

Explain in details how Postgress does WAL and how it would differ from what we have discussed we want to do.

Below is a **precise, systems-level explanation** of how **PostgreSQL WAL** actually works, followed by a **point-by-point contrast** with what *TruthDB's WAL* (as we've discussed) is trying to achieve.

I'll avoid fluff and focus on *mechanics, invariants, and consequences*.

1. What PostgreSQL WAL is (conceptually)

PostgreSQL WAL is:

A physical redo log used to guarantee crash recovery and durability for a mutable heap-based storage engine.

Key idea:

Data pages are NOT written atomically. WAL makes torn writes recoverable.

Postgres is *not* event-sourced. WAL is a **means**, not the **source of truth**.

2. What PostgreSQL WAL logs (very important)

Postgres WAL logs **physical changes**, not logical intent.

Examples:

- "On page X, offset Y, overwrite bytes A–B with bytes C"
- "Initialize page P with this full image"
- "Insert tuple T at slot S in heap page P"

This implies:

- WAL records are **storage-layout dependent**
- WAL format is tightly coupled to:
 - page size (usually 8 KB)
 - heap/index formats

- visibility rules (MVCC)

This is why:

- WAL records are **not portable**
 - WAL is **not replayable on a different engine**
 - WAL is **not a domain log**
-

3. WAL record structure (simplified but accurate)

Each WAL record contains:

- **LSN (Log Sequence Number)**
A byte offset in the WAL stream (monotonic).
 - **Resource Manager ID (RMGR)**
Identifies *who* knows how to replay this record
(heap, btree, transaction, clog, etc.)
 - **Redo payload**
 - Often page deltas
 - Sometimes full page images (FPI)
 - **Transaction metadata**
 - XID
 - commit / abort markers
-

4. When PostgreSQL writes to WAL

Normal write path

1. Client issues SQL
2. Executor modifies in-memory buffers (shared_buffers)
3. WAL records are generated **before dirty pages are flushed**
4. WAL records appended to WAL buffers
5. WAL buffers flushed to disk on:
 - COMMIT (sync commit)
 - background writer
 - WAL writer

Rule:

WAL must reach disk before corresponding data pages

(write-ahead rule)

5. Checkpoints (why WAL doesn't grow forever)

Postgres periodically performs a **checkpoint**:

- Flushes *all dirty pages* to disk
- Writes a checkpoint record to WAL
- After checkpoint, WAL before it can be recycled

This means:

- WAL is **bounded**
 - Recovery time is **bounded**
 - WAL is **not authoritative**
-

6. Crash recovery in PostgreSQL

When does recovery happen?

Every startup after an unclean shutdown.

How recovery works

1. Read last checkpoint record
2. Start replaying WAL **from checkpoint LSN**
3. For each WAL record:
 - Invoke RMGR redo function
 - Re-apply physical changes if needed
4. Stop at end of WAL
5. Database is now *consistent*

Important:

- WAL replay is **idempotent**
 - Pages have LSNs to avoid double-apply
 - Recovery logic is *complex and storage-specific*
-

7. Failure detection

Postgres "knows" it crashed because:

- `pg_control` indicates DB was not shut down cleanly
- WAL end does not contain a clean shutdown marker

There is **no per-write failure detection**.

Failure is detected **coarsely at startup**.

8. What PostgreSQL WAL is *not*

Thing	PostgreSQL WAL
Source of truth	✗ No
Event log	✗ No
Replayable business log	✗ No
Portable	✗ No
User-visible	✗ No
Append-only forever	✗ No

Now: How this differs from TruthDB WAL

This is where the architectural fork happens.

9. Core philosophical difference

PostgreSQL

WAL exists to protect mutable state

TruthDB (your design)

WAL *is* the state

This single sentence explains *everything* that follows.

