

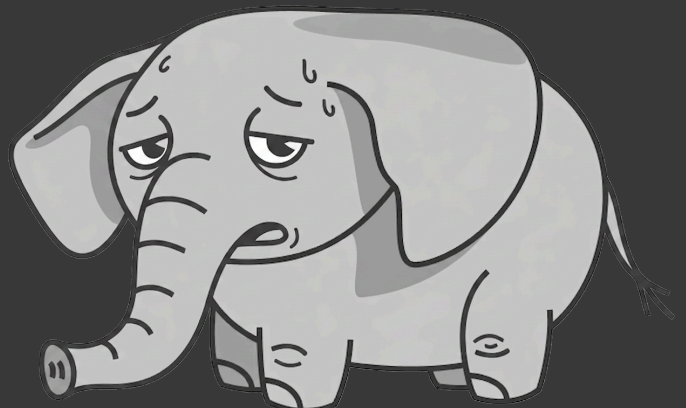


MOTHERDUCK · PRACTITIONER SERIES · A DATA ENGINEERING FIELD GUIDE

POSTGRES IS FULL.

When the database
that ships your
product can't answer
questions about it.

Authored by **Mehdi Ouazza**, DevRel at MotherDuck



I have watched this exact movie too many times. Postgres ships the product. Nobody plans for it to do analytics, it just sort of happens: a dashboard here, a customer-facing usage tab there, the weekly number someone in the business actually needs. For a year it is fine. Then one day the page takes ten seconds, the API gateway gives up at sixty, and a designer files a ticket about the spinner. Nothing about the workload is exotic. What changed is the shape of the problem, and the database never got the memo.

This happens at three-person seed startups and product squads inside thousand-person companies alike. Org size turns out to be a red herring. What actually happened is that analytical work piled onto the database that runs your product, and now the thing built to charge cards and look up users is also being asked to scan ten million rows and group them by week. Those are different jobs, and Postgres is quietly losing the second one.

I wrote this for the engineer who inherited that analytics work without asking for it: no formal data background, no appetite for a six-month warehouse project. My claim is simple, and I will defend it for the rest of the guide. You do not need to migrate off Postgres, and you do not need to build a data platform. You need to stop asking one database to do two opposite jobs. Keep Postgres as the system of record. Send the analytical queries somewhere built for them.

Two things up front, because they are the parts people do not believe. First, the payoff is enormous, orders of magnitude. A columnar engine is routinely 1000x faster on analytical queries ([ClickBench](#) puts numbers to it): the kind of gap where the report that took an hour comes back before you have finished reading the question. Second, the move is much smaller than "adopt a warehouse" sounds. Your app keeps speaking Postgres. You point the analytical queries at MotherDuck's Postgres wire-protocol endpoint, same `pg` driver, same SQL, one different host. No rewrite, no new client, no migration weekend.

A quick scope check. Postgres stays as your system of record, so nobody is migrating off anything. There is no six-month warehouse build here either; if someone quotes you that, close the tab. And yes, I work at MotherDuck, so apply the obvious grain of salt, but the product stays out of it until Chapter 6. Everything before that is strategy you could run with any columnar engine.

A note on Postgres vs. other OLTP databases. We center Postgres because it's the most common case, but the problem and the fix are not Postgres-specific - they apply to any row-based OLTP database (MySQL, SQL Server, and friends). The honeymoon, the patches, and the wall play out the same way, and the offload pattern in Chapter 6 transfers directly: swap DuckDB's Postgres scanner for the equivalent on your engine - e.g. the [MySQL scanner](#) - and the rest of the walkthrough is unchanged. Where something is genuinely Postgres-only, we say so.

A note on the code. The snippets throughout are excerpts, picked to show the *shape* of each step at a high level, not to be pasted into production. The full, runnable source - the Next.js webapp and the `uv` pipeline scripts behind Chapter 6 - lives in [motherduckdb/labs](#). Clone it, point it at your own Postgres, and the walkthrough runs end to end.



A note on the quotes throughout this guide. They are real, drawn from conversations with engineering teams running on Postgres. We have anonymized every one - the point is the pattern, not the company. If a quote sounds like it could be coming from your team, it probably is.

1. THE GROWTH CURVE



"We've built our star schema on a Postgres database as a starting point, always knowing we're going to be carrying this somewhere else when the need arises."

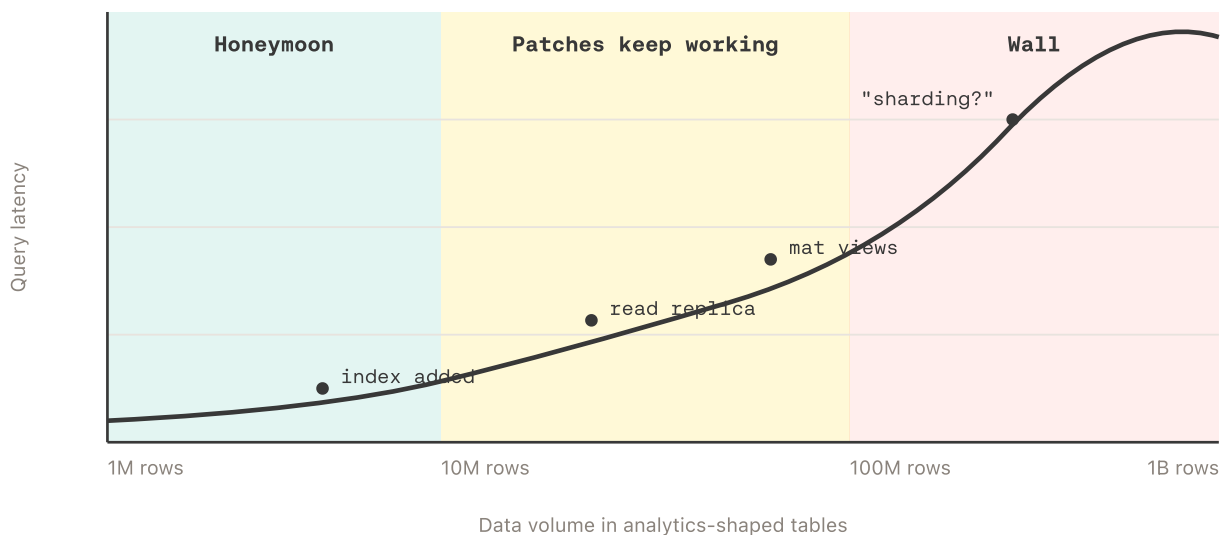
- an engineer at a B2B SaaS startup

Nobody decides to do analytics on Postgres. You back into it, one reasonable choice at a time. The data is already there, so you add an analytics table next to the production tables. The table is right there, so you put a dashboard on it. The first dashboard worked, so you add five more, because cloning a query is cheaper than building the right thing. Every one of those decisions is correct in isolation. That is what makes the trap so good.

Then the patches start. An index here, a read replica there. And here is the part that fools everyone: the patches work. They genuinely buy you months. Indexes paper over more analytical load than they have any right to. A replica papers over more after that. So you reasonably conclude the next patch will work too. It will not. The wall does not arrive gradually, it arrives all at once, because the patches were never fixing the problem, they were postponing it.

The curve looks roughly like this:

THE GROWTH CURVE EVERY POSTGRES-FIRST COMPANY WALKS



The numbers on the x-axis are not the point, and they move around: the more aggregations and group-bys in your mix, the earlier you hit the wall. (The next chapter explains why.) The shape is what is fixed. You spend a long time thinking you are fine, then a very short time discovering you are not.

