All the Dirt on VACUUM

Jim Nasby, OpenSCG

Overview

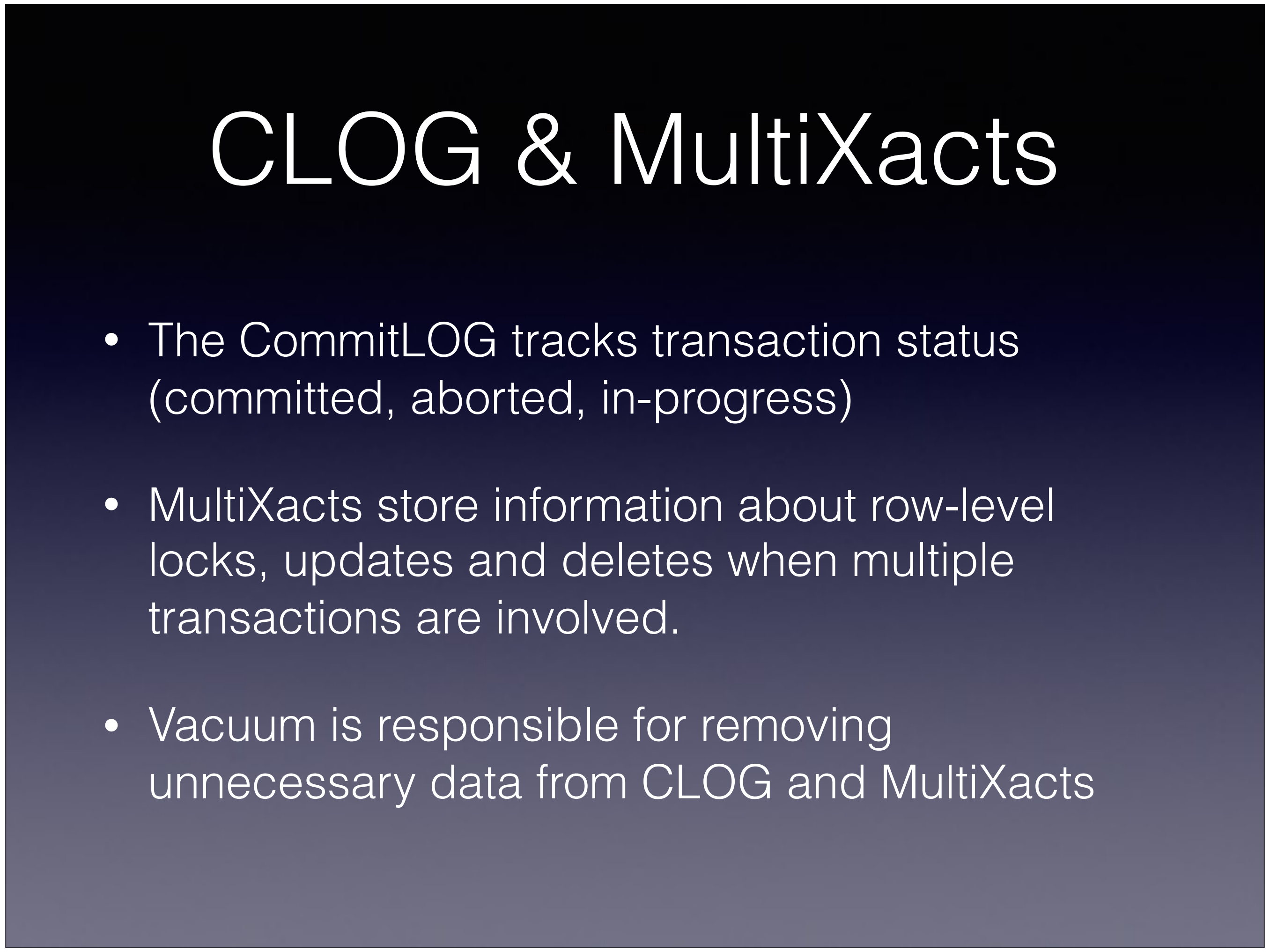
- In-depth look at vacuum in Postgres 9.6
- Code references in slide notes

Topics

- MVCC
- CLOG and MultiXacts
- What can be vacuumed
- Freezing
- HOT
- Vacuum
- Autovacuum

MVCC

- Postgres uses Multi Version Concurrency Control
- Rows are never deleted, they are only marked as deleted as of a specific transaction
- Updates are essentially a delete of the old tuple and an insert of a new tuple with the new values
- Eventually these old, dead tuples must be removed by “vacuuming”

