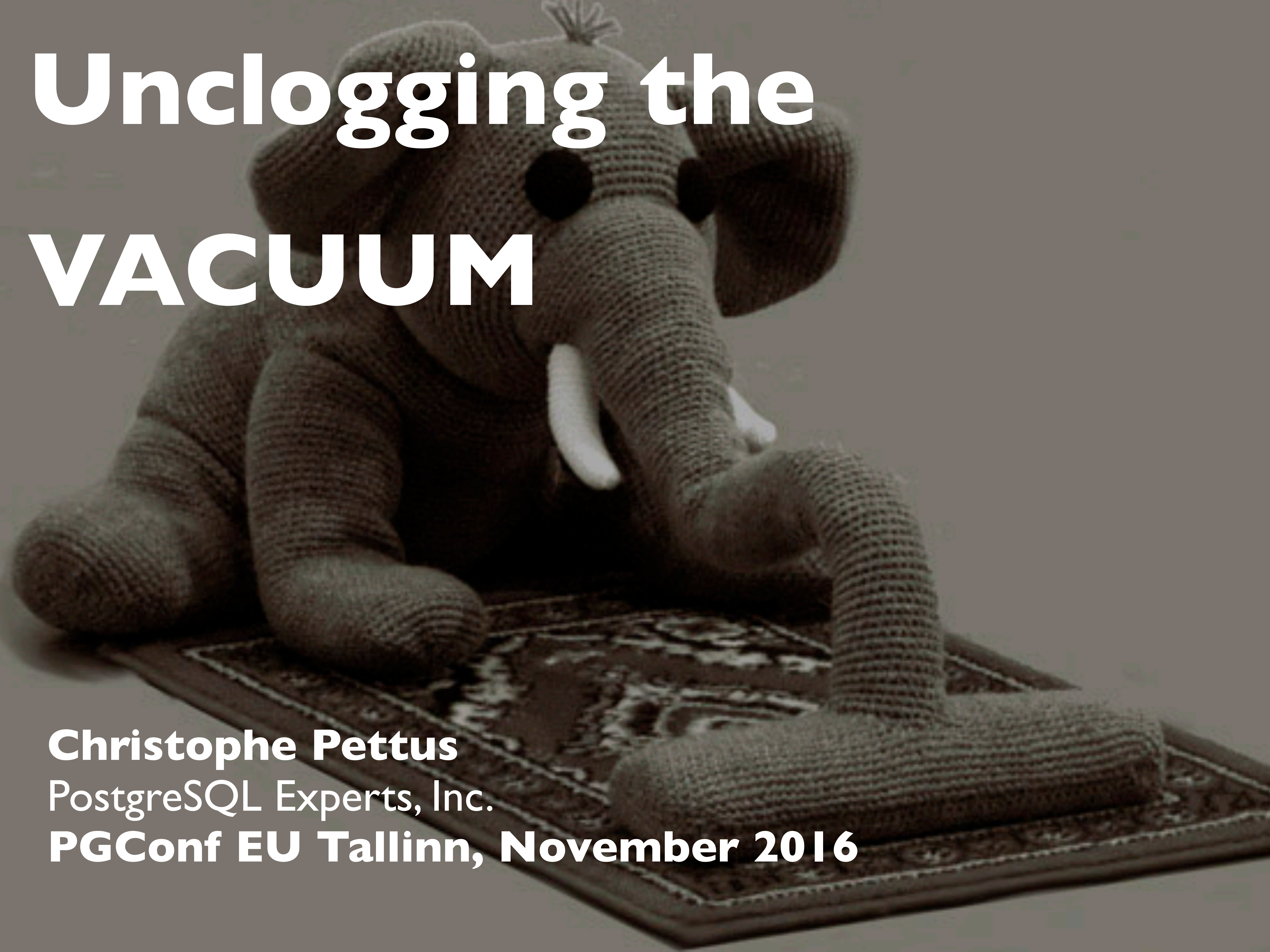


Unclogging the VACUUM

Christophe Pettus
PostgreSQL Experts, Inc.
PGConf EU Tallinn, November 2016

Greetings!

- Christophe Pettus
- CEO, PostgreSQL Experts, Inc.
- thebuild.com — personal blog.
- pgexperts.com — company website.
- Twitter @Xof
- christophe.pettus@pgexperts.com

VACUUM.

- Everyone's least-favorite PostgreSQL feature.
- Yet, essential to proper database operation.
- But why do we need it at all?
- Let's take a moment to find out.

The Problem.

- Process 1 begins a transaction.
- Process 2 begins a transaction.
- Process 1 updates a tuple.
- Process 2 tries to read that tuple.
- What happens?

