



How OSM Monitor stores its data

TL;DR : One big table of raw changesets, plus a set of small summary tables that answer most questions without reading the big one.

osm-monitor is a small Django app that ingests every OpenStreetMap changeset (about 189 million since 2005, plus ~50,000 a day) and serves a dashboard and a public API on top: volume over time, top editors and contributors, a world heatmap. It runs on a 2017 MacBook Pro (2 cores / 4 threads, 16 GB) under Docker via Colima, on TimescaleDB.

OSM Monitor used to keep every OpenStreetMap changeset in one PostgreSQL database. It uses two extensions: TimescaleDB for time-based data and PostGIS for geography.

One big raw table

Every changeset goes into one table. One row is one changeset: who made it, which editor and imagery they used, where on the map it was, and how many objects it changed.

Two programs fill this table:

- A **poller** reads OpenStreetMap's live feed every minute.
- An **importer** loads older history from the OpenStreetMap dump file.

Both use the same code, so loading the same changeset twice never creates a duplicate.

TimescaleDB splits the table into one piece per month, called a *chunk*. A question about last week only reads one or two chunks, not the whole table. After 30 days, a chunk is compressed: one month of data goes from 468 MB to 62 MB.

Many small summary tables

Counting millions of rows on every page load is too slow. So the database also keeps **summary tables** (TimescaleDB calls them *continuous aggregates*). Each one stores counts that are already added up, for example “changesets per editor per day.”

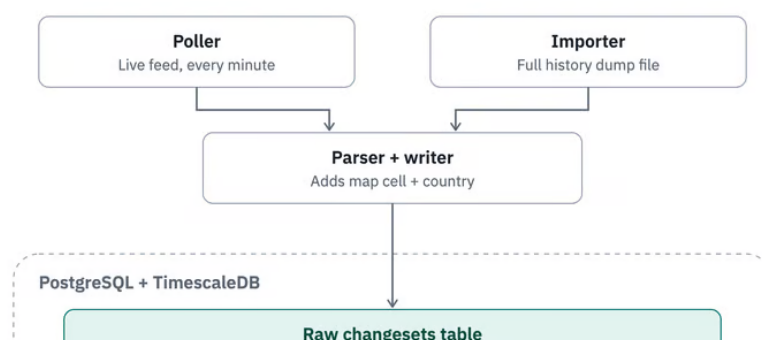
There are 23 of them:

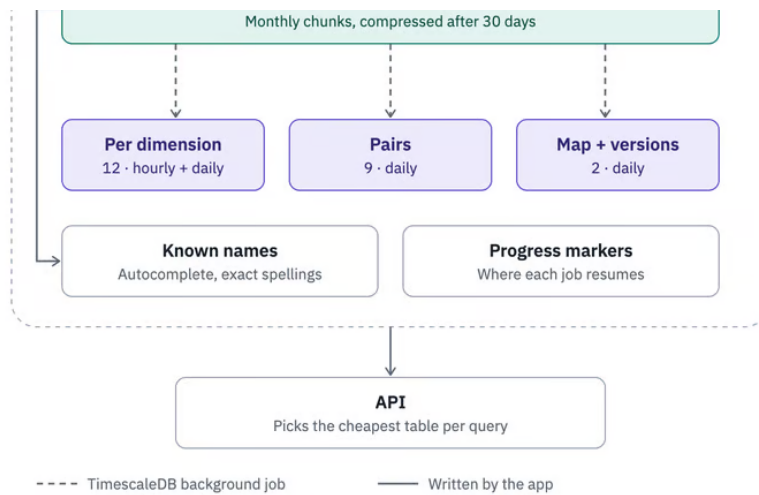
- total volume, per hour and per day
- one per dimension (contributor, editor, imagery, language, country), per hour and per day
- nine for pairs of dimensions, like “editor × imagery per day”
- one for the map, keyed by geohash (a short code for a map cell)
- one for editor versions

TimescaleDB updates these tables by itself every 30 to 60 minutes, but it only rechecks the last 7 days. After importing old data, we run one refresh command by hand.

A last small table holds every known name (contributors, editors, imagery). The search box uses it for autocomplete, so typing never scans the big table.

Poller

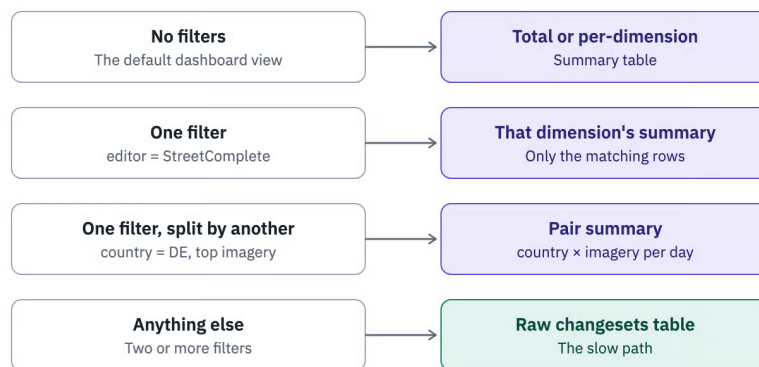




How a query picks a table

Before each API query runs, the app asks: *does a summary table already have this answer?*

The raw table is the slow path. It still skips most data by reading only the chunks in the time range, and by using small indexes stored with the compressed data.



