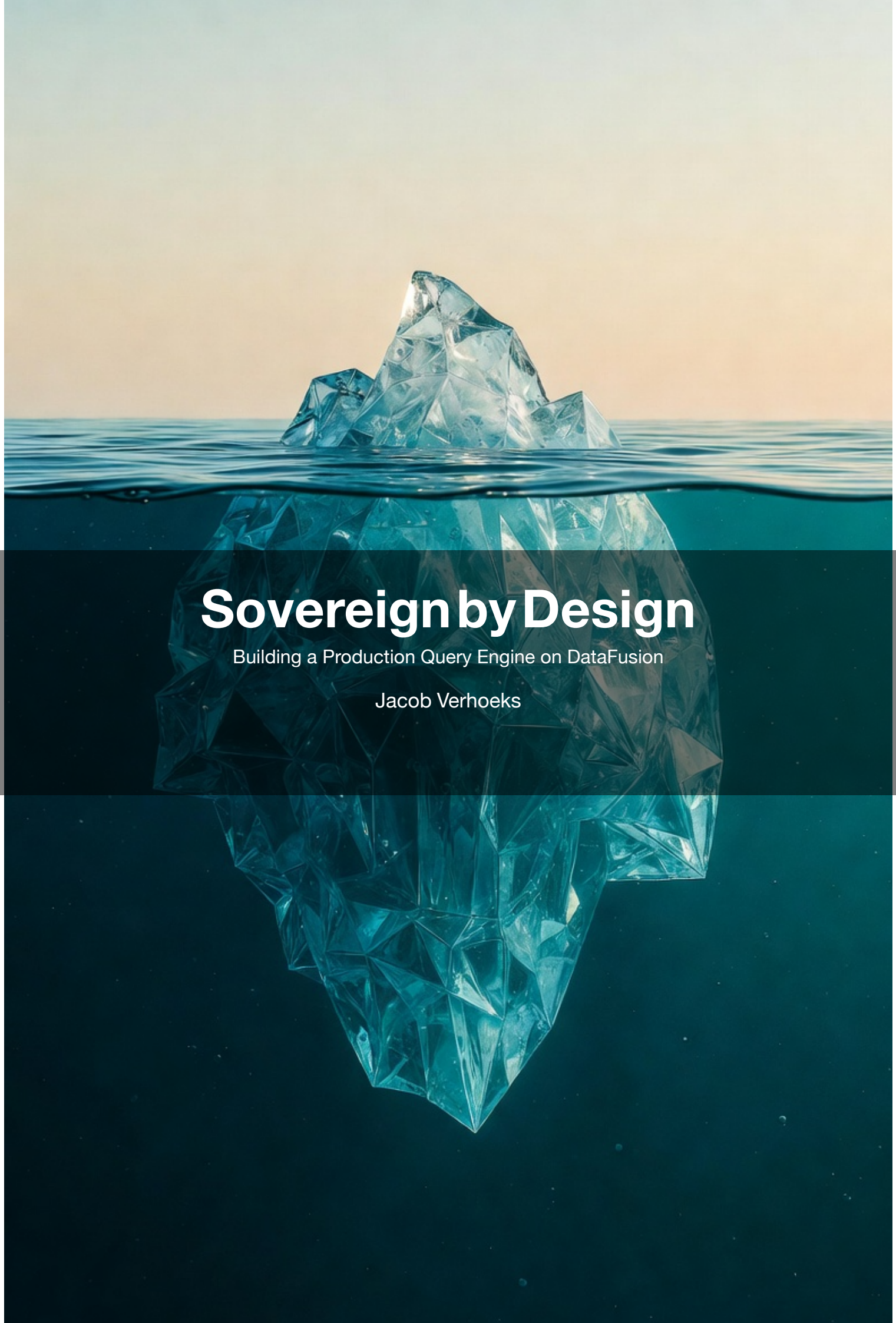

Sovereign by Design

Building a Production Query Engine on DataFusion

Jacob Verhoeks

Contents

Preface: The Sovereignty Thesis

> "The general who wins a battle makes many calculations in his temple before the battle is fought. The general who loses a battle makes but few calculations beforehand." — Sun Tzu, *The Art of War*

We had a Trino cluster. It worked. Mostly.

It ran our analytical queries against Iceberg tables in S3, served dashboards for the data team, and powered the nightly dbt pipeline that turned raw event data into something the business could actually read. We had it deployed on Kubernetes with a Helm chart we'd wrestled into shape over six months. We knew its quirks. We knew which config knobs to turn when queries got slow and which ones to leave alone because they'd been set by someone who'd left the company.

Then one morning the security team asked a simple question: "Who accessed the customer table last Tuesday?"

We couldn't answer it.

Not because we hadn't thought about auditing. We had. The problem was more fundamental. Every query that hit S3 came from the same identity: the Trino service account. When Alice queried customer data, it was the service account that read the files. When Bob ran an aggregation over financial records, same service account. When the dbt pipeline transformed everything at 2am, same service account again.

From S3's perspective, from CloudTrail's perspective, from the security team's perspective — it was all one actor. The Trino service account did everything and everyone did nothing.

That was the moment the idea started forming.

The Road Here

This book didn't start with Rust. It started with frustration.

For years I worked in the AWS data ecosystem — Glue, Snowflake, managed Spark, the full vendor stack. I wrote about it on dev.to, tracking each step of the journey in real time. The early articles are about Glue custom connectors and Snowflake integrations. They work. They're fine. They're also completely dependent on vendor decisions.

Then Apache Iceberg changed the game. In late 2024, I started experimenting with Iceberg REST APIs

— first through AWS Glue, then through Databricks Unity Catalog, then through DuckDB connecting to S3 Iceberg tables directly. Each experiment peeled back a layer. The table format was open. The data files were in your S3 bucket. The only thing still proprietary was the catalog — the thing that tells you which tables exist and where.