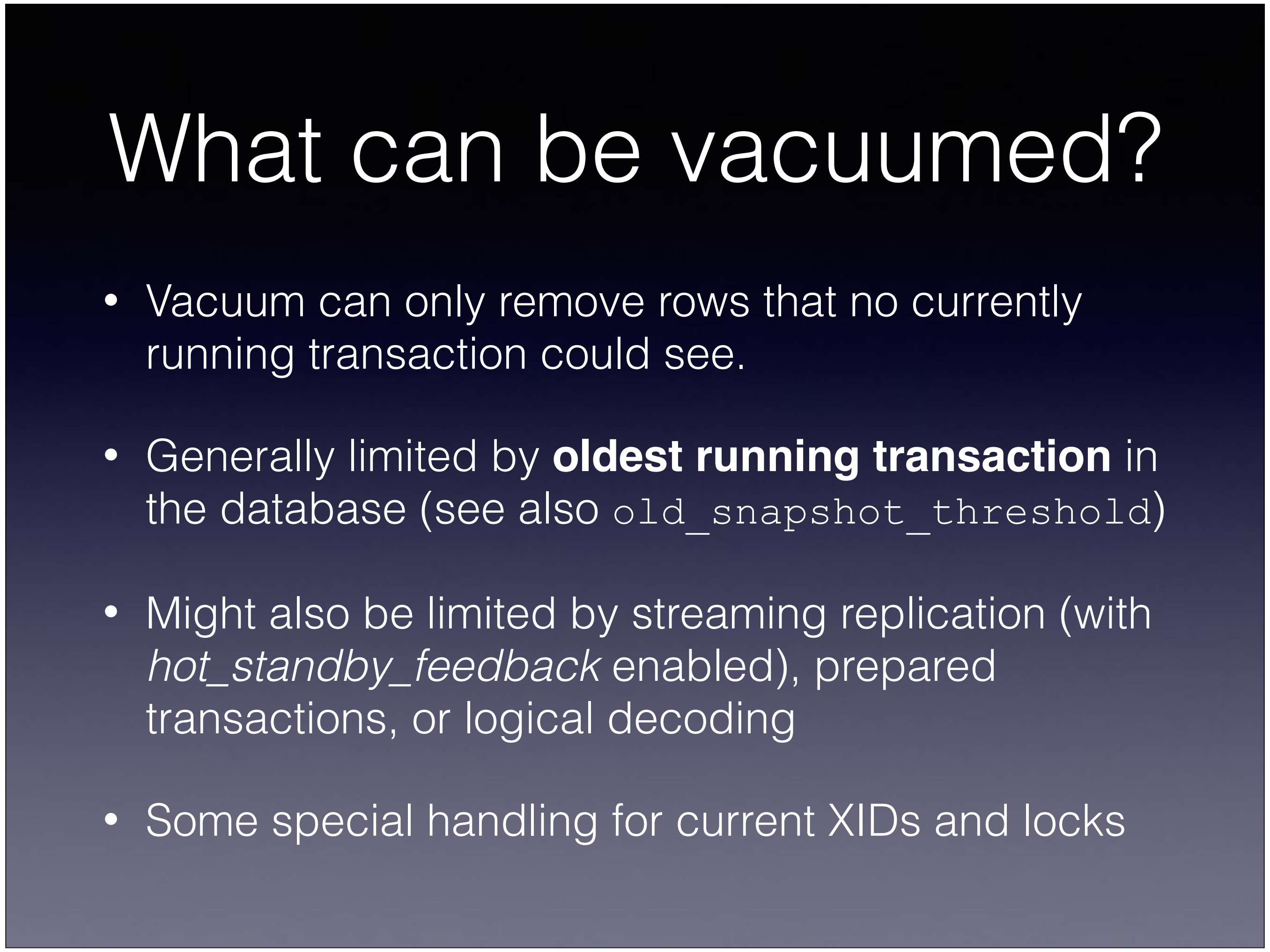


CLOG & MultiXacts

- The CommitLOG tracks transaction status (committed, aborted, in-progress)
- MultiXacts store information about row-level locks, updates and deletes when multiple transactions are involved.
- Vacuum is responsible for removing unnecessary data from CLOG and MultiXacts

CLOG and MultiXacts are stored as SLRUs (Simple Least Recently Used).

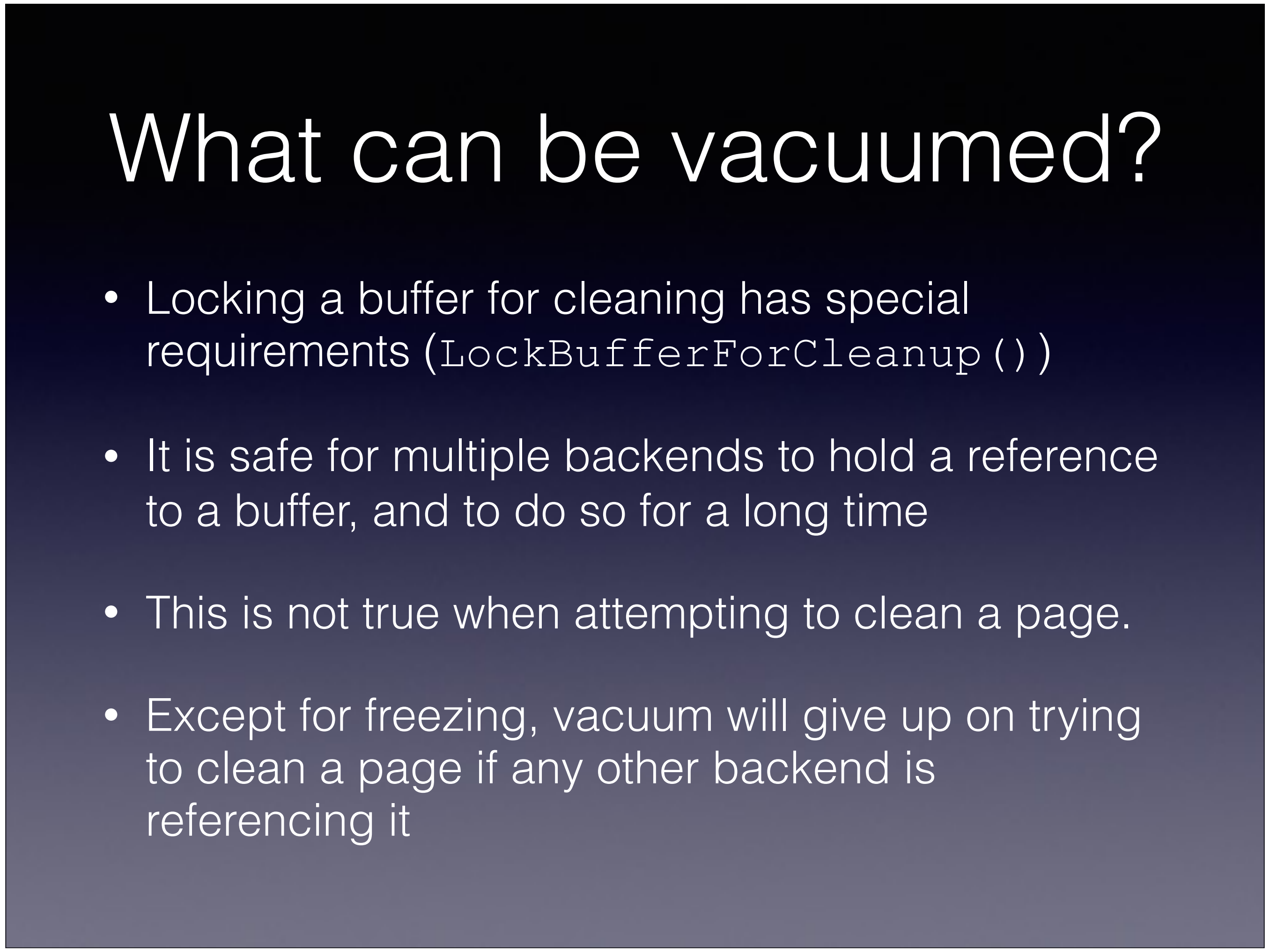
See src/backend/access/transam/clog.c, multixact.c and slru.c



What can be vacuumed?