Bad Things.

- Process 2 can’t get the new version of the tuple (ACID [generally] prohibits dirty reads).
- But where does it get the old version of the tuple from?
  - Memory? Disk? Special old-tuple area?
  - What if we touch 250,000,000 rows?

PGX
POSTGRESQL
EXPERTS, INC.

Some Approaches.

- Lock the whole database.
- Lock the whole table.
- Lock that particular tuple.
- Reconstruct the old state from a special area.
- None of these are particularly satisfactory.

Multi-Version Concurrency Control.

- Create multiple “versions” of the database.
- Each transaction sees its own “version.”
  - We call these “snapshots” in PostgreSQL.
- There is no privileged “real” snapshot.

Multiple Tuple Versions.

- Each version of the tuple is a real, first-class member of the database.
- And it takes up disk space.
- Even after no transaction can still "see" it, because of an UPDATE or DELETE.
- A tuple that is no longer visible to any transaction is a "dead" tuple.

Nothing's Perfect.

- Dead tuples are not immediately returned to free space
- Doing so would make COMMIT far too expensive.
- But these dead tuples build up over time.
- Which means:VACUUM!

VACUUM.

- VACUUM's primary job is to scavenge dead tuples.
- The space is reclaimed for new tuples, but is not released back to the operating system.
  - Except under relatively unusual situations.

VACUUM also...

- … can do an ANALYZE, which rebuilds the statistics that the planner uses to plan queries.
- Prevents the dreaded "xid wraparound."
- Posts updates to GIN indexes.
- More on those later.

VACUUM details.

- Standard VACUUM is incremental. It only works on pages that require vacuuming.
  - VACUUM FREEZE (<9.6) does a full table scan.
- autovacuum will stop on a table if some other process takes a lock that would prevent it from continuing.

Common Complaint #1

- "We deleted 50% of the rows of this very large table, but the disk space usage didn't go down."
- It almost never will, even after a standard VACUUM is complete.
- The space is, however, now available for reuse by new INSERTs and UPDATEs.

Bloat.

- All PostgreSQL databases have a certain amount of "bloat."
  - Bloat is disk usage over what a perfectly-packed database would have.
- My rule of thumb: ~50% bloat (2 x perfectly-packed) is normal.

Warning Signs.

- Disk space increasing much faster than the INSERT volume would indicate.
- But don't forget to include indexes, which can be larger than the data!
- Bloat percentage increasing, as opposed to absolute bloat in bytes.

autovacuum

- In 95% of all PostgreSQL installs, you never have to worry about VACUUM.
- Since version 8.0, autovacuum runs in the background, and manages it for you.
- The default configuration is suitable for most installations.
- Easy!

Complaints.

- Excessive bloat / space not being reclaimed.
- autovacuum using too much I/O.
- autovacuum getting "stuck".
- VACUUM FREEZE-related issues.

Excessive Bloat.

- What’s “excessive”?
- Depends on UPDATE / DELETE rate.
  - Higher will mean more “normal” bloat.
- Warning sign is database footprint increasing much faster than new tuples coming in.

Is autovacuum running?

- Is it turned on? (It is by default, but some enthusiastic people turn it off and forget.)
  - autovacuum = on
- Check pg_stat_user_tables to see last autovacuum run on the table… far in the past, or never?
- log_autovacuum_min_duration = 1000

Increase frequency.

- Increase the number of workers.
  - Often require for very large schemas (1,000+ tables).
  - Up to 5, 10, even 20 for huge schemas.
- Reduce autovacuum_naptime to let autovacuum run more often.

Per-table settings.

- tuples changed > autovacuum_ vacuum_threshold + (autovacuum_vacuum_scale_factor * table size in tuples)
- For large tables, this can result in a too-long delay.
- Can adjust per-table or system-wide.

Explicit locking.

- autovacuum “backs off” if a strong table-level lock is taken on a table.
- Schema changes, explicit LOCK statements.
- High frequency LOCKing + lots of UPDATEs / DELETEs = horrible bloat (common in queuing systems).

Statistics collector running?

- If the statistics collector fails, autovacuum doesn't have the data needed to run.
-
  ```
  21942   ??  Ss     0:00.00 postgres: stats collector process
  ```

- If the statistics collector fails, autovacuum doesn't have the data needed to run.

Index Bloat.

- Index bloat is often more severe than data bloat.
- Index structure means it is harder to reclaim space effectively.
- In general, this is not a serious issue, but…

Rebuilding Indexes.

- Indexes can be periodically rebuilt if they are badly bloated.
  - CREATE INDEX CONCURRENTLY
  - DROP INDEX
- Less downtime than a REINDEX.