When Apache Polaris appeared — a pure Iceberg REST catalog with no opinions about your query engine, your governance layer, or your cloud provider — the last proprietary piece fell away. For the first time, you could have a fully open data stack: open table format (Iceberg), open catalog protocol (REST), open storage (S3-compatible), and… what query engine?

DuckDB was single-node and embedded. Spark was a cluster framework, not a query engine. Trino couldn't pass user credentials through to storage. Every option had a gap.

DataFusion had none of these gaps — because it wasn't a product. It was a library. A Rust library that gave you a complete query engine in a single function call: parse SQL, plan it, optimise it, execute it, return Arrow batches. Everything else — auth, catalog, storage, distribution — was your problem.

That's when SQE started.

Why Build a SQL Engine at All?

People asked. A lot.

The first answer is simple: **because we can.** A team that builds a query engine from scratch develops an understanding of query planning, execution, and distribution that a team running a managed service never acquires. The exercise itself makes you better at operating *any* data infrastructure.

The second answer is more interesting: **to challenge ourselves to build the right tool.** Not the most general tool. Not the most feature-complete tool. The right one — for our security model, our data architecture, our operational constraints. Building SQE forced us to articulate what we actually needed, rather than accepting what was available.

The third answer only becomes clear in hindsight: **because the tools that exist don't match.** Trino's auth model is incompatible with zero-trust. Spark is too heavy for interactive queries. DuckDB is single-node. DataFusion is a library, not a product. The gap between "library" and "product" is exactly the gap this book fills.

The Open-Source Goal

SQE is built to be open-sourced. This isn't an afterthought or a marketing strategy. It's a design constraint. Every architectural decision in this book was made with the assumption that the code would be public, that strangers would read it, and that organisations with different security models, different catalogs, and different cloud providers would try to run it. That constraint shaped everything: pluggable auth (you shouldn't need our OIDC provider), pluggable catalog (you shouldn't need Polaris), pluggable policy (you shouldn't need OPA or Cedar), configurable everything (if we hardcoded it, you'd have to fork it).

A sovereign engine that only works in one environment isn't sovereign. It's just private. The goal is an engine that works in *any* environment — because the operator controls the configuration, not the developer. Open-sourcing a query engine also means something specific: the code becomes the documentation. Every trait, every config key, every error message is part of the public API. This book exists, in part, to bridge the gap between "here's the code" and "here's why the code is this way."

What Sovereignty Means

Sovereignty, in the context of data infrastructure, is a precise claim: every component in the pipeline runs under your control, with your policies, using credentials that trace back to an individual human. No shared secrets. No ambient authority. No "the engine has access to everything and we trust it to enforce the rules."

The requirements, once we wrote them down, were almost comically simple:

1. Every query runs as the authenticated user
2. The user's identity propagates to every system the query touches
3. Policy enforcement happens inside the engine, not around it
4. The engine has no ambient credentials
5. The audit trail shows which human accessed which data
6. The engine runs on your infrastructure, under your control
7. The source is open, the config is yours, the data never leaves

We looked at every major query engine. None had this model completely. The authentication model is so deeply embedded in query engine architecture that you can't bolt it on. You have to build it in from the first line of code.

Connection to *The Art of Agents*

This book is a companion to *The Art of Agents: Building Agentic AI Systems That Think Before They Code*. That book presents 13 principles for building agentic systems, structured around Sun Tzu's *Art of War*. Several of those principles shaped how we built SQE — particularly the Five Constants (Contract, Context, Terrain, Model, Protocol), which map directly onto the SQL standard, Iceberg metadata, the catalog landscape, DataFusion, and the OIDC + Flight SQL protocol stack. You don't need to have read *The Art of Agents* to follow this book, but if you have, you'll recognise the patterns.

How to Read This Book

Every chapter in this book is a problem being solved. Some problems are architectural ("how do we pass user identity through to S3?"). Some are operational ("what happens when a worker dies mid-query?"). Some are existential ("should we build this at all?"). But they're all problems, and the book follows the shape of solving them — what's in the way, what we tried, what worked, what we learned.

If you're the kind of person who reads technical books to see how someone else thought through a hard problem, you're the audience. If you're looking for API documentation, that's in `docs/book`.

Part I (Chapters 0–2) is the problem that started everything. The catalog landscape, the Iceberg stack, the question of whether to build at all.

Part II (Chapters 3–6) is the first real challenge: a working single-node engine. DataFusion, auth, Flight SQL, catalog integration. Each chapter is a door we had to get through.

Part III (Chapters 7–10) is where it gets interesting. Writes, security policy as plan rewriting, observability, configurability. Each chapter is an obstacle we didn't fully anticipate.

Part IV (Chapters 11–14) is the hard part. Distributed execution. Ballista. The load test that broke everything. This is where the most dead ends are.

Part V (Chapters 15–17) is the honest accounting. Deployment, benchmarks, and what we'd do differently — including the frank assessment of what AI-assisted development actually looks like in practice.

Writing While Building

This book is being written at the same time as SQE is being built. That's deliberate. Most technical books are written after the fact — the author finishes the project, then reconstructs the decisions from memory. The decisions get cleaner in hindsight. The wrong turns get edited out. The confusion gets smoothed over. This book doesn't do that. Each chapter is written close to when the feature was implemented. The frustration is fresh. The wrong turns are still in the git log. The design decisions haven't been retroactively rationalised into inevitability.

