Corruption War Stories

Christophe Pettus
PostgreSQL Experts

PGDay FOSDEM 2017

Christophe Pettus
PostgreSQL Experts, Inc.

pgexperts.com

thebuild.com

christophe.pettus@pgexperts.com

Twitter @xof

It will happen.

- Database corruption will happen to you.
- Sooner or later.
- Fortunately, it's super easy to recover!

Step 1:
Restore last-good backup

Step 2:
Receive the praise of a grateful nation.

Time for coffee!

Oh.

- You don't have a known-good backup?
- That's a shame.
- Sadly, even good backups can…
  - have hidden long-term corruption.
  - be too old.
  - *(whisper it)* be hit by PostgreSQL bugs.

Let's talk about…

- Preventing corruption.
- Finding corruption.
- Fixing corruption…
  - … if you can.

Preventing Corruption

PostgreSQL is very trusting.

- PostgreSQL assumes the file system is perfect.
- It cannot recover from any silent bad data write (unless you are very lucky).
- With 9.3 checksums, you at least get a warning.
  - So use them.

Hardware is cheap. Data is expensive.

- Use good-quality hardware.
- Be sure your hardware properly honors fsync, end to end.
  - The lack is more common than you think.
- Avoid network-attached devices for $PGDATA *and backups*.

Backup, backup, backup.

- What only exists on one drive you do not truly possess.
- Be sure you follow the right backup protocol for your technique.
  - pg_start_backup(), etc.
- Test your backups.
  - An untested backup strategy isn't one.

Prophylactic pg_dump

- pg_dump to /dev/null.
- Reads every single row in the database.
- Great for finding lurking corruption.
- Of course, if you can save the dump file, do so!

What causes corruption?

#1: Hardware failures.

- Underlying storage failure.
  - Bad disk, bad controller.
- Garbage writes during power loss.
  - Battery backup that didn't.
- Bad RAM.
  - Especially non-ECC RAM.

#2: Hardware “features.”

- Deferred or entirely missing fsync behavior.
  - Often done to flatter benchmark results.
- Network-attached-storage that does not handle detach gracefully.
- Soft-RAID edge conditions.

#3: PostgreSQL bugs.

- 9.x had a series of unfortunate replication bugs.
- Used to be extremely rare.
- With luck, will become extremely rare again.

#4: Operator error.

- Backups that do not include critical files.
- Backups that do not follow protocol.
- Backups that forget external tablespaces.
- rm -rf in the wrong directory.
- Bungled attempts at problem recovery.
  - Delete the wrong files to free space.

GitLab.com Database Incident - 2017/01/31

Note: This incident affected the database (including issues and merge requests) but not the git repo's (repositories and wikis).

YouTube Live stream - Follow us live debating and problem solving!

Timeline (all times UTC) 1

Recovery - 2017/01/31 23:00 (backup from ±17:20 UTC) 2

Problems Encountered 4

External help 5

Hugops (please add kind reactions here, from twitter and elsewhere) 5
Stephen Frost 5
Sam McLeod 5