Detecting Bloat.

- All bloat detection methods are somewhat uncertain.
  - https://github.com/pgexperts/pgx_scripts/tree/master/bloat
- Can be included in monitoring scripts.
  - Graph them, don't just set up alerts.

The Bloat Hammer

- Sometimes, you need to un-bloat a table.
- VACUUM FULL works great, but…
  - … it takes an exclusive lock on the table for the entire time it runs.
- Often not practical for a busy system.
- (Any table-rewriting DDL will also de-bloat the table.)

pg_repack

- http://reorg.github.io/pg_repack/
- Extension to repack tables without a long exclusive lock.
- Uses triggers to create a secondary table during the repack operation.
- Some gotchas and restrictions: read the documentation!

App-level fixes.

- Use TRUNCATE rather than DELETE if practical.
- Instead of doing mass deletes, consider a partitioned table where you just DROP the older tables.
- DROP TABLE just throws the files away… no VACUUM!

Things To Avoid.

- Very long-running transactions (or idle-in-transaction sessions).
- Very frequent updates on indexed columns (defeats HOT optimization).
- Gratuitous updates (no row changes, or one-update-per-column-change).

Explicit VACUUM.

- Do an explicit VACUUM ANALYZE after large UPDATE / DELETE changes to a particular table.
- Moves to work to being part of the bulk job, rather than some random point later.

Common Complaint #2

- “autovacuum is stuck.”
- It usually isn’t.
- No, really, it usually isn’t.
- But how can you tell?

Long autovacuums.

- Is the process doing I/O?
- How big is the table being vacuumed?
- How long since the last vacuum?
- Recent major bulk update/delete operations?
- Is it using an unusual amount of CPU?

maintenance_work_mem

- Sets maximum memory autovacuum will use for various operations.
- 1-2GB is usually about right, more if you have huge indexes.
- Be aware if you have also increased the number of workers!

(to prevent xid wraparound)

- Does this appear in pg_stat_activity in the “query” column for the autovacuum process?
- This means it is doing a VACUUM FREEZE.
- These tend to be long-running and high I/O.
- More in a bit.

Killing autovacuum processes.

- As a last resort, use pg_terminate_backend to terminate an autovacuum process.
- Don’t use kill -9!
- If it is a “(to prevent xid wraparound)” autovacuum, it will probably just start up again.
- If that doesn’t work, restart PostgreSQL.

Extra for Experts

- Attach strace to the autovacuum process.
- Doing I/O? Stuck on a semaphore?
- In (very) unusual situations, autovacuum can be stuck on a spinlock on a buffer page.
- Killing the process at the OS level is usually the only choice then.

Common Complaint #3

- "autovacuum is using too much I/O."
- VACUUM is high I/O.
  - "(to prevent xid wraparound)" even more so.
- Lots and lots and lots of cost-based configuration parameters to play with.

autovacuum_vacuum_cost_delay

- First place to look.
- Increase this to make autovacuum less “aggressive” while working on a specific table.
- Start at 50-100ms, increase until the I/O load comes back under control.

But.

- This will slow down the speed of autovacuum.
- If you have both autovacuum-too-slow and autovacuum-too-much-I/O problems…
- … it may be time to look at a more-hardware or app-level solution to the problem.

Analyze.

- Technically speaking, a separate operation from VACUUM.
- However, usually done as part of a vacuum (although you can do an explicit ANALYZE separately).
- Also handled by the autovacuum daemon.

Explicit ANALYZE.

- Always do an explicit ANALYZE after major database changes:
  - Restore from pg_dump backup.
  - pg_upgrade.
  - Large INSERT / UPDATE / DELETE bulk operations.

Autovacuum ANALYZE

- Similar tuple-change parameters as VACUUM.
- If you have increased the statistics target on a table…
- … consider changing these to make ANALYZE more frequent.

“(to prevent xid wraparound)”

- Otherwise known as VACUUM FREEZE.
- Not the same thing (exactly) as VACUUM.
- Often a nasty surprise the first time it happens, as it just appears after weeks or months.
- Very high I/O, as it has to (pre-9.6) scan and potentially rewrite whole table.

What is it?

- VACUUM FREEZE is required because transaction XIDs are 32 bits wide.
- 2^32 transactions is not all that many.
- Each tuple is “stamped” with the xid that created it.
- If allowed to wrap around, data could disappear from the database.

“Freezing your tuples.”

- VACUUM FREEZE marks tuples that are visible to all transactions.
  - < 9.4, with a special XID, ≥ 9.4, with a flag.
- This prevents data loss through XID wraparound.