The numbers tell the story: 316 commits (across all branches) in 15 days. From initial commit to distributed execution with a concurrent load test. From zero crates to a benchmark suite running TPC-H, TPC-DS, ClickBench, SSB, TPC-E, TPC-BB, and TPC-C. Writing two books and building an engine simultaneously is either very efficient or very foolish. Ask me again when they're all finished.

The Running Code

This book's running code is the SQE repository itself. Each chapter references specific crates, modules, and tests. Tagged commits mark the engine at each stage.

You don't need to check out tags to follow the book. The current `main` branch contains everything. But if you want to see the engine in a simpler state, the tags are there.

Acknowledgements

SQE exists because of the DataFusion community. The quality of DataFusion as a library — its extensibility, its performance, its documentation — made it possible for a small team to build a production query engine in months rather than years.

The iceberg-rust maintainers built the table format library that handles the Iceberg spec so we don't have to. The Polaris team at Snowflake open-sourced the catalog that proved REST-based table discovery works.

Rafael Herrero has been the other half of this project from the start. While I was deep in the query engine internals — DataFusion plans, Arrow batches, Iceberg commits — Rafael was building the deployment and operational layer that makes SQE actually runnable in production. The Kubernetes deployment, the Helm operator, the security hardening, the network policies, the mTLS configuration between coordinator and workers — that's Rafael's work. He's the person who took a binary that runs on a laptop and turned it into something that deploys securely across namespaces with proper RBAC, pod security standards, and automated rollouts. Many of the architectural decisions in this book — pluggable auth, TLS everywhere, the separation between coordinator and worker configs — exist because Rafael was asking the right deployment questions while I was writing Rust. Building a query engine is one thing. Operating it at the security standard Schuberg Philis demands is another, and that's where Rafael's expertise shaped the project.

The VPF Data & AI team at Schuberg Philis ran the queries, filed the bugs, and never once asked "why don't we just use Trino?" after the first week.

And to everyone who looked at this project and asked "why would you build a SQL engine?" — this book is the answer.

Jacob Verhoeks March 2026

The Catalog Wars

> Before you can query data, something has to tell you where it is. That something is the catalog. And everyone wants to own it.

Every data platform has a centre of gravity. It's not the compute engine. It's not the storage layer. It's the thing that answers two questions: what tables exist, and where are their files?

That thing is the catalog.

For fifteen years, we pretended the catalog didn't matter. It was Hive Metastore running on a MySQL instance that nobody patched, managed by a team that had inherited it from the team before them. It worked. Nobody thought about it. And that was the problem — because when the catalog is invisible, whoever controls it controls your data platform, and you don't even notice until you try to leave.

This chapter is about what happened when the industry noticed. The battle lines that formed. The bets that were placed. And why, after testing every major catalog implementation available in 2024 and 2025, we chose the simplest one.

The Hive Metastore Era

Hive Metastore was never designed. It was accreted. What started as a metadata store for Apache Hive grew into the default catalog for Spark, Presto, Trino, and every other engine that needed to know where Parquet files lived.

The protocol was Thrift RPC. The backend was a relational database — usually MySQL, sometimes PostgreSQL, occasionally Derby for development clusters that should never have reached production. The schema tracked databases, tables, partitions, and column statistics. It did this adequately.

The problem was coupling. Your catalog spoke Thrift, so your engine had to speak Thrift. Your catalog stored partition locations as absolute paths, so your storage layout was baked into your metadata. Your catalog ran as a single service, so your blast radius was one JVM crash away from every query in the organisation failing simultaneously.

Worse: Hive Metastore was a Java service that ran in the Hadoop ecosystem. When the industry moved to cloud object storage, the Metastore came along — usually as AWS Glue Catalog or a self-managed instance on Kubernetes. The protocol didn't change. The coupling didn't change. The single point of failure didn't change. We just moved the problem to a different data centre.

I ran Hive Metastore instances for years. Every migration was the same: export the metadata, transform the storage paths, import into the new instance, pray that the partition statistics survived. It worked often enough to not get replaced. It failed often enough to generate a steady stream of two-in-the-morning pages.

The REST Revolution

The Iceberg REST Catalog specification changed everything.

Apache Iceberg needed a catalog — a way to resolve table names to metadata file locations. The earliest Iceberg deployments used Hive Metastore or direct Hadoop filesystem calls. But the Iceberg community made a decision that, in retrospect, was more consequential than any feature in the table format itself: they defined a catalog protocol as HTTP REST.

Not Thrift. Not gRPC. Not a language-specific SDK. HTTP with JSON payloads and a well-defined OpenAPI specification.

This sounds unremarkable. HTTP APIs are everywhere. But for the data catalog space, it was a break from two decades of tight coupling. An HTTP catalog can be consumed by any language, any engine, any cloud. A Python script and a Rust query engine and a Java data pipeline can all talk to the same catalog with nothing more than an HTTP client library.

The specification defines the operations you'd expect:

- `GET /v1/{prefix}/namespaces` — list namespaces
- `POST /v1/{prefix}/namespaces/{namespace}/tables` — create a table
- `GET /v1/{prefix}/namespaces/{namespace}/tables/{table}` — load table metadata
- `POST /v1/{prefix}/namespaces/{namespace}/tables/{table}` — commit table updates

But the specification also defines something less obvious and far more important: credential vending. When a client loads a table, the catalog can return temporary storage credentials scoped to that specific table. The client doesn't need ambient S3 access. It doesn't need IAM roles pre-provisioned for every table. It asks the catalog for a table, and the catalog gives it both the metadata and the keys to read the files.

This is the mechanism that makes bearer token passthrough possible. The user authenticates to the catalog. The catalog decides what they can access. The catalog vends the storage credentials. The query engine is just a conduit — it passes the user's identity through and receives table-scoped credentials back.

