READ MODE · ANSWER FIRST

# How PostgreSQL MVCC Actually Works

An interactive, visual explanation of How PostgreSQL MVCC Actually Works, from setup through the complete system.

CHEAT SHEET · 6 ESSENTIAL IDEAS

## The whole story in 6 lines

How PostgreSQL MVCC Actually Works becomes easier to reason about when every stage is connected as one system.

1. A logical row can span several tuple headers, each with its own xmin/xmax. Readers later decide which one is visible.
2. A PostgreSQL snapshot is a cut through transaction ID space: xmin, xmax, and the set of xids still in progress.
3. Visibility does not flip at the first write. Other sessions keep treating the writer xid as in-progress until commit.
4. MVCC is mostly a predicate over tuple headers plus one snapshot. The database keeps versions; the reader chooses one.
5. Snapshot age, not row ownership, explains why one reader still sees old data while a newer reader sees the committed rewrite.
6. MVCC shifts work from blocking to cleanup. Vacuum and pruning are what keep old versions from accumulating forever.

STAGE 1 OF 8

# Setup


Welcome. PostgreSQL can handle many readers and writers at the same time without anyone waiting in line. The secret is called MVCC, which stands for Multi-Version Concurrency Control. Think of it like a library where every book has multiple editions on the shelf, and each visitor is allowed to read whichever edition was current when they walked in.

A heap tuple is one physical row stored on disk. PostgreSQL does not overwrite old rows. Instead it keeps multiple versions of the same logical row, each in its own heap tuple.

Every tuple carries two stamps called xmin and xmax. xmin records which transaction created the tuple. xmax records which transaction deleted or replaced it. Together they tell readers whether the tuple is still alive.

A snapshot is a frozen picture of which transactions are finished and which are still running. When a reader starts work, it grabs a snapshot and uses it to decide which tuple versions to trust.

A transaction ID, or xid, is a number PostgreSQL assigns to every write operation. Readers compare tuple stamps against their snapshot to figure out what is visible. Vacuum is the background cleanup process that eventually removes dead tuple versions nobody can see anymore.

Over the next six stages we will build up the complete picture: how versions form chains, how snapshots capture a consistent cut, how writers stamp tuples, how visibility rules choose the right version, how readers diverge without conflict, and how vacuum keeps it all tidy. Let us start with the physical foundation: version chains.

# Build Version Chains


02 BUILD VERSION CHAINS

Here is the first thing most people get wrong about PostgreSQL: it never overwrites a row in place. When you INSERT a row, PostgreSQL writes a heap tuple and stamps its xmin with the inserting transaction. That tuple is the first and only version of this logical row.

When an UPDATE arrives, PostgreSQL does not touch the old bytes. It writes a brand-new tuple version alongside the original. The old tuple stays exactly where it was. This is the core of MVCC: keep the old, write the new.

The older tuple gets an xmax stamp pointing to the updating transaction, and a CTID pointer that leads to the newer version. This is how versions link into a chain. Each link says "the next version lives over there."