The problem.

- Each page of the table must be inspected for freeze candidates.
- And rewritten if it has any.
- This generates a lot of I/O, and can happen at surprising times.
  - … like, during periods of heavy traffic.

“Table age”

- The important idea is how “old” the table is in terms of transaction xids.
- Can be determined by applying the age() function to pg_class.relfrozenxid.
- Highest possible value is 2^31-1, which is the disaster point.

```
fugu=> select relname, age(relfrozenxid) from pg_class where
age(relfrozenxid)<2147483647 order by age(relfrozenxid) desc;
              relname              |  age
-----------------------------------+-------
 catalog_announcement              | 21101
 pg_toast_16550                    | 21101
 pg_statistic                      | 21100
 pg_toast_2619                     | 21099
 pg_type                           | 21099
 pg_toast_97278                    | 21098
 engagement_track_log              | 21098
 pg_toast_97296                    | 21097
 sendgrid_webhook_log              | 21097
 auth_group                        | 21096
 auth_group_permissions            | 21096
 pg_toast_16612                    | 21096
```

vacuum_freeze_min_age

- First of the three major vacuum freeze parameters.
- If a page containing a tuple "this old" is consulted for other reasons, it is frozen.
- Lowering it can pre-freeze tuples. Little downside, since it's writing the page anyway.

vacuum_freeze_table_age

- If a table gets “this old”, when a normal vacuum is done on the table, it *also* does a vacuum freeze.
- Default is relatively low (150m transactions).
- Raising defers the vacuum freeze “switch-over”.

autovacuum_freeze_max_age

- When a table gets “this old”, a vacuum freeze will be done on the table by autovacuum…
- … even if autovacuum = off!
- Once it reaches this point, *let it run*. Don’t kill it; it’ll just keep coming back.

PGX POSTGRESQL EXPERTS, INC.

So, how do I prevent VACUUM FREEZE?

YOU CAN’T.

VACUUM FREEZE is essential.

- If the "oldest" table in the "oldest" database reaches 10m transactions to wraparound, warnings start appearing in the log.
- If the "oldest" table reaches 1m transactions to wraparound, the database shuts down.

That Sounds Bad.

- PostgreSQL will shut down and will only start in single-user mode.
- Then, you *have* to do the vacuum freeze.
- So, make sure you never ignore those warnings.
- You are regularly checking the logs for warnings and errors, right?

The “Coffin Corner.”

- On a busy database, it’s possible to reach the warning point, but have transactions being created too fast to avoid shutdown.
- So, make sure you don’t get to that point!
- Repeatedly killing autovacuum processes because of high I/O can cause this.
- or too high autovacuum_freeze_max_age.

Monitoring.

- Monitor the age of the oldest tuples in the database.
  - check_postgres.pl at bucardo.org
- Don’t set autovacuum_freeze_max_age so high that you don’t enough “room” to allow proper vacuum freeze operations.

Manual VACUUM FREEZEs

- autovacuum doesn’t prioritize tables.
- It’s a good idea to do manual VACUUM FREEZEs (via a cron job, etc.) of the “oldest” tables.
    - https://github.com/pgexperts/flexible-freeze
- Pick a low-traffic period to run it.

Binary Replication Notes.

- Vacuuming the primary vacuums the secondary automatically.
- But remember all vacuum changes must be sent down the replication stream.
- hot_standby_feedback = ‘on’ to reduce query cancellations due to vacuum.

Logical Replication Notes.

- Logical replicas are vacuumed independently of their primary.
- Incoming logical changes should be considered “application” workload.
- Same cautions about application workload apply.

Sidebar: GIN Index Posting.

- GIN indexes are expensive to update.
- Thus, updates are not immediately written into the index structure.
- Instead, they are written to a “posting list” that is merged into the index at VACUUM time.
- Generally, nothing you ever worry about.

But.

- Large, frequently-updated GIN indexes can have surprising I/O and CPU spikes when this update occurs.
- If list exceeds a certain size, posting is forced without a vacuum:
  - < 9.5: work_mem
  - ≥ 9.5: gin_pending_list_limit

GIN Posting Fixes.

- On ≥9.5, set gin_pending_list_limit to a smaller value to do more frequent postings (of less data).
- <9.5, the use of work_mem constrains you somewhat.
  - Manual vacuum may be the answer there.

Innovations!

9.6!

- PostgreSQL 9.6 contains many vacuum-related improvements.
- From a DBA's perspective, it's worth upgrading just to get those.
- (And parallel query is great, too.)

Incremental VACUUM FREEZE!