2. POSTGRES IS GREAT AT RUNNING YOUR APP, BAD AT ASKING IT QUESTIONS

Here is the whole disagreement in one sentence: Postgres is built to change data, and analytics is built to read it. Postgres is an OLTP database, Online Transaction Processing. It charges cards, inserts orders, looks up a user by id: small, fast operations that touch a row or two. That is genuinely what it is great at. People get the flip side backwards, though: Postgres is not even a fast bulk loader. Tiny changes are its gift; big batches belong to an analytical engine. MySQL and SQL Server sit in the same family.

OLAP, Online Analytical Processing, is the opposite shape. Aggregate, group, trend, compare this quarter to last. A few big queries that read one or two columns across millions of rows. DuckDB, MotherDuck, ClickHouse, BigQuery, Snowflake, and Redshift all live here.

The split comes down to physics, decided decades ago by how each engine lays bytes on disk. Postgres stores data row by row, which is exactly right for transactions: your app writes an order as one row and reads a user as one row, and row storage makes both cheap. But analytics never wants one row. It wants one column across millions of rows. Ask a row store for that and it has no choice but to read every row to get at the one field you asked for.

ROW STORAGE VS COLUMN STORAGE - THE SHAPE OF THE I/O

Row storage (Postgres)

AVG(amount) reads every row, all columns

id	user	day	amount

one row read = one disk fetch

Column storage (MotherDuck)

AVG(amount) reads one column, 1/4 the I/O

id	user	day	amount

one column read = one disk fetch

Postgres fetches one row by `id` in a single read, which is what it was built for. A columnar engine fetches `AVG(amount)` in a single read, for the exact opposite reason. Hand one engine both jobs and one of them has to lose. On Postgres, the loser is always analytics. There is no bug to file here. The design is working exactly as intended, for a workload that happens not to be yours.



This is the whole game, and the gap is enormous: routinely 1000x. On [ClickBench](#), an industry analytical benchmark, columnar engines like MotherDuck come in around three orders of magnitude ahead of Postgres. Row versus column decides it, and everything else in this guide is just dealing with the consequences.

Why indexes do not save you

The first reflex when a query is slow is to add an index. For OLTP queries that reflex is correct, it is basically Postgres's love language. For OLAP queries it does nothing, and worse, the indexes you pile on for analytics quietly tax every write your app makes. So the reflex does not just fail to help, it makes the thing Postgres was good at slower too.

Consider two queries on the same `orders` table.

OLTP shape - an index will make this fast:

```
SELECT * FROM orders
WHERE customer_id = 42
ORDER BY created_at DESC
LIMIT 10;
```

An index on `(customer_id, created_at)` is exactly what Postgres wants. The planner jumps to the relevant rows and returns them.

OLAP shape - no index will save this:

```
SELECT date_trunc('week', created_at) AS week, SUM(amount)
FROM orders
WHERE created_at >= NOW() - INTERVAL '90 days'
GROUP BY week
ORDER BY week;
```

The query has to scan ninety days of every order to compute weekly totals. The engine still does a sequential scan, still reads every column on every row to compute one of them. An index does not help.

Stacking indexes for the OLAP queries does not make them faster - and now every `INSERT` on `orders` has to update a dozen extra indexes that exist only to serve the weekly report. Write throughput drops. The OLTP path Postgres was built to handle gets slower because of the OLAP path it was not.

3. 7 SIGNS YOU'VE OUTGROWN POSTGRES FOR ANALYTICS

You will not get one clean symptom, you will get a handful at once, which is part of why this is hard to diagnose from the inside. I have ordered them by how often they actually come up when I talk to teams running on Postgres, not by how bad they sound on paper. The ones up top are the loud ones people lead with. The ones at the bottom are the quiet cousins, same root cause, different bruise. If you recognize three or more, you are not "approaching" the wall. You are on it.

1. Aggregates take minutes, not seconds



"It can take minutes, maybe up to 10 minutes sometimes... ideally we'd be able to run that on Postgres and everything would be super smooth, but sometimes we try to aggregate a lot of data in Postgres, and the latency spike, so..."

- an engineer at a SaaS company

Aggregations on Postgres are slow by design. As Chapter 2 showed, `SUM`, `COUNT`, `GROUP BY`, and window functions ask a row store to do a column store's job - every row read for every column scanned.

A bigger instance buys a few months. A partitioned table buys a few more. Adding indexes buys nothing - analytical scans do not filter to small sets. The shape of the I/O is the bottleneck, and the only real fix is a different engine on the read path. Postgres goes back to being the OLTP system it was always good at.

And this is rarely just an internal annoyance. The slow aggregate is usually the dashboard a customer is staring at, or the weekly number the business is waiting on. Slow analytics does not stay an engineering problem for long - it becomes a product and revenue one.

2. The dashboard tab is the top product complaint



"You've made all these retool dashboards, all these analytics things, all these crazy mat-view cycles that are doing a ton of work on Postgres and just constantly throttling it."

- an engineer at a 30-person AI company

When the same database serves both the transactional path and the customer-facing analytics tab, both get slower. The dashboard is not broken, exactly - it is consistently slow, and customers are telling you so. Support tickets reference it. Renewals reference it. The product manager has put it on the roadmap three quarters in a row.

The statement-timeout sign further down is the acute version, where the dashboard 500s and the gateway gives up. This is the chronic one: customers churn for reasons the retention dashboard cannot fully explain, and that retention dashboard, of course, also runs on Postgres. This is where graduating off Postgres stops being an engineering convenience and becomes a revenue issue.

3. Your "analytics replica" is still slow

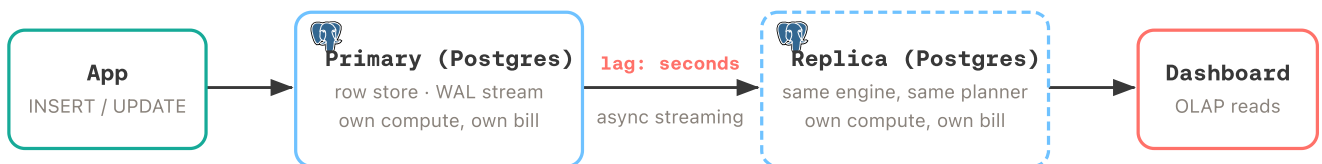


"We have a separate instance for analytics where we stream data from production instance via logical replication to the analytical instance. But yeah, majority of data is still in Postgres."

- an analytics lead at a media company

A read replica decouples reads from writes. It does not decouple OLAP from OLTP. Same row-based storage, same query planner, same fundamentals. The analytical queries take the same amount of time on the replica as on the primary - you have moved the slow queries to a different machine, not made them faster.

PRIMARY → REPLICA - WHAT IT GIVES YOU AND WHAT IT DOES NOT



A different machine running the same engine. Writes stop fighting reads, but the slow queries are still slow. Plus lag, patching breakage, twice the bill.

The cousin pain shows up at the worst possible time: **the replica breaks when someone patches the primary.** Logical replication has a lot of moving parts, and an OS-level update can quietly invalidate a standby. One engineer described it this way: *"They mindlessly patched it, just rebooted the machine. The patch broke replication, and I have the primary database, a streaming standby, a logical replica, and another one that goes out to Debezium and Kafka. They blew it out of the water, and we had a one-day outage."*

Chains of replicas build a tower of correctness assumptions on top of something that needs careful operational care. Towers like that fall down.

Aside from indexes and raw query speed, there are other signs you can watch to see if you are running out of runway on Postgres. The next five are operational patterns - they show up as failed jobs, customer complaints, and small fires that compound.