10. Authority and lifecycle

Aspect	PostgreSQL WAL	TruthDB WAL
Authority	Secondary	Primary
Retention	Recycled	Retained
Purpose	Crash safety	State + history
Replay	Internal only	Fundamental
Consumers	Storage engine	Engine + queries + replication

11. Logical vs physical

PostgreSQL

- WAL logs **how bytes changed**
- Replay requires identical storage engine
- Impossible to reconstruct intent

TruthDB

- WAL logs **domain-level facts**
 - events
 - commands
 - mutations
- Storage views are **derived**
- Replay is intentional and deterministic

This means:

- TruthDB WAL **must be stable across versions**
- PostgreSQL WAL **cannot be**

12. Recovery semantics

PostgreSQL recovery

- Happens only after crash
- Goal: restore last consistent heap state
- WAL replay is destructive (overwrites pages)

TruthDB recovery

- Happens on:
 - startup
 - replica catch-up
 - re-index
 - rebuild
 - verification
 - Goal: reconstruct *current truth* from log
 - WAL replay is **constructive**
-

13. Checkpoints vs snapshots

PostgreSQL	TruthDB
Checkpoint = flush dirty pages	Snapshot = derived optimization
Mandatory	Optional
Resets recovery window	Does not reset authority
WAL becomes discardable	WAL remains canonical

14. Failure model

PostgreSQL

- Assumes:
 - page corruption
 - torn writes
- WAL protects against *hardware realities*

TruthDB

- Assumes:
 - WAL append is atomic
 - corruption is detectable via checksums
- Failure is **record-level**, not page-level

This enables:

- per-record verification

- partial recovery
 - segment fencing
 - strong invariants
-

15. Why PostgreSQL *cannot* do what TruthDB wants

Postgres is boxed in by:

- Heap files
- MVCC visibility rules
- Physical page formats
- Decades of backward compatibility

Retrofitting:

- event sourcing
- immutable state
- deterministic replay

...would require **rewriting the entire storage engine**.

16. Why TigerBeetle is closer (but still different)

TigerBeetle:

- Treats log as authoritative
- Has deterministic replay
- Uses strict invariants

But:

- Domain is fixed (ledger)
- State machine is predefined

TruthDB:

- Wants **generalized truth storage**
 - With **WASM-driven interpretation**
 - And **multiple materialized views**
-

17. One-sentence summary

**PostgreSQL WAL exists to save mutable pages from crashing;
TruthDB WAL exists so that mutable pages don't matter at all.**

If you want, next step we can:

- map PostgreSQL concepts → TruthDB equivalents (checkpoint → snapshot, LSN → segment+offset, RMGR → domain handler)
- or formally define **TruthDB WAL invariants** (what *must* never happen)



Ok, stop. You are just regurgitating random noise from the internet. I want you to explain EVERYTHING in DETAILS. No abbreviation should go unexplained. Forget truthdb now. Dont just utter bullet point style phrases you dont even know the meaning of.

Thought for 13s >

Got it. I'll do this as a **full, bottom-up explanation** of PostgreSQL WAL, and I will **expand every abbreviation** the first time it appears. I'll also avoid "drive-by" phrases: whenever I introduce a term, I'll explain what it *is*, what it *does*, and why it exists.

I'll anchor the explanation to PostgreSQL's own docs and a few source references where it matters. PostgreSQL +3

0) What problem WAL solves in PostgreSQL

PostgreSQL stores table and index data in **files** on disk. Queries modify data in memory first (in a shared cache) and later those modified pages are written back to disk.

The key problem: **a crash can occur at any moment:**

- power loss
- operating system panic
- PostgreSQL process crash
- storage write being interrupted (a "torn" write: only part of a page reaches disk)

If PostgreSQL only wrote changed data pages directly to disk, then after a crash the on-disk files might be in a **half-updated** state: some pages reflect new changes, others don't, and some pages might be physically corrupted mid-write. That can destroy consistency.

Write-Ahead Logging (WAL) is the mechanism that ensures PostgreSQL can always recover the database to a consistent state after a crash.

1) WAL (Write-Ahead Logging) in one accurate sentence

WAL is an append-only log of changes such that PostgreSQL can re-apply ("redo") the changes after a crash, and it enforces the rule that the log is made durable before the corresponding data pages are allowed to reach disk.

That "write-ahead rule" is the core: **log first, data later**.

PostgreSQL's own docs describe checkpoints and recovery as scanning forward from a checkpoint location stored in `pg_control`. [PostgreSQL +1](#)

2) The basic storage model you must understand first

2.1 Page

A **page** is a fixed-size block of data used by PostgreSQL storage files, typically **8 kilobytes** (8 KB) by default. A table file is basically a sequence of such pages.

2.2 Buffer

A **buffer** is an in-memory copy of a page. PostgreSQL keeps many pages cached in memory for speed.

2.3 shared_buffers

`shared_buffers` is PostgreSQL's main shared memory area that holds these cached pages (buffers). It is shared among PostgreSQL backend processes (server processes handling client connections).

