# Nine Circles of Inferno
# or Explaining the PostgreSQL Vacuum.

PgDay 2016, Saint Petersburg

Lesovsky Alexey
lesovsky@pgco.me

# Outline.

MVCC Basics

Circle I. Postmaster.

Circle II. Postmaster and Autovacuum Launcher.

Circle III. Autovacuum Launcher and Workers.

Circle IV. Autovacuum Workers.

Circle V. Process a single database.

Circle VI. Prepare for Vacuum.

Circle VII. Process one heap relation.

Circle VIII. Scan heap relation.

Circle IX. Vacuum heap relation.

# Multiversion Concurrency Control (MVCC).

Multiversion Concurrency Control:

1. Allows to offer high concurrency;

2. During significant database read/write activity;

3. Readers never block writers and writers never block readers.

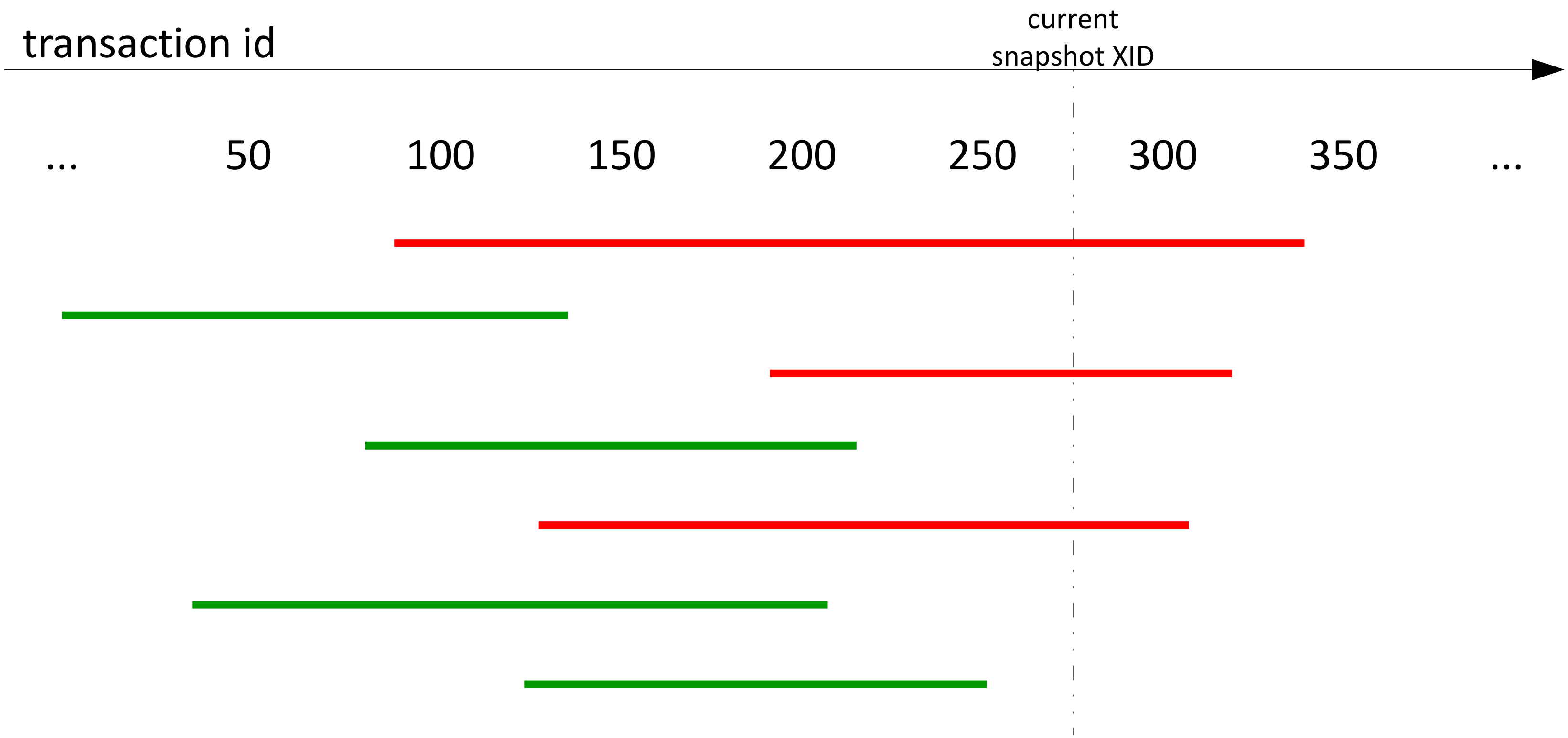
# Multiversion Concurrency Control (MVCC).

| | | |
|---|---|---|
| created: 123<br>deleted: | insert row | INSERT by 123 |
| created: 123<br>deleted: 456 | delete old version | UPDATE by 456 |
| created: 456<br>deleted: | insert new version | |
| created: 456<br>deleted: 789 | delete row | DELETE by 789 |

# Multiversion Concurrency Control (MVCC).

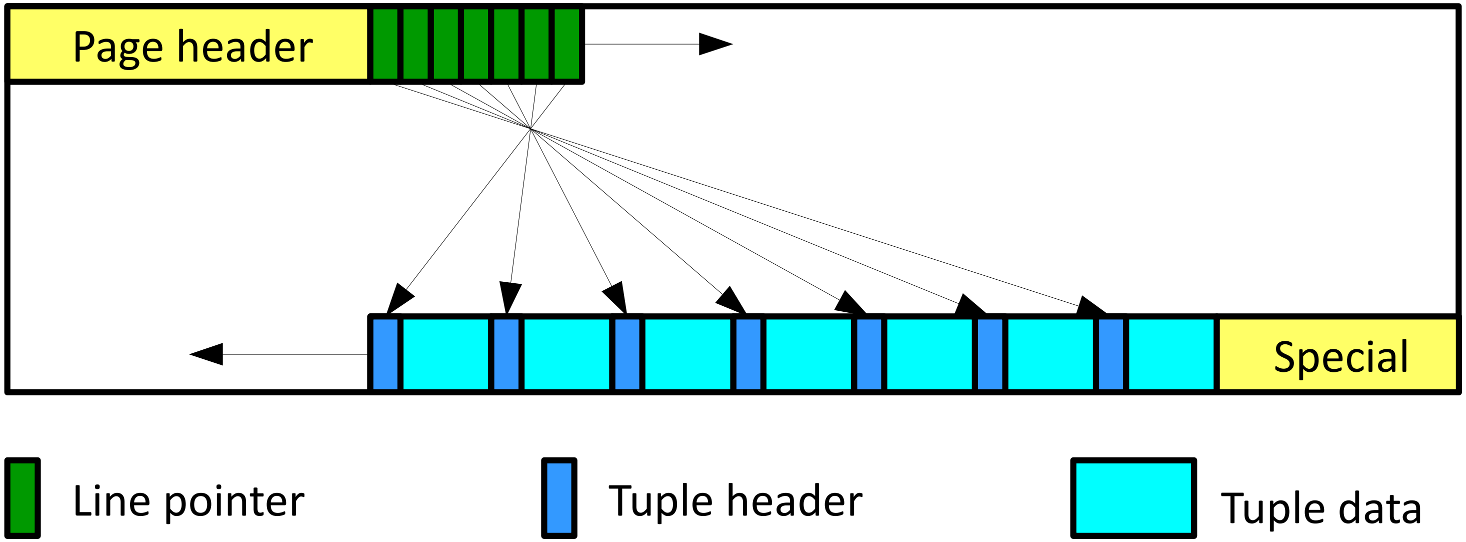
```
/*
 * information stored in t_infomask:
 */
...
#define HEAP_XMIN_COMMITTED   0x0100        /* t_xmin committed */
#define HEAP_XMIN_INVALID     0x0200        /* t_xmin invalid/aborted */
#define HEAP_XMIN_FROZEN      (HEAP_XMIN_COMMITTED|HEAP_XMIN_INVALID)
#define HEAP_XMAX_COMMITTED   0x0400        /* t_xmax committed */
#define HEAP_XMAX_INVALID     0x0800        /* t_xmax invalid/aborted */
#define HEAP_XMAX_IS_MULTI    0x1000        /* t_xmax is a MultiXactId */
```

# Multiversion Concurrency Control (MVCC).

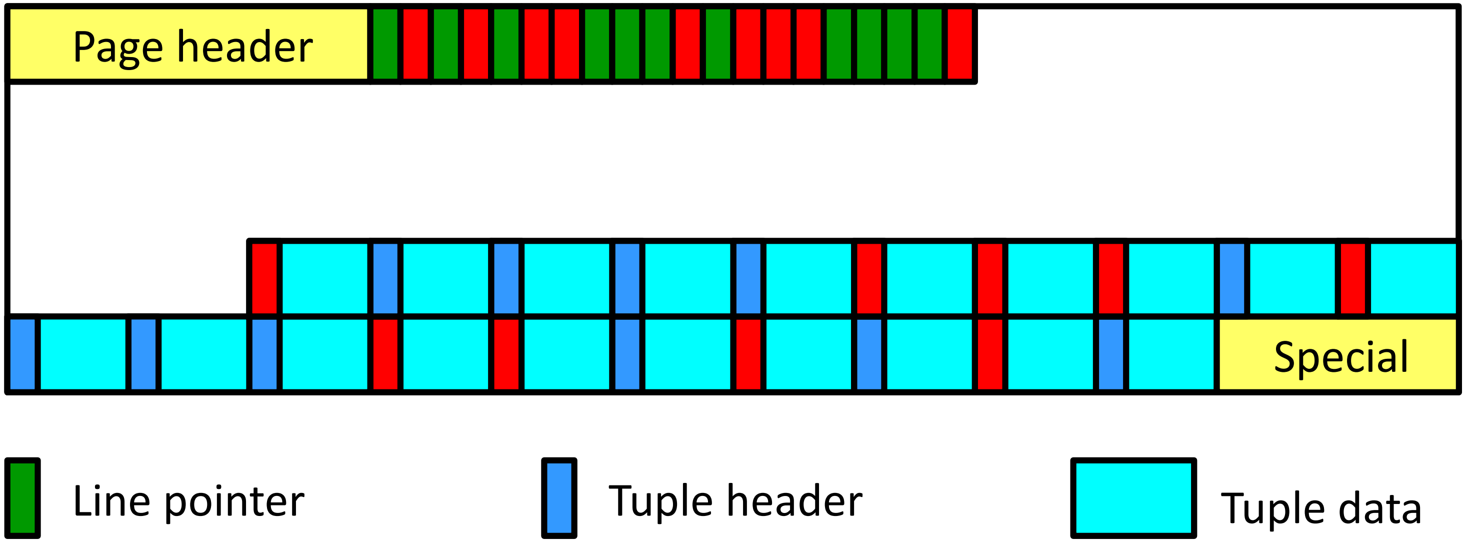


**Green** is visible, **Red** is not visible.

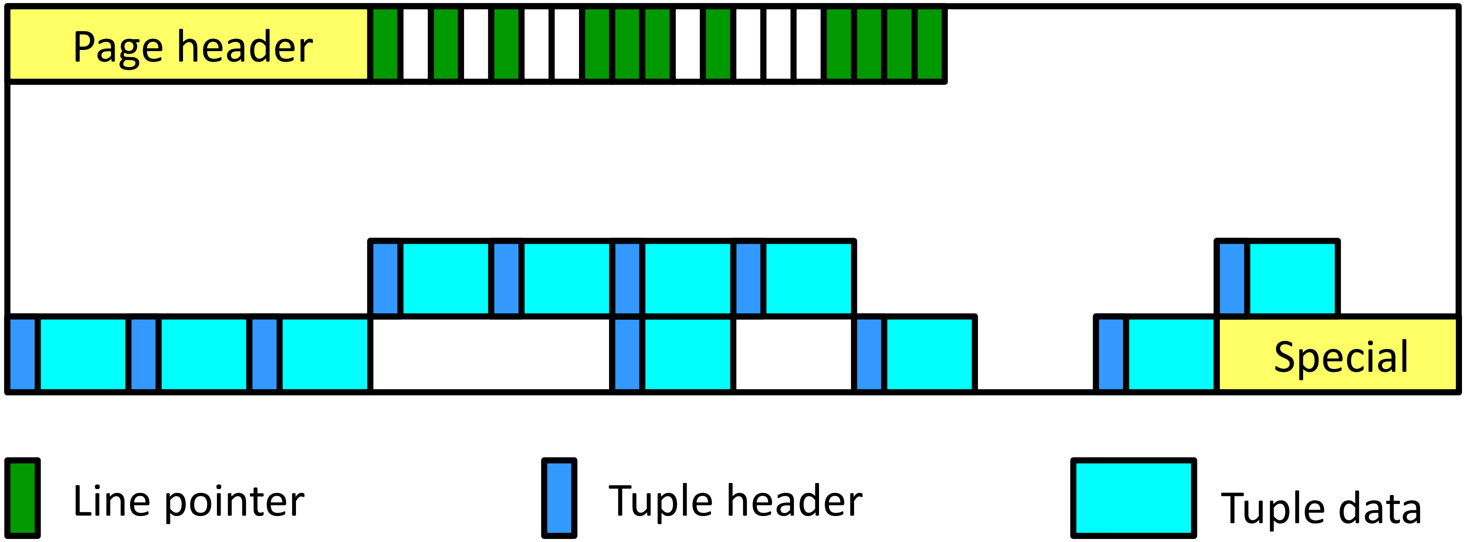
# Multiversion Concurrency Control (MVCC).