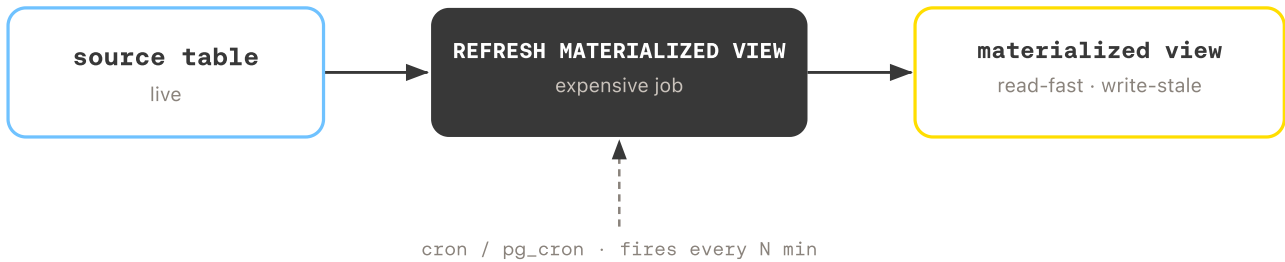
4. Materialized views are either stale or expensive



"Having to refresh materialized views that live in Postgres that we need to manually refresh every time the thing updates."

- an engineer at a health tech startup

Materialized views make reads fast until the answer is wrong. Then refresh becomes the new bottleneck. The mental model:



Postgres offers two flavors of refresh, both with sharp edges:

- `REFRESH MATERIALIZED VIEW` blocks reads against the view while it runs. On a large table, this is unusable for production dashboards.
- `REFRESH MATERIALIZED VIEW CONCURRENTLY` does not block, but it is slower and requires a unique index. On a really large table, the refresh takes long enough that the answer is already stale by the time it finishes.

The pattern that emerges: a cron job that refreshes a thing whose freshness no one is quite sure of; a `last_refreshed_at` column to expose to the dashboard so users know how recent the numbers are; a check for "is the refresh job already running" because two overlapping refreshes are their own problem. The team is now operating a small ETL pipeline inside the transactional database, on a primitive that was not designed for one.

5. Statement timeouts on customer-facing pages

Customer-facing analytics is the dashboard you ship inside your product - the usage tab, the billing report, the per-account performance view that your customers see and interact with. The query is triggered by the end user; the latency is part of the product experience.



"Query performance is very challenging right now. We get statement timeouts because we have an API server that proxies the user, then does a query builder, and queries the Postgres instance, and because the API gateway is configured to 60 seconds..."

- an engineer at a logistics startup

When the dashboard tab times out, the bug stops being internal. A customer opens the analytics tab in the product. The API gateway proxies the query to Postgres. Postgres takes ninety seconds. The gateway times out at sixty and returns a generic 500. The customer sees an error and opens a support ticket. Support tells them to try again later. Sometimes it works. Often it doesn't.

The patches are familiar: add a `LIMIT`, pre-aggregate, run the query async and email the results, tell engineering this is a known issue. None of these are fixes. They are hostage negotiations with a database that no longer wants to play the analytical role you gave it.

6. Analytical indexes are taxing your write path

An index added for a weekly report is a write cost every transaction has to pay. Chapter 2 covered the mechanism - analytical scans cannot be served by indexes, but stacking indexes for them taxes the OLTP path. The symptom in the wild is the discovery moment: opening `pg_stat_user_indexes` and finding indexes no one remembers creating, scanned twelve times in the past month, costing four percent on every write.

The pattern that leads there is familiar: a partial index for "active users only", a multi-column index because someone filtered by two things at once, another index because a different report was slow. Indexes proliferate. The cost is invisible until something else slows down and the trail leads back to them.

7. ETL is `pg_dump piped to psql` and the team is afraid of it

Pipeline by archaeology. There is one script. One person wrote it. It runs at 3 AM. It mostly works. When it doesn't, the morning is bad.

It usually looks something like this:

```
#!/usr/bin/env bash
# sync.sh - cron: 0 3 * * * DO NOT TOUCH (we mean it)
set -e # added the morning after it half-failed and we shipped a day of garbage

pg_dump --no-owner -t orders -t order_items "$PROD_DATABASE_URL" \
  | psql "$WAREHOUSE_URL"

# TODO: make this incremental - full dump is 40 min now
# TODO: the 2-3 AM rows sometimes load twice? ask Dave (left in 2023)
```

The team has half-meant to "do this properly" for two years. The script is now load-bearing. Nobody wants to be the one to touch it. The original author left twelve months ago.

Blame the situation, not Postgres: this is the operational debt that piles up when analytics piggybacks on a system never built to host it. Move the analytics workload off Postgres and the `pg_dump` script can finally retire.

CHAPTER 4. THE PATCHES THAT DON'T SCALE (AND WHY)

You have probably tried some of these, and I want to be clear: they are not dumb. Every one is a reasonable engineer making a reasonable call, and every one buys real time. That is exactly why they are dangerous. A patch that did nothing would get abandoned in a week. A patch that buys you three months gets mistaken for a solution, and then you spend the next quarter shopping for the next patch instead of fixing the actual problem. I am not here to talk you out of them. I am here so that when you hit each ceiling, you recognize it as a ceiling and not as "we just need one more."

PATCH	WHAT IT GIVES YOU	WHY IT EVENTUALLY FAILS
Read replica	Decouples read load from writes; the primary stays fast for the app	Still row-based, still Postgres' query planner; the slow queries don't get faster, just moved
Materialized views	Pre-computed answers; reads cheap	Refresh cost or staleness; brittleness on schema changes; you're operating mini-ETL inside the OLTP DB
Bigger instance	A few months of headroom	Linear scaling on a non-linear problem; you double the bill and buy six months
Citus / sharding	Horizontal scale for OLTP	Roughly a 10x gain on analytics, not the ~1000x of a columnar engine; real ops complexity, and non-shard-aligned queries still fan out cross-node
Parquet in S3, no engine	Cheap, durable storage	A data swamp: no query layer, and no ACID, so you inherit duplicated and half-written datasets to untangle
A second Postgres "for analytics"	Isolation from the production database	Two production-grade Postgres instances to operate, patch, secure, and back up; the analytical queries are still slow on the second one too

Two of these - **read replicas** and **materialized views** - got their own treatment in Chapter 3 (symptoms and). Both are real patches that buy real time; both have the ceilings described there. Below are the others.

Citus / sharding is the move teams reach for when they have concluded Postgres just needs to scale horizontally. For OLTP-shaped workloads (user data partitioned by tenant, for example), this can work. For OLAP-shaped workloads, the sharding key rarely aligns with the analytical query's shape, and queries fan out across every shard. And the payoff is modest: on [ClickBench](#), Citus is roughly a 10x improvement over plain Postgres on analytics, where a columnar engine is closer to 1000x. You take on real operational complexity for a fraction of the win. One engineer who tried it: *"We've tried views on Postgres, especially with Citus, and it doesn't scale well."* Another, at a SaaS startup: *"Like Postgres, you kind of hit a ceiling at a certain point where you have to start sharding."* The recurring theme: teams expecting Citus to fix analytics, discovering it fixes a different problem.

"Just put it in S3 as Parquet and figure it out later" is the engineer's instinct toward an accidental data lake - a data swamp. It is not wrong as a building block; Parquet is a fine storage format. But Parquet without a query engine is a folder of files, and worse, a folder with no ACID transactions. The moment two jobs write at

once, or one dies halfway, you get duplicated and half-written datasets that nobody can fully trust. Untangling that eats far more time than the storage ever saved. The engine question does not go away by skipping it.

A second Postgres "for analytics" is the move teams make when they want isolation from production but stay inside familiar territory. It is the most expensive version of the patch ladder: two production-grade Postgres instances, twice the operational surface, and the analytical queries are still slow on the second one too. The instinct is sound (isolate the OLAP load), but the destination is wrong (it should be a different *engine*, not a different *Postgres*).

The pinch quote that lands the chapter is from a 40-person company we spoke to:



"Snowflake's too expensive, Postgres is just like not quite enough for where we need to go."

- a founder at a 40-person company

Between a rock and a hard place. The next chapter is the way out.

