EDB
Postgres® for the AI Generation

Beginner’s Guide to Hacking Postgres

Pavan Deolasee,
Senior Principal Engineer,
6th March 2025

Why hack on Postgres?

- Postgres is world’s most advanced open source database system
- Relational database concepts are time tested and evolved over a period of time
- Easy to understand, well commented, highly organised source code
- Well defined primitives and building blocks
- Highly extensible system
- BSD licensed code, giving a lot of flexibility
- Thriving and supportive community, but can use more reviewers and submitters
- Postgres internals skills are always high in demand
- Become a better programmer

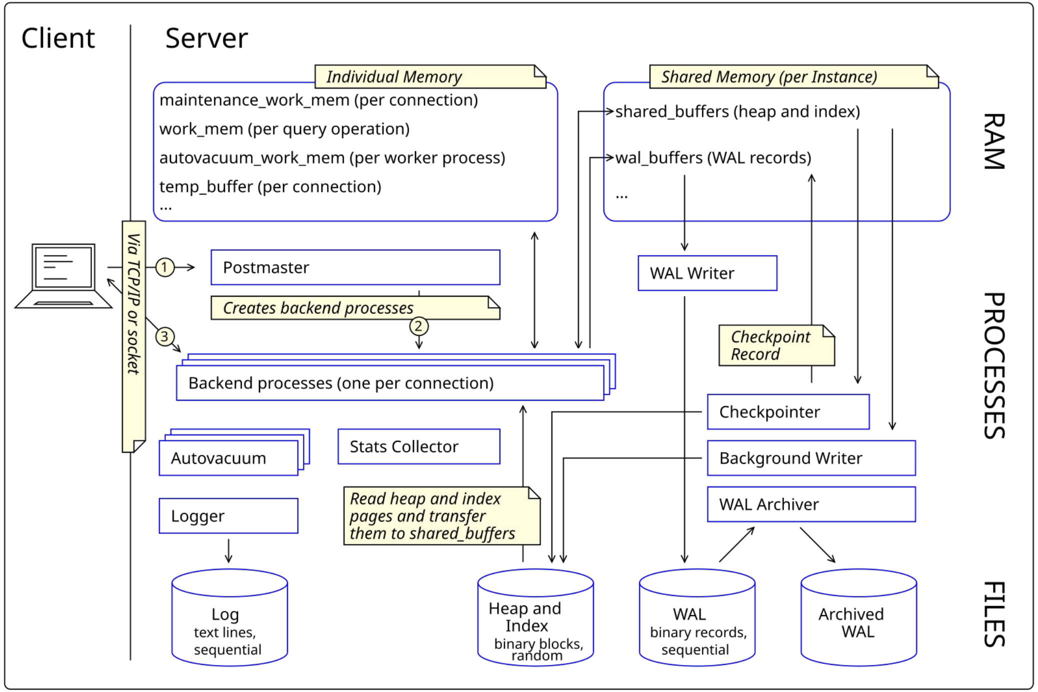
What skills do I need?

- Algorithms and data structures
- Understanding of structured programming languages (C preferred, but not necessary)
- Understanding of system programming (file systems, synchronization primitives, IPC)
- Understanding of relational database system concepts (transactions, buffer pool, ACID, SQL)
- Tools (editor, IDE, debugger)
- Aptitude to learn new things
- Communication skills

Agenda

- PostgreSQL Architecture Overview
- Preparing environment and obtaining Postgres sources
- Source code layout
- Building/installing Postgres from source
- Important Postgres subsystems
- Important tools and resources

PostgreSQL Architecture



Client
Server
Individual Memory
maintenance_work_mem (per connection)
work_mem (per query operation)
autovacuum_work_mem (per worker process)
temp_buffer (per connection)
...
Shared Memory (per Instance)
shared_buffers (heap and index)
wal_buffers (WAL records)
...
RAM
Via TCP/IP or socket
1
Postmaster
Creates backend processes
2
3
Backend processes (one per connection)
WAL Writer
Checkpoint Record
Checkpointer
Background Writer
WAL Archiver
Autovacuum
Stats Collector
Logger
Read heap and index pages and transfer them to shared_buffers
PROCESSES
Log text lines, sequential
Heap and Index binary blocks, random
WAL binary records, sequential
Archived WAL
FILES