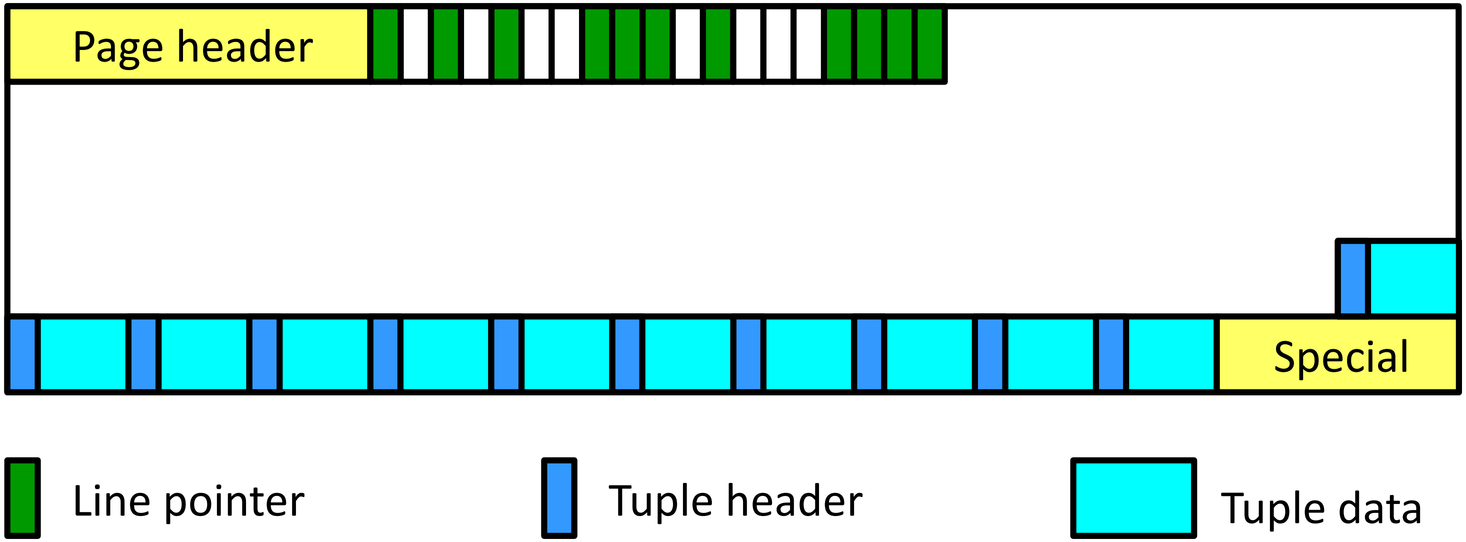
# Multiversion Concurrency Control (MVCC).



# Multiversion Concurrency Control (MVCC).



# Multiversion Concurrency Control (MVCC).



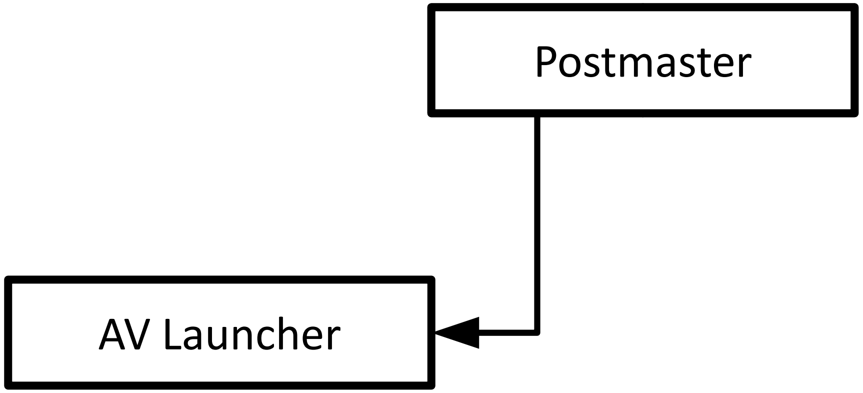
# MVCC. Questions?

Questions?

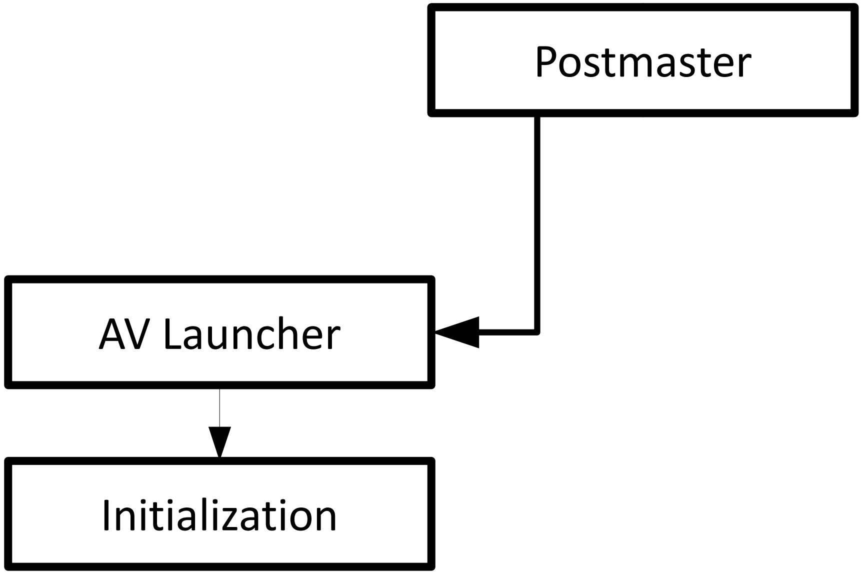
# I. Postmaster.

```
/*
 * postmaster.c
 *    This program acts as a clearing house for requests to the
 *    POSTGRES system.  Frontend programs send a startup message
 *    to the Postmaster and the postmaster uses the info in the
 *    message to setup a backend process.
 */
```

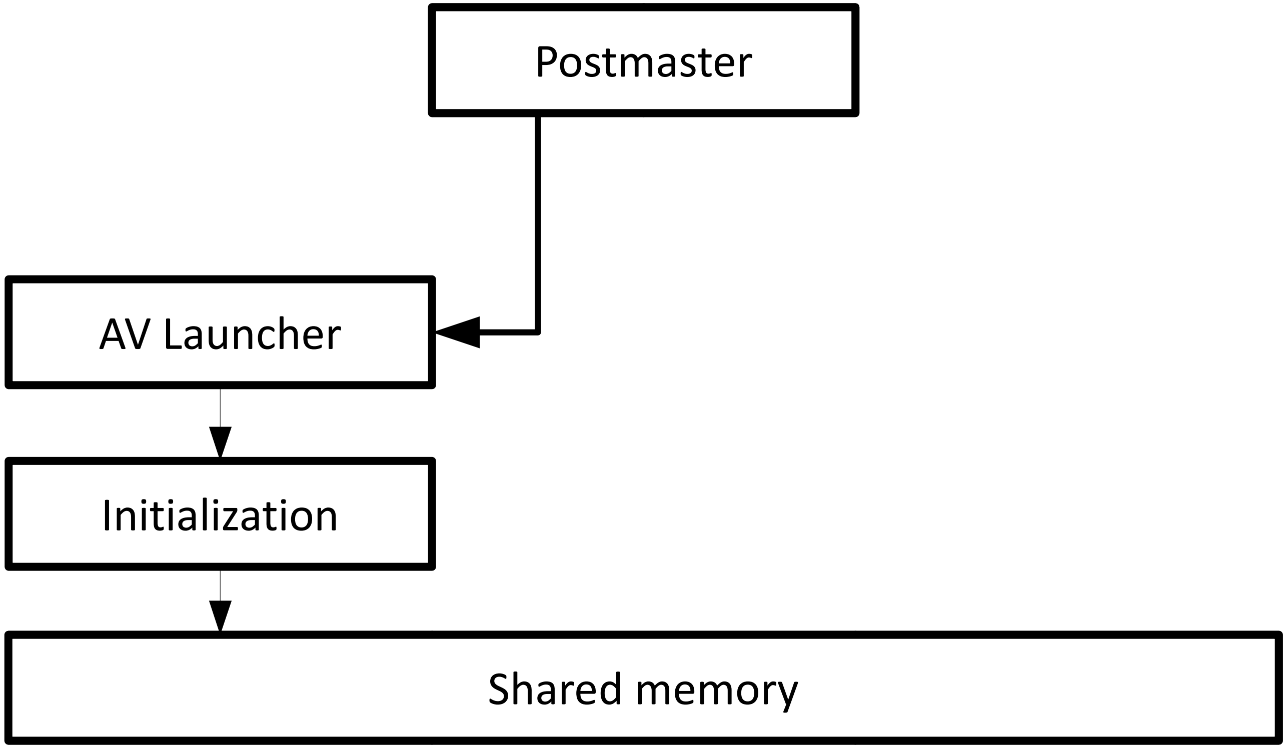
# I. Postmaster.



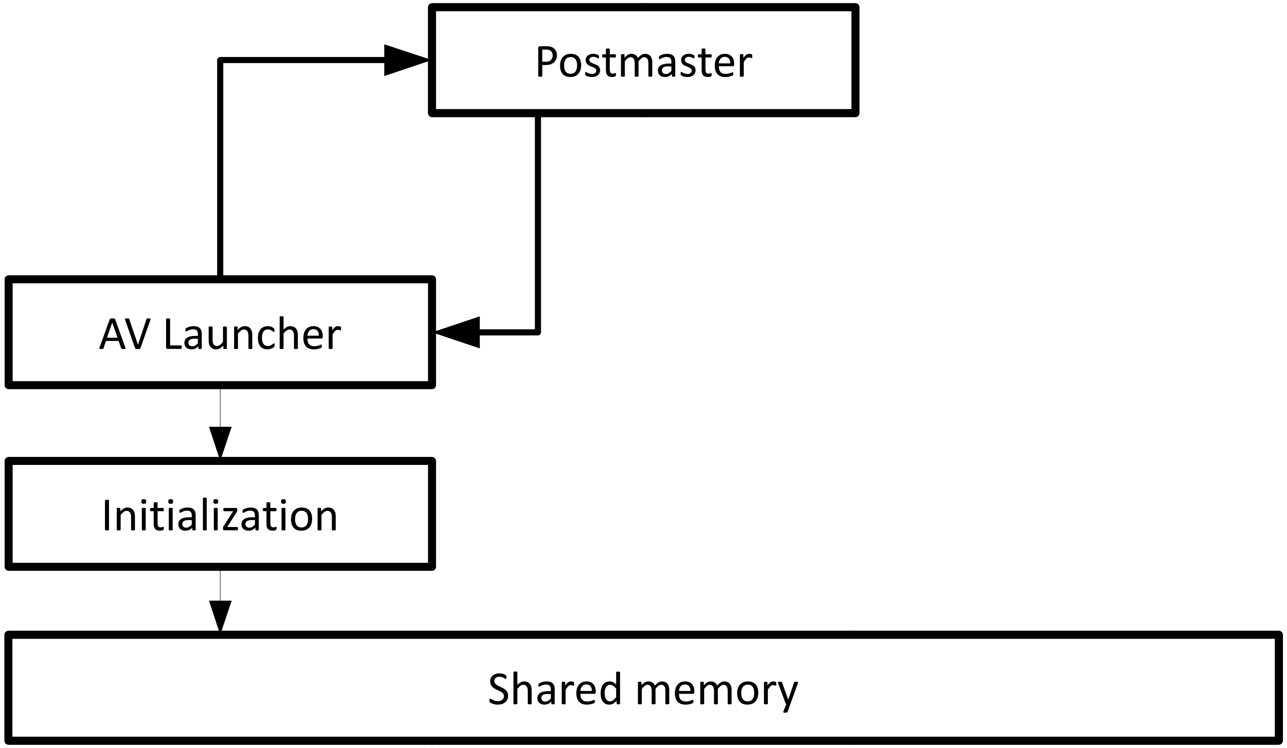
# I. Postmaster.