- In 9.6, VACUUM FREEZE is now incremental rather than whole-table.
- Huge improvement!
- All-frozen pages are stored in the visibility map.
- One big VACUUM FREEZE required after upgrade.

VACUUM Progress!

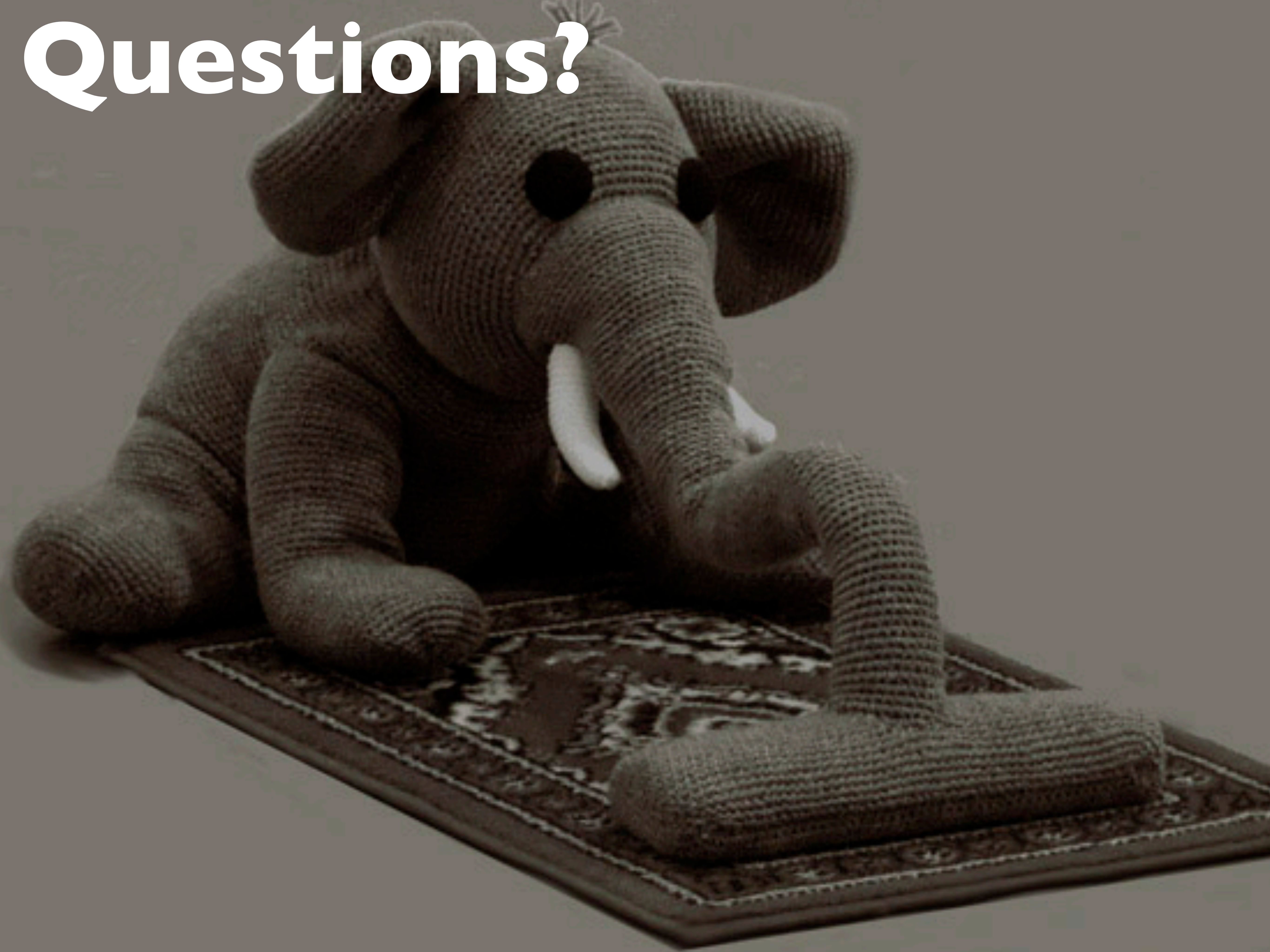
- pg_stat_progress_vacuum view.
- One row per autovacuum process.
- Shows phase of autovacuum, number of blocks scanned, total blocks.
- Finally can answer the "roughly how much longer will it be?" question.

Controllable GIN Posting

- gin_clean_pending_list()
- Updates the pending list independent of a VACUUM.
- Handy to separate the operations to reduce I/O, get the GIN index back to normal speed, etc.

So, Upgrade!

Questions?



Thank you!

Christophe Pettus
@xof

thebuild.com
pgexperts.com