Source: https://commons.wikimedia.org/

Preparing to Build Postgres

- Preparing build environment
  - git, flex, bison, C compiler, required libraries
  - `sudo apt install build-essential`
  - `sudo yum groupinstall 'Development Tools'`
- Obtaining code

```
$ git clone https://git.postgresql.org/git/postgresql.git

Cloning into 'postgresql'...
remote: Enumerating objects: 17668, done.
remote: Counting objects: 100% (17668/17668), done.
remote: Compressing objects: 100% (9709/9709), done.
remote: Total 1045612 (delta 12315), reused 10367 (delta 7887), pack-reused 1027944
Receiving objects: 100% (1045612/1045612), 351.25 MiB | 3.46 MiB/s, done.
Resolving deltas: 100% (902378/902378), done.
Updating files: 100% (7062/7062), done.
```

Power of GIT

- Some important commands
  - git fetch
  - git checkout [-b]
  - git pull [--rebase]
  - git log
  - git diff
  - git rebase [-i]
  - git merge
  - git commit [-a]
  - git worktree add

Source Code Layout

```
$ ls postgresql/
COPYRIGHT         HISTORY            README.md     config        configure.ac  doc
meson_options.txt GNUmakefile.inMakefile      aclocal.m4    configure     contrib
              meson.build       src

$ ls postgresql/src/
DEVELOPERS    Makefile.global.in     backend       common   include
makefiles     nls-global.mk          port          test     tools
Makefile      Makefile.shlib    bin            fe_utils  interfaces
meson.build   pl                     template      timezone tutorial
```

Source Code Layout

```
$ ls postgresql/src/backend/
Makefile backup   commands      foreign       libpq         nls.mk
parser   port     replication   statistics    tsearch       access   bootstrap
common.mkjit      main          nodes         partitioning  postmaster    rewrite
storage           utils         archive       catalog       executor      lib
meson.build       optimizer     po            regex         snowball      tcop
```

Source Code Layout

```
$ ls src/bin
initdb            pg_basebackup      pg_config         pg_dump
pg_test_fsync     pg_verifybackup    pgbench           scripts
pg_amcheck        pg_checksums       pg_controldatapg_resetwal
pg_test_timingpg_waldump        pgevent            pg_archivecleanup
pg_combinebackup  pg_ctl             pg_rewind         pg_upgrade
pg_walsummary     psql
```

Source Code Layout

```
$ ls src/test
authenticationexamples           icu      isolation     kerberos
ldap               locale            mb        modules       perl
postmaster         recovery          regress   ssl           subscription
```

Source Code Layout

```
$ ls contrib
dblink        lo            pg_buffercachepg_walinspect      pg_freespacemap
amcheck       btree_gin     pg_prewarm         intarray
pg_stat_statements          pgstattuple        auto_explain      file_fdw
postgres_fdw                pageinspect        pg_trgm           test_decoding
cube                        hstore             passwordcheck     pg_visibility
```

Configure

- Configure

```
CFLAGS="-O0" ./configure \
     --enable-debug \
     --enable-depend \
     --enable-tap-tests \
     --enable-cassert \
     --prefix=$HOME/pgsql/install
```

- Look for errors reported by configure command and fix those (most often development libraries are missing)

Make Targets

01 **Build**

- make
- make -C contrib
- make world

02 **Install**

- make install

03 **Test**

- make check
- make check-prove
- make -C src/test/isolation check
- make check-world

Start Postgres Server

- Run initdb
  - `~/pgsql/install/bin/initdb -D ~/pgsql/data`
- Edit postgresql.conf, pg_hba.conf (may not be required for development env)
- Start the server

  -
    ```
    ~/pgsql/install/bin/pg_ctl -D ~/pgsql/data -l ~/pgsql/postgres.log start
    waiting for server to start.... done
    server started
    ```
- Check