2.4 Dirty page (dirty buffer)

A page/buffer is **dirty** when it has been modified in memory but the corresponding on-disk page file has not yet been updated.

2.5 Why dirty pages are dangerous without WAL

Because the system can crash:

- after some dirty pages are written,
- before other dirty pages are written,
- or while a page is being written (torn write)

So the database files may become inconsistent unless PostgreSQL has a second source of truth to repair from.

3) What "WAL record" means (and what it contains)

A **WAL record** is a single "entry" appended to the WAL stream describing some change that must be recoverable.

PostgreSQL WAL is generally **redo logging** (not undo logging): it records information needed to **re-apply** changes during crash recovery.

3.1 LSN (Log Sequence Number)

LSN stands for **Log Sequence Number**.

It is not a "record number". It is a **byte position** in the WAL stream. Think: "offset into the WAL". It increases monotonically as WAL is appended.

PostgreSQL uses LSNs everywhere to say "this page includes WAL up to position X" or "flush WAL up to position Y".

3.2 XID (Transaction ID)

XID stands for **Transaction Identifier** (in code it's often `TransactionId`).

Transactions in PostgreSQL get an XID (with some special cases, but for this explanation: assume "each real writing transaction has an XID").

WAL records often carry the XID so recovery and replication can reason about transaction boundaries and commit/abort markers.

3.3 RMGR (Resource Manager)

RMGR stands for **Resource Manager**.

This is critical: PostgreSQL has many subsystems that can produce WAL:

- heap (table storage)
- btree (B-tree index)
- hash index
- gist, gin indexes
- transaction log structures
- etc.

Each WAL record includes a **resource manager identifier** that tells PostgreSQL **which redo function** knows how to interpret and re-apply that record during recovery.

This is why WAL is not "generic": WAL records are interpreted by code specific to the resource manager that created them.

In PostgreSQL source, the WAL record header contains fields including:

- `xl_rmid` (resource manager id)
- `xl_xid` (transaction id)
- `xl_tot_len` (total length)
- `xl_info` (extra info bits)
- `xl_prev` (previous record pointer)

and a checksum field in modern versions. Doxygen Postgre... +2

3.4 FPI (Full Page Image)

FPI stands for **Full Page Image**.

Sometimes PostgreSQL logs not just a "delta" (change description), but a full copy of an entire page into WAL.

Why? Because if a data page write is torn/corrupted, then a later delta record might be impossible to apply safely. A full page image lets recovery restore the page completely.

PostgreSQL's WAL record/block format has explicit support for logging full-page writes for modified blocks. Doxygen Postgre...

4) Where WAL lives on disk: `pg_wal` and segments

The WAL stream is stored as **segment files** in the directory:

- `pg_wal` (newer PostgreSQL)
- historically `pg_xlog` (older versions)

A **segment** is a fixed-size file (commonly 16 MB by default). WAL is appended across segments.

This matters for:

- recycling old segments
 - archiving segments
 - replication shipping
-

5) Who writes WAL, and when

PostgreSQL uses multiple processes (in the OS-process sense) with distinct jobs.

5.1 Backend process (the server process handling a client)

When you connect via psql or an application, PostgreSQL forks/spawns a backend process for that connection (exact mechanics vary by platform/config).

That backend executes queries and modifies pages in `shared_buffers`.

When it performs a change that must be crash-safe, it constructs WAL records and inserts them into WAL buffers.

5.2 WAL buffers

WAL is first accumulated in memory in **WAL buffers** (a separate shared memory area from `shared_buffers`).

WAL buffers are then written out to disk.

5.3 WAL writer (background process)

The **WAL writer** is a background process that periodically flushes WAL buffers to disk to reduce how often backends must do it themselves.

PostgreSQL documentation explicitly describes that writing WAL buffers is done by functions like `XLogWrite` and can be triggered by `insert/flush` or by the WAL writer.

PostgreSQL +1

5.4 background writer

The **background writer** is a background process that writes dirty data pages from `shared_buffers` out to the table/index files to reduce spikes where client backends must write pages themselves. PostgreSQL

5.5 checkpointer

The **checkpointer** is the process responsible for performing **checkpoints** (explained next), coordinating mass flushing of dirty pages and writing a checkpoint record to WAL.

PostgreSQL +1

6) Checkpoint: what it is, exactly, and why it exists

A **checkpoint** is a point in time at which PostgreSQL ensures:

All data changes described by WAL **up to the checkpoint point** are reflected in the main data files on disk.