Impact

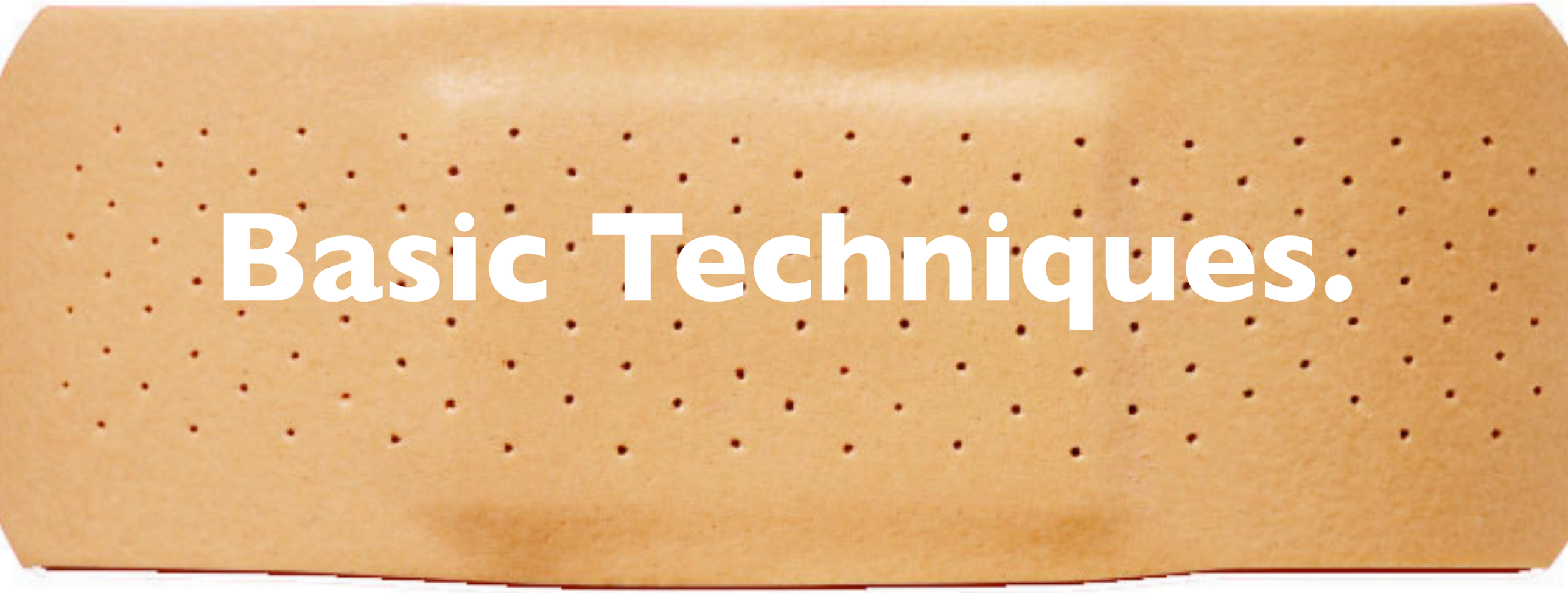
1. ±6 hours of data loss
2. 4613 regular projects, 74 forks, and 350 imports are lost (roughly); 5037 projects in total. Since Git repositories are NOT lost, we can recreate all of the projects whose user/group exist cannot restore any of these projects' issues, etc.
3. ±4979 (so ±5000) comments lost
4. 707 users lost potentially, hard to tell for certain from the Kibana logs
5. Webhooks created before Jan 31st 17:20 were restored, those created after this time are lost

Timeline (all times UTC)

1. **2017/01/31 16:00/17:00 - 21:00**
   a. YP is working on setting up pgpool and replication in staging, creates an LVM snapshot to get up to date production data to staging, hoping he can re-use this for bootstrapping roughly 6 hours before data loss.
   b. Getting replication to work is proving to be problematic and time consuming (estimated at ±20 hours just for the initial pg_basebackup sync). The LVM snapshot is not usable on could figure out. Work is interrupted due to this (as YP needs the help of another collegue who's not working this day), and due to spam/high load on GitLab.com
2. **2017/01/31 21:00** - Spike in database load due to spam users - Twitter | Slack
   a. Blocked users based on IP address
   b. Removed a user for using a repository as some form of CDN, resulting in 47 000 IPs signing in using the same account (causing high DB load). This was communicated with th team.
   c. Removed users for spamming (by creating snippets) - Slack
   d. Database load goes back to normal, some manual PostgreSQL vacuuming is applied here and there to catch up with a large amount of dead tuples.
3. **2017/01/31 22:00** - Replication lag alert triggered in pagerduty Slack
   a. Attempts to fix db2, it's lagging behind by about 4 GB at this point
   b. db2.cluster refuses to replicate, `/var/opt/gitlab/postgresql/data` is wiped to ensure a clean replication
   c. db2.cluster refuses to connect to db1, complaining about `max_wal_senders` being too low. This setting is used to limit the number of WAL (= replication) clients
   d. YP adjusts `max_wal_senders` to `32` on db1, restarts PostgreSQL
   e. PostgreSQL complains about too many semaphores being open, refusing to start
   f. YP adjusts `max_connections` to `2000` from `8000`, PostgreSQL starts again (despite 8000 having been used for almost a year)
   g. db2.cluster still refuses to replicate, though it no longer complains about connections; instead it just hangs there not doing anything
   h. At this point frustration begins to kick in. Earlier this night YP explicitly mentioned he was going to sign off as it was getting late (23:00 or so local time), but didn't due to the repli of a sudden.