```
$ ~/pgsql/install/bin/pg_ctl -D ~/pgsql/data status
pg_ctl: server is running (PID: 53198)
/Users/pavan/pgsql/install/bin/postgres "-D" "/Users/pavan/pgsql/data"
```

Postmaster Lifecycle

- Initialise various subsystems, perform XLOG recovery, bring the database in a consistent state (**src/backend/postmaster/startup.c)**
- Create/initialise shared memory segments
- (Re)start various system background processes (and even user background processes, on demand)
  - Autovacuum launcher
  - WAL Writer
  - Background Writer
  - Stats collector
- Bind to the TCP socket and starts listening
- Accept client connection, fork a new backend and handover further processing to the backend (**src/backend/postmaster/postmaster.c**)

Backend Lifecycle

- Postmaster accepts client connection, forks a new backend, closes its copy of the socket connection
- Backend takes over client communication
- Performs authentication and enter into frontend-backend protocol
- Receives commands from the client, validate, execute and sends results back
  - Simple query protocol (simple SQL queries, replication commands)
  - Extended query protocol (parse, bind, execute)
  - Copy protocol
- Exits when client connection ends

Query Lifecycle

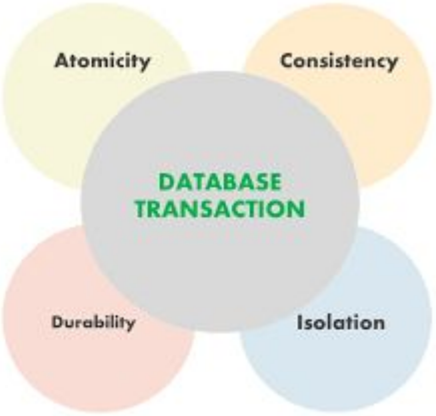
- Parsing
  - translate query string into tokens (**src/backend/parser/gram.y, scan.l**)
- Analyzing
  - validate tokens (e.g. is it really a table or does the column exist),
  - translate raw parse tree into Query (**src/backend/parser/analyze.c**)
- Planning
  - Generate different execution paths (**src/backend/optimizer/path**)
- Optimizing
  - Optimize the query Plan (**src/backend/optimizer/plan**)
- Execution
  - Finally execute the Plan and get the results (**src/backend/executor**)

Accessing Data

- Same mechanism for user data and system catalogs
- Postgres provides many different ways to access the data efficiently
- Sequential access of the heap
- BTree access for point and range queries
- Hash access for efficient filtering over a number of WHERE clauses
- Tablesample access for a random subset of data
- Table Access Method for user defined strategies
- **src/backend/access/**

Transaction Management

- Transactions are at the core of any database system
- Postgres supports transactions and subtransactions, and a variety of serializability modes
- MVCC allows readers to read without waiting for writers to finish
- Write-ahead-logs (WAL) for transaction durability
- **src/backend/access/transam/**



Buffer Management

- Shared buffer pool, carved out of the shared memory segment
- When it needs to access a page, a backend asks the buffer manager to get the page in the buffer pool
- Buffer manager implements a hash table, mapping page identifiers (a combination of relation identifier and block number) to actual pages
- Buffer manager keeps a pin of the page, to ensure it's not removed from the pool while a process is still accessing it
- **src/backend/storage/buffer/** implements the core of buffer manager

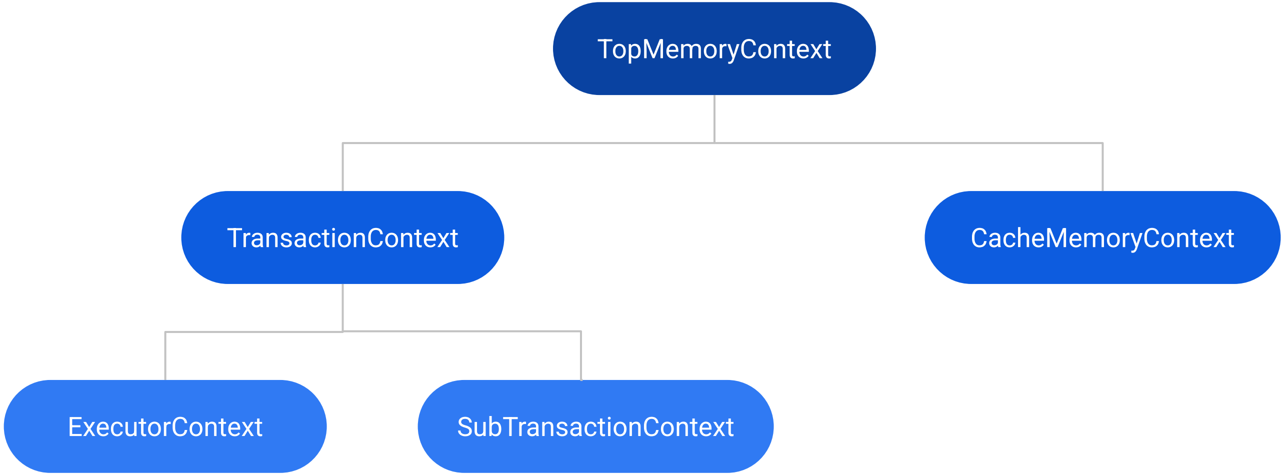
Storage Management

- Tables and indexes are mapped to files in the file system
- Each file can be maximum 1GB in size (segment)
- Large tables and indexes are thus made up of many such segments
- Storage manager keeps track of open files, improves performance by caching open file descriptors
- Also implements support for temporary files
- **src/backend/storage/file/**

Memory Management

- Postgres provides robust infrastructure for memory management
- Allocations are tracked via `MemoryContext`, which are hierarchical in nature
  - `palloc/palloc0`
  - `MemoryContextAlloc()`
- Allocated memory can be freed in a single shot by deleting or resetting the `MemoryContext`
  - `MemoryContextReset()`
  - `MemoryContextDelete()`
- Or it can be explicitly freed
  - `pfree()`

Examples of MemoryContext