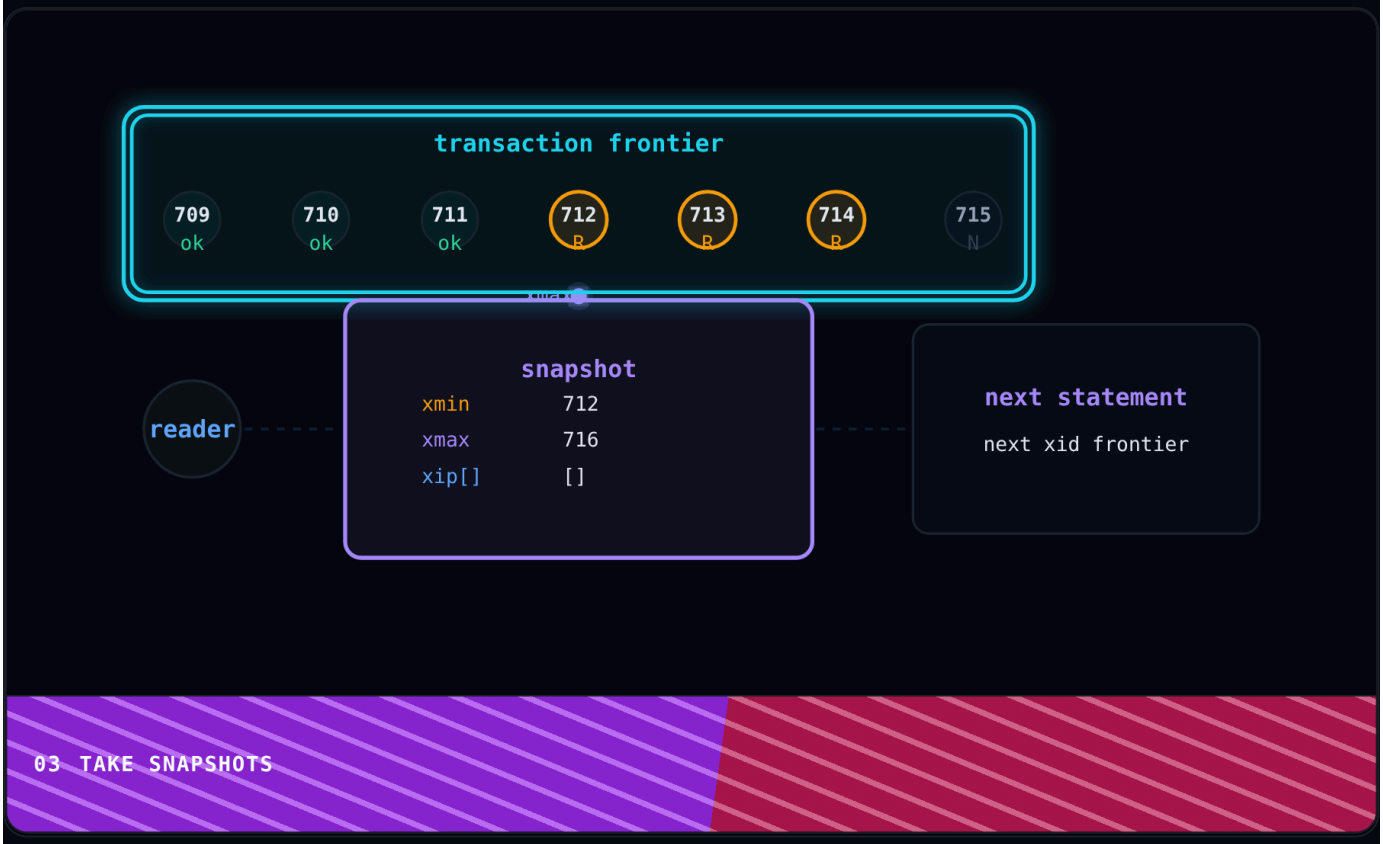
Notice how every version keeps its own xmin and xmax. These are the stamps that snapshots will judge later. A reader does not need to ask anyone "is this row locked?" It just reads the stamps and decides for itself.

Try changing the Index path control in the sidebar to Indexed. A HOT (Heap-Only Tuple) update can reuse the old index entry because only the heap chain grows. An indexed update needs a fresh pointer into the heap. HOT updates are cheaper, but they only work when the updated columns are not in any index.

By the end, one logical row is really a chain of physical tuple versions. The index gets you into the chain. The chain gives you candidates. But who decides which candidate is the right one? That is what snapshots do, and that is where we are headed next.

STAGE 3 OF 8

# Take Snapshots




03 TAKE SNAPSHOTS

Now that we know rows live as version chains, the question is: how does a reader pick the right version? The answer starts here. Before a reader trusts any tuple header, it captures a snapshot. Think of it as freezing a photograph of which transactions are done and which are still running.

The snapshot sets xmin to the oldest transaction that is still active. Any transaction older than xmin is definitely committed or aborted, so the reader already knows how to judge it.

It also sets xmax to the next transaction ID that has not been assigned yet. Anything at or beyond xmax is automatically invisible because it started after the snapshot was taken.

Between xmin and xmax there may be transactions that started but have not committed yet. Those get copied into the xip list. The snapshot now has three pieces: a floor, a ceiling, and a list of gaps in between.

Try switching the Moment control in the sidebar. When the reader arrives early, fewer transactions are running and the xip list is short. Arrive mid-write and the list grows. The timing of the snapshot changes what the reader is allowed to see.