5. WHAT GRADUATING ACTUALLY LOOKS LIKE

The whole strategy fits on a sticky note: **Postgres stays. Analytics moves.**

That is the part people overthink. Postgres remains the system of record. Your app keeps writing to it and reading its hot rows from it, and nothing on the application side changes. The dashboards, the weekly reports, the customer-facing tab, the ad-hoc questions, those move to a columnar engine built for them. OLTP and OLAP stop fighting over the same machine for the simple reason that they are no longer on the same machine. You are not migrating. You are giving the second workload its own room.

Two questions decide the shape of the move: **how the data gets there**, and **where the analytics actually runs**. Treat them as orthogonal.

Axis 1 - How does the data get to the analytics store?

Four extraction patterns, ordered by how common they are in real-world deployments. The freshness column reflects what teams actually achieve, not vendor marketing.

PATTERN	FRESHNESS	WHEN IT MAKES SENSE	HOW IT LANDS ON MOTHERDUCK
Scheduled batch (ELT)	minutes to hours	The default for most teams reading this guide. Hourly or six-hourly sync, low ops cost.	A DuckDB process runs the Postgres scanner against your database and writes the result into MotherDuck. Host the job on MotherDuck Flights (Python runtime inside MotherDuck), on Vercel Cron, on GitHub Actions, or in an existing orchestrator.
CDC streaming (logical replication / WAL)	seconds	Sub-minute freshness genuinely needed (real-time customer dashboards, fraud signals).	Managed services like Estuary or PeerDB stream Postgres changes into MotherDuck. Self-hosted: Debezium + Kafka. More moving parts; reach for it only when you truly need seconds.
Query federation (no copy)	live	Ad-hoc exploration, small tables, no pipeline wanted.	A DuckDB client (or a Flight) attaches to Postgres at query time via the postgres scanner. No movement, no copy. Pushes the analytical scan back onto Postgres - fine for small data, painful at scale.
Object-store offload (Iceberg / Parquet on S3)	minutes	Open format wanted. Storage decoupled from compute. Multi-engine setups (DuckDB and Spark and Trino).	A Flight (or a pg_duckdb / pg_lake job) writes Postgres data to Parquet or Iceberg on S3. MotherDuck reads directly from S3. The fastest-growing architecture in 2026; Snowflake open-sourced pg_lake for exactly this.

For most teams reading this guide, **scheduled batch is the right answer**. It is what we build the Chapter 6 walkthrough around. The other patterns are good signposts when the assumptions change - sub-minute freshness, regulatory data residency, multi-engine setups.

A note on [MotherDuck Flights](#), since it shows up in three of the four patterns: a Flight is a Python runtime that lives inside MotherDuck. It connects to your data via the DuckDB Python client, runs the SQL or Python you wrote, and exits. Good for scheduled syncs, periodic compaction, reverse ETL, and operational one-shots.

Not a general-purpose compute platform - scheduled jobs only. Because a Flight is just plain Python, it is agent-native: a coding agent can write and ship one directly, and there is a ready-made [Postgres ingest recipe](#) to start from.

Axis 2 - Where does the analytical engine actually live?

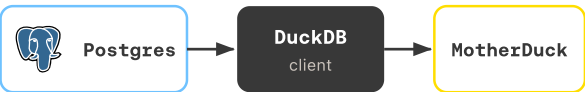
Four MotherDuck product surfaces ship today, and they split cleanly into two jobs. Two are about **analytics and the pipeline** - getting the data in and running the engine: a DuckDB client driving the Postgres scanner, or `pg_duckdb` embedded inside Postgres for when the data cannot leave at all. Two are about **analytics and the dashboard** - serving it: the Postgres wire-protocol endpoint your app already speaks, and Embedded Dives for customer-facing views.

THE FOUR SURFACES - PIPELINE VS DASHBOARD

ANALYTICS & PIPELINE - GET THE DATA IN

1. DuckDB client + scanner

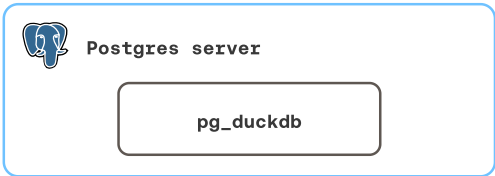
A small process pulls from PG, writes to MD.



Best for: hourly / daily scheduled sync. Most teams.

2. pg_duckdb extension

DuckDB embedded inside your Postgres. No data leaves.



Best for: compliance, cannot move data out.

ANALYTICS & DASHBOARD - SERVE IT

3. Postgres wire endpoint

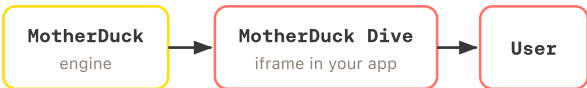
Your app speaks Postgres, points at MD.



Best for: existing BI tools and app dashboards.

4. Embedded Dives

Customer-facing views. Wasm under the hood.



Best for: customer-facing analytics. SDK handles Wasm.

What MotherDuck does today, plainly

MotherDuck does not have a one-click "connect to my Postgres and replicate" button. Connection is always initiated from a process: one you run, or one MotherDuck runs for you.

The process you run is a DuckDB client - on your laptop, in your Vercel project, in your existing pipeline. It opens a connection to your Postgres via the DuckDB postgres scanner and writes the result into MotherDuck.

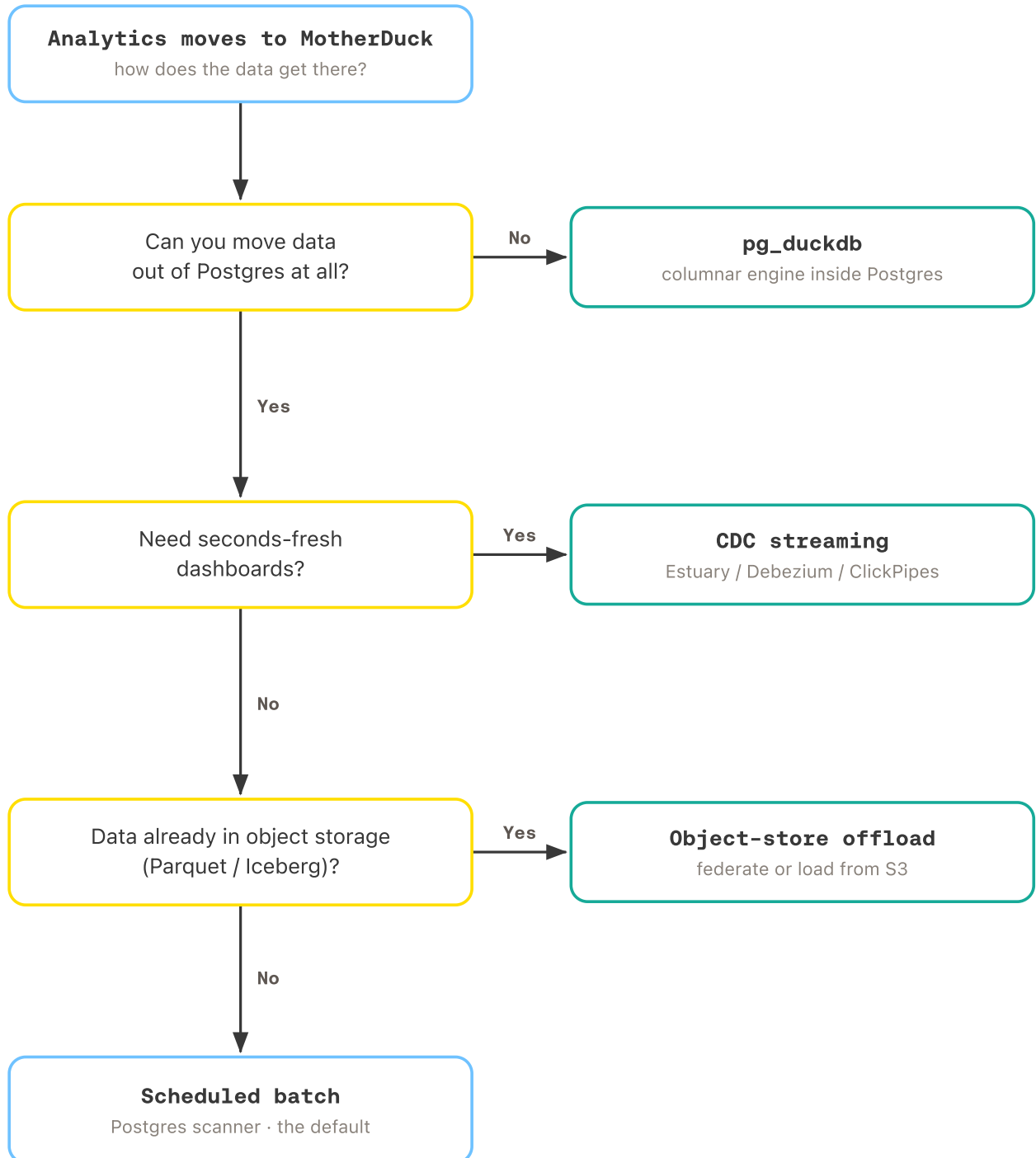
The process MotherDuck runs for you is a **Flight**: a Python runtime that lives inside MotherDuck, schedules itself, connects to your Postgres, runs your sync, and exits. Same architecture as the laptop version, hosted instead of self-managed.