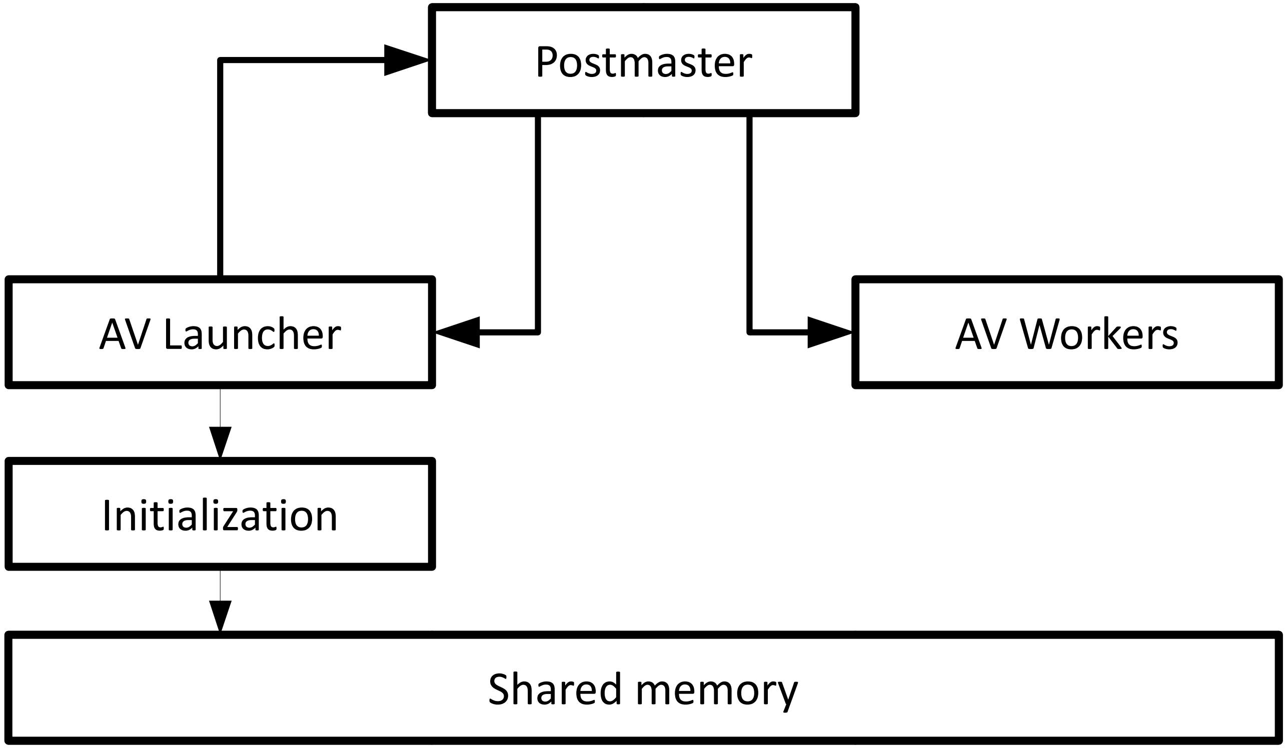
# I. Postmaster.



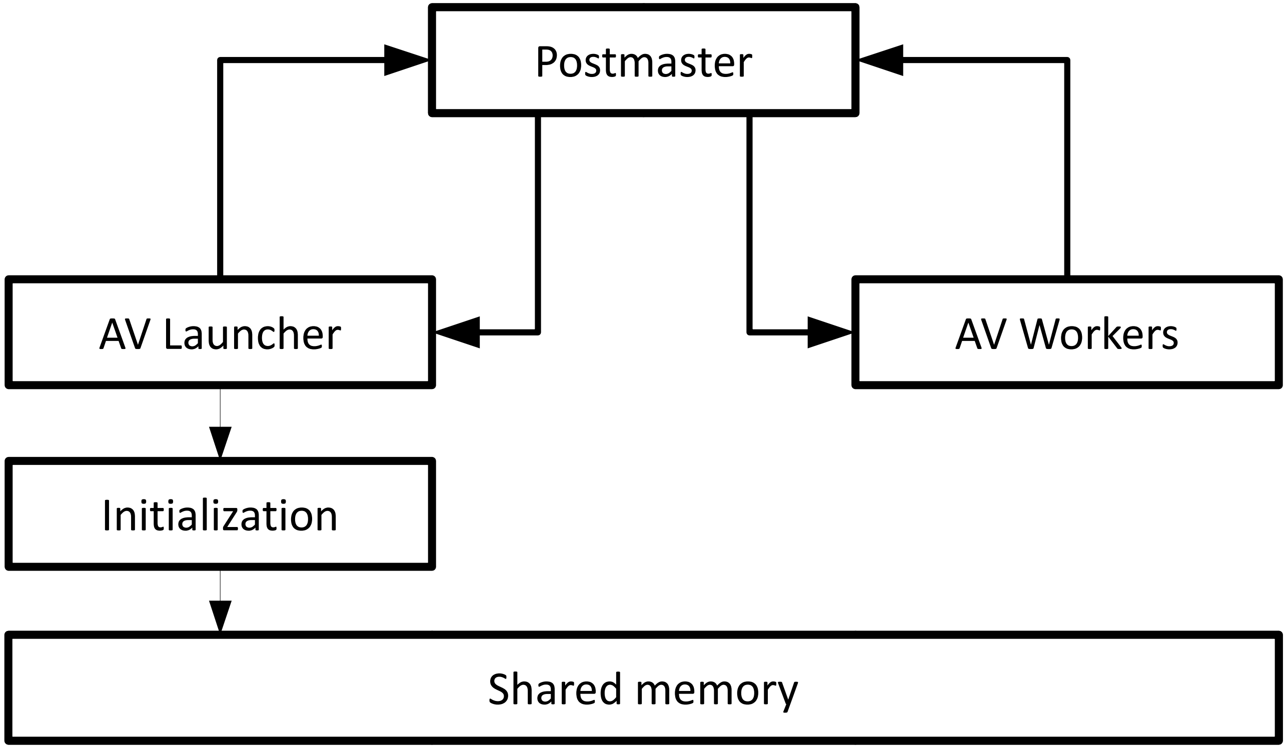
# I. Postmaster.



# I. Postmaster.



# I. Postmaster.



# I. Postmaster, briefly.

autovac_init()

    Check **track_counts**, or

    WARNING: "`autovacuum not started because of misconfiguration`".

ServerLoop() – infinite loop:

    Run background processes (checkpointer, bgwrter, walwriter);

    Run autovacuum launcher;

    Other stuff...

# I. Postmaster, briefly.

autovac_init()

Check **track_counts**, or

WARNING: "`autovacuum not started because of misconfiguration`".

ServerLoop() – infinite loop:

Run background processes (checkpointer, bgwrter, walwriter);

Run autovacuum launcher;

Other stuff...

AV Launcher will restarted on the next itertaion, if current attempt is failed.

(AV Launcher is not starting in binary upgrade mode)

# I. Postmaster, briefly.

autovac_init()

- Check **track_counts**, or
- WARNING: "`autovacuum not started because of misconfiguration`".

ServerLoop() – infinite loop:

- Run background processes (checkpointer, bgwrter, walwriter);
- Run autovacuum launcher;
- Other stuff...

AV Launcher will restrted on the next itertaion, if current attempt is failed.
(AV Launcher is not starting in binary upgrade mode)

fork()

# I. Postmaster, briefly.

```
/usr/pgsql-9.5/bin/postgres -D /var/lib/pgsql/9.6/data
  \_ postgres: logger process
  \_ postgres: checkpointer process
  \_ postgres: writer process
  \_ postgres: wal writer process
  \_ postgres: autovacuum launcher process
  \_ postgres: stats collector process
```

# II. AutoVacLauncherMain()... I'm the Launcher now.

AutoVacLauncherMain() base initialization:

- Connecting to semaphores and shared memory;
- File descriptors for debug and input/output;
- Init file, storage, buffer managers;
- Self-registering in shared memory, create work structures.

# II. AutoVacLauncherMain()... I'm the Launcher now.

Init as Postgres backend:

- Adding to ProcArray and ProcSignal;
- Finish buffer pool initialization;
- Access to XLOG;
- Init Relation-, Catalog-, Plan- caches, allow PortalManager;
- Init stats and fill RelationCache.

Create new memory context and switch to.

# II. AutoVacLauncherMain()... When something goes wrong.

Error handling:

- Reset timeouts;
- Write error message to the server log;
- Abort current transaction;
- Switch to main memory context;
- Reset error context;
- Reset and remove all children memory contexts;
- Reset stats snapshot;

Prevent all interrupts during error handling.

# II. AutoVacLauncherMain()... Preparing to work.

Set options:

- zero_damaged_pages=false, default_transaction_isolation="read commited";
- statement_timeout=0, lock_timeout=0;

Start worker immediately if running in emergency mode.

Build database list.

# II. AutoVacLauncherMain()... Preparing to work.

Build database list - rebuild_database_list():

- Start workers for all databases during naptime interval (but min. 110ms);
- Sort by next_worker (desc).

# II. AutoVacLauncherMain()... Main loop.

Loop until shutdown request (SIGTERM):

- If free workers available, determine sleep interval;
- Sleep with WaitLatch and post-sleep signal handling:
  - If postmaster died – exit immediately;
  - SIGTERM – graceful shutdown with "`autovacuum launcher shutting down`";
  - SIGHUP – reload config, rebalance costs, rebuild database list (changed naptime?);
  - SIGUSR2 – worker finished or worker startup failed... sleep 1s, try restart worker.

# II. AutoVacLauncherMain()... Main loop.

Loop until shutdown request (SIGTERM):

- Check free workers list;
- Check worker status in "startingWorker” stage
  - If worker stucks more than 60s (or naptime), cancel worker with "`worker took too long to start; canceled`", otherwise skip iteration.

## II. AutoVacLauncherMain()... Main loop.

When all conditions are satisfied we can have two cases:

- Normal
  - Get database from list and compare next_worker with current time.
    - Run worker or skip iteration.

# II. AutoVacLauncherMain()... Main loop.

When all conditions are satisfied we can have two cases:

- Normal
  - Get database from list and compare next_worker with current time.
    - Run worker or skip iteration.
- First start after initdb (there is no stat and database list is empty)
  - Run worker as is.

# II. AutoVacLauncherMain()... Launch worker.

Functions launch_worker() and do_start_worker()

Get database change next_worker (now() + naptime) and place it to the list head.

Rebuild list, if database is not in the list.

# II. AutoVacLauncherMain()... Launch worker.

do_start_worker():

- Check number of free workers. Exit silently if there is no free workers.
- Create memory context and switch to. Get fresh stats snapshot. Build own databases list.
- Get recent transaction ID, determine xidForceLimit and multiForceLimit
  - recentXid – autovacuum_freeze_max_age
- Choose a database.

# II. AutoVacLauncherMain()... Launch worker.

do_start_worker():

- Check number of free workers. Exit silently if there is no free workers.
- Create memory context and switch to. Get fresh stats snapshot. Build own databases list.
- Get recent transaction ID, determine xidForceLimit and multiForceLimit
  - recentXid – autovacuum_freeze_max_age
- Choose a database:
  - Database with wraparound risk with oldest datfrozenxid;
  - Database with wraparound risk with oldest datminmxid;
  - Database with oldest autovacuum time;
- Skip recently-vacuumed databases and databases without stats.

# II. AutoVacLauncherMain()... Launch worker.

At this moment the database candidate should be determined.

- If no candidate – rebuild database list, exit from function.

Update shared memory structures (freeWorkers, database name, launch time)

Place these info into startingWorker structure.

Send signal to the postmaster (setup flag in shared memory and send SIGUSR1).