Glue: The Catalog You Get for Free

AWS Glue Catalog was our first encounter with a production catalog at scale. It ships free with every AWS account. It backs Athena, Redshift Spectrum, EMR, and Lake Formation. If you're on AWS and you have data in S3, Glue is already your catalog whether you chose it or not.

For a long time, Glue only spoke its own API — the `aws-sdk-glue` interface. You called `GetTable`,

`GetPartitions`, `CreateTable`. It was AWS-specific, but it worked. Every AWS-native tool supported it.

Then in late 2024, AWS added an Iceberg REST endpoint to Glue. On paper, this was the best of both worlds: the operational simplicity of a managed service with the open protocol of Iceberg REST.

I tested it with PyIceberg as soon as it was available.

From the blog: "Glue Iceberg Rest Api and PyIceberg" (December 2024) — I walked through configuring PyIceberg against Glue's REST endpoint, creating tables, running scans. The API was functional. The limitations surfaced within hours.

The Glue REST endpoint worked for basic operations. You could list namespaces, load tables, read metadata. But the implementation had gaps. Some Iceberg REST operations weren't supported. The credential vending model was tied to IAM — you couldn't pass an OIDC bearer token to Glue's REST endpoint and get back S3 credentials for a specific user. Glue assumed you were already authenticated via IAM. The REST API was a facade over the same Glue internals, not a first-class implementation of the Iceberg REST specification.

The real problem was architectural. Glue's catalog data lives in AWS's managed infrastructure. You can't export it to run on another cloud. You can't run a local instance for development. You can't inspect the underlying storage to debug metadata inconsistencies. The catalog is a black box that happens to speak HTTP.

For a single-cloud deployment where AWS is a permanent commitment, Glue is fine. It's reliable, it scales, it costs nothing extra. But we were building an engine that could run anywhere — on any cloud, on any S3-compatible storage, in any data centre. A catalog that only exists inside AWS doesn't fit that model.

There's also the Collibra angle. We use Collibra for data governance — classification, lineage, access policies. When we explored Collibra Protect with Snowflake and Iceberg tables, the governance layer worked because Snowflake was the enforcement point. But Glue has no equivalent enforcement surface. Lake Formation tries, but it's a separate system with separate concepts and separate permissions. Your governance tool says "mask this column for this role." Then you have to translate that into Lake Formation policies, Glue catalog permissions, and IAM policies. Three translations of one intent. Each translation is a place where the intent gets lost.

Dead end: Glue as the primary catalog. We used Glue for nearly two years of development and experimentation. It's how we learned the Iceberg REST protocol. It's how we validated that a query engine could talk to a catalog over HTTP. But it's also how we learned that a managed catalog is a dependency disguised as a convenience. When we tried cross-cloud scenarios — querying the same tables from both AWS and a local development environment — Glue couldn't follow. And when we tried to enforce governance policies across Glue tables, the translation layers between Collibra, Lake Formation, and IAM became their own maintenance burden.

Unity Catalog: Openness as Strategy

Databricks open-sourced Unity Catalog in mid-2024. This was a significant move. Unity had been Databricks' proprietary catalog for years — the thing that made Databricks workspaces aware of tables, columns, permissions, lineage. Open-sourcing it meant anyone could run a Unity Catalog instance and use it as their Iceberg REST catalog.

I tested it the same week Databricks published the Iceberg REST compatibility layer.

From the blog: "Unity Catalog Iceberg Rest Api and PyIceberg" (December 2024) — testing Unity's REST compliance with PyIceberg. It worked. Table creation, metadata loading, namespace management — all functional over the standard Iceberg REST API.

Unity Catalog's Iceberg REST support was more complete than Glue's. You could run it as a standalone server, connect from any Iceberg client, and it genuinely implemented the specification. For basic catalog operations, Unity was a credible open-source option.

But Unity Catalog is not just a catalog. It's a governance platform. It tracks permissions, column-level access controls, data lineage, model registrations. These are features, not bugs — but they create a gravity well. Once you adopt Unity's permission model, your security enforcement is tied to Unity. Once you use Unity's lineage tracking, your observability is tied to Unity. The catalog becomes the control plane, and the control plane becomes the platform.

This is exactly Databricks' strategy, and it's not a secret. Open-source the catalog, make it compelling, and the governance layer pulls teams toward the Databricks ecosystem. It's smart business. It's also the opposite of what we needed.

We needed a catalog that did one thing: map table names to metadata locations. No opinions about governance. No opinions about security enforcement. No opinions about which query engine talks to it. A catalog that would be equally happy being called by our Rust engine, a Python script, a dbt model, and a Java Spark job — without any of them needing to understand Unity's permission model.

The open-source version of Unity and the managed Databricks version are different beasts. The open-source version lacks many of the governance features. This is reasonable — the governance layer is the product. But it means evaluating Unity requires deciding which Unity you're evaluating: the ambitious platform, or the stripped-down catalog.

We chose neither. Not because Unity is bad, but because it is too much. A catalog should be a dumb registry that speaks a standard protocol. Unity wants to be smart. Smart catalogs make decisions you didn't ask for.

The Cross-Cloud Problem

Between the Glue experiments and the Unity evaluation, I spent time on a problem that clarified everything: cross-cloud table access.

The scenario: tables registered in Snowflake, queryable from tools outside Snowflake. Snowflake had added Iceberg table support — you could create tables backed by Parquet files in your own S3 bucket,

with metadata managed by Snowflake. This meant the data was nominally open, but the catalog was still Snowflake.

From the blog: "Bridging Clouds: Access Snowflake Iceberg Tables via Glue and Spark" (October 2024) — I built a bridge between Snowflake's Iceberg tables and AWS Glue, making tables visible to Spark EMR jobs. The bridge worked. It also revealed how deeply each catalog assumes it's the only catalog.