The four surfaces above cover every Postgres-to-analytics workload pattern we have seen.

A decision tree

The analytical engine is MotherDuck either way. The real decision is **how the data gets there**. Four movement patterns cover almost everyone, and most teams land on the first one.

MOVING DATA TO MOTHERDUCK - PICK YOUR PATH



Most readers land on scheduled batch sync. That is the path Chapter 6 walks through end to end.

Embedded Dives and the power of Wasm

[WebAssembly \(Wasm\)](#) is a portable binary format that runs at near-native speed inside the browser; DuckDB compiles to it, so the [full query engine runs client-side](#) - no server round-trip for the query.

For customer-facing analytics - the dashboard tab inside your product, the per-account usage view, the partner-portal report - the cleanest architecture is to embed a **MotherDuck Dive** directly in your app. A Dive is a shareable, interactive analytics view; embedded in an iframe, it lives inside your customer's session in your application.

The power: under the hood, the embed serves the analytical query via DuckDB Wasm - meaning the user's browser runs the filters, aggregations, and drill-downs locally, against data MotherDuck has prepared. The Wasm bundle is a few megabytes. Filters and drill-downs are instant, with no round-trip to your server. MotherDuck's SDK handles the Wasm pipeline and the data delivery; you handle the embed and the per-customer scoping.

This is the practical answer to "I want my customers to see fast analytics dashboards in my product without standing up an analytics service of my own." It is also what we build the Wasm side of Chapter 6 around.

6. WALKTHROUGH - A MULTI-SHOP E-COMMERCE ANALYTICS SETUP ON VERCEL

Enough theory. Let me show you the whole thing working, with a dataset you can attach and numbers you can reproduce. The scenario is a multi-shop storefront platform, a Shopify-style product where lots of shops sell through one Next.js app on Vercel, backed by a managed Postgres. Sales land in a wide `order_items` fact table, and two kinds of analytics now lean on it: the internal dashboards (revenue across all shops, by plan tier, by month) and the customer-facing ones (each shop owner's own revenue-by-category view). Both are slow on Postgres, for every reason the last two chapters just walked through. The schema is deliberately boring, the shape almost every product ends up with, so the numbers here transfer to yours.

We keep Postgres as the system of record and move analytics to MotherDuck. The walkthrough assumes a stack most readers already have (Next.js on Vercel with any managed Postgres, e.g. Supabase, Neon, RDS, or PlanetScale for Postgres), and it covers two things: a scheduled batch sync via a MotherDuck **Flight**, and an Embedded Dive powering the customer-facing dashboard.

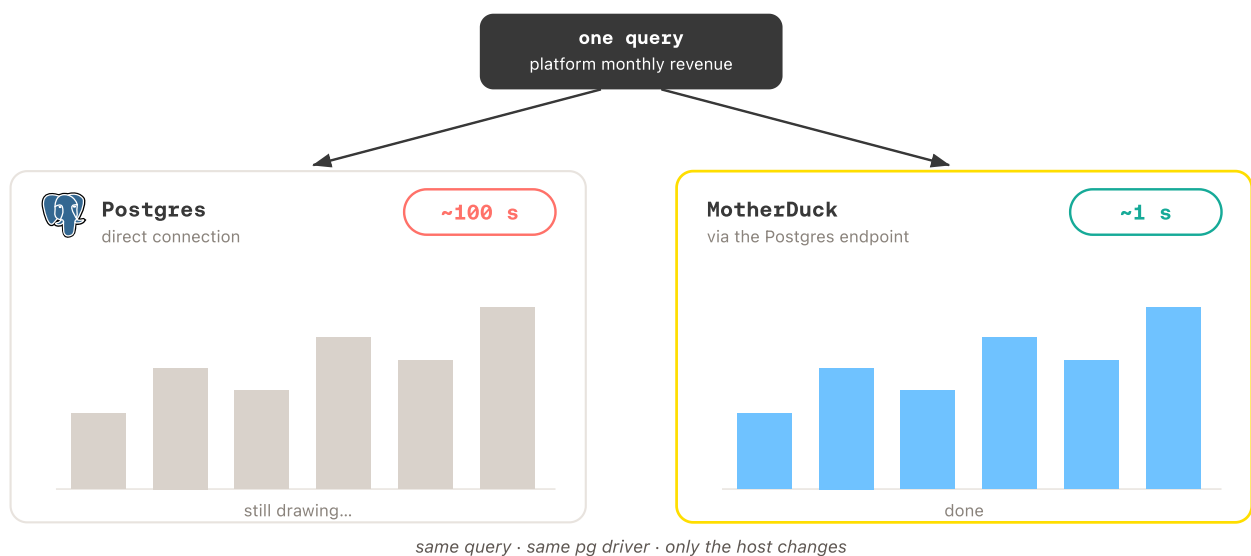
Everything below is a real, runnable reference project. The dataset is a ready-made multi-shop e-commerce dataset, available as a public MotherDuck share, so you can reproduce the before/after numbers yourself. Attach it and follow along:

```
-- Run this snippet to attach the database
ATTACH 'md:_share/multishop_commerce/ac3d36cc-f295-4c66-bf13-371b998f12e8';
```

Copy the snippets to see the *shape*; the full repo wires them together.

What you'll build

THE COMPARISON - SAME QUERY, TWO ENGINES, SIDE BY SIDE



Six steps. A handful of files. The dataset is already made.

What you'll end up with:

1. A **side-by-side comparison page**: the same query, drawn as a chart from two engines at once - one panel hitting Postgres, the other hitting MotherDuck through the Postgres endpoint. Hit run and watch Postgres grind for ~100 seconds while MotherDuck has already drawn its bars in ~1.

2. A small **Python pipeline** that pulls `orders` / `order_items` from Postgres into MotherDuck with the Postgres scanner - the same logic you explore interactively, packaged, and schedulable as a Flight.
3. The whole thing as a runnable repo (webapp + pipeline) you can clone and point at your own database.
4. A customer-facing per-shop drill-down powered by an Embedded Dive.

The webapp and pipelines live in [motherduckdb/labs](#) : a `webapp/` (Next.js) and a `pipeline/` of single-file `uv` scripts.