# III. AutoVacLauncherMain()... Launch worker.

```
/* We're OK to start a new worker */
```

# III. Postmaster again.

sigusr1_handler – action is depends on flag in shared memory:

PMSIGNAL_BACKGROUND_WORKER_CHANGE

PMSIGNAL_RECOVERY_STARTED

PMSIGNAL_BEGIN_HOT_STANDBY

PMSIGNAL_ROTATE_LOGFILE

PMSIGNAL_START_AUTOVAC_WORKER – StartAutovacuumWorker()

PMSIGNAL_START_WALRECEIVER

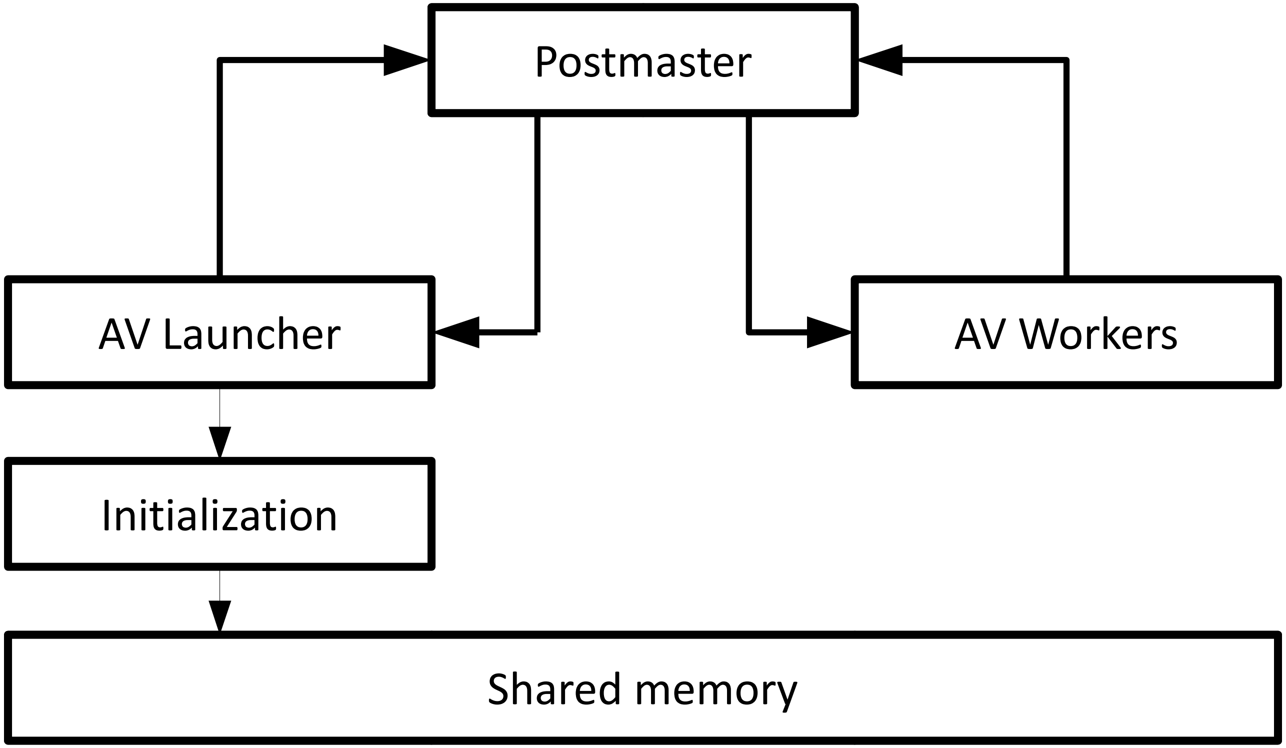
...

# III. Postmaster. StartAutovacuumWorker().

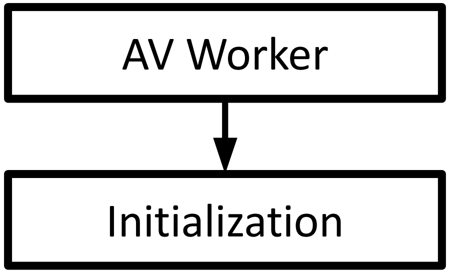
StartAutovacuumWorker():

- Is allowed to accept new connections?
  - Do not accept in startup/shutdown/inconsistent recovery state/limit reached.
  - If denied, set failed flag and signal AV Launcher (SIGUSR2).
- Allocate memory and slot for backend, fork() inside StartAutoVacWorker();
- If fork() is successful, set BACKEND_TYPE_AUTOVAC to backend.
  - If failed, "`could not fork autovacuum worker process: %m`".
  - Or init postmaster child, close postmaster sockets and run AutoVacWorkerMain().
- Exit from function.

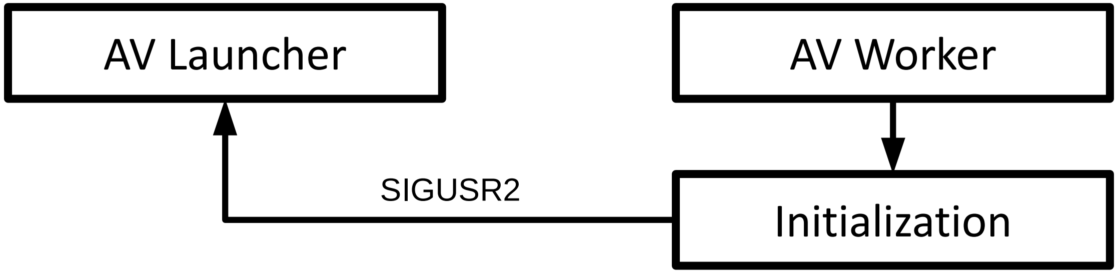
# III. Questions?



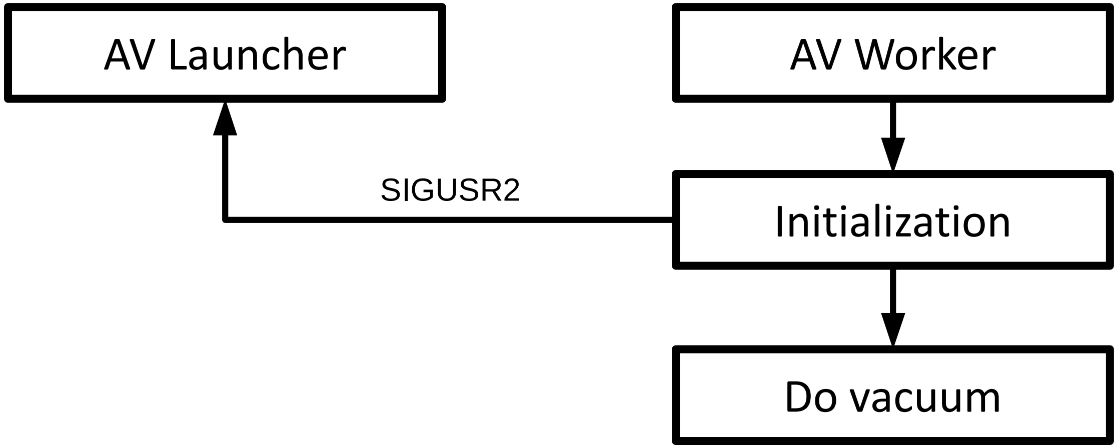
# IV. Autovacuum Worker.



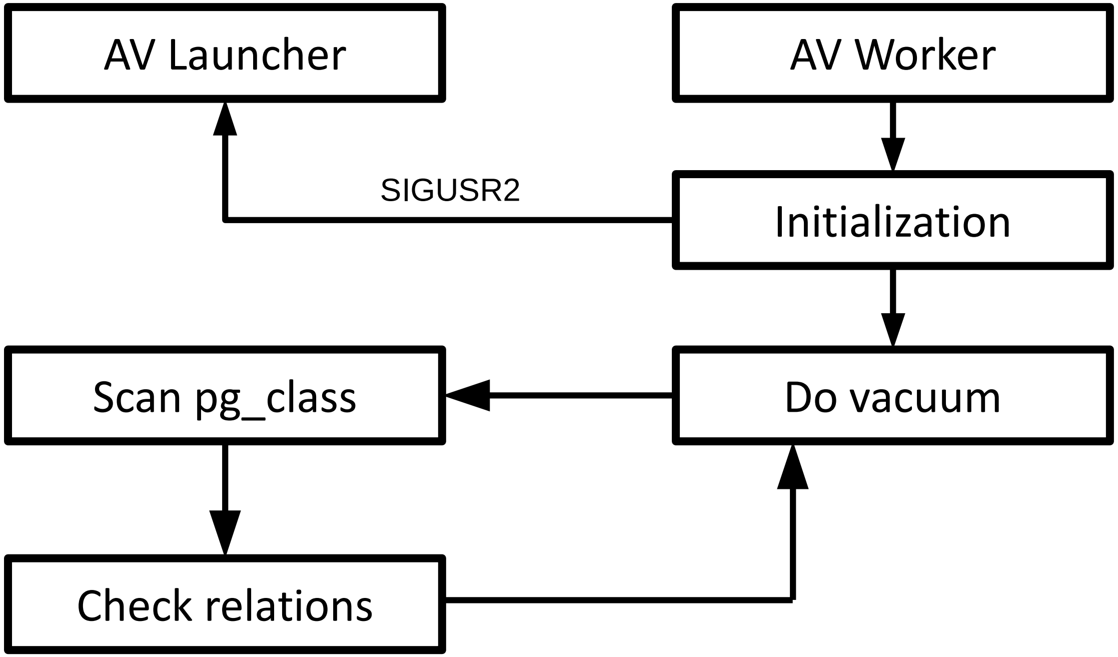
# IV. Autovacuum Worker.



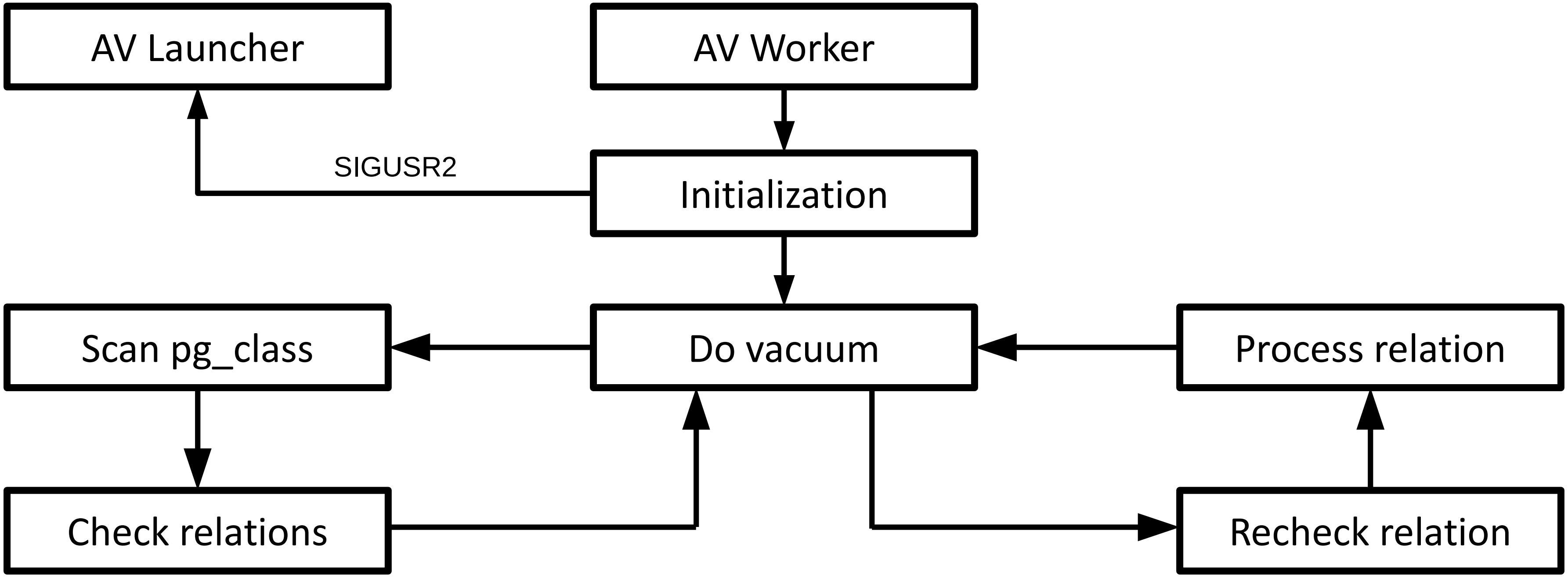
# IV. Autovacuum Worker.



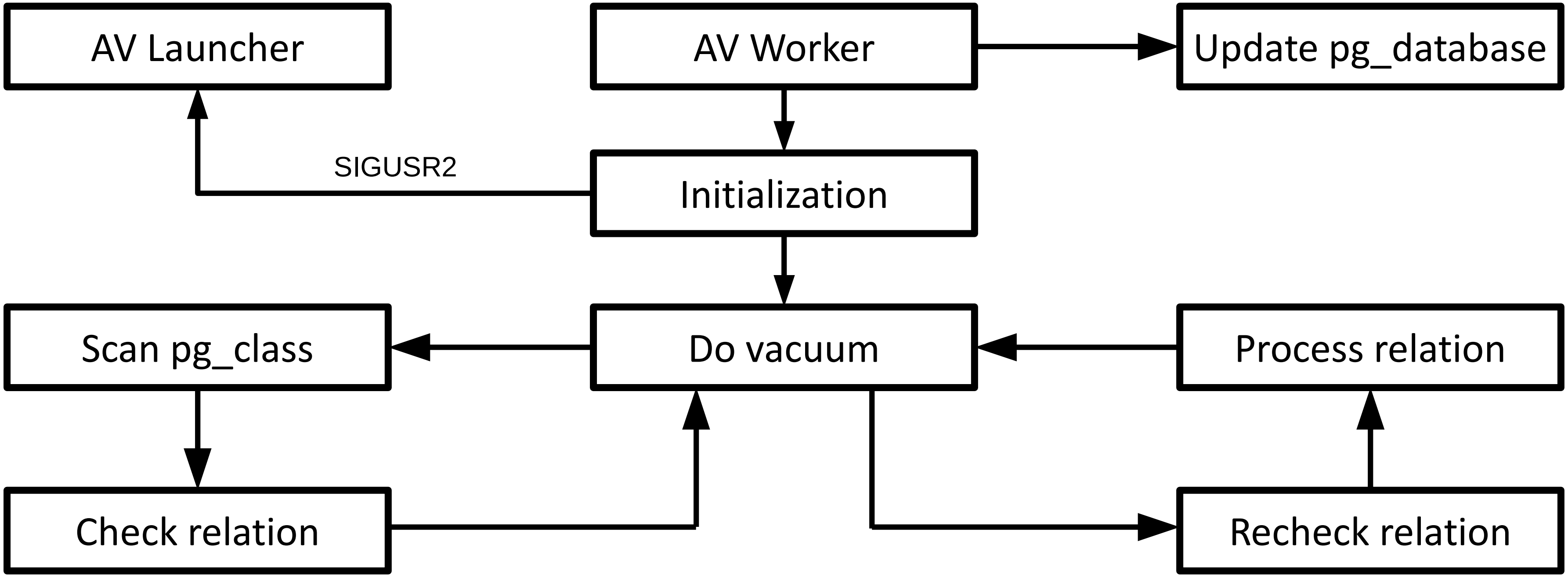
# IV. Autovacuum Worker.