- Vacuum can only remove rows that no currently running transaction could see.
- Generally limited by **oldest running transaction** in the database (see also `old_snapshot_threshold`)
- Might also be limited by streaming replication (with *hot_standby_feedback* enabled), prepared transactions, or logical decoding
- Some special handling for current XIDs and locks

See HeapTupleSatisfiesVacuum() for details.

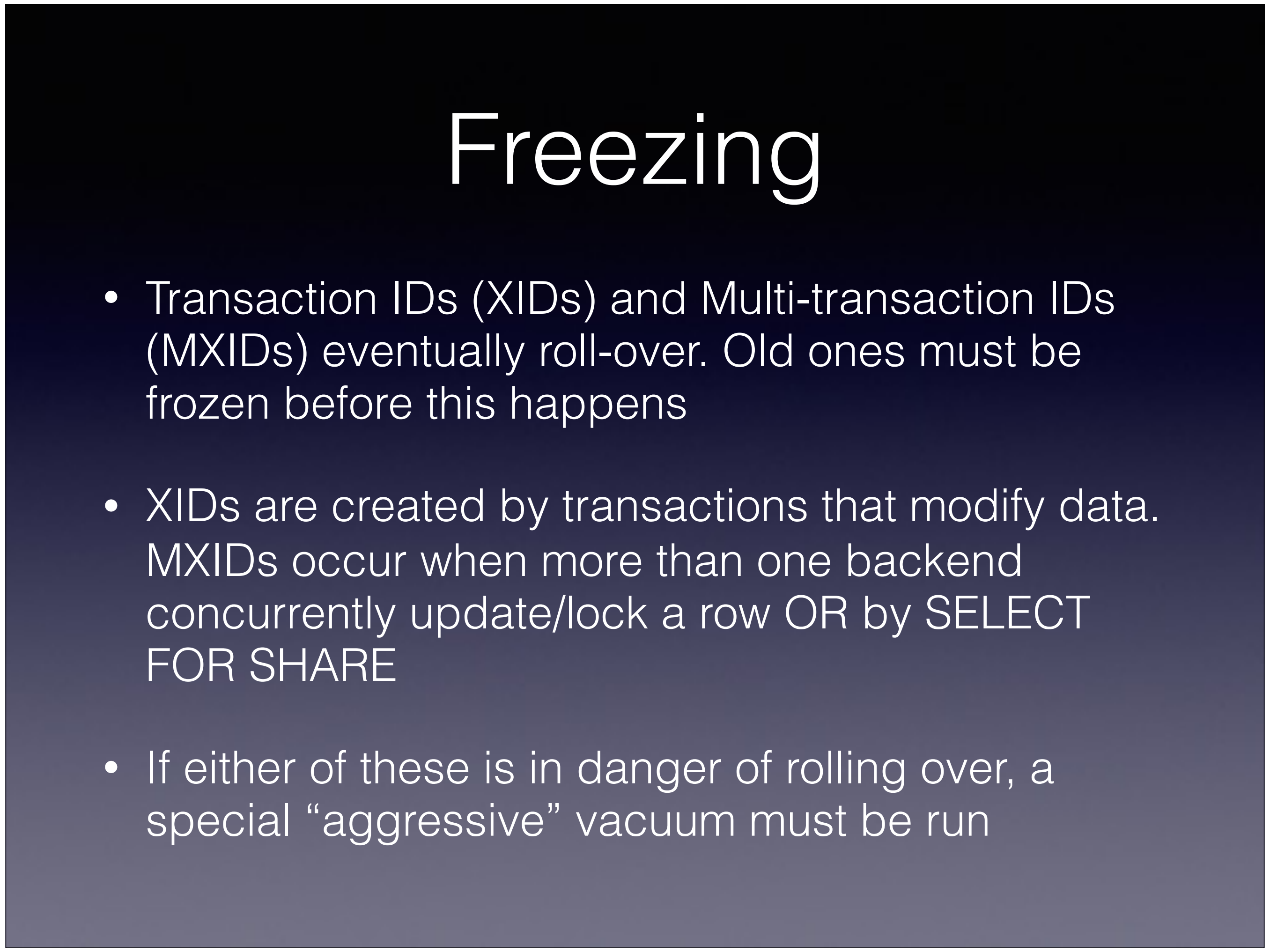


What can be vacuumed?

- Locking a buffer for cleaning has special requirements (`LockBufferForCleanup()`)
- It is safe for multiple backends to hold a reference to a buffer, and to do so for a long time
- This is not true when attempting to clean a page.
- Except for freezing, vacuum will give up on trying to clean a page if any other backend is referencing it

Every time a backend takes a reference to a buffer, in gets a “pin”. See src/backend/storage/buffer/README.