Step 1 - The storefront, and the query that hurts

The app's admin dashboard answers one obvious question - *revenue per month, across all shops, by plan tier* - with one obvious query:

```
SELECT s.plan_tier,
       date_trunc('month', o.ordered_at) AS month,
       SUM(oi.line_total)                AS revenue,
       COUNT(DISTINCT o.order_id)        AS orders
FROM order_items oi
JOIN orders o ON o.order_id = oi.order_id
JOIN shops s ON s.shop_id = o.shop_id
WHERE o.status = 'paid'
GROUP BY 1, 2 ORDER BY 1, 2;
```

It is the exact OLAP shape from Chapter 2: a full scan of a tens-of-millions-row `order_items` fact, joined and aggregated. On a mid-tier Postgres (~2 vCPU / 4 GB RAM) with the fact table larger than RAM, this lands in the *tens of seconds to minutes*. That is the "before."



Optional - set up the demo data. If you're following along without your own slow Postgres, load the ready-made dataset into a throwaway database:

```
uv run pipeline/seed_postgres.py # DuckDB reads the MotherDuck share, writes it
into Postgres
```

If your data is already in Postgres - and it is - skip this. You don't move data *into* Postgres for this; you offload it *out*. The seed script exists only so the walkthrough has a "before" to measure.

Step 2 - Explore with the Postgres scanner

Before building any pipeline, just *look* at the data with DuckDB's Postgres scanner. The quickest way in is the DuckDB CLI - install it with `curl -s https://install.motherduck.com | sh` (see the [DuckDB CLI docs](#) for other platforms). Any DuckDB client works the same way - Node, Java, Python, whatever your stack already speaks; the CLI is just the fastest way to start poking. Attach Postgres and run the same aggregate through DuckDB:

```

-- attach to your Postgres (READ_ONLY is a safety belt - never write to the source)
ATTACH 'postgres://USER:PASSWORD@HOST:5432/DBNAME' AS pg (TYPE postgres, READ_ONLY);

-- the same platform revenue query, now run by DuckDB's columnar engine
SELECT s.plan_tier, date_trunc('month', o.ordered_at) AS month, SUM(oi.line_total) AS revenue
FROM pg.public.order_items oi
JOIN pg.public.orders o ON o.order_id = oi.order_id
JOIN pg.public.shops s ON s.shop_id = o.shop_id
WHERE o.status = 'paid'
GROUP BY 1, 2 ORDER BY 1, 2;

```

The `postgres` extension is built into DuckDB; you don't install anything. This is the fastest way to explore a Postgres database analytically - no pipeline, no copy, no setup.

But it is **exploration, not a production read path**. The scanner pulls the rows it needs *out of Postgres at query time*, so the scan load still lands on your OLTP database - the very thing you're trying to relieve. It's perfect for "let me poke at this"; it's the wrong place to point a dashboard. The fix is to copy the data once and read from the copy - which is exactly what the next step packages up.

Step 3 - Package the exploration into a Flight

The exploration in Step 2 is the pipeline - it just needs a home that isn't your laptop and isn't your OLTP database. That home is a **Flight**: a single Python file that MotherDuck runs on its own compute on a schedule. A Flight is a full Python + DuckDB runtime, so the Postgres scanner works inside it exactly as it did locally; the Postgres credentials come from a MotherDuck **secret**, injected as an env var at runtime.

```

# flights/sync_postgres.py (abridged from MotherDuck's flight-postgres-ingest plan)
import os, duckdb

# The 'pg' secret injects pg_HOST, pg_PORT, ... Map them to libpq env vars so the
# password never lands in SQL, then attach read-only with an empty conn string.
for key, libpq in [("HOST", "PGHOST"), ("PORT", "PGPORT"), ("DATABASE", "PGDATABASE"),
                  ("USER", "PGUSER"), ("PASSWORD", "PGPASSWORD"), ("SSLMODE", "PGSSLMODE")]:
    if os.environ.get(f"pg_{key}"):
        os.environ[libpq] = os.environ[f"pg_{key}"]

def main():
    con = duckdb.connect("md:") # token injected by MotherDuck
    con.execute("CREATE DATABASE IF NOT EXISTS multishop_commerce; USE multishop_commerce")
    con.execute("INSTALL postgres"); con.execute("LOAD postgres")
    con.execute("ATTACH ' ' AS pg (TYPE postgres, READ_ONLY)")

    # Discover the source tables, then full-refresh each with a single CTAS:
    # atomic (swaps in one step), idempotent (re-running replaces), streaming
    # (the scan pipes into the write, so memory stays bounded even on big tables).
    tables = con.execute(
        "SELECT table_schema, table_name FROM postgres_query('pg', "
        "'SELECT table_schema, table_name FROM information_schema.tables "
        "WHERE table_type = 'BASE TABLE' AND table_schema = 'public'')"
    ).fetchall()
    for schema, table in tables:
        con.execute(f"CREATE OR REPLACE TABLE \"{table}\" AS SELECT * FROM pg.\"{schema}\".\"{table}\"")

if __name__ == "__main__":
    main()

```

The credentials live in a MotherDuck secret, created once. Keys are UPPERCASE and reach the flight as `pg_HOST`, `pg_PORT`, and so on:

```
CREATE SECRET pg IN motherduck (  
  TYPE flights,  
  PARAMS MAP {  
    'HOST': 'your-host', 'PORT': '5432', 'DATABASE': 'your_db',  
    'USER': 'readonly_user', 'PASSWORD': 'your_password', 'SSLMODE': 'require'  
  }  
);
```

This is MotherDuck's own Postgres-ingest pattern, trimmed for the page. The full plan adds schema/table include-exclude filters, per-table retries, and a `flight_tracker` audit table, and you can deploy it as-is: [flight.py in the flight-postgres-ingest example](#), walked through in the [cookbook](#).

Test it locally first. A Flight is just a script. Set `MOTHERDUCK_TOKEN` plus the `pg_*` vars and run it on your laptop before you ever schedule it:

```
MOTHERDUCK_TOKEN=... pg_HOST=... pg_DATABASE=... pg_USER=... pg_PASSWORD=... \  
uv run flights/sync_postgres.py
```

Step 4 - Deploy and schedule the Flight from your repo

The same file, now hosted and on a cron. A small script reads `sync_postgres.py` and registers it with MotherDuck via `MD_CREATE_FLIGHT` - so deploying the pipeline is a command in your project, not a click in a console:

```
// scripts/deploy-flight.ts (abridged) - run with: pnpm deploy:flight  
const source = readFileSync("flights/sync_postgres.py", "utf8");  
  
await conn.run(`  
  SELECT flight_id FROM MD_CREATE_FLIGHT(  
    name := 'multishop-sync',  
    access_token_name := 'analytics-token',  
    source_code := $py${source}$py$,  
    requirements_txt := 'duckdb\ntenacity',  
    flight_secret_names := ['pg'],           -- pg_HOST, pg_PORT, ... arrive as env vars  
    config := MAP { 'TARGET_DATABASE': 'multishop_commerce' },  
    schedule_cron := '0 * * * *'           -- hourly, UTC  
  )  
`);  
// then: CALL MD_RUN_FLIGHT(flight_id := '...') to trigger the first run
```

You can also deploy MotherDuck's [flight.py](#) as-is - point the MCP `create_flight` tool at it with `flight_secret_names := ['pg']` and the same `TARGET_DATABASE` config, and skip writing your own.

That's the full sync layer. The pipeline runs on MotherDuck's compute, on a schedule, with run logs in the MotherDuck UI. Keeping the extraction here, rather than wiring a scanner job into your app, means the heavy lifting lives next to the data and the reads stay simple. The Flight is the home for the extraction; the next step is the home for the reads.