What to do?

- Buy good hardware, demand your cloud provider do so, or have multi-tier redundancy.
- Make backups, and test them.
- Stay up on PostgreSQL releases, and read the release notes.



Basic Techniques.

Save all the parts!

- Stop PostgreSQL.
- Do a full file-system level backup.
- Keep that backup safe.
- Make changes methodically, and document each step.

Index Corruption.

- The most common kind of corruption.
- Drop the index in a transaction, and confirm that solves the problem.
- If so, rebuild the index.
- If not, it's probably not index corruption.

Take a pg_dump.

- pg_dump reads every row, and…
- … creates a logically-good snapshot.
- Restore that into a clean database.

Bad Data Page.

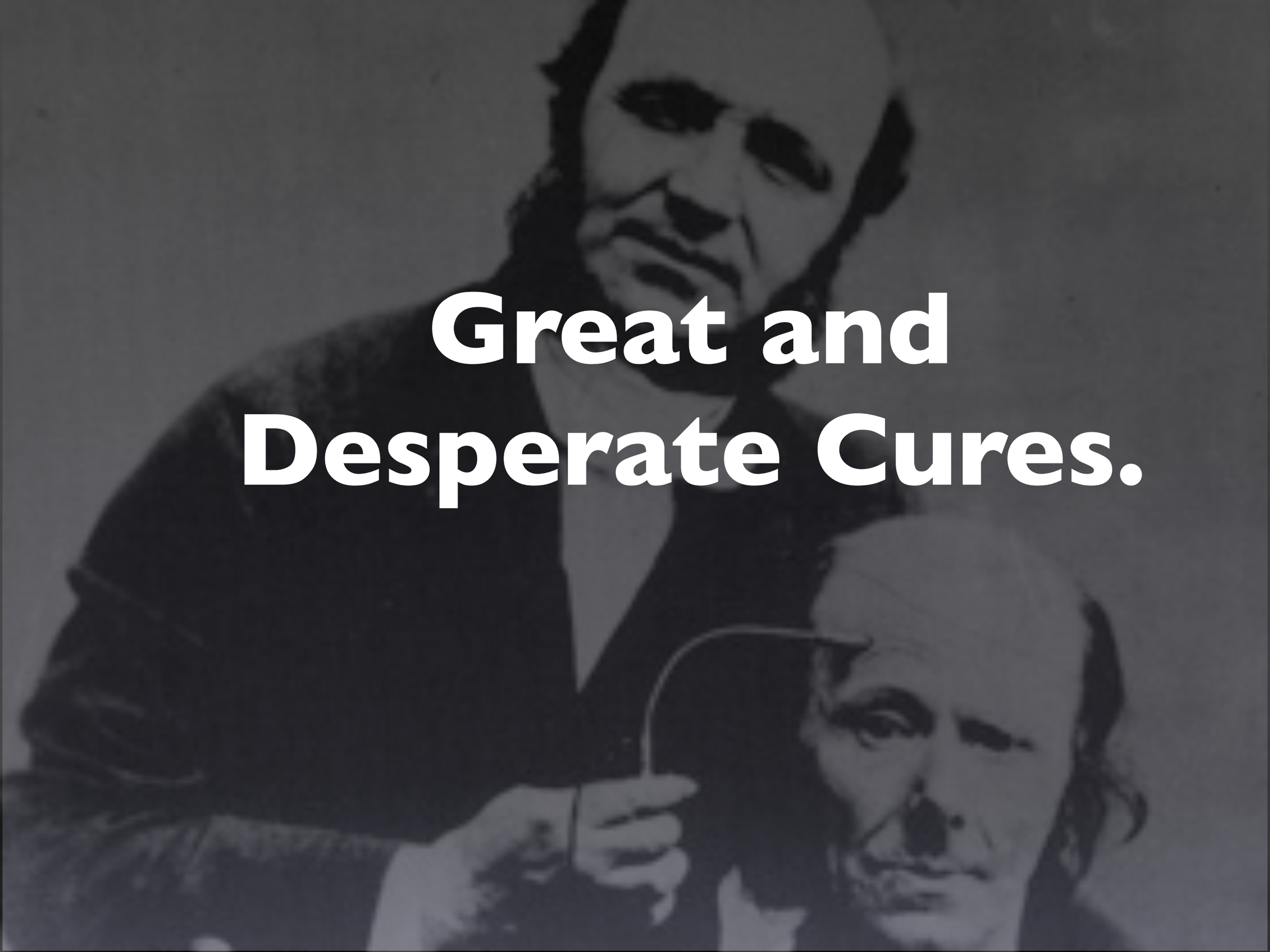
- Checksum failures, complaints about bad headers, etc.
- Can you do a pg_dump of the table?
- zero_damaged_pages = on.