# IV. Autovacuum Worker.



# IV. Autovacuum Worker.



# IV. AutoVacWorkerMain().

Base init (signals, file descriptors, filemgr, bufmgr, smgr, shm, local struct);

Set zero_damaged_pages=false, statement_timeout=0, lock_timeout=0;

Set default_transaction_isolation="read commited", synchronous_commit=local;

Get database name from av_startingWorker;

Set itself in runningWorkers list and reset av_startingWorker;

Send SIGUSR2 to AV Launcher.

# IV. AutoVacWorkerMain().

Base init (signals, file descriptors, filemgr, bufmgr, smgr, shm, local struct);

Set zero_damaged_pages=false, statement_timeout=0, lock_timeout=0;

Set default_transaction_isolation="read commited", synchronous_commit=local;

Get database name from av_startingWorker;

Set itself in runningWorkers list and reset av_startingWorker;

Send SIGUSR2 to AV Launcher.

But, if av_startingWorker is empty:

Log "`autovacuum worker started without a worker entry`" and exit process.

# IV. AutoVacWorkerMain().

Init as Postgres backend:

- Adding to ProcArray and ProcSignal.
- Finish buffer pool initialization.
- Get access to XLOG.
- Create relation-, catalog-, plan- caches, allow PortalManager.
- Init stats.
- Fill relacache from system catalog.
- Become a superuser.
- Check database existance, database directory and other checks.

Remember recentXid and recentMulti, exec do_autovacuum().

## V. Process a single database.

```
/*
 * do_autovacuum() -- Process a database table-by-table
 */
```

# V. do_autovacuum().

Fetch database stat.

Start a transaction.

Compute the multixact age for which freezing is urgent.

Find the pg_database entry, select the default freeze ages (min_age, table_age).

- Use 0 for templates and nonconnectable databases.
- Otherwise system-wide default.

Open pg_class relation.

# V. do_autovacuum(). Create tables list.

The catalog pg_class catalogs tables and most everything else that has columns or is otherwise similar to a table. -- official documentation.

https://www.postgresql.org/docs/current/static/catalog-pg-class.html

# V. do_autovacuum(). Create tables list.

Scan pg_class twice to determine which tables to vacuum.

- Relations and materialized views.
- TOAST tables.

```
* The reason for doing the second pass is that during it we
* want to use the main relation's pg_class.reloptions entry if the TOAST
* table does not have any, and we cannot obtain it unless we know
* beforehand what's the main table OID.
```

# V. do_autovacuum(). Create tables list.

First pass:

- Skip all, except regular relations and materialized views (pg_class.relkind);
- Fetch stat and reloptions (pg_class.reloptions);
- relation_needs_vacanalyze();
  - Need vaccum, analyze or wraparound?
- Check if it is a temp table (pg_class.relpersistence).

Depending on relation_needs_vacanalyze() place relation to list.

If relation has TOAST (pg_class.reltoastrelid), remember its assocaition.

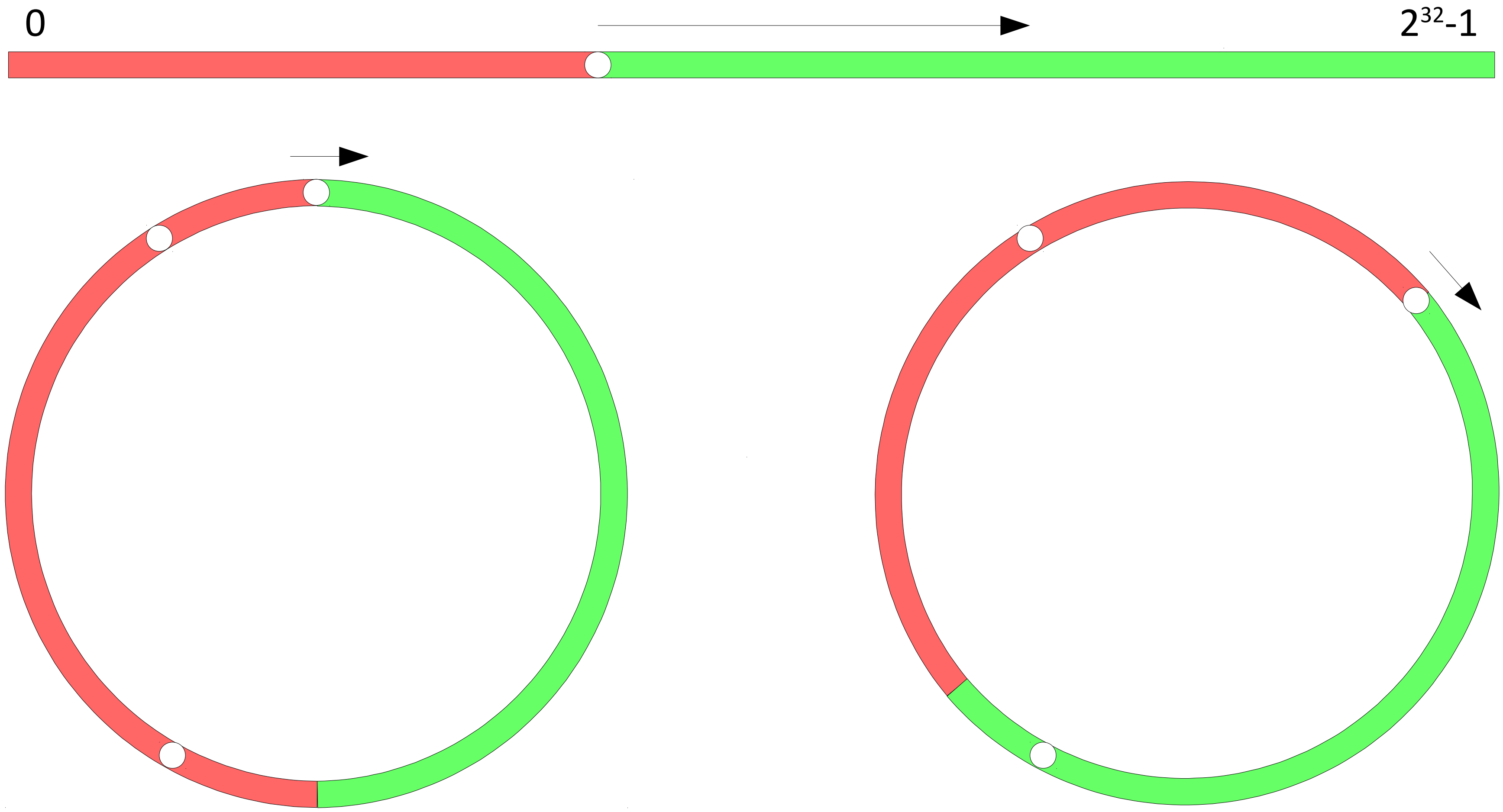
- Need for second pass, because we don't automatically vacuum toast tables along the parent table.

# V. do_autovacuum(). Create tables list.

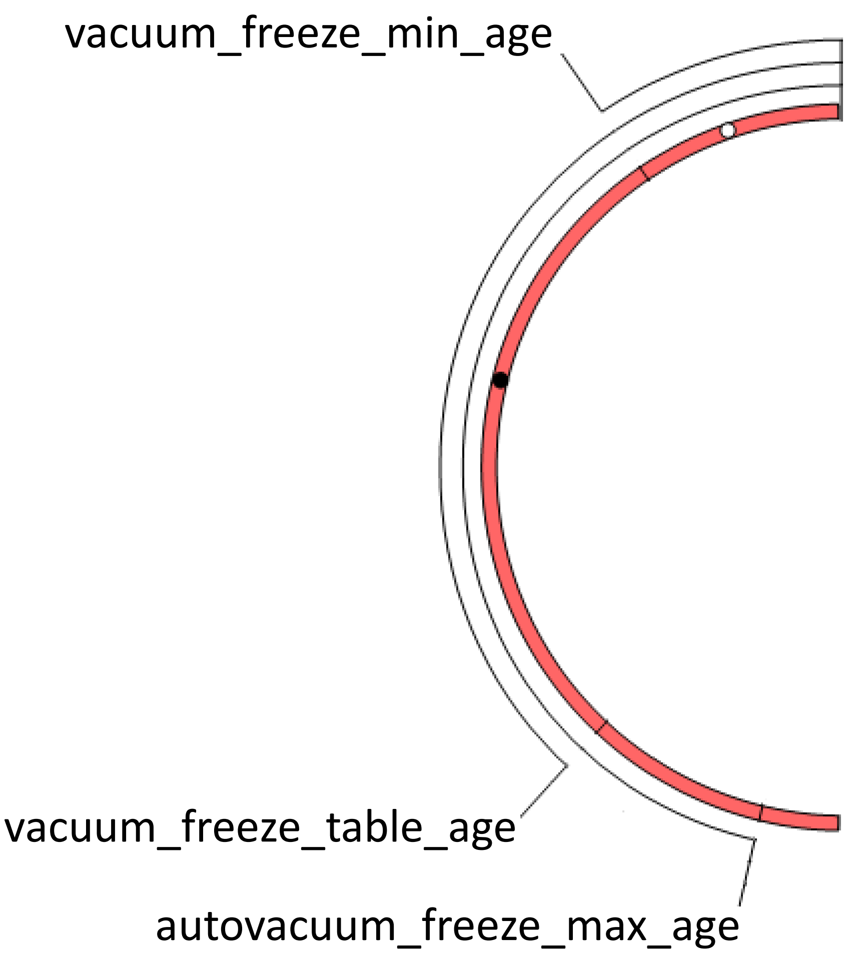
Second pass:

- Skip temporary tables;
- Extract reloptions (pg_class), or use parent tables reloptions (through associations);
- relation_needs_vacanalyze():
  - Check TOASTs only for vacuum.
- Append table to list.

# Wraparound.



# Wraparound.



recentXid – current transaction.

vacuum_freeze_min_age - cutoff age that vacuum should use to decide whether to freeze row versions while scanning a table

vacuum_freeze_table_age - vacuum performs a whole-table scan if the age is reached.

autovacuum_freeze_max_age - vacuum operation is forced to prevent transaction ID wraparound within the table.

# V. do_autovacuum() -> relation_needs_vacanalyze()

Check whether a relation needs to be vacuumed or analyzed (or both).

Determine vacuum/analyze equation parameters.

- Use reloptions (from main table or a toast table);
- or the autovacuum GUC variables;
- for freeze_max_age choose min values from reloptions and GUC.

# V. do_autovacuum() -> relation_needs_vacanalyze()

Force vacuum if table is at risk of wraparound:

xidForceLimit = recentXid – freeze_max_age;

multiForceLimit = recentMulti – multixact_freeze_max_age;

Force vacuum if pgclass.relfrozenxid or relminmxid precedes Limits.

If not wraparound and AV is disabled in relopts, skip the table.

Skip tables without stats, unless we have to force vacuum for anti-wrap purposes.

# V. do_autovacuum() -> relation_needs_vacanalyze()

```
           View "pg_catalog.pg_stat_all_tables"
       Column        |     Type     | Modifiers | Storage
---------------------+--------------+-----------+---------
relid                | oid          |           | plain
schemaname           | name         |           | plain
relname              | name         |           | plain
n_tup_ins            | bigint       |           | plain
n_tup_upd            | bigint       |           | plain
n_tup_del            | bigint       |           | plain
n_tup_hot_upd        | bigint       |           | plain
n_live_tup           | bigint       |           | plain
n_dead_tup           | bigint       |           | plain
n_mod_since_analyze  | bigint       |           | plain
...
```

# V. do_autovacuum() -> relation_needs_vacanalyze()

```
reltuples = classForm->reltuples;
vactuples = tabentry->n_dead_tuples;
anltuples = tabentry->changes_since_analyze;

vacthresh = (float4) vac_base_thresh + vac_scale_factor * reltuples;
anlthresh = (float4) anl_base_thresh + anl_scale_factor * reltuples;

*dovacuum = force_vacuum || (vactuples > vacthresh);
*doanalyze = (anltuples > anlthresh);
```

# V. do_autovacuum() -> relation_needs_vacanalyze()

```
autovacuum_vacuum_threshold = 50         # min number of row updates
                                         # before vacuum
autovacuum_analyze_threshold = 50        # min number of row updates
                                         # before analyze
autovacuum_vacuum_scale_factor = 0.2     # fraction of table size
                                         # before vacuum
autovacuum_analyze_scale_factor = 0.1    # fraction of table size
                                         # before analyze
```

# V. do_autovacuum(). Continue.

Table has now checked for vacuum, analyze (or both) or wraparound.