Mechanically, a checkpoint involves two key actions:

1. PostgreSQL **flushes dirty pages** (pages changed in memory) from `shared_buffers` to their data files on disk (heap and index files).
2. PostgreSQL writes a **checkpoint WAL record** saying, essentially:
 - "At WAL location L, the database files are consistent up through point R (redo point)"
 - and it stores the checkpoint location in `pg_control`

PostgreSQL docs: "At checkpoint time, all dirty data pages are flushed to disk and a special checkpoint record is written to the WAL file." PostgreSQL +1

6.1 Why checkpoints matter

Without checkpoints, crash recovery might need to replay WAL from a very old point. Checkpoints bound recovery time: recovery replays only WAL from the last checkpoint's redo point forward.

Docs (older but clear): crash recovery looks at the latest checkpoint record to find the redo point from which to start redo. PostgreSQL +1

7) The "write-ahead" rule, precisely

The write-ahead property means:

A dirty data page is not allowed to be written to disk unless the WAL describing the changes on that page has already been flushed to durable storage.

Otherwise you get the deadly scenario:

- data page reaches disk with new data
- but WAL did *not* reach disk
- crash occurs
- recovery has no record of that change
- database becomes inconsistent

PostgreSQL enforces this by tracking, for each modified page, the **LSN** of the WAL record that last modified it, and ensuring WAL is flushed at least to that LSN before the page write is considered safe.

8) fsync and durability: what “durable” means here

When we say “WAL is flushed to disk”, we mean two things:

1. the bytes have been written to the operating system (write syscall)
2. the operating system has been forced to push them to stable storage (an `fsync` - type operation)

PostgreSQL's documentation for WAL configuration mentions:

- `XLogWrite` writing WAL buffers
- `issue_xlog_fsync` performing the sync depending on configured method and explains that some sync methods imply the write call itself ensures syncing.

PostgreSQL

So: durability here means “survives OS crash/power loss”, not just “was copied into the OS page cache”.

9) Commit path: what must happen for COMMIT to succeed

When a transaction commits, PostgreSQL must ensure that if it reports success to the client, then after a crash the committed changes can be recovered.

For a “synchronous commit” (the common safe mode):

- The commit generates WAL records including a commit record.
- PostgreSQL must flush WAL to disk at least through that commit record's LSN before acknowledging success.

Dirty data pages themselves can be written later; WAL guarantees redo can reconstruct.

This is the essence of WAL-based durability: **commit is about WAL persistence, not immediate data page persistence.**

10) Crash recovery: step-by-step, with the actual files involved

Crash recovery happens when PostgreSQL starts and detects it was not shut down cleanly.

10.1 `pg_control`

`pg_control` is a small control file in the data directory that stores critical state, including the location of the latest checkpoint record.

PostgreSQL docs (WAL internals): after a checkpoint is made and WAL flushed, the checkpoint position is saved in `pg_control`. At start of recovery, server reads `pg_control` then the checkpoint record, then performs redo scanning forward from the indicated location. PostgreSQL

10.2 Recovery algorithm in plain language

1. Read `pg_control` to find the last checkpoint's WAL location.
2. Read the checkpoint record from WAL.
3. Determine the **redo start point** (the LSN from which redo must begin).
4. Scan WAL forward from that redo point:
 - For each WAL record:
 - look at RMGR (resource manager id)
 - call that resource manager's redo function
 - apply changes to data pages as needed
5. Stop when you reach the end of WAL (the last valid record).
6. Database is now consistent and can accept writes again.

10.3 Why redo is safe to repeat (idempotence)

During recovery, PostgreSQL can encounter WAL records for changes that might already be on disk (because some dirty pages were flushed before crash). The system must avoid corrupting data by applying changes twice.

PostgreSQL prevents this by using page LSNs and record structure so the redo code can check whether a change is already reflected in the page.

(That is why the LSN is more than a "counter": it's used as a correctness guard.)

11) How PostgreSQL knows WAL records are valid (integrity)

Modern PostgreSQL WAL records include checksums (CRC32C) so that during recovery it can detect partial/corrupted WAL records at the end (common after a crash mid-write). The record header includes a checksum field in source (`xl_crc`).

Doxygen Postgre... +1

Recovery will stop at the last valid record; any garbage after that is ignored because it was never fully written.

12) "Continuous archiving" and why WAL is used for backup/PITR

PITR stands for **Point-In-Time Recovery**.