The bridging exercise taught me something I should have seen earlier. Every catalog assumes it owns the table. Glue stores the metadata location in its own database. Snowflake stores it in its own metadata layer. Unity stores it in its own persistence backend. When you bridge between them, you're synchronising metadata between two systems that each believe they're the source of truth.

This doesn't scale. Two catalogs means reconciliation logic. Three catalogs means a metadata mesh that nobody wants to maintain. The industry's answer to multi-cloud data access was to synchronise catalogs — to keep Glue and Snowflake and Unity all aware of the same tables. Every synchronisation layer adds latency, introduces consistency windows, and creates a new failure mode.

The right answer is one catalog. One source of truth for table metadata. Every engine, every tool, every cloud — they all talk to the same catalog. The catalog doesn't live inside any engine. It doesn't live inside any cloud provider's managed service. It runs where you put it, speaks HTTP, and has no opinions about what connects to it.

Field report: The Snowflake-to-Glue bridge worked in production for about six months. During that time, we hit metadata drift three times — tables that were updated in Snowflake but not yet synchronised to Glue, causing Spark jobs to read stale data or fail on schema mismatches. Each incident took about four hours to diagnose because the root cause was always "which catalog has the current version?" Every time, someone asked: "Why do we have two catalogs?" We never had a good answer.

The DuckDB Detour

Before settling on building our own engine, I spent time with DuckDB — the embedded analytical database that runs anywhere, needs no cluster, and processes Parquet at surprising speed.

From the blog: "DuckDB S3 Tables with Iceberg using Iceberg Rest API" (January 2025) — I connected DuckDB to an Iceberg REST catalog, read tables from S3, and ran analytical queries locally. No Spark. No Trino. No cluster management. Just a binary and a catalog URL.

DuckDB + Iceberg REST was the closest thing to the engine we wanted before we built one. It proved the model: a lightweight engine that talks to a REST catalog and reads Parquet from S3. The experience confirmed that the REST catalog protocol was the right abstraction layer. The catalog answered "what tables exist and where are their files," and DuckDB did the rest.

But DuckDB is embedded, single-node, and — crucially — doesn't support bearer token passthrough for per-user identity. Every DuckDB query runs as whatever credentials are configured at startup. For a personal analytics tool or a CI pipeline, that's fine. For a multi-user query engine where security auditing requires per-user attribution, it's a non-starter.

The DuckDB experiments mattered because they stripped away everything except the catalog interaction. No distributed complexity. No JVM overhead. Just: talk to the catalog, get metadata, read files, return results. That clarity shaped how we designed `sqe-catalog`. The catalog interface should be that simple, even when the engine behind it is distributed.

Gravitino and Nessie: Interesting but Different

Two other catalogs deserve mention because they solve adjacent problems.

Apache Gravitino is a meta-catalog — a federation layer that can front multiple underlying catalogs (Hive Metastore, Iceberg REST, JDBC catalogs) behind a single API. If you have three Hive Metastores and a Glue catalog and you want a unified namespace, Gravitino is the answer. It's technically impressive and solves a real problem for organisations with sprawling catalog infrastructure.

We didn't need federation. We needed to start clean. Gravitino solves the problem of having too many catalogs. Our problem was choosing one catalog to rule them all. Different starting points, different solutions.

Project Nessie adds git-like semantics to table metadata — branches, tags, commits, diffs. You can create a branch of your data lake, make changes, and merge them back. For data pipeline development, this is compelling. You can test a transformation on a branch without affecting production tables.

Nessie speaks the Iceberg REST protocol (or close to it), which made it a genuine contender. But the versioning model adds complexity to every catalog operation. A `load_table` call needs to know which branch or tag to resolve. A `commit` needs to handle merge conflicts. These are features we didn't need for a query engine — and features have a carrying cost, even when you don't use them.

LakeFS solves a similar versioning problem but at the storage layer rather than the catalog layer. It presents a versioned S3-compatible API, so tools think they're reading regular S3 but get branching and merging for free. We briefly considered it as a complement to a simple catalog. It's a good product solving a different problem than the one we had.

Polaris: The Catalog That Does Nothing Extra

Apache Polaris started life as Snowflake's internal Iceberg catalog. Snowflake contributed it to the Apache Software Foundation in 2024, and it entered incubation as an Apache project. The pedigree matters — this isn't an academic exercise or a startup's side project. It's a catalog that ran at Snowflake's scale, extracted and open-sourced.

Polaris implements the Iceberg REST specification. Not a subset. Not a superset with proprietary extensions. The specification.

When we first deployed Polaris, the contrast with every other catalog was immediate. There was no governance layer to configure. No permission model to learn beyond the Iceberg REST spec's own token-based auth. No UI to understand. You start the server, point it at a storage backend, and it speaks REST.

The configuration for SQE's catalog connection is four lines in a TOML file:

```
[catalog]
polaris_url = "https://polaris.internal:8181/api/catalog"
warehouse = "production"
metadata_cache_ttl_secs = 30
```

The Rust code that creates a per-session catalog connection reflects this simplicity:

```
pub struct SessionCatalog {
    inner: Arc<RwLock<RestCatalog>>,
    polaris_url: String,
    warehouse: String,
    bearer_token: String,
    token_fingerprint: String,
    storage_config: StorageConfig,
    http_client: reqwest::Client,
}
```

Each user session gets its own `SessionCatalog` instance, configured with the user's bearer token. The token goes straight to Polaris in the `Authorization` header. Polaris validates it and returns table metadata — including, critically, vended S3 credentials scoped to that user's permissions.