Really Bad Data Pages.

- Can you SELECT around them?
- Do a COPY out of the good data, drop table, COPY back in.
- Or do a CREATE TABLE from the SELECT, rename appropriately.
- DELETE just the bad rows by ctid, if you can isolate them.

Finding bad data pages.

- Iterate through rows in PL/pgSQL…
- … with an exception block around the SELECT.
- Catch and log any rows that throw an exception.
- Very helpful for finding TOAST corruption.



Great and Desperate Cures.

Known unknown knowns.

- All corruption is, by its nature, a one-off situation.
- Be sure to determine the extent of it before continuing.
- Be sure you can step backwards!

There are no recipes.

REMEMBER.

WORK ON A COPY.

First things first.

- Are there errors in dmesg indicating a hardware or OS problem?
  - Is the OOM killer terminating backends?
  - Disk I/O errors?
  - Can you cp -R the data directory to /dev/null?

Very, very bad data pages.

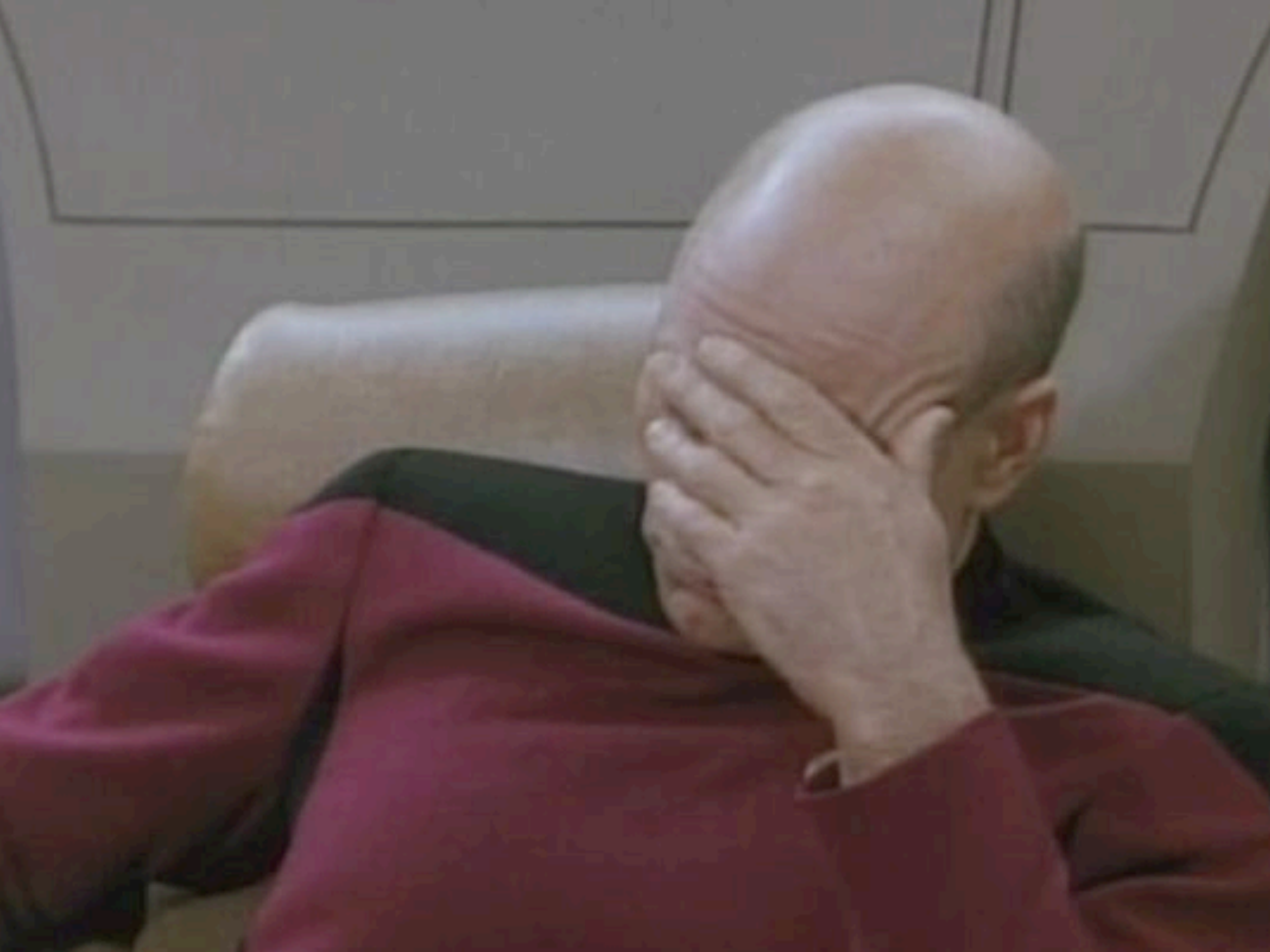
- As in, the backend crashes when it touches them.
- Isolate pages, use dd to zero out those blocks.
- Be sure to drop and recreate all indexes on the table!