```
src/backend/utils/mmgr
```

EDB
Postgres® for the AI Generation

Lock Manager

- Heavyweight locks or lmgr locks

  (**`src/backend/storage/lmgr/lmgr.c`**)
  - Used to lock objects such as tables, rows, transactions
  - Can be held for very long time, often for the entire transaction
  - Many modes and a conflict matrix
- LWLocks (**`src/backend/storage/lmgr/lwlock.c`**)
  - Interprocess synchronization
  - Short duration locks
  - read/write mode
- Spinlocks (**`src/backend/storage/lmgr/s_lock.c`**)
  - Interprocess synchronization
  - Extremely short duration (few CPU cycles, not across IO, must not sleep etc)
  - Single mode

System Caches

- Postgres maintains a cache of system objects for quick and fast lookup
  - No need to read the catalog from storage everytime
  - Cache in the backend local memory
  - Invalidated when system objects are modified

  src/backend/utils/cache/catcache.c
- lsyscache
  - A set of convenience routines to lookup objects
  - E.g. get_relname_relid(), get_rel_relkind(), get_rel_relispartition(), get_func_name()

  src/backend/utils/cache/lsyscache.c

Interprocess Communication

- Shared Memory Segments
    - BufferPool, PGPROC Array etc
- Shared Memory Queues
- Shared Memory Hash Tables
    - Lock Table, shared buffer pool hash table
- Dynamic Shared Memory
    - Parallel workers
- Latches
    - Wait for events, signal procs waiting for events
- Signals

`src/backend/storage/ipc/`

Error Handling

- User level errors are reported via `ereport()`.
    - Allows additional information to be included (SQLSTATE, detail, hint, backtrace etc)
- Implemented via `longjmp`.
    - Control is passed back to the top level error handler
    - Transaction is aborted, resources are released (locks, memory, buffers, open files etc) and backend is prepared to handle next set of commands (usually ROLLBACK first)
- Developers can write their own exception handlers via `PG_TRY/PG_CATCH` blocks.
- Assertions to detect unexpected states
    - Postgres uses assertions freely (enabled with `–enable-cassert` flag)
    - Turned off in production builds

Tools

- Source code navigation tools
  - My favorite is ctags + cscope, but use whatever you are comfortable with
- Editor
  - My favorite is vim, but use whatever works for you. Emacs, gvim, Visual Studio, even cursor-ai
- Editor Plugins
  - cscope/ctags integration, key shortcuts, git plugins
- Debugger
  - gdb, lldb
- Profilers
  - gprof

Sample Debugging

- Either user backend, auxiliary process or a background worker process