One last wrinkle: READ COMMITTED grabs a fresh snapshot for every statement, so the view can shift between queries. REPEATABLE READ keeps the same snapshot for the entire transaction. Switch the Isolation control to see how the next-statement behavior changes. Now that we have snapshots, let us see how writers stamp the tuples that snapshots judge.

STAGE 4 OF 8

# Write New Tuples


We just learned how readers grab snapshots. Now let us see the other side: what a writer actually does to the heap. Remember the version chain from Stage 1? The writer is about to add a new link to that chain.

The writer reserves a transaction ID and immediately stamps it into the old tuple's xmax field. This is the moment the old version gets marked. But remember: xmax alone does not make the change visible. The snapshot rules from Stage 2 still apply.

For an UPDATE, a brand-new tuple appears with xmin set to the writer's xid. For a DELETE, there is no new tuple. The old tuple's xmax is the only evidence. Switch the Change control in the sidebar to Delete and watch the new tuple disappear.

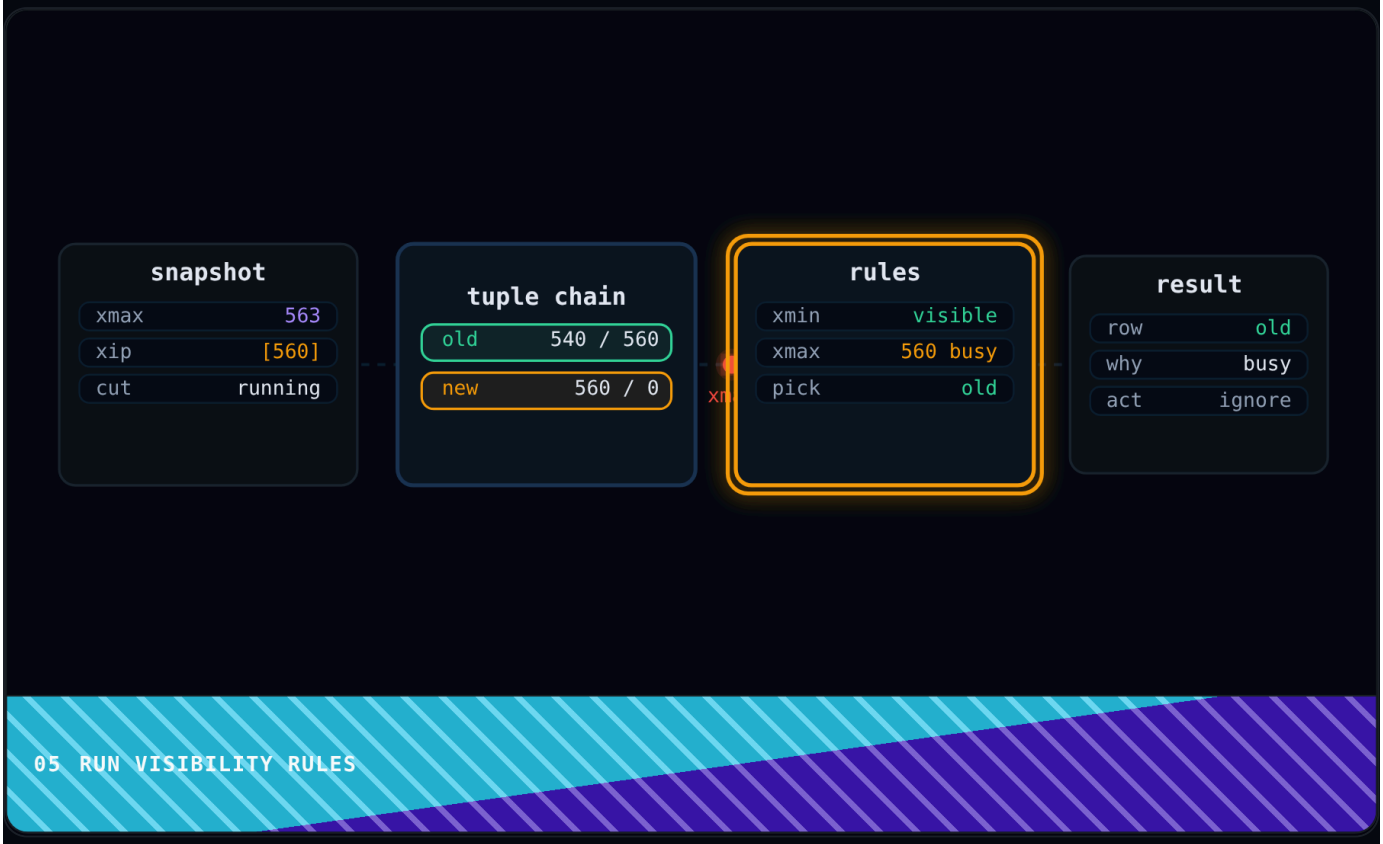
Here is the critical insight: other readers still see the old version. Their snapshots treat the writer's xid as "in progress," so the xmax stamp does not count yet. Nobody is blocked. Nobody is waiting.