The `SessionCatalog::new` method tells the whole story. It builds a properties map with the token, the URI, and the warehouse, then hands it to `iceberg-rust`'s `RestCatalogBuilder`:

```
let mut props = HashMap::new();
props.insert("token".to_string(), bearer_token.to_string());
props.insert("uri".to_string(), polaris_url.to_string());
props.insert("warehouse".to_string(), warehouse.to_string());

let catalog = RestCatalogBuilder::default()
    .with_storage_factory(Arc::new(OpenDalStorageFactory::S3 {
        configured_scheme: "s3".to_string(),
        customized_credential_load: None,
    }))
    .load(
        format!("sqe-session-{}", &token_fingerprint),
        props,
    )
    .await?;
```

That's it. No IAM role configuration. No SDK-specific authentication dance. No catalog-specific client library. An HTTP client, a bearer token, and a URL.

Credential Vending: The Mechanism That Makes It Work

The REST specification's credential vending is the feature that separates a catalog from a metadata registry. When SQE loads a table from Polaris, the response includes not just the metadata location

and schema, but also temporary S3 credentials that can read the table's files.

These credentials are scoped. They grant access to the specific S3 prefix where that table's data lives. They expire. They're tied to the identity that requested them.

Our `CredentialCache` extracts these vended credentials from the table config:

```
pub fn extract_from_table_config(
    config: &HashMap<String, String>,
) -> Option<VendedCredentials> {
    let access_key = config.get("s3.access-key-id").cloned()?;
    let secret_key = config.get("s3.secret-access-key").cloned()?;
    let session_token = config.get("s3.session-token").cloned();
    // ...
    Some(VendedCredentials {
        access_key,
        secret_key,
        session_token,
        expiry,
    })
}
```

The property names — `s3.access-key-id`, `s3.secret-access-key`, `s3.session-token` — are defined in the Iceberg REST specification. Any conforming catalog returns them. This means our credential extraction code works with Polaris, but it would also work with any other catalog that implements vending correctly.

This is the sovereignty argument in miniature. The code depends on a specification, not on a product. If Polaris ceased to exist tomorrow, any compliant REST catalog would be a drop-in replacement for the credential vending flow.

Sovereignty principle: Credential vending moves the security boundary from the engine to the catalog. The engine never holds ambient storage credentials. It receives scoped, temporary credentials through a standard protocol. This means you can swap the catalog without touching the engine's security model. The security model is the protocol, not the product.

The DataFusion Bridge

One practical challenge: DataFusion's `CatalogProvider` trait is synchronous for some operations. You implement `schema_names()` and it returns `Vec<String>` — no async, no futures, no await. But listing namespaces from a REST catalog is inherently asynchronous. You're making an HTTP call.

Our solution is a cached snapshot. When `SqeCatalogProvider` is constructed, it makes one async call to list namespaces and caches the result:

```
impl SqeCatalogProvider {
    pub async fn try_new(
        session_catalog: Arc<SessionCatalog>,
        storage_config: StorageConfig,
```

```
        warehouse: String,
    ) -> sqe_core::Result<Self> {
        let namespaces = session_catalog.list_namespaces().await?;
        let cached_namespaces: Vec<String> = namespaces
            .iter()
            .map(|ns| /* ... */)
            .collect();
        Ok(Self {
            session_catalog,
            storage_config,
            warehouse,
            cached_namespaces,
        })
    }
}
```

The synchronous `schema_names()` method then returns the cached list. Tables are loaded lazily — `table()` is async, so it can call Polaris on demand.

This matters because it's a pattern you'll see throughout SQE: bridging between DataFusion's trait interfaces and the async reality of a REST catalog. DataFusion was designed with embedded catalogs in mind — Hive Metastore clients or in-memory registries. A remote REST catalog introduces latency and failure modes that the trait signatures don't anticipate. Every bridge we build is a small bet that the trait designers will eventually make async-friendly, and a pragmatic workaround until they do.

Why "No Opinions" Is the Architecture

Every catalog we evaluated had opinions. Glue has opinions about identity (IAM). Unity has opinions about governance (its own permission model). Nessie has opinions about versioning (branches and tags). Gravitino has opinions about federation (its meta-catalog layer).

Polaris has one primary opinion: tables have names and metadata locations. It does have its own role-based access control — catalog roles, principal roles, and privilege grants — which is a real governance layer. But it's a thin one. It controls who can see which tables and namespaces. It does not control what happens to the data after the table is loaded. It does not inject row filters. It does not mask columns. It does not rewrite queries.

This is a meaningful distinction. A catalog that controls access to table metadata is doing its job. A catalog that controls what happens inside your query engine is doing your job. Polaris stays in its lane. Your governance — row-level security, column masking, audit policies — can be OPA. Or Cedar. Or a custom policy engine built into your query engine (which is what we did in Chapter 8). Your versioning can be handled by Iceberg's own snapshot semantics. Your federation isn't needed because you have one catalog.