Close pg_class.

Choose a buffer access strategy.

| | |
|---|---|
| BAS_BULKREAD: | ring_size = 256 * 1024 / BLCKSZ; |
| BAS_BULKWRITE: | ring_size = 16 * 1024 * 1024 / BLCKSZ; |
| BAS_VACUUM: | ring_size = 256 * 1024 / BLCKSZ; (32kB) |

Process list.

# V. do_autovacuum(). Process list.

Check for interrupts (Reread config if SIGHUP received).

Check table for vacuuming by another worker (and skip).

Recheck table with table_recheck_autovac().

Announce table in shared memory.

Setup cost parameters.

Do balance with autovac_balance_cost().

# V. Cost-based vacuum.

```
vacuum_cost_delay = 0                   # 0-100 milliseconds
vacuum_cost_page_hit = 1                # 0-10000 credits
vacuum_cost_page_miss = 10              # 0-10000 credits
vacuum_cost_page_dirty = 20             # 0-10000 credits
vacuum_cost_limit = 200                 # 1-10000 credits

autovacuum_vacuum_cost_delay = 20ms     # default vacuum cost delay for
                                        # autovacuum, in milliseconds;
                                        # -1 means use vacuum_cost_delay
autovacuum_vacuum_cost_limit = -1       # default vacuum cost limit for
                                        # autovacuum, -1 means use
                                        # vacuum_cost_limit
```

# V. Cost-based vacuum. autovac_balance_cost().

The idea here is that we ration out I/O equally.

The amount of I/O is determined by cost_limit/cost_delay

- autovacuum_vac_cost_limit or vacuum_cost_limit;
- autovacuum_vac_cost_delay or vacuum_cost_delay;

Nothing to do, if not set (<= 0).

# V. do_autovacuum() -> autovac_balance_cost()

Calculate the total base cost limit of participating active workers.

1) cost_limit_base = cost_limit = 200, cost_delay = 10ms, n_workers = 5.

2) cost_total += cost_limit_base / cost_delay = 20 + 20 + 20 + 20 + 20 = 100

# V. do_autovacuum() -> autovac_balance_cost()

Calculate the total base cost limit of participating active workers.

1) cost_limit_base = cost_limit = 200, cost_delay = 10ms, n_workers = 5.

2) cost_total += cost_limit_base / cost_delay = 20 + 20 + 20 + 20 + 20 = 100

Adjust workers limit to balance the total of cost limit to autovacuum_vacuum_cost_limit.

3) cost_avail = cost_limit / cost_delay = 200 / 10 = 20

4) limit = cost_avail * cost_limit_base / cost_total = 20 * 200 / 100 = 20 * 2 = 40

5) w->cost_limit = max(min(limit, cost_limit_base), 1) = limit = **40**

# V. Cost based vacuum. Delay interval.

```
msec = VacuumCostDelay * VacuumCostBalance / VacuumCostLimit;
if (msec > VacuumCostDelay * 4)
    msec = VacuumCostDelay * 4;

pg_usleep(msec * 1000L);
VacuumCostBalance = 0;
```

# V. do_autovacuum(). Process list.

Remember table name (database.schema.relation)

- If failed (drop table?), skip table.

Do all work in autovacuum_do_vac_analyze()

- If error occurs?

# V. do_autovacuum(). Process list.

Remember table name (database.schema.relation)

- If failed (drop table?), skip table.

Do all work in autovacuum_do_vac_analyze().

- If error occurs:
  - Hold interrupts;
  - Report to postgres log;
  - Abort the transaction;
  - Reset error context, memory contexts;
  - Start new transaction, resume interrupts.

# V. do_autovacuum().

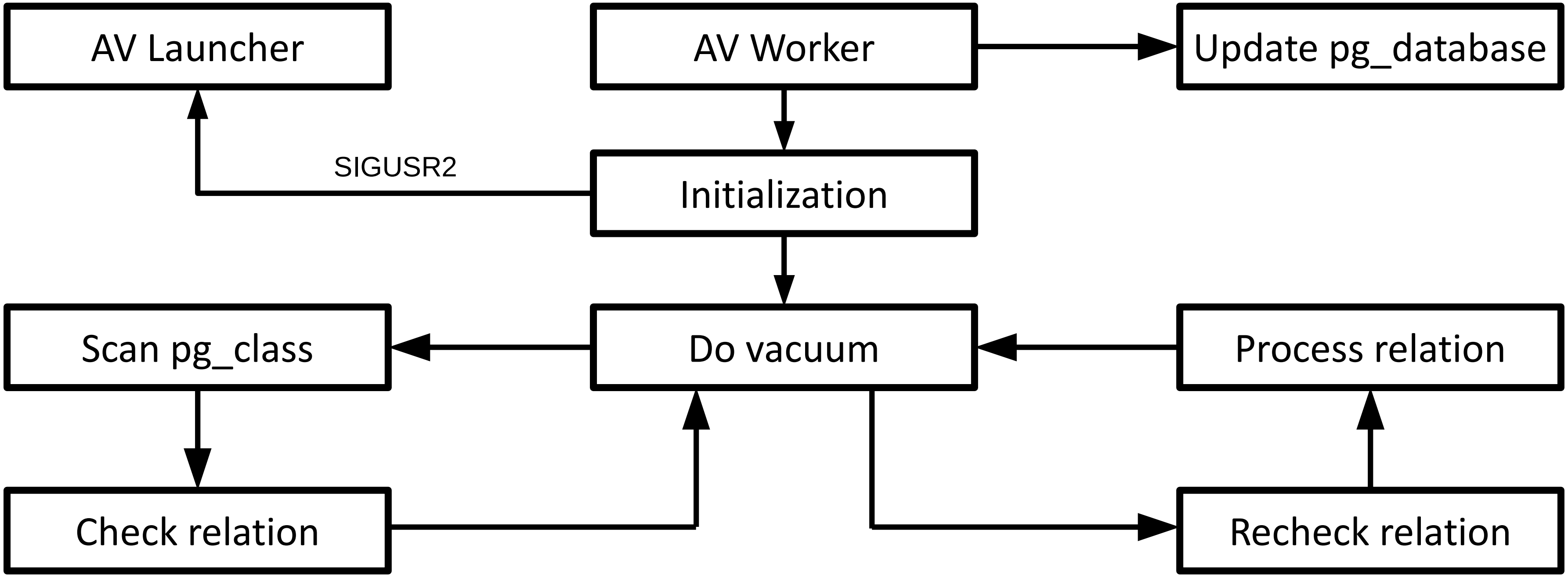
All tables has been processed.

Update pg_database.datfrozenxid.

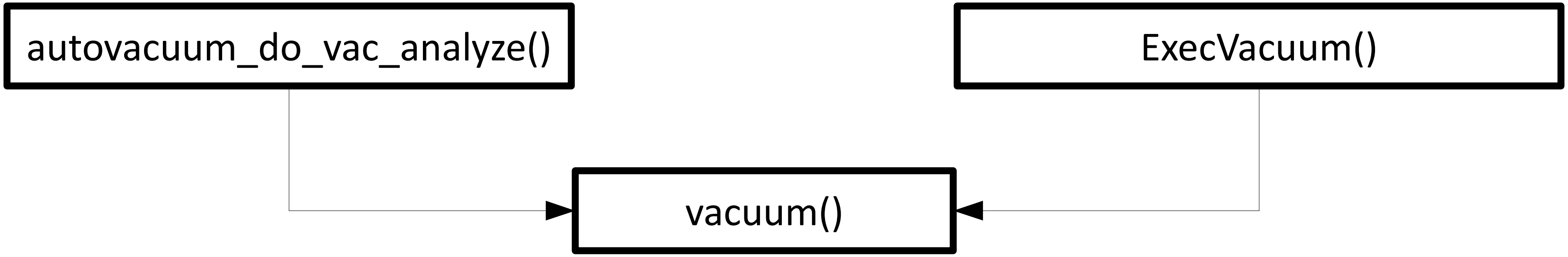
Truncate pg_clog if possible.

Finally close out the last transaction.

# V. Questions?


AV Launcher
AV Worker
Update pg_database
SIGUSR2
Initialization
Scan pg_class
Do vacuum
Process relation
Check relation
Recheck relation

# VI. Prepare for Vacuum.



autovacuum_do_vac_analyze()

ExecVacuum()

vacuum()

autovacuum_do_vac_analyze() - vacuum and/or analyze the specified table.

ExecVacuum() - primary entry point for manual VACUUM and ANALYZE commands.

vacuum() - primary entry point for VACUUM and ANALYZE commands.

# VI. vacuum()

Choose buffer access strategy (if not defined yet).

Decide whether we need to start/commit our own transactions.

For VACUUM (with or without ANALYZE): always do so, so that we can release locks as soon as possible.

- Use own xact in (auto)vacuum and autoanalyze;
- Remove the topmost snapshot from the active snapshot stack;
- Commit transaction command (started in do_autovacuum()).

For ANALYZE (no VACUUM): if inside a transaction block, we cannot start/commit our own transactions.

# VI. vacuum()

If VacuumCostDelay > 0, use cost-based vacuum and zeroing counters.

Process relation, check vacoptions:

- VACOPT_VACUUM - run vacuum_rel();
- VACOPT_ANALYZE - run analyze_rel();

Finish up processing:

- If own xacts used, start transaction command – this matches the CommitTransaction waiting for us in PostgresMain().
- Update pg_database.datfrozenxid, and truncate pg_clog if possible.

The end.

# VII. vacuum() -> vacuum_rel().

```
/*
 *  vacuum_rel() -- vacuum one heap relation
 *
 *      Doing one heap at a time incurs extra overhead, since
 *      we need to check that the heap exists again just before
 *      we vacuum it.  The reason that we do this is so that
 *      vacuuming can be spread across many small transactions.
 *      Otherwise, two-phase locking would require us to lock
 *      the entire database during one pass of the vacuum cleaner.
 *
 *      At entry and exit, we are not inside a transaction.
 */
```

# VII. vacuum_rel().

Begin a transaction and get a transaction snapshot.

Set PROC_IN_VACUUM or PROC_VACUUM_FOR_WRAPAROUND in ProcArray

Check for user-requested abort.

Determine the lock type:

- Exclusive lock for a FULL vacuum;
- ShareUpdateExclusiveLock for concurrent vacuum.

Open the relation and get the appropriate lock on it.

- If autovacuum and lock failed, log "`skipping vacuum of %s --- lock not available`".
- If open failed (relation removed?), remove snapshot, commit transaction, finish.

# VII. vacuum_rel().

Check permissions (superuser, the table owner, or the database owner).

Check that it's a vacuumable relation (regular, matview, or TOAST).

Ignore tables that are temp tables of other backends.

Get a session-level lock for protecting access to the relation across multiple transactions.

(we can vacuum the relation's TOAST table secure in the knowledge that no one is deleting the parent relation.)

Remember the relation's TOAST relation for later (except autovacuum).

Switch to the table owner's userid.

# VII. vacuum_rel().

```
/*
 * Do the actual work --- either FULL or "lazy" vacuum
 */
```

VACOPT_FULL:

- close relation before vacuuming, but hold lock until commit.
- cluster_rel() - VACUUM FULL is now a variant of CLUSTER; see cluster.c.

Otherwise:

- lazy_vacuum_rel()

# VII. vacuum_rel().

Table vacuum is finished now.

Restore userid and security context.

Close relation.

Complete the transaction and free all temporary memory used.

If  TOAST exists, vacuum it too (use vacuum_rel()).

Release the session-level lock on the master table.

# VII. vacuum_rel() -> lazy_vacuum_rel().

```
/*
 *   lazy_vacuum_rel() -- perform LAZY VACUUM for one heap relation
 *
 *       This routine vacuums a single heap, cleans out its indexes, and
 *       updates its relpages and reltuples statistics.
 *
 *       At entry, we have already established a transaction and opened
 *       and locked the relation.
 */
```

# VII. vacuum_rel() -> lazy_vacuum_rel().

Set xid limits for freezing:

- freeze_min_age, freeze_table_age;
- multixact_freeze_min_age, multixact_freeze_table_age;
- oldestXmin – distinguish whether tuples are DEAD or RECENTLY_DEAD;
- freezeLimit – below this all Xids are replaced by FrozenTransactionId;
- xidFullScanLimit – full-table scan if **relfrozenxid** older than this;
- multiXactCutoff – cutoff for removing all MultiXactIds from Xmax;
- mxactFullScanLimit – full-table scan if **relminmxid** older than this.