Long-running transactions prevent vacuuming from being effective

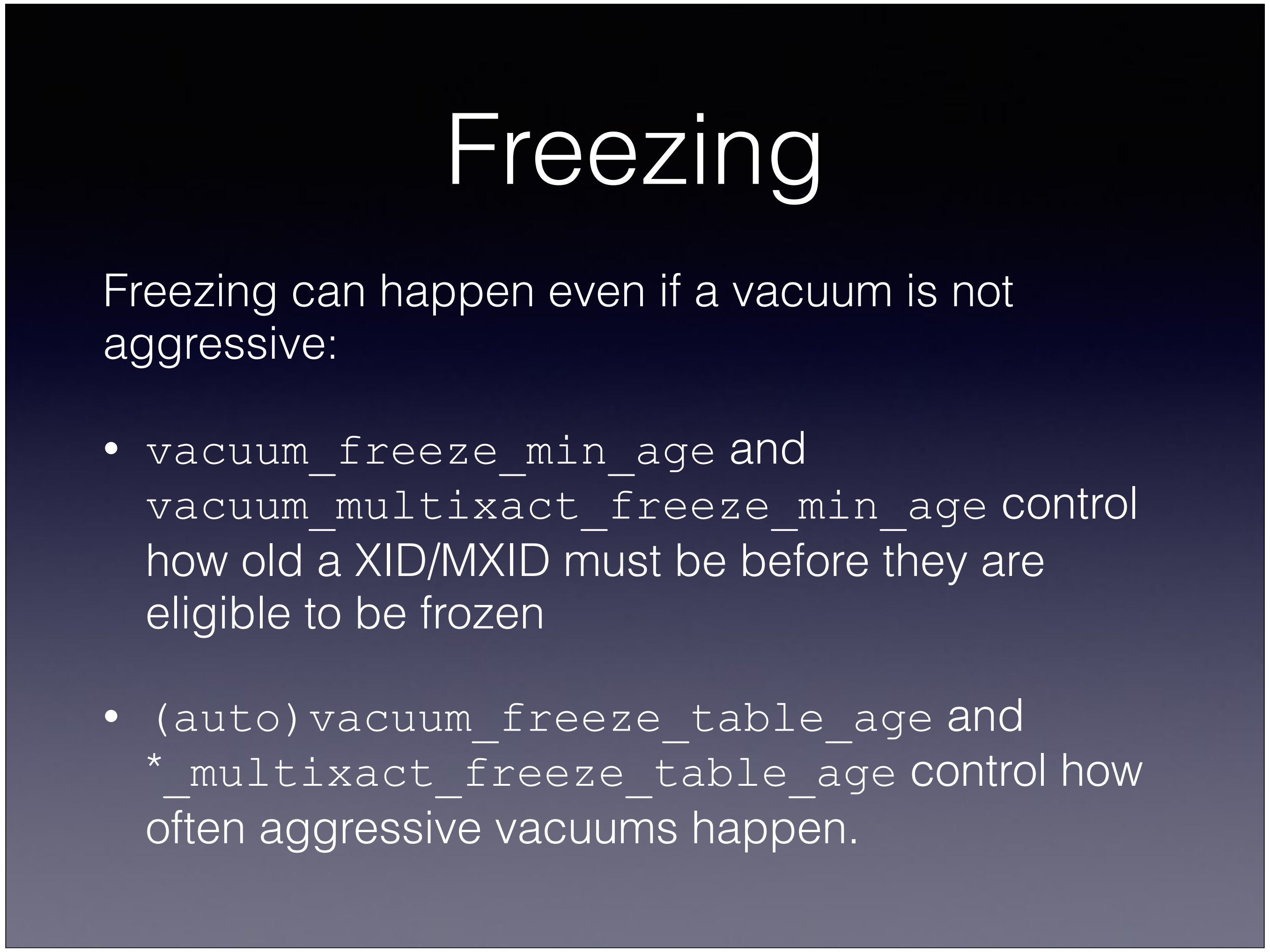


Freezing

- Transaction IDs (XIDs) and Multi-transaction IDs (MXIDs) eventually roll-over. Old ones must be frozen before this happens
- XIDs are created by transactions that modify data. MXIDs occur when more than one backend concurrently update/lock a row OR by SELECT FOR SHARE
- If either of these is in danger of rolling over, a special “aggressive” vacuum must be run

See src/backend/access/heap/README.tuplock

Extremely high rates of update transactions, FOR SHARE LOCK, or concurrent FK checks can cause freeze problems



Freezing

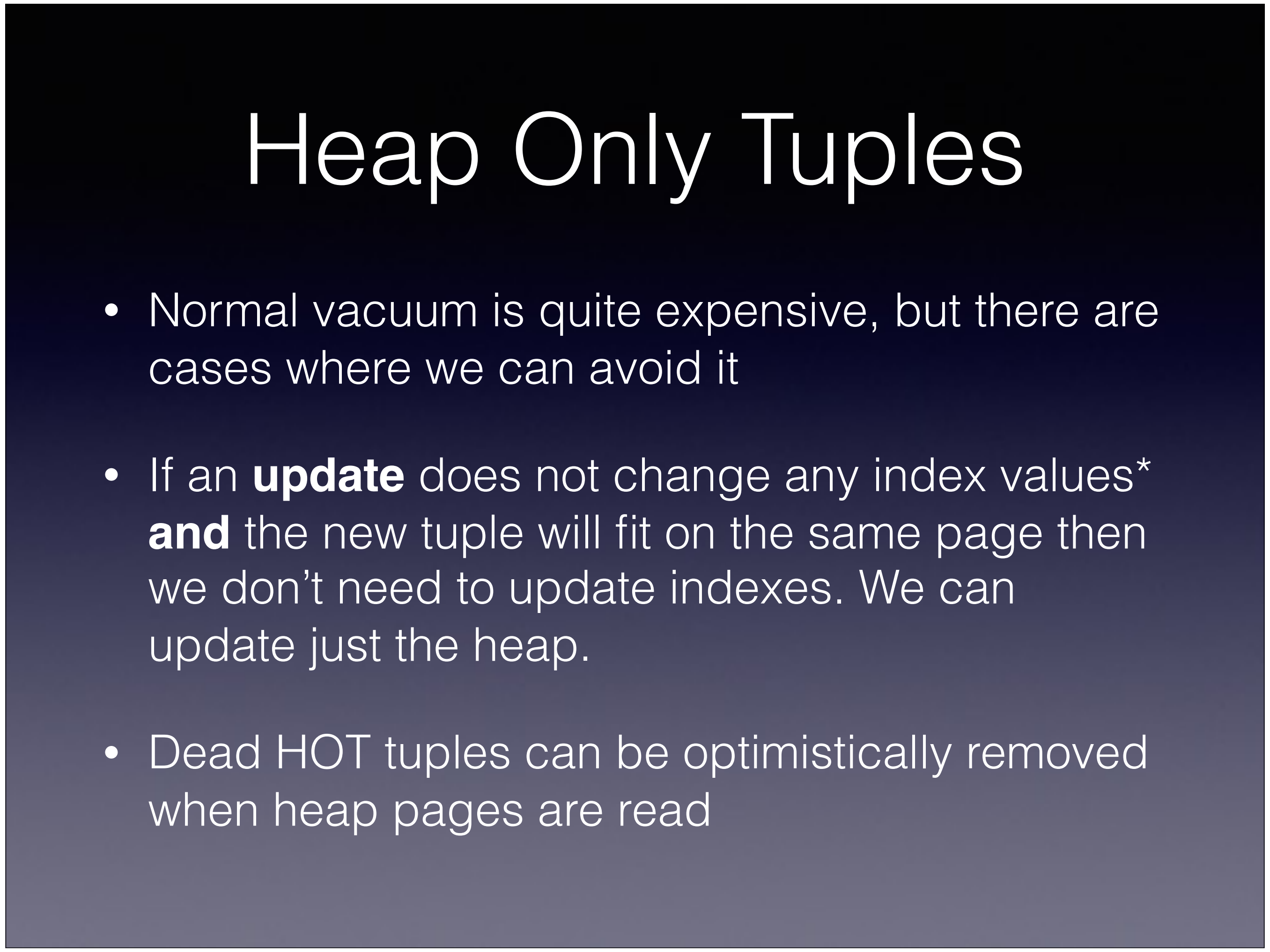
Freezing can happen even if a vacuum is not aggressive:

- `vacuum_freeze_min_age` and `vacuum_multixact_freeze_min_age` control how old a XID/MXID must be before they are eligible to be frozen
- `(auto)vacuum_freeze_table_age` and `*_multixact_freeze_table_age` control how often aggressive vacuums happen.

*_min_age are computed from the oldest running XID in the system.

*_table_age are computed from pg_class.relfrozenxid and pg_class.relminmxid.

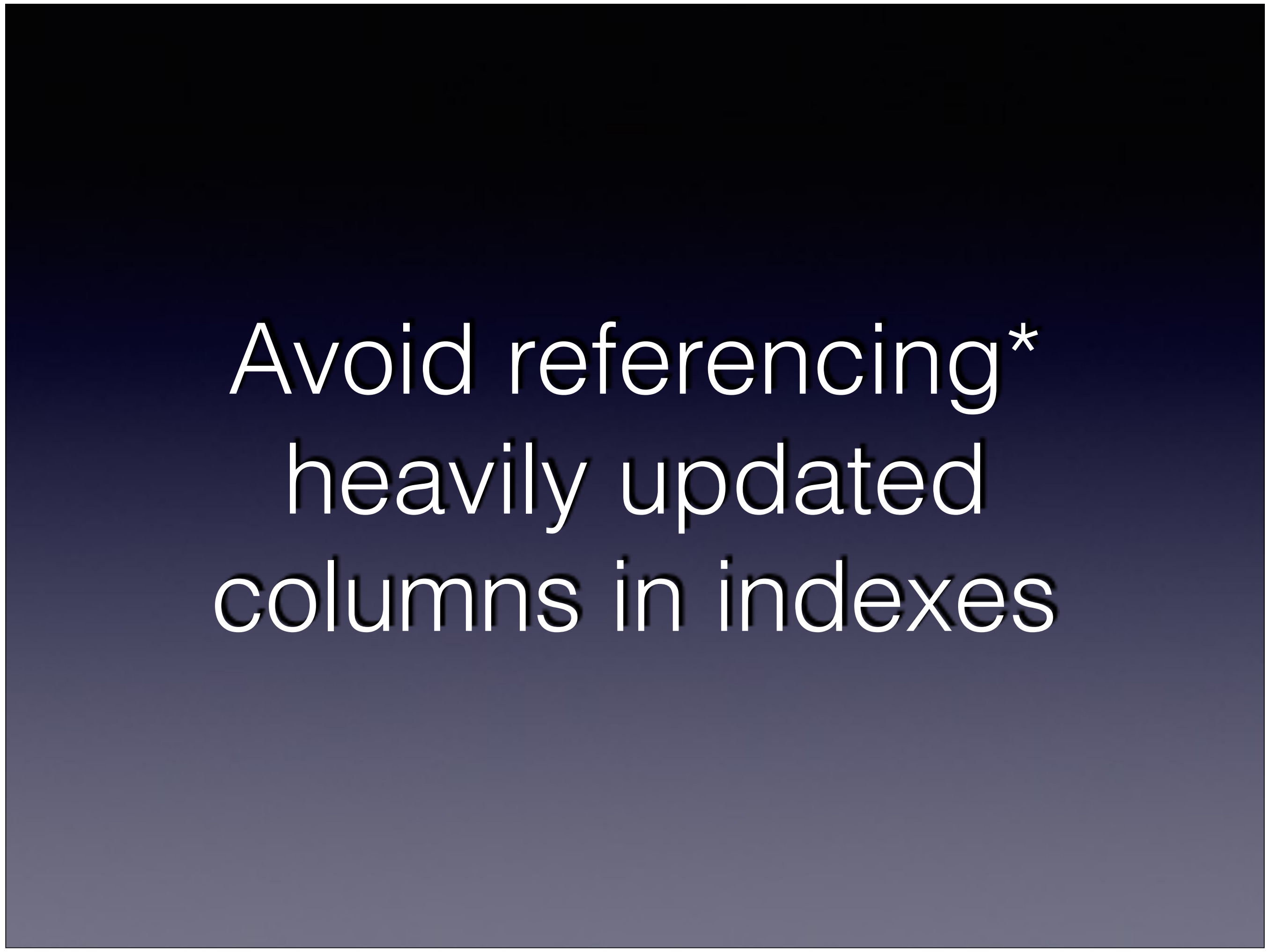
Long-running transactions prevent freezing from happening



Heap Only Tuples

- Normal vacuum is quite expensive, but there are cases where we can avoid it
- If an **update** does not change any index values* **and** the new tuple will fit on the same page then we don’t need to update indexes. We can update just the heap.
- Dead HOT tuples can be optimistically removed when heap pages are read

* Indexed values means any column referenced anywhere in an index, including predicates and functions. See src/backend/access/heap/README.HOT.