```
postgres=# SELECT pg_backend_pid();
 pg_backend_pid
----------------
          55381
(1 row)
```

```
$ lldb -p 55381
NAME        PASS     STOP     NOTIFY
=========== ======= ======= =======
SIGUSR1     true     false    not set
```

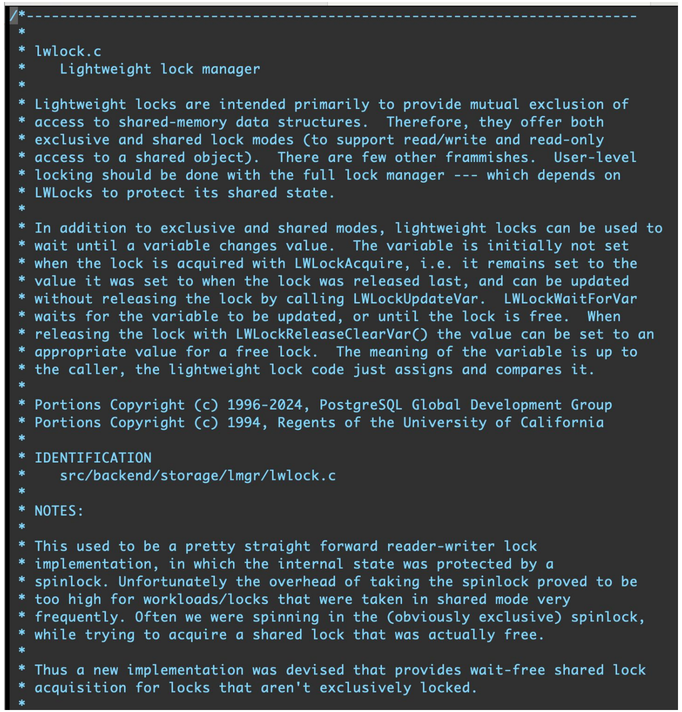
Code Flow