Why this design

Most questions are “what happened in this time range, by one or two dimensions?” Summary tables answer those quickly, and they grow much more slowly than the raw table.

Today we store one year, about 23 million changesets. The goal is the full history since 2005, about 190 million. The summary tables should keep the common dashboard views fast at that size. Queries with several filters at once still read the raw table, so those are the ones to watch.

(Sept 2026)

Putting ClickHouse next to TimescaleDB without changing a single API response

I wanted to know whether ClickHouse would do better on this exact workload, on this exact machine.

The starting point

Before touching anything, I recorded a baseline: 46 fixed API calls (every endpoint, from one week to 21 years of history, with and without filters).

- **Storage:** the raw hypertable compresses from 91 GB to 19 GB. On top of it sit **23 continuous aggregates** (pre-computed rollups) taking **32 GB**.
- **Ingestion:** a full-history load took **~14 hours**, then **4 h 26 min** to refresh the aggregates and **~55 min** to compress.

too fast to be a good fit.

- **Queries:** anything an aggregate covers answers in **10–100 ms**. Everything else gets slow fast: top contributors for a year takes 1.5 s, two filters at once 1–2.5 s, the heatmap for a year 4.7 s. The full-history world map and top contributors **don't finish within the 30 s limit**.

TimescaleDB is fast where I pre-computed the answer, and those 23 aggregates exist precisely because the slow paths kept showing up, one at a time. That's the shape of workload column stores are built for.

Step 1: make the database swappable, provably

The API views mixed what the API means (parameters, defaults, response shapes) with how the data is fetched (which of the 23 aggregates answers which question). I split them:

- **changesets/api/**: the public endpoints, which parse, validate and ask a backend.
- **changesets/analytics/**: a small contract (summary, timeseries, toplist, geo_cells, ...) plus one implementation per database.

The interface sits at the level of *operations*, not SQL, so each database stays idiomatic: the TimescaleDB backend routes between aggregates, a ClickHouse backend can simply scan. The write side got the same treatment: ingestion parses each batch once and hands it to every configured writer, with one contract (an idempotent upsert that only replaces a changeset whose change count grew).

How I know nothing changed:

- **46/46** API responses byte-identical before and after, and the OpenAPI schema didn't move by a line.
- A differential write test imports real replication data inside a rolled-back transaction, set up so every path (create, update, skip) runs. It caught a real bug: the XML parser had *always* left changeset IDs as strings, Postgres had silently cast them for months, and my new writer compared them with integers, so nothing matched.

Step 2: take geography out of the database

Country and geohash were computed by a PostGIS trigger. I moved them into Python at ingest (Shapely + pyproj), at 35 μ s per changeset. Parity on 3.17 million rows: geohash identical everywhere, country different on 13 (0.0004%), all on a "nearest country within 5 km" fallback where PostGIS doesn't even agree with itself. The simplified borders have long edges: PostGIS treats them as curves along the Earth's surface, GeoJSON as straight lines in lat/lon, and over a long edge those drift apart by hundreds of meters.

Performance lessons from the TimescaleDB side

- **An index that couldn't be used:** the import started at 260/s. A nearest-country fallback cast geometries to geography, which made the spatial index useless, costing **135 ms** per miss. Subdivided polygons plus a bounding-box pre-filter: **0.25 ms**.
- **Monitoring that choked the database:** pg_stat_statements stored every 2,000-row insert with its parameters until its file hit **1 GB**, which Datadog DBM re-read nonstop. With that fixed: **~3,500/s**.
- **A planner that gives up:** from about **10** (id, created_at) pairs in a DELETE, TimescaleDB locks **all 524** chunk relations. That deadlocked with compression. One bounded DELETE per day: 2 relations.
- **Compression vs case-insensitive filters:** bloom filters only help equality. UPPER(user) = UPPER(x) took **990 ms** per month of data, against **23 ms** for an exact match.

First ClickHouse results

The schema is deliberately plain: a mirror of the Postgres table, `ReplicatingMergeTree(changeset_count) monthly partitions, no materialized views or`

replacing merge tree(changes_count), monthly partitions, no materialized views or projections. The rule: **measure before optimizing**.

- **Ingestion: ~9,800 changesets/s**, against ~3,500/s on the same pipeline (**~2.7x**). ClickHouse uses about 60% of one core; the bottleneck is now *my Python XML parser*.
- **No read-before-write**: 5 old changesets that took **~120 s** to go through the TimescaleDB writer's existence check went into ClickHouse instantly.
- **Disk**: [TBD]
- **Queries over the same 46 API calls**: [TBD]

[Archive](#) [RSS feed](#) [Mastodon](#) [QR Code](#)

Made with [Montaigne](#) and by anton 