Avoid referencing* heavily updated columns in indexes

* Indexed values means any column referenced anywhere in an index, including predicates and functions. See src/backend/access/heap/README.HOT.

VACUUM

- There are 4 major variations on vacuuming
- `autovacuum` is a built-in process that attempts to automatically vacuum anything that needs it
- `VACUUM FULL` completely rebuilds a table from scratch
- `VACUUM` is a regular, manually run vacuum
- `VACUUM FREEZE` is a manual vacuum that forcibly freezes everything it can

autovacuum

- Generally doesn’t need to be tweaked in 9.6
- There is no way to control when it runs; do not attempt to do so with `autovacuum_naptime`
- If you do have slow periods (ie: weekends) running regular `vacuum` via cron helps autovacuum
- Frequent manual `vacuum` of heavily updated tables is still a good idea
- Documented in sections 2.1.6 and 18.10

VACUUM FULL

- Since 9.0, completely rebuilds table and indexes from scratch, similar to `cluster`
- **Takes an exclusive lock on table**
- Because this is essentially a Create Table As Select + indexes, it's not really vacuuming anything
- See also https://github.com/reorg/pg_repack

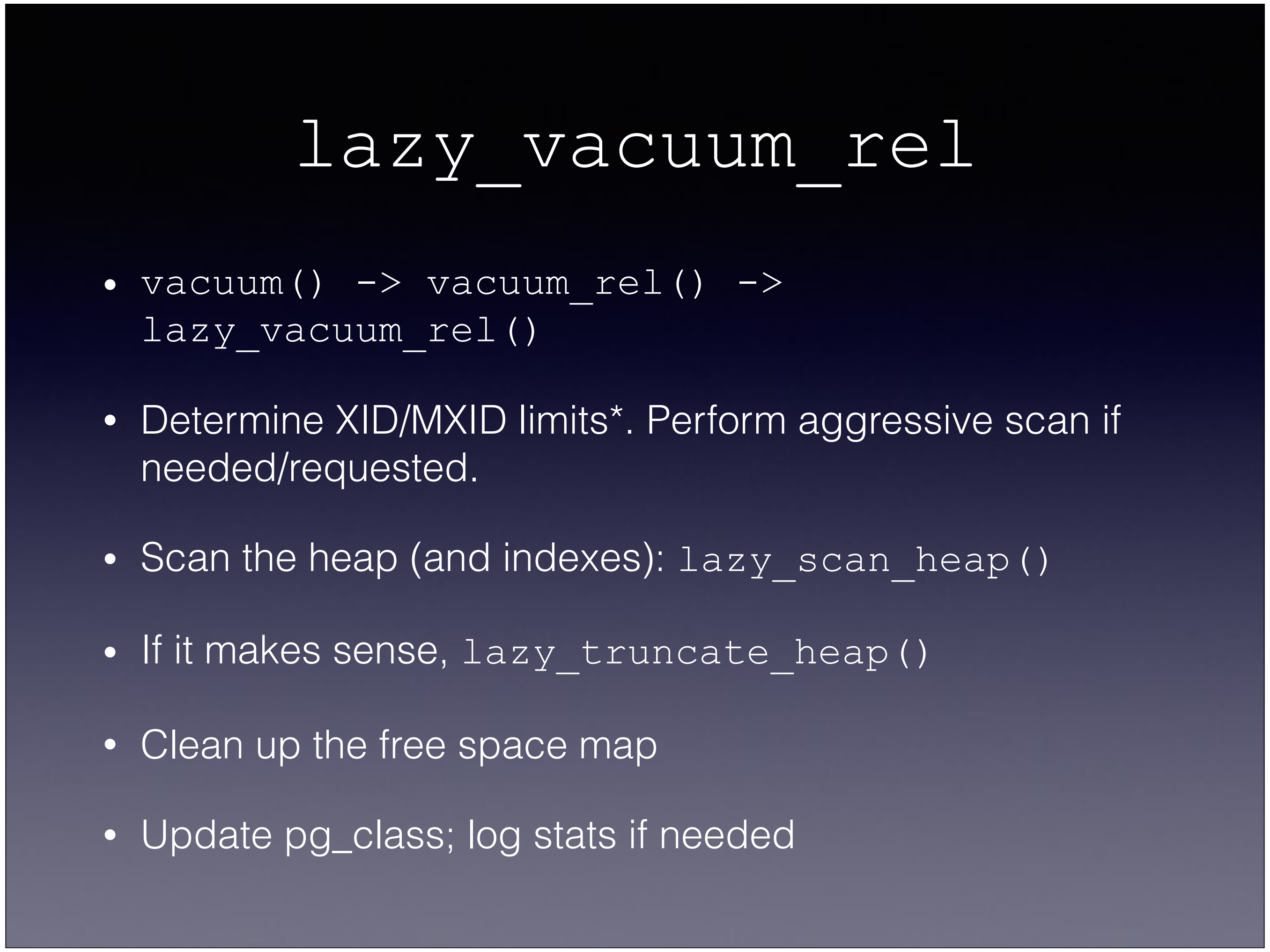
VACUUM

- Can not be run in a transaction (or function)
- See also `vacuumdb` shell command
- For each table
  - Scan heap, remembering tuples to remove
  - Scan indexes, removing tuples
  - Remove tuples from heap
  - If ANALYZE option specified, do analyze.
- Update datfrozenxmin and datminmxid

If `maintenance_work_mem` isn’t large enough, any vacuum (except full) must make multiple passes over indices

`vacuum_rel`

- `vacuum() -> vacuum_rel()`
- Vacuums a single relation
- Non-freeze autovacuum may skip relation
- Does a bunch of mundane stuff, then calls either `cluster_rel` (for a VACUUM FULL) or `lazy_vacuum_rel()`
- Before returning, calls itself to vacuum the TOAST table (but not for autovac)