```
(lldb) bt
* thread #1, queue = 'com.apple.main-thread', stop reason = signal SIGSTOP
  * frame #0: 0x0000000198f42f40 libsystem_kernel.dylib`kevent + 8
    frame #1: 0x00000001009a235c postgres`WaitEventSetWaitBlock(set=0x000000014f80c2e0, cur_timeout=-1, occurred_events=0x000000016fad9910,
nevents=1) at latch.c:1737:7
    frame #2: 0x00000001009a16b4 postgres`WaitEventSetWait(set=0x000000014f80c2e0, timeout=-1, occurred_events=0x000000016fad9910, nevents=1,
wait_event_info=100663296) at latch.c:1525:8
    frame #3: 0x000000010073530c postgres`secure_read(port=0x000000014f80aa90, ptr=0x0000000100f66400, len=8192) at be-secure.c:214:3
    frame #4: 0x0000000100742b48 postgres`pq_recvbuf at pqcomm.c:923:7
    frame #5: 0x00000001007429e4 postgres`pq_getbyte at pqcomm.c:969:7
    frame #6: 0x00000001009f3458 postgres`SocketBackend(inBuf=0x000000016fad9b18) at postgres.c:374:10
    frame #7: 0x00000001009efe3c postgres`ReadCommand(inBuf=0x000000016fad9b18) at postgres.c:497:12
    frame #8: 0x00000001009ef364 postgres`PostgresMain(dbname="postgres", username="pavan") at postgres.c:4699:15
    frame #9: 0x00000001009e6b44 postgres`BackendMain(startup_data="", startup_data_len=4) at backend_startup.c:105:2
    frame #10: 0x00000001008c4320 postgres`postmaster_child_launch(child_type=B_BACKEND, startup_data="", startup_data_len=4,
client_sock=0x000000016fad9d78) at launch_backend.c:277:3
    frame #11: 0x00000001008cc0e4 postgres`BackendStartup(client_sock=0x000000016fad9d78) at postmaster.c:3594:8
    frame #12: 0x00000001008c8e18 postgres`ServerLoop at postmaster.c:1676:6
    frame #13: 0x00000001008c7d04 postgres`PostmasterMain(argc=3, argv=0x000060000329d360) at postmaster.c:1374:11
    frame #14: 0x000000010074bdac postgres`main(argc=3, argv=0x000060000329d360) at main.c:199:3
    frame #15: 0x0000000198bf3154 dyld`start + 2476
```

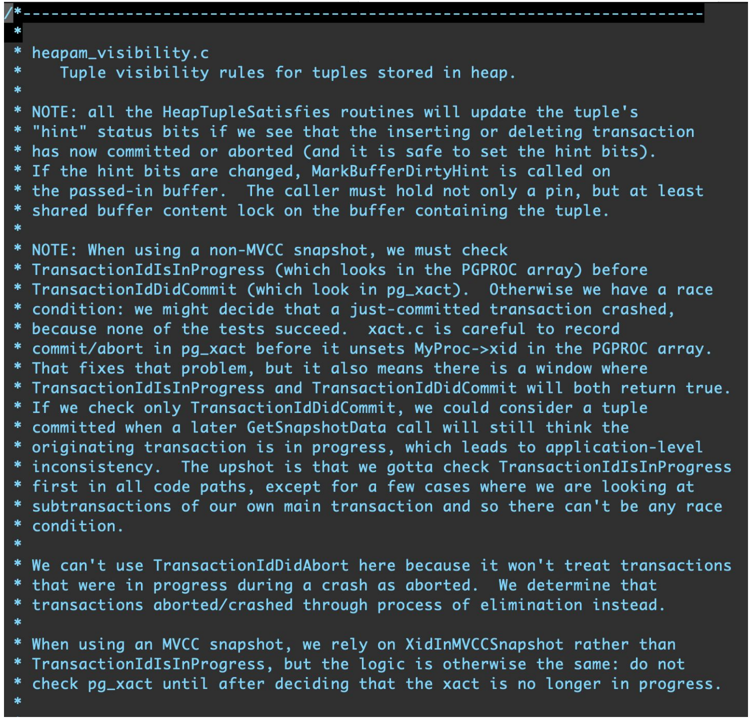
Other Useful Tricks

- Useful GUCs
  - log_min_messages
  - log_line_prefix
- Developer GUCs (may require compilation with assert enabled)
  - debug_print_plan/debug_pretty_print
  - trace_locks
  - trace_userlocks
  - trace_lwlocks
  - debug_deadlock
  - wal_debug
  - pre_auth_delay/post_auth_delay

Resources - Source Code