PostgreSQL can archive WAL segments to external storage. If you have:

- a base backup (a full snapshot of data files)
- plus a continuous sequence of WAL segments after that snapshot

...then PostgreSQL can restore the base backup and replay WAL forward up to a chosen time/LSN/transaction to reconstruct the database as-of that point.

PostgreSQL documentation calls this "Continuous Archiving and Point-in-Time Recovery" and states you need a continuous sequence of archived WAL files extending back at least to the start of the base backup. PostgreSQL

13) What the core WAL-related abbreviations mean (quick index)

I'm listing these again just as an index (not as "hand-wavy bullets"), because you asked for zero unexplained abbreviations:

- **WAL**: Write-Ahead Logging (log must be durable before data pages)
- **LSN**: Log Sequence Number (byte position in WAL stream)
- **XID**: Transaction ID (identifier for a transaction)
- **RMGR**: Resource Manager (subsystem that knows how to redo a record)
- **FPI**: Full Page Image (a full page copy logged into WAL)
- **PITR**: Point-In-Time Recovery (restore to a chosen point using WAL)
- **CRC32C**: a checksum algorithm used to detect corrupted records

14) What I did *not* yet explain (so you can push me deeper)

If you want “everything” down to implementation detail, the next layers are:

1. The exact physical layout of a WAL record on disk (headers, block references, payload chunks).
2. How a WAL record can reference multiple modified pages (“blocks”) and include per-block data.
3. The difference between logging “main data” and “block data”.
4. How full-page images interact with “first modification after checkpoint” rules.
5. How replication reads WAL (physical streaming replication) and what “timeline” means.

If you pick **one** of those, I'll go all the way down to bytes/structures and explain every field name (like `xl_tot_len`, `xl_info`, etc.) using PostgreSQL's own record header definitions. Doxygen Postgre... +2

      Sources

How does postgres ensure that `pg_control` is valid?
Seems to imply we need an endless recursion of wals.

Good question — and you're right to be suspicious.

The short answer is: **PostgreSQL does *not* ensure `pg_control` via another WAL.**
There is **no infinite recursion**. Instead, PostgreSQL relies on a **carefully bounded trust base** and very explicit rules about what *must* be trusted at startup.

I'll explain this **from first principles**, slowly, with no hand-waving.

1. What `pg_control` actually is (precisely)

`pg_control` is a **small, fixed-size control file** stored in the PostgreSQL data directory.

It contains **metadata about the database cluster as a whole**, not table data.

Examples of what it stores (explained, not just listed):

- **Database system identifier**
A 64-bit value generated at `initdb`.

Used to ensure WAL files belong to *this* cluster and not some other one.

- **Latest checkpoint record location (LSN)**

A pointer into the WAL stream saying:

"This is the last checkpoint we *believe* was successfully written."

- **State flags**

Whether PostgreSQL was:

- shut down cleanly
- crashed
- in recovery
- promoted from standby
- etc.

- **Configuration constants**

Page size, WAL segment size, alignment assumptions — things that must not change.

Crucially:

`pg_control` is **not part of normal data**, and is **not updated frequently**.

2. Why `pg_control` cannot itself be WAL-protected

This is the core of your question.

If you tried to WAL-protect `pg_control`, you would need:

- a WAL record describing the update to `pg_control`
- which would itself need to be protected
- which would need another WAL
- ... infinite recursion

So PostgreSQL **explicitly stops the chain**.

There is a small set of things that are *assumed* durable without WAL.

This is not a flaw — it is a **deliberate base case**.

Every durable system has one.

3. The durability base case (the key concept)

PostgreSQL's durability stack looks like this:

pgsql

 Copy code

Hardware / filesystem durability

↓

pg_control ← trusted root

↓

WAL

↓

Data pages

The recursion stops at **hardware + filesystem guarantees**.

PostgreSQL assumes:

If the filesystem says “this file was fsynced”, then the bytes are there after a crash.

That assumption is **outside PostgreSQL**.

No database can avoid making *some* assumption at that level.

4. How PostgreSQL writes pg_control safely (mechanically)

This is where the subtlety is.

4.1 Write pattern

PostgreSQL updates pg_control using a **very strict protocol**:

1. Write the entire pg_control structure in one operation
2. Include a **CRC checksum** covering the contents
3. Call fsync on the file
4. Optionally fsync the directory (on some platforms)

There is **no incremental update**.