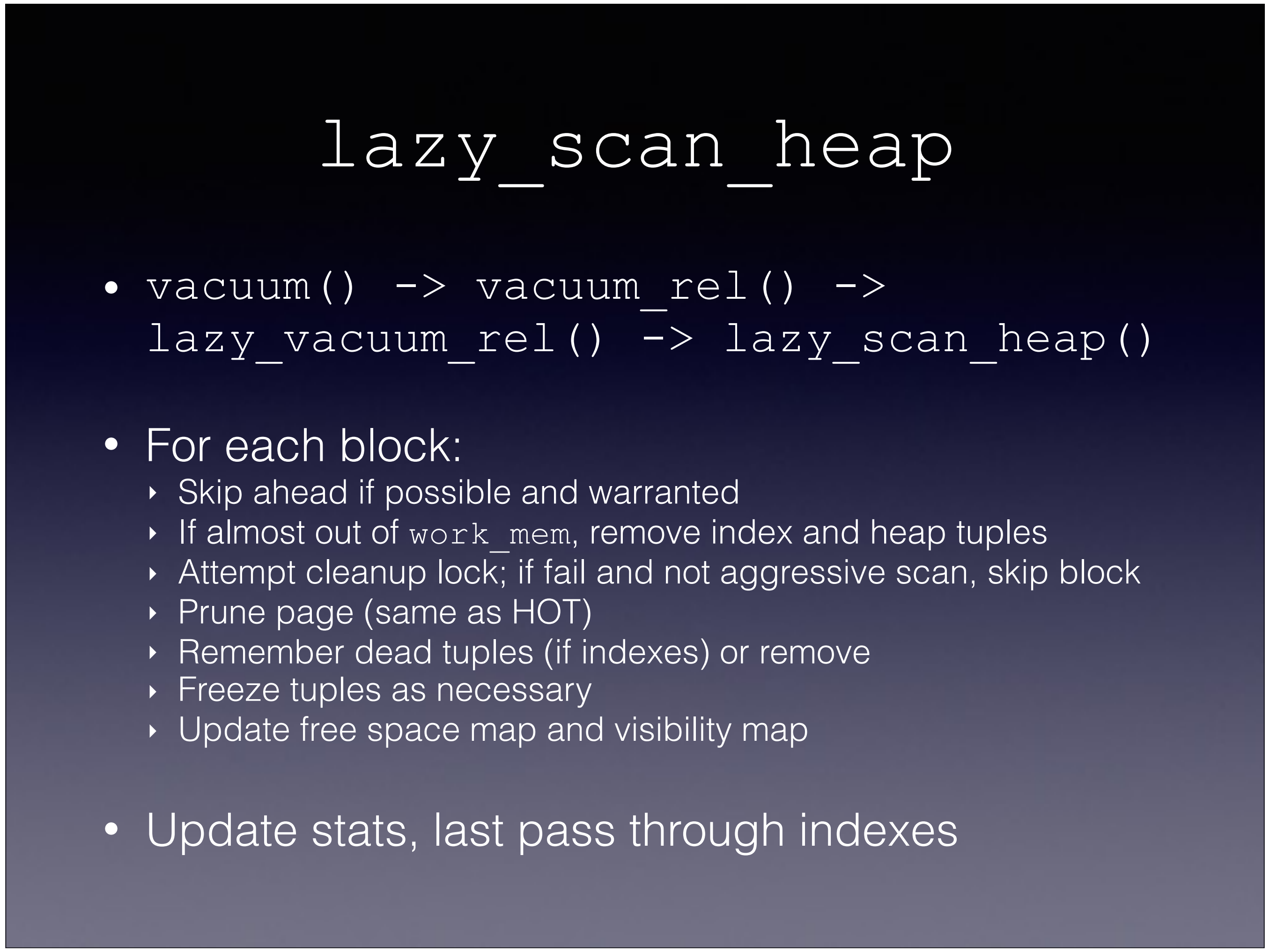


lazy_vacuum_rel

- `vacuum() -> vacuum_rel() -> lazy_vacuum_rel()`
- Determine XID/MXID limits*. Perform aggressive scan if needed/requested.
- Scan the heap (and indexes): `lazy_scan_heap()`
- If it makes sense, `lazy_truncate_heap()`
- Clean up the free space map
- Update pg_class; log stats if needed

There was a bug in some versions where we updated relfrozenxid and relminmxid even if we hadn’t scanned the whole table, potentially resulting in data loss.

* See vacuum_set_xid_limits()



lazy_scan_heap

- vacuum() -> vacuum_rel() -> lazy_vacuum_rel() -> lazy_scan_heap()
- For each block:
  - Skip ahead if possible and warranted
  - If almost out of `work_mem`, remove index and heap tuples
  - Attempt cleanup lock; if fail and not aggressive scan, skip block
  - Prune page (same as HOT)
  - Remember dead tuples (if indexes) or remove
  - Freeze tuples as necessary
  - Update free space map and visibility map
- Update stats, last pass through indexes

Block skipping depends on whether the vacuum is “aggressive” or not. If it’s aggressive, only blocks where all tuples are known to be frozen can be skipped. Otherwise, blocks that contain only visible tuples (which includes any all-frozen blocks) can be skipped.

lazy_vacuum_index

-
  ```
  vacuum() -> vacuum_rel() ->
  lazy_vacuum_rel() ->
  lazy_scan_heap() ->
  lazy_vacuum_index()
  ```

- Call index-specific vacuum method. These methods must scan the entire index, checking each index pointer against the list of remembered dead tuples

It is very difficult to reduce the size of a bloated index. Don’t let bloat happen, and if it does, reindex.