Compare relfrozenxid/relminmxid with cutoff values.

# VII. vacuum_rel() -> lazy_vacuum_rel().

Open all indexes of the relation.

Do the vacuuming with lazy_scan_heap().

Close indexes.

Compute whether we actually scanned the whole relation.

scanned_pages + frozenskipped_pages = rel_pages

Optionally truncate the relation.

Report that we are now doing final cleanup (pg_stat_*)

Update Free Space Map.

Update statistics in pg_class:

- relpages, reltuples, relallvisible, relhasindex;
- Update refrozenxid/relminmxid **only when** full table scan.

# VII. vacuum_rel() -> lazy_vacuum_rel().

Report results to the stats collector (n_live_tupe, n_dead_tuples)

Report to postgres log, if log_min_duration >= 0.

Finish.

# VII. Return to the vacuum_rel(). Remind.

Table has been vacuumed.

Restore userid and security context.

Close relation.

Complete the transaction and free all temporary memory used.

If  TOAST exists, vacuum it too (use vacuum_rel()).

Release the session-level lock on the master table.

# VIII. lazy_vacuum_rel() -> lazy_scan_heap()

```
/* lazy_scan_heap() - scan an open heap relation */
```

# VIII. lazy_vacuum_rel() -> lazy_scan_heap()

Allocate memory for dead tuples storage (autovacuum_work_mem);

Check pages which can be skipped:

- ALL_FROZEN and ALL_VISIBLE flags (according to the visibility map):
- If not full scan, skip all-visible pages;
- Skip all-frozen pages.
- Force scanning of last block – check for relation truncation.

After each block exec vacuum_delay_point();

# VIII. lazy_vacuum_rel() -> lazy_scan_heap()

Start loop from first unskippable block:

- Looking for the next unskippable block;
- Check dead tuples storage, if close to overrun, do cycle of vacuuming;
- Read the buffer. Account costs.
- Try to acquire lock for buffer clean up (need for HOT pruning).

  Block will be skipped if lock failed.

# VIII. lazy_vacuum_rel() -> lazy_scan_heap()

Check the page for xids that need to be frozen:

- Always vacuum an uninitialized page;
- Skip an empty page.
- Check normal pages:
  - Dead and redirect items never need freezing;
  - Check to see whether any of the XID fields of a tuple (xmin, xmax, xvac) are older than the specified cutoff XID or MultiXactId.

# VIII. lazy_vacuum_rel() -> lazy_scan_heap()

Contunue main buffer loop...

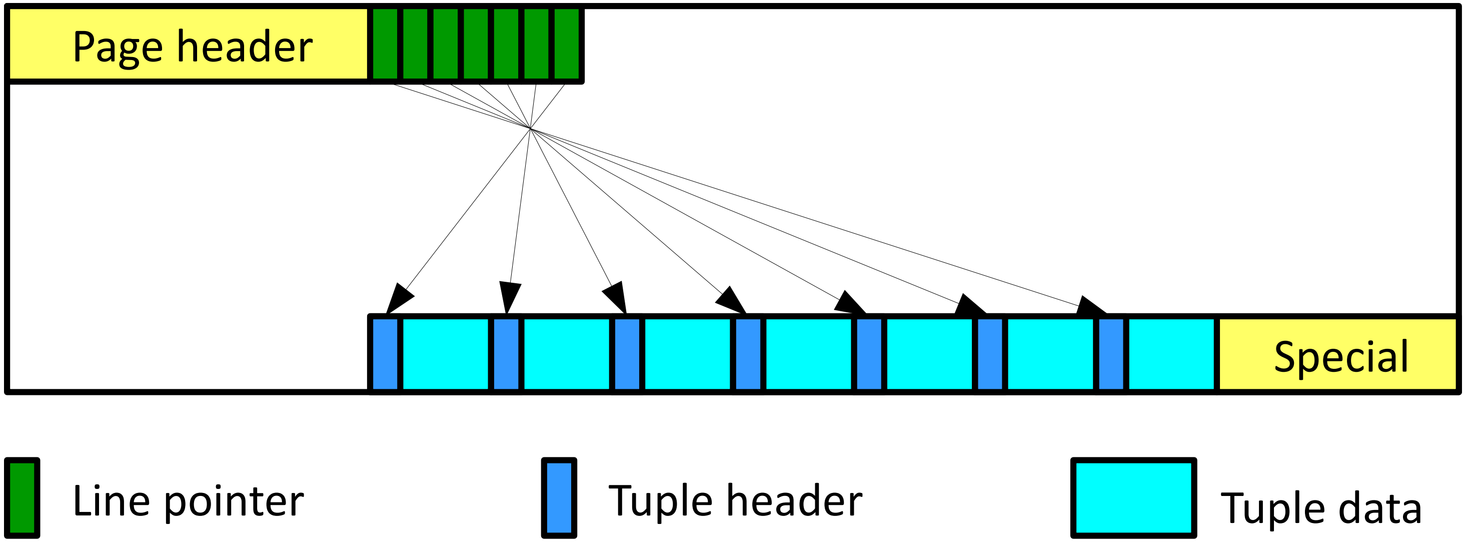
If page is new, init it:

- WARNING: "`relation %s page %u is uninitialized --- fixing`";
- Mark buffer as dirty, update Free Space Map.

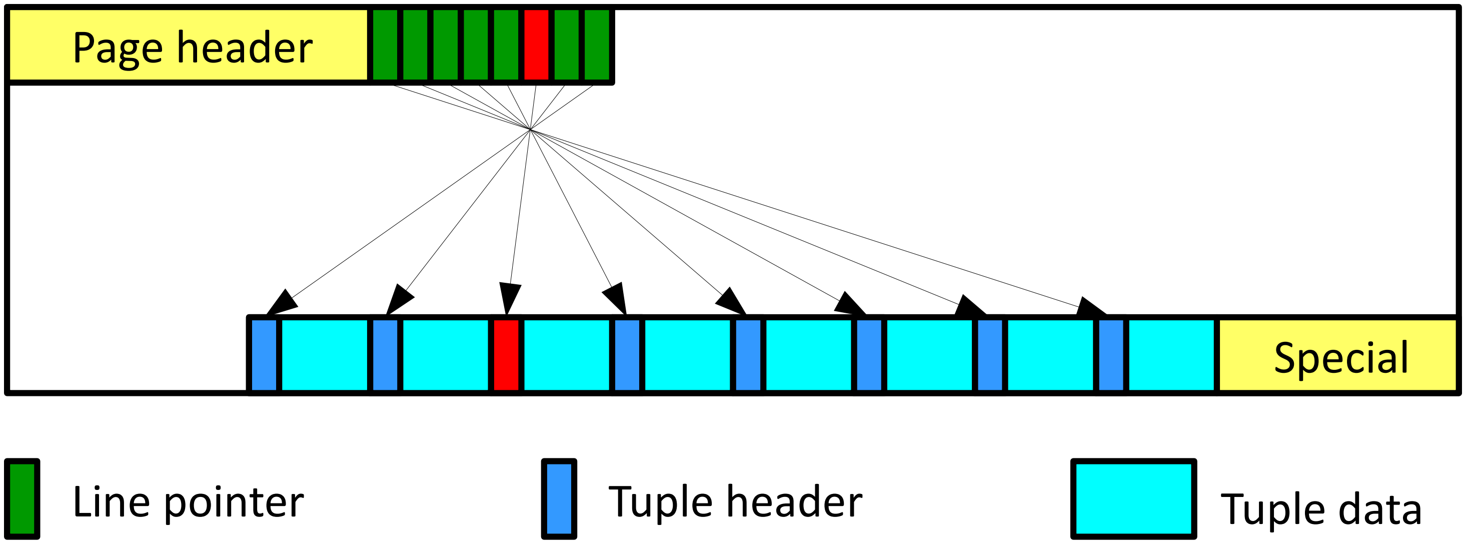
If page is empty:

- Mark it as ALL_VISIBLE and ALL_FROZEN;
- Mark buffer dirty, write a WAL record, update Visibility Map and Free Space Map.

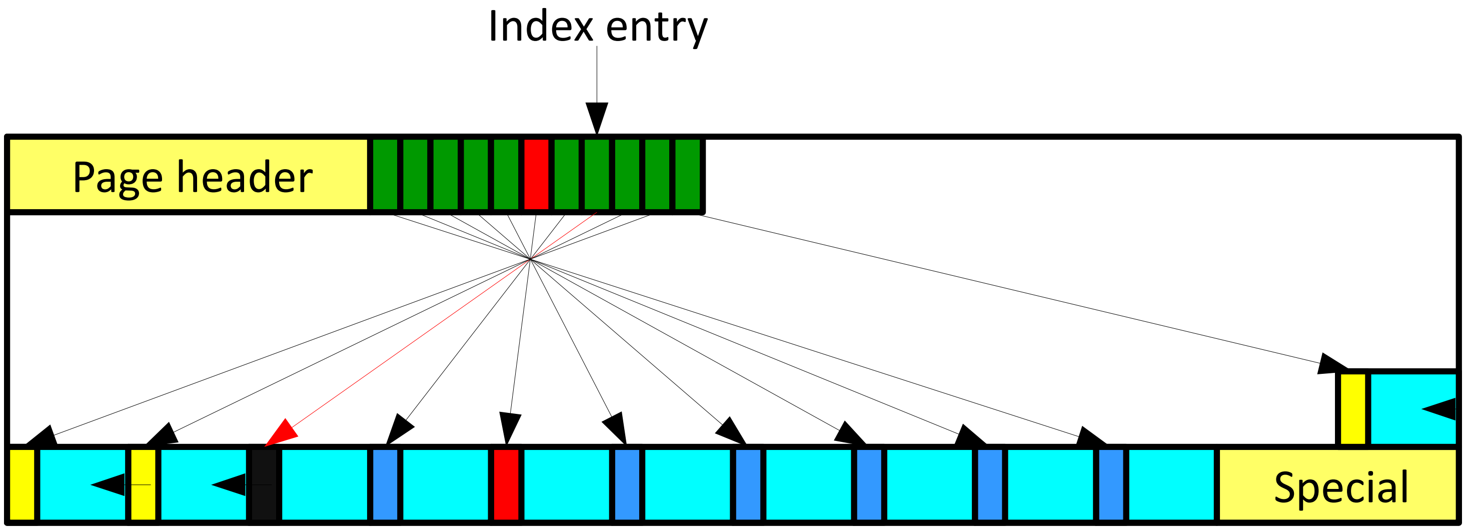
# HOT Update (Heap Only Tuple).



# HOT Update (Heap Only Tuple).

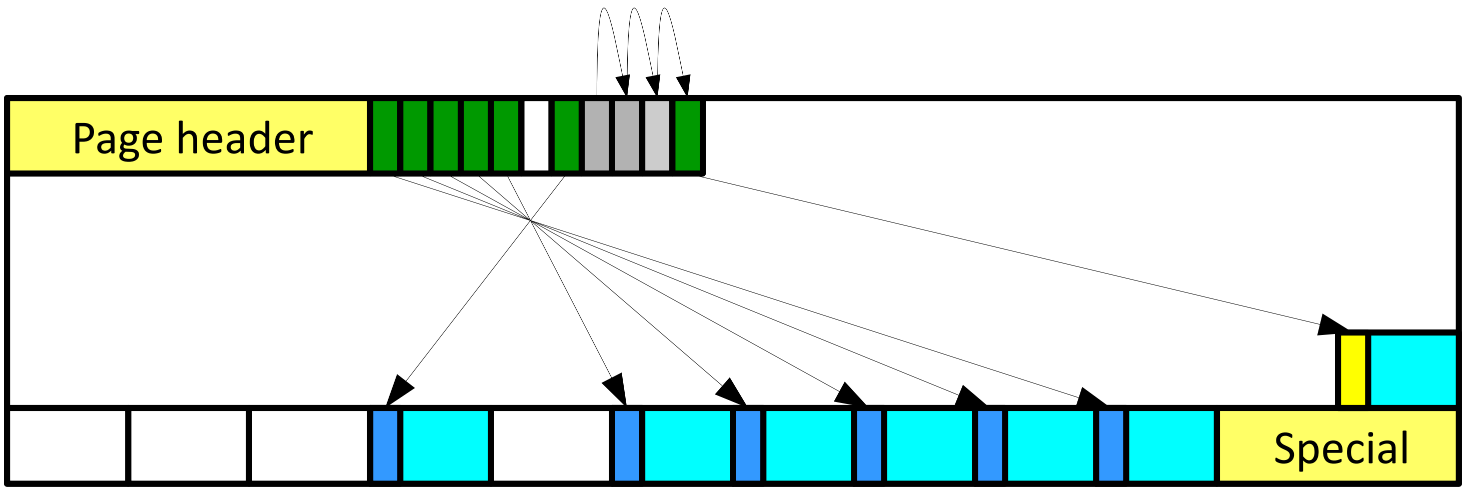


# HOT Update (Heap Only Tuple).



| | | | | | |
|---|---|---|---|---|---|
| 🟩 | Line pointer | 🟦 | Tuple header | ▭ | Tuple data |
| ⬛ | Heap Hot Updated | 🟨 | Heap Only Tuple | | |