COMMIT flips the writer's xid to "committed" in the pg_xact log. Now every future snapshot will treat those stamps as real. ROLLBACK does the opposite: future snapshots pretend the stamps never happened. Try toggling the Outcome control to see how the final state changes.

The heap now has either a new live version, a committed delete, or the original row restored after rollback. The writer's job is done. But we still have not answered the most important question: how does a reader actually decide which version to return? That is the visibility rules, and that is our next stage.

STAGE 5 OF 8

# Run Visibility Rules




05 RUN VISIBILITY RULES

This is where everything comes together. We have version chains from Stage 1, snapshots from Stage 2, and writer stamps from Stage 3. Now the reader walks the chain and applies two simple checks to each tuple.

Check one: is the creator visible? The reader looks at the tuple's xmin and compares it against the snapshot. If xmin belongs to a transaction that was committed before the snapshot was taken, the creator is visible.

If xmin is too new or still in the snapshot's xip list, this version was written by a transaction the reader cannot see yet. The reader skips it and keeps walking the chain.

Check two: is the deleter or replacer visible? The reader now looks at xmax. If xmax belongs to a committed transaction visible to the snapshot, then this version has been superseded. The reader must skip it and look for the newer one.

The first tuple whose xmin passes and whose xmax does not pass is the answer. Try switching the Reader control to After. When the snapshot was taken after the writer committed, the new version wins. Switch to Before and the old version wins. Same chain, different snapshot, different answer.

That is the entire visibility algorithm: two checks, one snapshot, one answer. The database keeps all the versions. The reader's snapshot decides which one is real. This explains something important: two readers with different snapshots can look at the exact same chain and get different rows. Let us see that in action next.

STAGE 6 OF 8

# Readers Diverge


06 READERS DIVERGE

We just learned the visibility rules. Now let us watch them in action with two readers and one writer sharing the same row. Reader A starts first and grabs an older snapshot. Remember from Stage 2: that snapshot is a frozen photograph of which transactions are done.

While Reader A is working, a writer creates a newer tuple version for the same logical row. This is the same write process from Stage 3: the old tuple gets an xmax stamp, and (for updates) a new tuple appears with xmin set to the writer's xid.

Reader A queries the row again. Its snapshot predates the writer's xid, so the visibility rules from Stage 4 say: the old version's xmin is visible, and its xmax is not visible yet. Reader A still sees the old tuple. No lock, no wait, no conflict.

Reader B starts now with a newer snapshot. If the writer has committed, Reader B's visibility rules resolve differently: the old version's xmax is now visible, so Reader B skips to the new version. Same chain, different answer.

Try switching the Reader A control to Refreshes. Under READ COMMITTED, the next statement grabs a fresh snapshot. If the writer committed in between, Reader A may suddenly see the new version on its next query. Under REPEATABLE READ, Reader A stays pinned to the old view for the entire transaction.

This is the payoff of MVCC. Readers and writers coexist because visibility is snapshot-based, not lock-based. Nobody freezes the row for everyone else. But there is a cost: all those old tuple versions pile up. Who cleans them? That is where vacuum comes in, and it is our final technical topic.

STAGE 7 OF 8

# Vacuum Dead Versions


07 VACUUM DEAD VERSIONS

Every update and delete we saw in Stage 3 left an old tuple behind. Those dead versions do not disappear when the writer commits. They sit on the heap page taking up space and slowing down scans. Try increasing the Dead control in the sidebar to see how quickly the page fills up.