It's always rewritten as a whole.

4.2 Atomicity assumption

PostgreSQL relies on the fact that:

- pg_control is **small** (well under a filesystem block)
- It is written sequentially
- It is checksummed

If a crash occurs:

- Either the old version is intact
- Or the new version is intact
- Or the file is corrupted → checksum fails

There is **no attempt to “recover”** `pg_control` via WAL.

5. What happens if `pg_control` is corrupted?

This is critical.

If PostgreSQL cannot read a valid `pg_control` :

- Startup **fails immediately**
- PostgreSQL **refuses to guess**
- The cluster is considered **unusable without manual intervention**

This is intentional.

PostgreSQL's philosophy is:

Silent corruption is worse than refusal to start.

So there is **no recovery mechanism** beyond:

- filesystem snapshots
 - backups
 - replication copies
-

6. Why this does NOT require “WAL of WAL”

Because WAL itself is **not self-describing from nothing**.

At startup, PostgreSQL needs *some* starting point:

- Which WAL segment to read
- Where the last checkpoint is
- Which timeline is current

That information **must live outside WAL**, otherwise you get exactly the infinite regress you described.

So PostgreSQL draws a hard line:

WAL protects *data consistency*

`pg_control` anchors *system identity and recovery starting point*

7. Why WAL files themselves don't need another WAL

Same reason.

WAL files are protected by:

- fixed record headers
- length fields
- CRC checksums
- sequential append-only structure

During recovery:

- PostgreSQL scans WAL forward
- Stops at the **last valid record**
- Ignores garbage beyond that

This works because WAL is **append-only** and **self-delimiting**.

But again: this assumes the **filesystem does not reorder or fabricate bytes**.

8. The implicit trust model (made explicit)

PostgreSQL trusts exactly these layers:

1. CPU and RAM
2. Filesystem obeys fsync semantics
3. `pg_control` checksum
4. WAL checksums
5. Page LSNs and invariants

Everything *above* that is verified.

Everything *below* that is assumed.

There is no further recursion.

9. Why this is unavoidable (not a PostgreSQL quirk)

This is not PostgreSQL-specific.

Any durable system must choose a **root of trust**:

- Filesystem metadata
- A superblock
- A boot sector
- A control file
- A hardware write barrier

If you don't stop somewhere, **durability is logically impossible**.

So the answer to your intuition is:

Yes — an “endless recursion of WALs” *would* be required **if** you tried to WAL-protect everything.

PostgreSQL avoids that by explicitly declaring `pg_control` as a trusted, checksummed root.

10. One precise sentence (no fluff)

`pg_control` is not WAL-protected because it *defines* the starting point for WAL recovery; PostgreSQL instead ensures its validity through whole-file rewrite, checksums, and `fsync`, and treats corruption as fatal rather than attempting recursive recovery.

If you want next:

- we can walk through the **exact fields** of `pg_control` and how each is used during startup, **byte by byte**
- or examine **how recovery behaves if the checkpoint LSN points into missing WAL**
- or compare this to how filesystems (ext4, XFS) handle their own superblocks

Just say which direction.



Ok, what does tigerbeetle do? Does it also have a similar file?

Short answer first (so we anchor intuition):

TigerBeetle does *not* have a `pg_control` -style “one small magic file”.

Instead, it makes the WAL itself self-describing and self-verifying, and it deliberately moves the root of trust *into* the log.

Now I'll explain **exactly how**, without abbreviations left unexplained, and why this avoids the problem you pointed out.

1. The problem TigerBeetle set out to eliminate

PostgreSQL has:

- mutable heap files
- a separate control file (`pg_control`)
- WAL as *supporting infrastructure*

TigerBeetle's authors explicitly wanted to avoid:

- “special” metadata files
- hidden recovery state
- filesystem-dependent assumptions about small files

So the design goal was:

**If you have the log, you have the truth.
If you don't, the system must refuse to start.**

This leads directly to their solution.

2. TigerBeetle's core storage model (must understand this first)

TigerBeetle is built around **one persistent structure**:

An append-only log of fixed-size frames

Everything else is derived from it.

There are:

- no heap files
- no separate metadata file

- no "last checkpoint pointer" outside the log
-

3. What a "frame" is in TigerBeetle (precisely)

A **frame** is a fixed-size record written sequentially to disk.