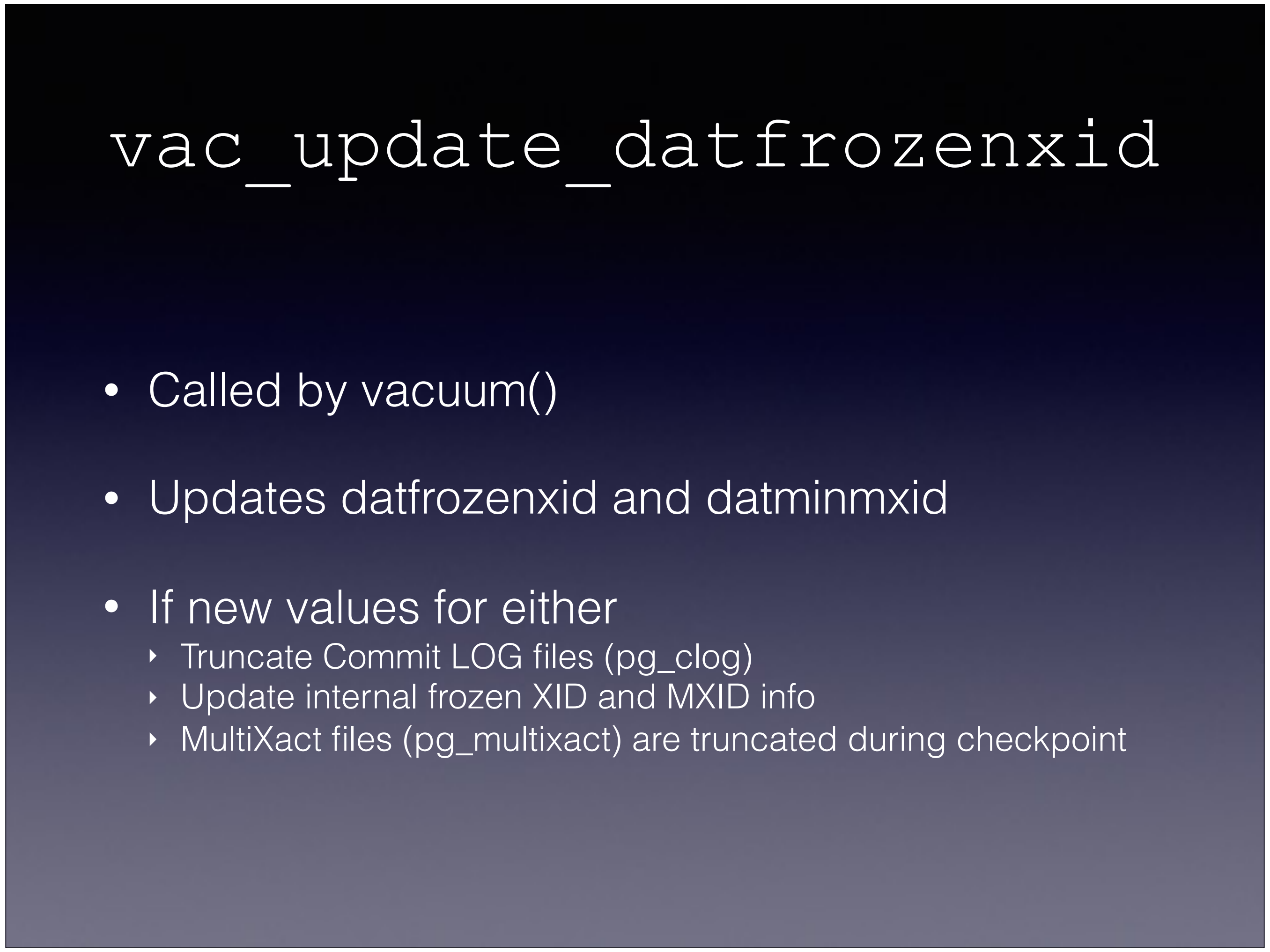
lazy_vacuum_heap

- vacuum() -> vacuum_rel() -> lazy_vacuum_rel() -> lazy_scan_heap() -> lazy_vacuum_heap()
- Removes marked-dead tuples from heap
- On each page, defragment page and record free space

lazy_cleanup_index

-
  ```
  vacuum() -> vacuum_rel() ->
  lazy_vacuum_rel() ->
  lazy_scan_heap() ->
  lazy_cleanup_index()
  ```

- Call index-specific cleanup method. For btree, this simply cleans up the index free space map.



vac_update_datfrozenxid

- Called by vacuum()
- Updates datfrozenxid and datminmxid
- If new values for either
  - Truncate Commit LOG files (pg_clog)
  - Update internal frozen XID and MXID info
  - MultiXact files (pg_multixact) are truncated during checkpoint

See ForceTransactionIdLimitUpdate() and vac_truncate_clog()

autovacuum

- Has two parts, launcher and workers
- The launcher prioritizes databases by
  - Most in need of XID freeze (usually none)
  - Most in need of MXID freeze (usually none)
  - Least recently autovacuumed, skipping any database vacuumed less than `autovacuum_naptime` ago.
- Multiple workers can work on the same database at once
- Workers will be canceled if they interfere with other backends
- You can compare how many autovacuum workers are running against `autovacuum_max_workers` to see if autovaccum is running into problems.

autovacuum worker

- Get list of heap tables & materialized views that need vacuum or analyze
- Get list of TOAST tables that need vacuuming
- TEMP tables are ignored
- For each relation; attempt to get lock. Skip if unavailable (unless freeze is needed)
- vac_update_datfrozenxid
- exit

autovacuum does not prioritize tables within a database

autovacuum can
become ineffective for
high demand tables if
too many large tables
need vacuum at once

vacuum cost delay

- Well documented; please see section 19.4.4

- The critical idea is that once we hit (auto)vacuum_cost_limit we sleep for (auto)vacuum_cost_delay. Increasing limit speeds vacuum; increasing delay slows vacuum.

- Don’t slow vacuum too much

- On many systems you should set page_dirty lower than page_miss

In closing…

- Long running transactions hurt vacuum
- High transaction rates, use of FOR SHARE LOCK and concurrent FK checks increase the need to FREEZE
- Indexes referencing heavily updated columns prevent HOT
- Make maintenance_work_mem large for vacuum
- It's very difficult to reduce the size of a bloated index
- autovacuum can only do so much

Questions?

Jim.Nasby@OpenSCG.com

https://github.com/decibel/presentations