Vacuum is a background worker that periodically scans heap pages looking for dead tuple versions it can safely remove. It cannot just delete everything marked dead, because some reader might still need an old version. Remember Reader A from Stage 5? Its older snapshot might still reference a dead tuple.

The safety check is called OldestXmin. It is the xmin of the oldest snapshot still active anywhere in the system. Any dead tuple whose deleting transaction is older than OldestXmin is safe to remove, because no living snapshot can possibly see it.

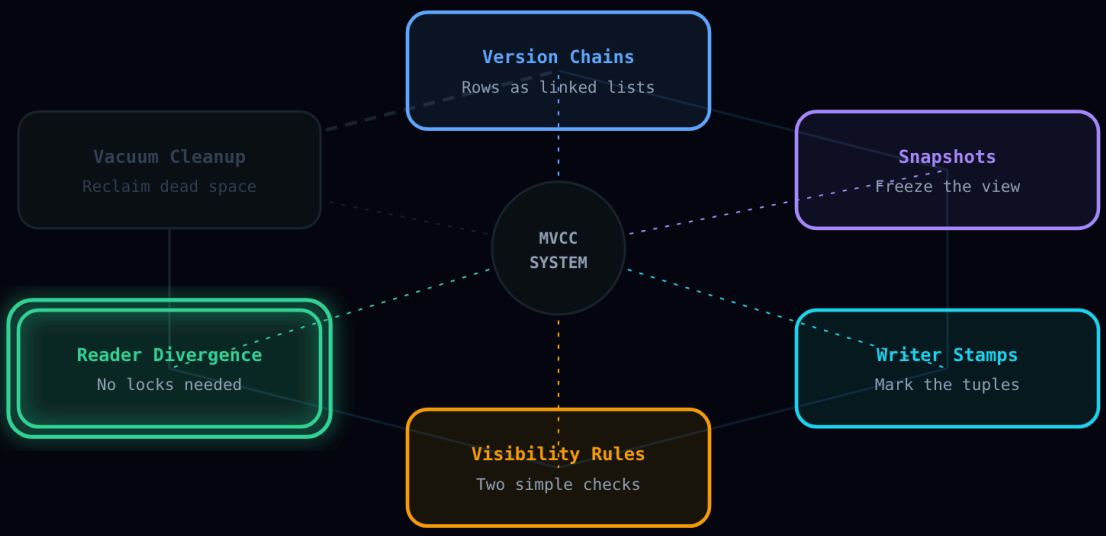
Toggle the Long reader control off. When no old snapshot exists, the horizon advances and vacuum can reclaim every dead slot. Turn it back on and watch one dead tuple stay protected. That protected tuple is the price of letting long readers see consistent data without locks.

Once the horizon clears, vacuum prunes the removable versions and marks their slots as reusable. New inserts and updates can claim those slots, keeping the table from growing endlessly. The page is cleaned up without stopping any reader.

That cleanup loop is what keeps MVCC practical at scale. Without vacuum, every table would grow forever. With it, old versions eventually turn back into free space. Now let us step back and see the whole picture together.

STAGE 8 OF 8

# Recap




08 RECAP

We started with version chains. PostgreSQL never overwrites a row. Every UPDATE creates a new tuple and links it to the old one. That chain of physical versions is the foundation everything else builds on.

Then we learned how snapshots work. A reader freezes a photograph of which transactions are done and which are still running. That photograph becomes the lens through which the reader judges every tuple it encounters.

Writers stamp tuples with their transaction ID. They mark the old version's xmax and create a new version's xmin. But nothing becomes visible to other readers until the writer commits.

The visibility rules are the heart of MVCC: check xmin, check xmax, and the first tuple that passes both checks is the answer. Two readers with different snapshots can get different answers from the same chain.

That is exactly what reader divergence looks like. Old readers keep seeing old data. New readers see committed changes. Nobody waits. Nobody locks. The snapshot decides.

Finally, vacuum sweeps up the dead versions that no snapshot can see anymore. Without it, the table would grow forever. With it, MVCC stays practical at any scale.

Here is the whole picture: PostgreSQL keeps every version, lets each reader freeze its own snapshot, applies two simple visibility checks, and cleans up behind itself. That is how it achieves lock-free reads alongside concurrent writes. The tradeoff is storage cost and cleanup overhead, but the payoff is that readers and writers never block each other.