Each frame contains:

1. Header
2. Payload
3. Checksum

3.1 Header (identity + ordering)

The header includes:

- a **cluster identifier** (so logs cannot be mixed)
- a **replica identifier**
- a **monotonic sequence number**
- flags describing frame purpose (command batch, commit marker, snapshot, etc.)

This replaces:

- PostgreSQL's system identifier
- checkpoint location
- timeline metadata

3.2 Payload (the actual data)

Depending on frame type, payload may contain:

- client requests (logical operations)
- state machine commands
- snapshot data
- internal coordination metadata

Importantly:

- payload is **domain-level**, not page-level
- no physical storage layout is logged

3.3 Checksum (integrity)

Each frame has a cryptographic checksum over header + payload.

If *any* byte is corrupted:

- the frame is invalid
- recovery stops at the last valid frame

No guessing.

4. How TigerBeetle avoids `pg_control`

PostgreSQL needs `pg_control` to answer:

- "Where do I start replay?"
- "Was the last shutdown clean?"
- "Which WAL belongs to this cluster?"

TigerBeetle answers **all of those from the log itself**.

4.1 Replay start point

TigerBeetle always replays:

From frame zero forward

There is no "checkpoint pointer" stored outside the log.

Snapshots exist (explained below), but they are **inside the log**.

4.2 Clean vs unclean shutdown

TigerBeetle does **not care**.

On startup:

- it scans frames sequentially
- validates checksums
- stops at the first invalid or missing frame
- rebuilds in-memory state

There is no special "clean shutdown marker" that is trusted outside the log.

4.3 Cluster identity

Cluster identity is embedded **in every frame header**.

So:

- if you point TigerBeetle at the wrong disk
- or mix files from another cluster

It will detect the mismatch immediately and abort.

5. Snapshots in TigerBeetle (important distinction)

TigerBeetle *does* have snapshots, but:

Snapshots are log entries, not external state

A snapshot frame:

- captures a compact representation of state
- is written *into* the same append-only log
- is checksummed like everything else

On recovery:

1. Scan log
2. Find latest valid snapshot frame
3. Load snapshot
4. Replay frames *after* snapshot

So snapshots are:

- optional
- accelerators
- never authoritative

This is very different from PostgreSQL, where:

- heap files are authoritative
 - WAL is auxiliary
-

6. Why TigerBeetle does NOT need a "WAL of WAL"

This answers your recursion concern directly.

TigerBeetle's invariant is:

There is exactly one persistent truth: the log.

That means:

- no metadata outside the log
- no pointers into the log stored elsewhere
- no "control file" that must itself be protected

The recursion ends because:

- there is nothing *below* the log that must be recovered

If the log is corrupt:

- startup fails
- operator intervention is required

No silent repair.

7. What TigerBeetle trusts (its root of trust)

TigerBeetle explicitly trusts only:

1. The storage device writes bytes in order
2. A write that returns success will not later return different bytes
3. Checksums detect corruption

That's it.

There is no:

- filesystem metadata dependency
- directory fsync dependency
- small-file atomicity assumption

This is why TigerBeetle uses **direct I/O** and controls alignment carefully.

8. Comparison to `pg_control` in one table

Aspect	PostgreSQL	TigerBeetle
Control metadata	Separate file (<code>pg_control</code>)	None
Recovery start	Pointer stored outside WAL	Derived from log
Clean shutdown marker	Stored in control file	Not needed
Cluster identity	Control file + WAL	Every frame

Aspect	PostgreSQL	TigerBeetle
Snapshot location	External (heap files)	In-log
Corruption handling	Try to recover	Refuse to start

9. Why PostgreSQL *cannot* easily copy this model

PostgreSQL:

- mutates data pages in place
- has many file types
- relies on MVCC visibility rules
- needs fast random access to heap pages

TigerBeetle:

- uses a single deterministic state machine
- has no ad-hoc queries
- has a fixed domain
- rebuilds everything from log deterministically

So the difference is not “PostgreSQL is sloppy” vs “TigerBeetle is clean”.

It's:

Different problem → different root of truth

10. The key sentence (no fluff)

TigerBeetle does not have a `pg_control` equivalent because it treats the append-only log as the sole durable authority, embedding identity, ordering, and recovery state directly into each checksummed frame, thereby eliminating the need for an external control file and avoiding recursive durability problems.

If you want next, we can:

- dissect a TigerBeetle frame layout field by field
- compare this to Kafka's log segments
- or analyze what breaks if you try to hybridize PostgreSQL and TigerBeetle approaches

Just point.

