is not a viable solution when new database demand can hit in seconds.
- Adjusting the primary database size generally means downtime, at the worst possible moment.

Scale doesn’t scale.

- Moore’s Law is dead now.
- Individual core performance will be (by historical standards) flat for the foreseeable future.
- Lower-performance, lower-power, higher-core-count CPUs (and arrays of them) are the new normal.
- PostgreSQL’s current process model is not a good fit for this environment.

What we want.

Oracle Real Application Clusters (RAC)

Oracle Real Application Clusters (RAC)

(that doesn’t suck)

A single sharding / HA solution.

- Cloud computing has fundamentally changed how we view system resources.

- “Servers” are both easy to provision and transient.

- Firing up twelve servers to handle six shards is completely reasonable now.

- Sharding solutions that don’t take HA into account are making a bad situation worse.


OLD MAN YELLS AT CLOUD

I was that man.

- Yes, “the cloud” is just other people’s computers.
- But it has fundamentally changed how operations work.
- The ability to dynamically provision computing resources is revolutionary.
- Especially compared to the Big Glass-House Server model we had through the early 2000s.
- PostgreSQL has not kept up with this model.

How it was.

- One server that provides an endpoint.
- This server completely encapsulates the service.
- That server “owns” the storage resources required to provide the service.
- “Scaling up” requires that you get a bigger server, faster networking, faster I/O…
- … or you do a lot of application development work.

How it is.

- Computing resources (“servers”) come and go all the time, due to scaling, host machine failure, etc.

- Storage resources are widely shared across applications and computing resources, and provide the persistence.

- Multiple computing resources share the same underlying storage.

- Scaling up means spinning up more compute resources to work on the same underlying data.

What we want.

… with PostgreSQL functionality.

This means…

- A single endpoint for both reading and writing.
- Adding more resources is done automatically to meet bursts or sustained demand, or manually to handle prospective demand.
- The cluster is tolerant of node failures, and heals without manual intervention.
- That pony would be nice, too, while you're up.

PostgreSQL vs The Future.

- The PostgreSQL model is rooted in the old, server-centric world.

- That world is passing and will never come back.

- New approaches to scaling are required.

- If we want PostgreSQL to thrive in this world, we need to start thinking about this.

Lots of hard problems.

- Distributed lock manager.
- Distributed transaction manager.
- Distributed foreign keys (and other relationships).
- Parallelizing everything.
- Resilience to individual compute node failures (restartable partial queries, etc.).
- Front-end routing (with its own high availability story).

But we need to move forward.

- The old “single glass house server” model is dead.
  - Even if it hasn’t quite stopped moving yet.
  - The rapid adoption of containers shows that this is something we’ve been waiting for.
  - This will be a whole new codebase!
  - But in 1999, so was PostgreSQL.

In Sum.

“Professionals talk ~~logistics~~ operations.”

- PostgreSQL relies too much on third parties for its full operation solution.

- For the vast majority of our customers, they are not waiting for new core RDBMS features.

- Replication didn’t stop with Slony. Operational features need core support, too.

- Third parties are embracing / enhancing / extinguishing community PostgreSQL through operational convenience.

Thank you!

Christophe Pettus

CEO, PostgreSQL Experts, Inc.

christophe.pettus@pgexperts.com

thebuild.com

twitter @xof

https://2018.nordicpgday.org/feedback