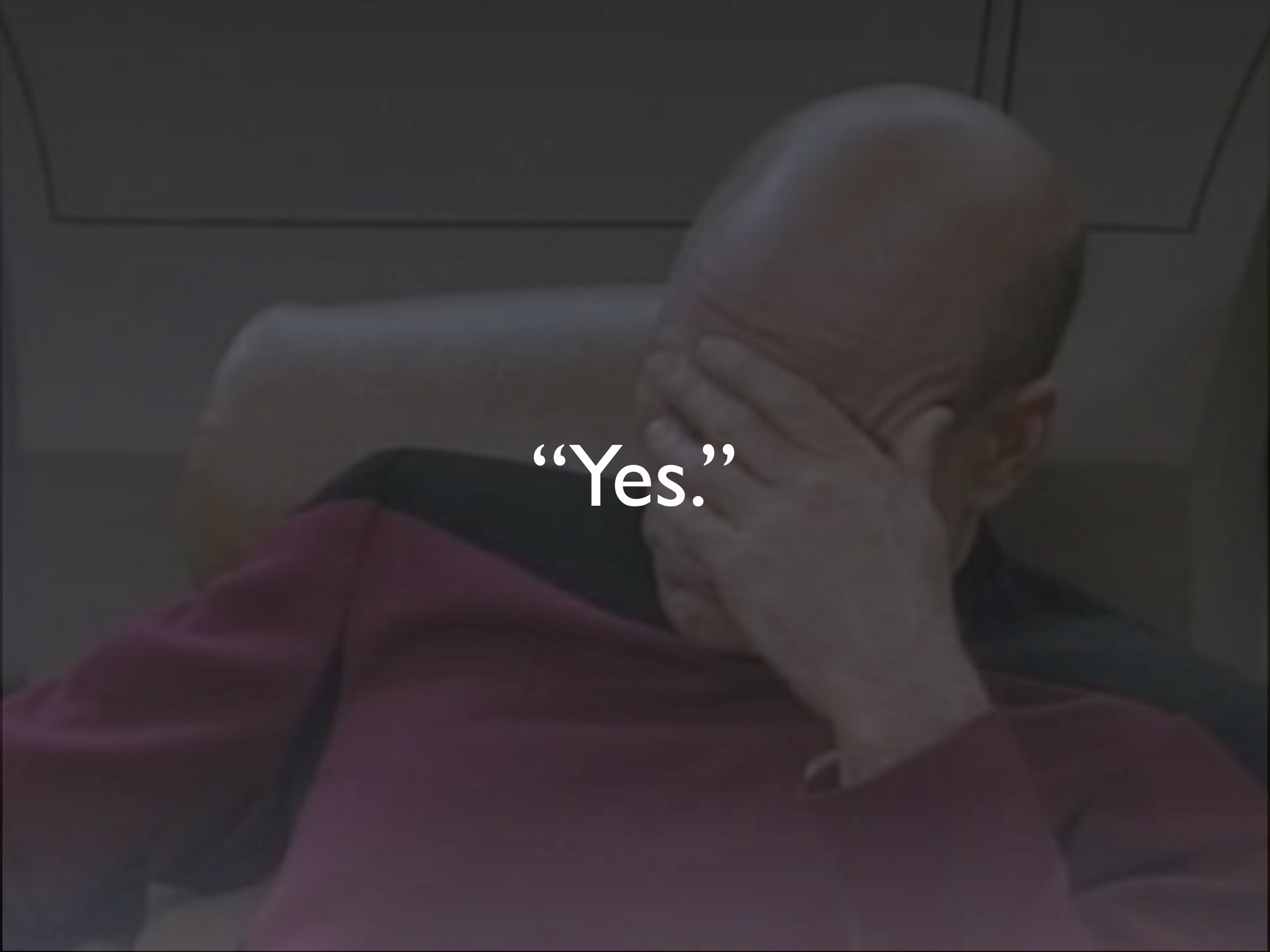
WAL files corrupt or missing.

- You went on vacation…
- The system ran out of disk space…
- And they called you to say that it won't start now.
- "We just deleted some log files."
- "Which ones?"

“pg_xlog”

“Is that bad?”




“Yes.”

pg_resetxlog

- Tells PostgreSQL that WAL files it needs for crash recovery… it doesn't need.
- Can get the server to start with missing log files.
  - Read the instructions carefully!
- High risk of inconsistent data! Check the database very thoroughly!

pg_clog

```
PG::InternalError: ERROR: could not
access status of transaction 2924295225
DETAIL: Could not read from file
"pg_clog/0AE4" at offset 212992:
Success.
```

pg_clog corruption.

- Good news: Rarely subtle.
- Missing file.
- Truncated file.

Patching.

- Replace a missing file with all-zeros file.
- 00 for a transaction means “in progress.”
- Previously-committed transactions can thus disappear.
- Be prepared to do more cleanup in this case.

System catalog corruption.

- The nightmare scenario.
- Very hard to recover from, unless the corruption is very small.
- Probably requires expert attention to do recovery or scavenging.

War Stories.

```
PG::InternalError: ERROR: could not
access status of transaction 2924295225
DETAIL: Could not read from file
"pg_clog/0AE4" at offset 212992:
Success.
```

So, how did we get here?

- Network connectivity issue caused secondary to be promoted to primary.
- New secondary couldn't handle load.
- Beefier primary was initialized from secondary, but…
- … on startup, these errors popped out.

What happened?

- Same network problems that began the situation caused the rsync building the primary to abort.
- No one noticed in the rush to get the primary back on line.

The fix…

- … the missing clog file could be copied from the secondary and dropped into place.
- Problem solved!
- Very lucky that clog file was available.

The moral?

- No matter how bad a disaster is…
- … rushing can make it worse.
- Make sure that you are not introducing new problems as you are repairing old ones.

```
ERROR:  missing chunk number 0 for
toast value 968442 in pg_toast_263610
```

So, how did we get here?