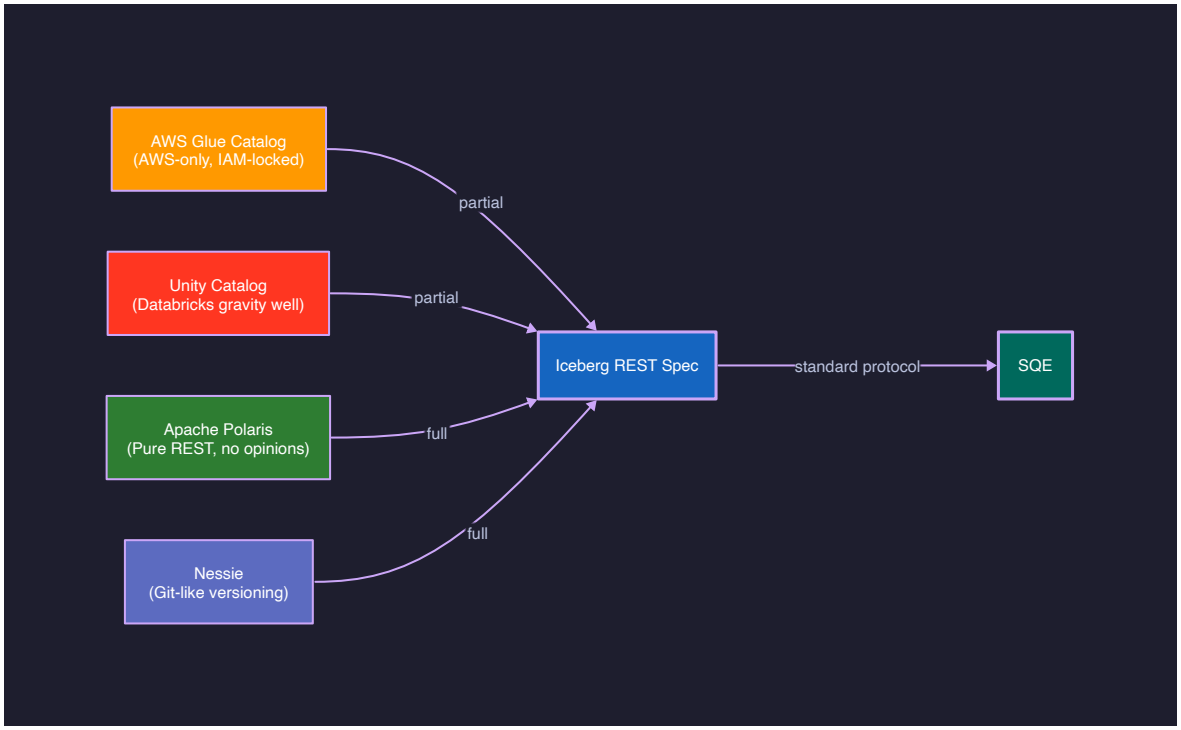

Figure 1: Catalog landscape: comparing REST spec compliance, auth models, and governance opinions across Glue, Unity, Polaris, Gravitino, and Nessie

Catalog	REST Spec	Auth Model	Governance	Opinions
Glue	Partial	IAM	Lake Formation	AWS-native, single-cloud
Unity (OSS)	Full	Built-in	Limited (OSS)	Databricks ecosystem gravity
Unity (Managed)	Full	Built-in	Full	Databricks platform
Polaris	Full	OIDC/OAuth2	None	Table registry only
Gravitino	Via proxy	Pluggable	None	Multi-catalog federation
Nessie	Near-full	Pluggable	None	Git-like versioning

The “Opinions” column is the one that matters. When a catalog has opinions about governance, those opinions compete with your own governance design. When a catalog has opinions about auth, those opinions shape your engine’s auth model. When a catalog has no opinions, you’re free.

Freedom has a cost: you have to make those decisions yourself. You have to implement governance, choose an auth model, design a security enforcement layer. For some teams, that cost is too high, and a catalog with built-in governance (Unity, Glue + Lake Formation) is the right choice.

For us, the cost was the point. We were building a query engine precisely because we wanted to make those decisions ourselves. A catalog with no opinions is the right foundation for an engine with strong opinions.

Running Polaris in Practice

Polaris runs as a Java application. You can deploy it from a JAR, a Docker container, or a Kubernetes Helm chart. The storage backend for catalog metadata can be an in-memory store (for development), a local file system, or a production database.

For our development and test environment, we run Polaris in memory:

```
polaris:
  image: apache/polaris:latest
  environment:
    POLARIS_BOOTSTRAP_CREDENTIALS: "root:s3cr3t"
  ports:
    - "8181:8181"
```

The entire catalog starts in under two seconds. No database to provision. No persistent volume to mount. For integration testing, this is transformative — every test run gets a clean catalog, and there's no cleanup needed.

For production, Polaris backs its metadata to a relational database (PostgreSQL is the typical choice). The metadata is small — table names, schema definitions, partition specs, snapshot references. The heavy lifting is in the Iceberg metadata files themselves, which live in object storage. The catalog is a pointer to those files, not a copy of them.

This architectural clarity — the catalog points, storage holds — is what makes the system composable. You can inspect the Iceberg metadata files directly with a tool like PyIceberg or `iceberg-rust` without going through the catalog at all. The catalog is the preferred path, but it's not the only path. Your data is never locked behind a catalog API.

One thing we learned quickly: Polaris's in-memory mode is not just a convenience for testing. It became the foundation of our entire integration test suite. Every test starts a fresh Polaris instance, creates namespaces and tables, runs queries through the full SQE stack, and tears everything down. No shared state between tests. No flaky failures from leftover metadata. The entire test stack — Polaris in-memory plus RustFS for S3-compatible storage — starts in under five seconds and requires no cloud credentials. We went from "integration tests need a running AWS environment" to "integration tests run on a laptop at an airport."

That shift in development velocity is hard to overstate. When testing the catalog integration is cheap, you test it more. When you test it more, you find problems earlier. When you find problems earlier, the catalog code is better. The tool shaped the practice.

Field report: During one debugging session, we needed to inspect a table's manifest list to understand a failed compaction. Instead of building a debug tool for the catalog, we pointed PyIceberg

directly at the S3 location from the catalog's `metadata_location` field and read the manifests manually. The table format is the source of truth. The catalog is an index.

The Centre of the Data Platform

I made a claim at the start of this chapter: the catalog is the centre of the data platform. Let me make it concrete.