# HOT Update (Heap Only Tuple).



🟩 Line pointer

🟦 Tuple header

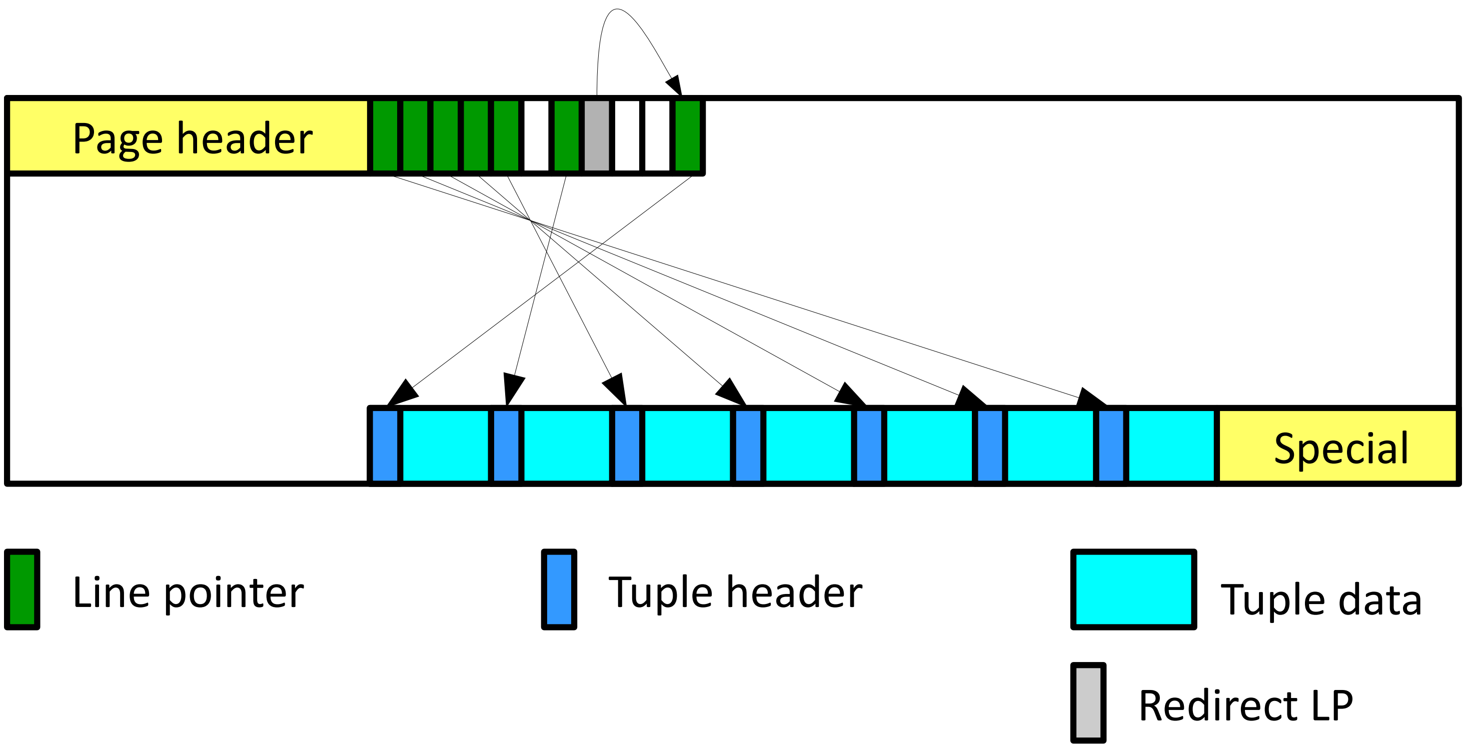
▭ Tuple data

⬛ Heap Hot Updated

🟨 Heap Only Tuple

▨ Redirect LP

# HOT Update (Heap Only Tuple).



Page header

Special

🟩 Line pointer

🟦 Tuple header

🟦 Tuple data

⬜ Redirect LP

# VIII. lazy_vacuum_rel() -> lazy_scan_heap()

Prune all HOT-update chains in page:

- Scan the page item pointers, looking for HOT chains.
  - Skip redirects, unused and already dead.
- Prune item pointers or a HOT chains (don't actually change the page here):
  - Prune dead or broken HOT chain;
  - Rebuild redirects.

# VIII. lazy_vacuum_rel() -> lazy_scan_heap()

Apply changes within crit section:

- Update all redirected line pointers;
- Update all now-dead line pointers;
- Update all now-unused line pointers;
- Finally, repair fragmentation.

Clear the "page is full" flag, mark page dirty, emit a WAL.

End crit section.

(If prunable not found, do nothing)

# VIII. lazy_vacuum_rel() -> lazy_scan_heap()

Scan the page, collect vacuumable items, check for tuples requiring freezing.

Check item pointers:

- Skip unused, dead, redirects. Check only normal.

  HeapTupleSatisfiesVacuum():

  - HEAPTUPLE_DEAD: vacuumable (but skip, if it's a HOT chain member).
  - HEAPTUPLE_LIVE: good tuple, do not vacuum.
  - HEAPTUPLE_RECENTLY_DEAD: must not remove it from relation.
  - HEAPTUPLE_INSERT_IN_PROGRESS and

    HEAPTUPLE_DELETE_IN_PROGRESS: do nothing, page is not ALL_VISIBLE.

Remeber vacuumable tuples in vacrelstats.

# VIII. lazy_vacuum_rel() -> lazy_scan_heap()

Check non-removable tuples to see if it needs freezing.

- Prepare tuple, if true (prepare infomask in local structure).

If any tuple is frozen:

- Start crit section;
- Mark the buffer dirty;
- Set bits into tuple infomask (from local structure);
- Write a WAL record recording the changes;
- End crit section.

# VIII. lazy_vacuum_rel() -> lazy_scan_heap()

Vacuum page right now, if there are no indexes (lazy_vacuum_page()).

Update Visibility Map and Free Space Map.

Finish loop, all blocks scanned.

# VIII. lazy_vacuum_rel() -> lazy_scan_heap()

Save stats, compute new pg_class.reltuples.

If any tuples need to be deleted, perform final vacuum cycle.

- Remove index entries;
- Remove tuples from heap with lazy_vacuum_heap().

# IX. lazy_scan_heap() -> lazy_vacuum_heap()

lazy_vacuum_heap() - second pass over the heap.

Loop over collected dead tuples (vacrelstats) – do not visit pages with no dead tuples.

- Before start vacuum_delay_point();
- Read buffer by item pointer and account costs;
- Try to lock buffer for cleanup – **skip page if no lock**;
- Vacuum page with lazy_vacuum_page();
- Update Free Space Map.

# IX. lazy_vacuum_heap() -> lazy_vacuum_page()

lazy_vacuum_page() -- free dead tuples on a page and repair its fragmentation.

Start crit section.

- Loop over collected dead tuples (within page), set ItemID as unused (LP_UNUSED).
- Repair page fragmentation;
- Mark buffer dirty, write to XLOG.

End crit section.

Update Visibility Map.

# VIII. lazy_vacuum_rel() -> lazy_scan_heap()

Now that we've compacted the page, Visibility Map updated.

Update FreeSpaceMap.

Write log message: "`%s: removed %d row versions in %d pages`".

Post-vacuum cleanup and statistics update for each index (pg_class)

Write message about what we did to postgres log.

The End?

# Links

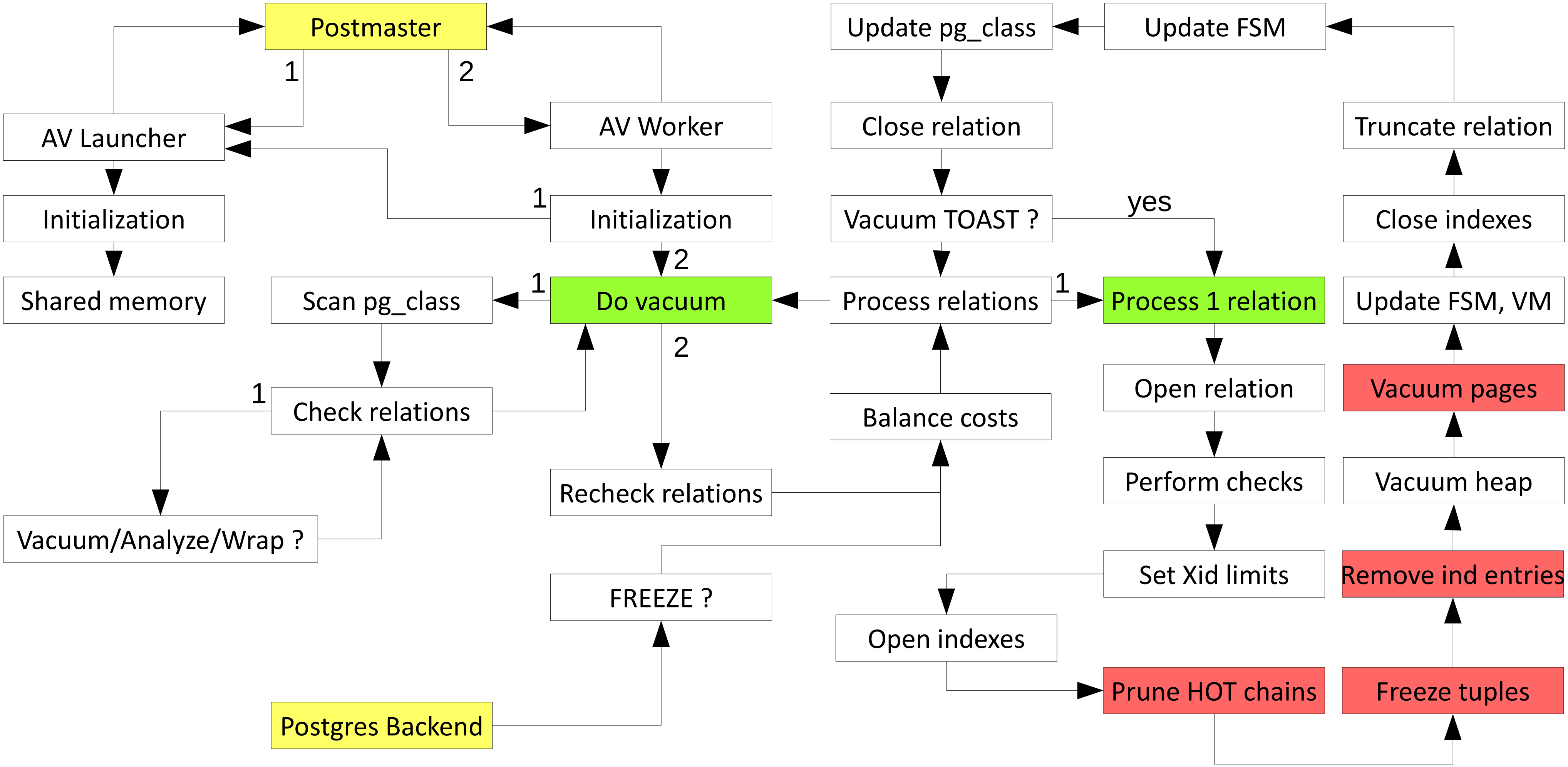
Alexey Lesovsky – lesovsky@pgco.me

See slides on SlideShare: http://www.slideshare.net/alexeylesovsky/

PostgreSQL official documentation:

- Vacuum: https://www.postgresql.org/docs/current/static/routine-vacuuming.html
- Autovacuum:
  - https://www.postgresql.org/docs/current/static/routine-vacuuming.html#AUTOVACUUM
  - https://www.postgresql.org/docs/current/static/runtime-config-autovacuum.html
- Progress Reporting: https://www.postgresql.org/docs/devel/static/progress-reporting.html
- PageInspect contrib module: https://www.postgresql.org/docs/current/static/pageinspect.html

# Questions?


Postmaster
1
2
AV Launcher
AV Worker
Initialization
Initialization
1
Shared memory
Scan pg_class
1
Do vacuum
2
2
Check relations
1
Vacuum/Analyze/Wrap ?
Recheck relations
FREEZE ?
Postgres Backend
Update pg_class
Update FSM
Close relation
Vacuum TOAST ?
yes
Process relations
1
Process 1 relation
Balance costs
Open relation
Perform checks
Set Xid limits
Open indexes
Prune HOT chains
Truncate relation
Close indexes
Update FSM, VM
Vacuum pages
Vacuum heap
Remove ind entries
Freeze tuples