- New primary provisioned by promoting from secondary.
- Errors started appearing almost immediately.
- But only one row, on one table…

Spooky!

- … and only on some queries.
- Isolating the record using primary key found nothing; the record retrieved just fine.
- Reindexing the TOAST table? No help.
- Iterating through the table did find it, however.

What happened?

- Two levels of corruption.
- Bad TOAST entry, and…
- … two rows with the same primary key.
  - One referring to the “bad” row.
- Index scans only found the “good” row.
- Seq scans found both.

The fix…

- Delete the "missing" row by ctid.
- Iterate through all other tables to confirm no other corrupt rows.
- Rebuild indexes.

The moral?

- Read the release notes.
- This was directly related to an existing bug in PostgreSQL.
- But the hosting provider* hadn't upgraded PostgreSQL promptly.
- * cough AWS cough

Uh, Christophe?
About that upgrade...

So, how did we get here?

- New primary provisioned by promoting from secondary.
- New primary put into service, old primary decommissioned.
- Everything looks fine for a few hours, until…

Spooky!

- Some existing rows are missing.
- Some existing rows are duplicated, as if both old and new from an UPDATE had been committed.
- No error messages.

What happened?

- PostgreSQL bug.
  - Since fixed.
  - clog values were not properly being transmitted from primary to secondary under high-write-load conditions.
  - So, some rolled back, etc.

The fix…

- Enough information in the database (date/time stamps) to delete the bad rows, and copy the missing ones from the old database.
- Hand-crafted scripts to do so.
- Never, ever want to do that again.

The moral?

- Do not exclude that it could be a PostgreSQL problem.
- Do thorough sanity checks on promoted primaries.
- Have a client or employer who understands open source software.

Nothing works.
Everything is broken.
We're all going to
die.

So, how did we get here?

- Database running on desktop hardware.
- Disk did not honor fsync.
- Power failure… with a UPS that hadn't been tested in a while.

Spooky!

- PostgreSQL started up correctly, but…
- Backend crashed when touching certain tables.
- Those tables were central to the application.

What happened?

- Broad corruption on four tables…
  - … and the sequences on those tables.
- Couldn't do a system-wide pg_dump.
- Had to do touch / crash / recovery to find the damaged tables.

The fix…

- Schema-only dump to get a blank database.
- pg_dump to restore undamaged tables.
- Restore damaged tables from an old (but still useful) backup.
  - Old backup had an obsolete schema, so transformation required.
- Manually reset sequences from data.

The moral?

- Desktop hardware is not a great choice for a business-critical server.
- No location is too small to have a secondary.
- Even very old backups can be useful in a dire emergency.

```
ERROR: database is
not accepting
commands to avoid
wraparound data loss
in database "oops"
```

So, how did we get here?

- "Too many autovacuums going on"
- `autovacuum_freeze_max_age = 2000000000`
- `vacuum_freeze_table_age = 1000000000`

- Yay! No more autovacuum freeze jobs!

Spooky!

- On Halloween, no less…
- … wraparound warnings appeared in the log.
- … but they weren't monitoring the logs closely.

What happened?

- By the time they noticed, it was too late.
- xids being used faster than autovacuum could freeze the table.
- Eventually hit shutdown mode.

The fix…

- Manually vacuum the "oldest" tables to get the database back on-line.
- Aggressively vacuum the "oldest" tables as the system goes back into production.
- Ignore complaints about how much I/O was being done.

The moral?

- Don't crank up the autovacuum freeze settings unless you do manual vacuums.
- Monitor errors and warnings in the PostgreSQL logs.
- Don't terminate autovacuum freeze processes thinking you will "deal with it later."

In sum.

Remember the basics.

- Good hardware.
- Test your backups.
- Stay up on PostgreSQL news and apply upgrades promptly.
- Monitor your log output.
- Get plenty of rest.

Thank you!

Christophe Pettus
PostgreSQL Experts, Inc.

pgexperts.com

thebuild.com

christophe.pettus@pgexperts.com

Twitter @xof