```
/*-------------------------------------------------------------------------
 *
 * lwlock.c
 *    Lightweight lock manager
 *
 * Lightweight locks are intended primarily to provide mutual exclusion of
 * access to shared-memory data structures.  Therefore, they offer both
 * exclusive and shared lock modes (to support read/write and read-only
 * access to a shared object).  There are few other frammishes.  User-level
 * locking should be done with the full lock manager --- which depends on
 * LWLocks to protect its shared state.
 *
 * In addition to exclusive and shared modes, lightweight locks can be used to
 * wait until a variable changes value.  The variable is initially not set
 * when the lock is acquired with LWLockAcquire, i.e. it remains set to the
 * value it was set to when the lock was released last, and can be updated
 * without releasing the lock by calling LWLockUpdateVar.  LWLockWaitForVar
 * waits for the variable to be updated, or until the lock is free.  When
 * releasing the lock with LWLockReleaseClearVar() the value can be set to an
 * appropriate value for a free lock.  The meaning of the variable is up to
 * the caller, the lightweight lock code just assigns and compares it.
 *
 * Portions Copyright (c) 1996-2024, PostgreSQL Global Development Group
 * Portions Copyright (c) 1994, Regents of the University of California
 *
 * IDENTIFICATION
 *    src/backend/storage/lmgr/lwlock.c
 *
 * NOTES:
 *
 * This used to be a pretty straight forward reader-writer lock
 * implementation, in which the internal state was protected by a
 * spinlock. Unfortunately the overhead of taking the spinlock proved to be
 * too high for workloads/locks that were taken in shared mode very
 * frequently. Often we were spinning in the (obviously exclusive) spinlock,
 * while trying to acquire a shared lock that was actually free.
 *
 * Thus a new implementation was devised that provides wait-free shared lock
 * acquisition for locks that aren't exclusively locked.
 *
```



```
/*-------------------------------------------------------------------------
 *
 * heapam_visibility.c
 *    Tuple visibility rules for tuples stored in heap.
 *
 * NOTE: all the HeapTupleSatisfies routines will update the tuple's
 * "hint" status bits if we see that the inserting or deleting transaction
 * has now committed or aborted (and it is safe to set the hint bits).
 * If the hint bits are changed, MarkBufferDirtyHint is called on
 * the passed-in buffer.  The caller must hold not only a pin, but at least
 * shared buffer content lock on the buffer containing the tuple.
 *
 * NOTE: When using a non-MVCC snapshot, we must check
 * TransactionIdIsInProgress (which looks in the PGPROC array) before
 * TransactionIdDidCommit (which look in pg_xact).  Otherwise we have a race
 * condition: we might decide that a just-committed transaction crashed,
 * because none of the tests succeed.  xact.c is careful to record
 * commit/abort in pg_xact before it unsets MyProc->xid in the PGPROC array.
 * That fixes that problem, but it also means there is a window where
 * TransactionIdIsInProgress and TransactionIdDidCommit will both return true.
 * If we check only TransactionIdDidCommit, we could consider a tuple
 * committed when a later GetSnapshotData call will still think the
 * originating transaction is in progress, which leads to application-level
 * inconsistency.  The upshot is that we gotta check TransactionIdIsInProgress
 * first in all code paths, except for a few cases where we are looking at
 * subtransactions of our own main transaction and so there can't be any race
 * condition.
 *
 * We can't use TransactionIdDidAbort here because it won't treat transactions
 * that were in progress during a crash as aborted.  We determine that
 * transactions aborted/crashed through process of elimination instead.
 *
 * When using an MVCC snapshot, we rely on XidInMVCCSnapshot rather than
 * TransactionIdIsInProgress, but the logic is otherwise the same: do not
 * check pg_xact until after deciding that the xact is no longer in progress.
 *
```

EDB Postgres for the AI Generation

Resources - Builtin Contrib Modules

- pg_stat_statements, auto_explain
- postgres_fdw
- test_decoding
- pg_prewarm

Additional Resources

- src/backend/access/transam/README
- src/backend/access/transam/README.parallel
- src/backend/access/hash/README
- src/backend/access/brin/README
- src/backend/access/rmgrdesc/README
- src/backend/access/nbtree/README
- src/backend/access/heap/README.HOT
- src/backend/access/heap/README.tuplock
- src/backend/optimizer/plan/README
- src/backend/optimizer/README
- src/backend/nodes/README
- src/backend/utils/misc/README
- src/backend/utils/resowner/README
- src/backend/utils/mmgr/README
- src/backend/utils/fmgr/README
- src/backend/storage/lmgr/README.barrier
- src/backend/storage/lmgr/README
- src/backend/storage/lmgr/README-SSI
- src/backend/storage/page/README
- src/backend/storage/freespace/README
- src/backend/storage/smgr/README
- src/backend/storage/buffer/README
- src/backend/executor/README
- src/backend/replication/README

Thank You

EDB
Postgres® for the AI Generation