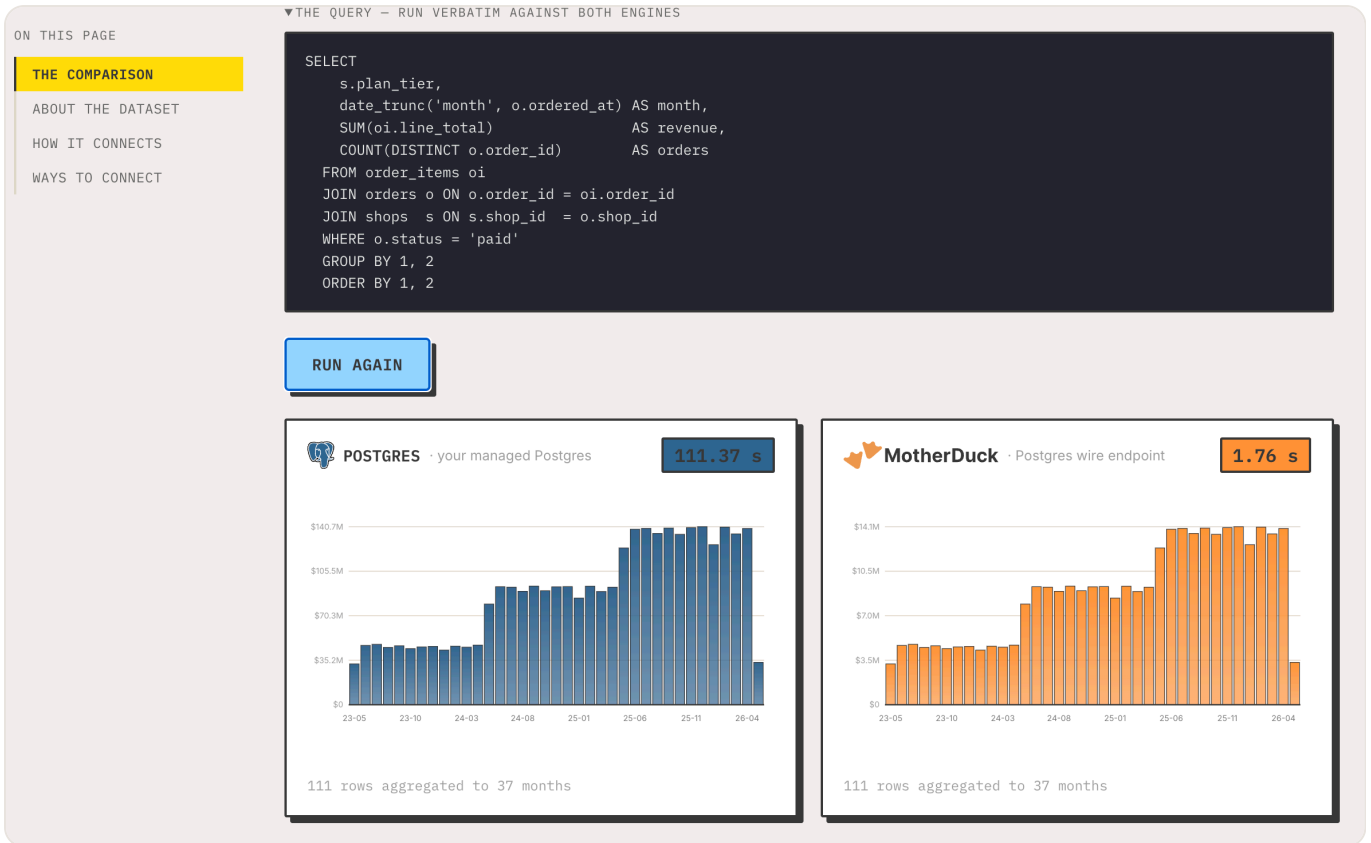
Step 5 - Serve the dashboard from MotherDuck (the "after")

Here is the part that makes the whole guide feel like a re-route rather than a rewrite. MotherDuck speaks the **Postgres wire protocol**, so your dashboard talks to it with the *same* `pg` driver and the *same* SQL it used for Postgres. The before/after is literally one config value - which host the connection points at:

```
// lib/db.ts - the before/after switch
export function poolFor(source: "postgres" | "motherduck") {
  if (source === "motherduck") {
    return new Pool({
      host: process.env.MD_PG_HOST,           // pg.<region>.aws.motherduck.com
      user: "motherduck",
      password: process.env.MOTHERDUCK_TOKEN, // the token is the credential
      database: process.env.MD_DATABASE,
      ssl: { rejectUnauthorized: false },
    });
  }
  return new Pool({ connectionString: process.env.POSTGRES_URL }); // your managed PG
}
```

The Server Component never changes - it just reads from `appPool()`, whose host is chosen by `DATA_SOURCE`. No driver swap, no SQL rewrite, nothing new to install: it is the standard `pg` driver your app already uses, pointed at a different host. That is the whole appeal of the endpoint - the simplest possible read path.

The demo makes the difference impossible to miss by running **both at once**. The comparison page draws the same query as a chart from each engine, calling `/api/chart?source=postgres` and `/api/chart?source=motherduck` independently. Each panel renders the moment its engine answers, so you literally watch a timer tick up while Postgres grinds, while the MotherDuck panel has already drawn its bars. Same query, same driver, two hosts, side by side.



The comparison page in the demo: same query, same pg driver, two hosts. Postgres 111.37 s, MotherDuck 1.76 s.

The benchmark. The two-panel race is the demo; here are the measured numbers behind it. The setup, so they're reproducible and fair:

BENCHMARK SETUP

Postgres	PlanetScale Postgres 18.4, single node, ½ vCPU / 4 GB RAM (the "PS-40" tier), Amazon EBS, <code>us-east-1</code>
MotherDuck	queried over the Postgres wire endpoint with the same <code>pg</code> driver and SQL - no special client
Data shape	500 shops · 12 categories · 50K products · 500K customers · 20M orders · 39.4M order_items · ~3 years of history
Size	the <code>order_items</code> fact is ~6-8 GB - larger than the 4 GB of RAM , so Postgres can't cache it and must scan from disk
Indexes (on Postgres)	primary keys on <code>shops</code> / <code>products</code> / <code>customers</code> , a unique index on <code>orders(order_id)</code> , and join/filter indexes on <code>order_items(order_id)</code> , <code>order_items(shop_id)</code> , <code>orders(shop_id)</code> - then <code>ANALYZE</code> . The indexes a real app would have.
Method	identical SQL both sides; 3 runs each; <code>cold</code> = first run, <code>best</code> = fastest of the three

```
$ uv run benchmark.py
Dataset: 39,382,720 order_items, 20,000,000 orders, 500 shops

— Platform-wide monthly revenue
Postgres    cold  107.56 s  best  101.40 s  (111 rows)
MotherDuck  cold   1.30 s  best   210 ms  (111 rows)
→ MotherDuck is 483× faster (best-of-3)  ✓ row counts match

— Single-shop category trend, 12 months
Postgres    cold   39.80 s  best   38.83 s  (143 rows)
MotherDuck  cold   0.99 s  best   181 ms  (143 rows)
→ MotherDuck is 214× faster (best-of-3)  ✓ row counts match
```

These are real numbers, and the key point is that **Postgres is tuned the way you'd actually run it** - proper indexes, fresh stats - and the platform aggregate still takes ~100 seconds warm. MotherDuck answers the same SQL in ~200 ms. Try a few queries of your own; the gap moves around, but on real analytical scans it stays in this range. Note the second one: even with an index on `order_items(shop_id)`, a large tenant's 12-month rollup is ~39 s on Postgres, because it's still an analytical *scan*, not a point lookup - exactly the Chapter 2 distinction.

Same query, same driver, same result set - a different engine on the read path. Postgres keeps serving the app; the dashboard stops competing with it.

Step 6 - Customer-facing per-shop drill-down

The admin dashboard is internal. The other audience is each **shop owner**, who should see *their own* revenue-by-category, filterable, in your product. Shipping a server round-trip per filter click is the slow pattern Chapter 3 warned about; the clean answer is an **Embedded Dive** - a shareable MotherDuck analytics view you drop into your app in an iframe, scoped to one `shop_id`:

```
// app/dashboard/ShopDrilldown.tsx
<iframe src={`${process.env.NEXT_PUBLIC_DIVE_URL}?shop_id=${shopId}`} />
```

Under the hood the Dive runs on **DuckDB-Wasm in the customer's browser**: the heavy filtering and aggregation happen locally on data MotherDuck has prepared, so every drill-down is instant and never touches your server. You handle the embed and the per-shop scoping; MotherDuck's SDK handles the Wasm pipeline. (If you'd rather run the Wasm yourself, hand the browser a scoped Parquet - `COPY (SELECT ... WHERE shop_id = $1) TO ...` - and query it with `@duckdb/duckdb-wasm`.)

Cost ballpark for this setup

Andrew and Jacob's internal MotherDuck use case is the upper end of "small team" - 150GB of Postgres data synced every six hours, ~10 minutes per sync. Your team's setup is likely smaller. MotherDuck bills active CPU time plus storage (more on the model in Chapter 7), so an analytics workload that sits idle most of the day costs far less than an always-on warehouse - plug your own numbers into the [pricing page](#) rather than trusting a guessed figure here.

What if this exact shape doesn't fit?

This walkthrough is one shape. The decision tree in Chapter 5 points to four others.

- **Cannot move data out of Postgres** (regulated, network-restricted) → replace the Flight (Steps 3-4) with `pg_duckdb` installed inside your Postgres server. Same SQL surface, no data movement, all queries run inside Postgres but use the columnar engine. *Caveat:* `pg_duckdb` runs inside the same Postgres process, so analytical queries share compute with your OLTP workload. Run it on a read replica if you want the OLAP load isolated from the production primary.
- **BI tool already speaks Postgres** (Metabase, Tableau, Power BI configured for Postgres) → you already have the connection. Point the BI tool at MotherDuck's Postgres endpoint instead of your primary - the same swap Step 5 makes for the dashboard. The tool does not know it is not Postgres. Existing dashboards work, faster.
- **Need real-time freshness** → keep the dashboard read path (Step 5) but replace the scheduled Flight with a CDC pipeline (Estuary, or Debezium → Kafka). The rest of the architecture stays the same.

We build the guide around one shape to keep the walkthrough concrete. The others are listed so you do not feel cornered if yours is different.

CHAPTER 7. OPERATING IT ON DAY 2

Shipping the walkthrough is the easy part. Operating it through Postgres schema changes, growing data, and the inevitable on-call moment is what determines whether this becomes the architecture your team actually trusts. This chapter is the operational shape - the things worth knowing on day one of running a real analytics setup, even a small one.

Schema drift

Postgres schemas change. Migrations land. Columns get added, renamed, occasionally dropped. The sync job has to handle this without falling over.

Wildcards are the friend. `SELECT *` in sync loads is fine and resilient. When a new column appears in Postgres, the next sync brings it into MotherDuck. DuckDB and MotherDuck handle column additions automatically.

Type changes and renames are the cases that bite. New columns are easy; *changed* ones are not. An upstream migration that flips a column from `TEXT` to `INTEGER`, or renames it, looks fine in Postgres and quietly breaks the sync downstream - a `RENAME COLUMN` reads as "old column gone, new column appeared," and a type change can fail the load or silently coerce values. The sync layer has no way to know the new `INTEGER` column *is* the old `TEXT` one. You can paper over it reactively (treat each as a deploy: pause the sync, reconcile the type or rename in MotherDuck, resume), but you're always a step behind.

The durable fix is upstream, not in the pipeline. Don't build elaborate schema-diffing into the sync job - it's a database, not a state machine. Instead, agree a **data contract** with the producers of those tables: a lightweight, explicit agreement on the schema and types the analytics layer depends on, so a breaking change is a coordinated decision rather than a 6 AM surprise. (See [Data Contracts](#), Sanderson, Freeman & Schmidt, O'Reilly.) Even an informal "these columns are load-bearing for analytics, ping us before you change them" goes a long way at this size.

Cost

MotherDuck's pricing is active CPU time plus storage. You pay for the seconds you are running queries, plus the data you store. This is different from traditional warehouses where you pay for an always-on cluster regardless of utilization.

For teams new to usage-based pricing, the mental adjustment is: **idle costs nothing**. Analytics setups have a heavily peaked usage curve - Monday morning everyone refreshes their dashboards, then it is quiet. The always-on pricing model overcharges for the quiet hours. Active CPU pricing does not.

Security

Secrets. Your Postgres connection string lives in your Vercel environment (or AWS Secrets Manager, or wherever you keep app secrets). Your MotherDuck token lives next to it. Treat them with the same care you treat your database password - they grant the same level of access.

Network. Most managed Postgres providers (Neon, Supabase, RDS, Aurora) ship with TLS by default. Use it. If you're connecting from Vercel's edge functions to a Postgres in a VPC, you'll need a public endpoint or a private connection (Neon's branching with private endpoints, AWS PrivateLink, etc.).

Least privilege. Don't sync as the Postgres superuser. Create a dedicated `replicator` role with `SELECT` on the analytics-shaped tables only:

```
-- in Postgres
CREATE USER replicator WITH PASSWORD 'use-a-good-one';
GRANT CONNECT ON DATABASE myapp TO replicator;
GRANT USAGE ON SCHEMA public TO replicator;
GRANT SELECT ON events, users, orders TO replicator;
```

The blast radius if this credential leaks is roughly "read access to data you'd already share with a BI tool." Manageable.

Observability

There are two layers to wire up: **did the job run** and **did the job do the right thing**.

Where the sync job lives determines the first layer. On **MotherDuck Flights**, every run and its logs are captured in the MotherDuck UI, and MotherDuck also keeps a queryable **query history** - every query that ran, with its duration and the compute it burned. Both are just data, which is the point: you can **build your own observability** on top of them. Build a [Dive](#) that reads the Flight logs and the query-history table and you have a live dashboard of job duration, failures, sync lag, slow queries, and cost drivers - no extra tooling, the same engine watching itself. (We run our own internal observability exactly this way.) On **GitHub Actions** or a third-party orchestrator (Dagster, Airflow, Prefect), use whatever monitoring that tool already provides; on **Vercel Cron**, wire failures into your existing alerting (Sentry, PagerDuty, a Slack webhook).

The second layer matters more. The common failure mode is sneakier than a crash: the function ran fine but quietly copied no new data. Add a sanity check at the end of the sync:

```
// at the end of the sync handler
const { current_max, last_synced_max } = await conn.runAndReadOne(`
  SELECT
    (SELECT MAX(created_at) FROM events) AS last_synced_max,
    (SELECT MAX(created_at) FROM pg.public.events) AS current_max
`);
if (current_max - last_synced_max > '2 hours'::INTERVAL) {
  notify('Sync lag exceeded 2 hours');
}
```

The first sign of trouble that a user will see is "the dashboard's numbers look old." Beat them to it. Add a `last_synced_at` timestamp to the dashboard footer so your operator sees the staleness before a user files a ticket.

CLOSING

Postgres ships products. It always has, it still does, and nothing in this guide is a knock on it. Choosing Postgres was the right call. The mistake comes later, when you ask it to be two databases at once and then blame it for being bad at the half it was never built for.

So don't break up with it. Re-route. The OLTP path stays exactly where it is, the OLAP path moves to an engine that reads columns instead of rows, and your app never notices the difference. Your dashboards will. That is the whole trick, and the reason it feels too easy is that it is supposed to. One workload finally gets the room it was quietly starving for, and you never had to take on a platform to give it.