When a user runs `SELECT * FROM production.events` in SQE, the catalog answers four questions:

1. **Does this table exist?** The catalog resolves `production.events` to a metadata location — an S3 path pointing to an Iceberg metadata JSON file.
2. **What does it look like?** The metadata file (fetched from S3 using vended credentials) contains the schema, partition spec, sort order, and current snapshot.
3. **Can this user read it?** The catalog's auth layer — OIDC token validation — determines whether the user's identity has permission to load this table's metadata. If not, the table doesn't exist (not an access denied error, just absence — no information leakage).
4. **How does the engine access the files?** The catalog vends temporary S3 credentials scoped to this table's storage location. The engine uses these credentials to read Parquet files.

Four questions. Every one answered by the catalog. If the catalog is wrong about any of them, the query fails. If the catalog is unavailable, no query can run. If the catalog is compromised, table visibility and storage credentials are both compromised.

This is why the catalog is the centre, even though it holds very little data itself. It holds the pointers, the permissions, and the credentials. Everything else — the actual data, the query execution, the result delivery — depends on those three things.

Control the catalog, and you control what data exists (namespace and table management), who can see it (authentication and authorization), and how it's accessed (credential vending). That's the data platform.

What We Actually Built

The `sqe-catalog` crate is the result of everything in this chapter. It's several thousand lines of Rust across eight core modules:

- **`rest_catalog.rs`** — The `SessionCatalog` struct: one per user session, wraps `iceberg-rust`'s `RestCatalog` with the user's bearer token. Handles tables, views, namespaces.
- **`catalog_provider.rs`** — The DataFusion bridge: implements `CatalogProvider` using cached namespace snapshots. Maps Iceberg namespaces to DataFusion schemas.
- **`schema_provider.rs`** — Implements `SchemaProvider` for a single namespace. Loads tables and views lazily. Handles the sync-to-async bridge for `table_names()`.
- **`table_provider.rs`** — Wraps an Iceberg `Table` as a DataFusion `TableProvider`. Converts schemas, pushes filter predicates down.

- **`credential_vending.rs`** — Extracts and caches vended S3 credentials from catalog responses. TTL-based cache using `moka`.
- **`info_schema.rs`** — Virtual `information_schema` tables (tables, columns, schemata) for SQL standard compliance.
- **`system_catalog.rs`** — Virtual `system` tables for runtime introspection.
- **`iceberg_scan.rs`** — The physical scan operator that reads Iceberg data via the table's `FileIO` and vended credentials.

Since then, the crate has grown to include `manifest_cache`, `footer_cache`, `circuit_breaker`, `s3_io`, `sort_order`, `read_parquet`, `iceberg_metadata_tvf`, `pruning_stats`, and several more — the caching and I/O layers that made SQE competitive with Trino.

The crate depends on `iceberg` and `iceberg-catalog-rest` from iceberg-rust, `datafusion` for the trait implementations, `moka` for caching, and `reqwest` for direct REST calls (views, which iceberg-rust doesn't support natively yet).

None of these modules knows or cares that the catalog is Polaris. They speak the Iceberg REST protocol. If we swapped Polaris for a compliant Nessie instance, or a future version of Unity's REST endpoint, or a catalog that doesn't exist yet — the code wouldn't change. The config file would change. That's it.

The Lesson

I spent most of 2024 testing catalogs. Glue, Unity, Snowflake's Iceberg support, cross-cloud bridges, Gravitino, Nessie. I wrote about each one. I built prototypes with each one. I hit limitations with each one.

The pattern was always the same. Start with the managed option. It works quickly. Hit a cross-cloud scenario, or a custom auth requirement, or a governance model that doesn't match the built-in one. Realise the catalog has opinions, and those opinions don't match yours. Either live with the mismatch or migrate.

The catalog wars aren't about features. Every major catalog can list tables and return metadata. The wars are about control. Every catalog vendor wants the catalog to be the gravitational centre of their ecosystem. Glue pulls you toward AWS. Unity pulls you toward Databricks. Snowflake's catalog pulls you toward Snowflake. They provide genuine value in exchange for that gravity.

Polaris has no ecosystem to pull you toward. It's a catalog. It stores table metadata. It validates tokens. It vends credentials. It does nothing else. For a project whose entire premise is sovereignty — running independently of any vendor's platform — that absence of ambition is the feature.

The Iceberg REST specification is the real winner. Not Polaris specifically, but the protocol. The protocol means your catalog choice is reversible. It means your engine doesn't know or care which catalog implementation answers its HTTP calls. It means the most important component in your data platform — the one that answers "what tables exist and where are their files" — is a standard interface that you can implement, replace, or self-host without touching a line of engine code.

AI Logbook: The AI produced the `SessionCatalog`, `CredentialCache`, and all eight `sqe-catalog`

modules in a single session from a spec that named each module and its purpose. The human chose Polaris over Unity, Glue, and Nessie after months of hands-on evaluation that the AI never saw. The credential vending extraction code — parsing `s3.access-key-id` from the REST response config map — was correct on the first pass; the three days of debugging Polaris 0.9-vs-1.0 timestamp format differences were entirely human detective work.

The next chapter covers what happens after the catalog tells you where the data is. The data itself — Apache Iceberg's table format, from metadata trees to manifest files to the Parquet files at the bottom. The catalog is the index. Iceberg is the book.

Tables Made of Files

An Iceberg table is not a table. It is a versioned filesystem with opinions about schema.

A database table exists inside a database. You move the database, you move the tables. You lose the database, you lose the tables. You want to query the table from a different engine, you either export it or you connect that engine to the database. The table belongs to the database. The database owns the table. This is how it has been since the 1970s.

A data lake file exists on storage. Anyone with access can read it. But nobody knows what the columns mean, whether the schema changed last Tuesday, or which files belong to a logical table called "orders." You have data, but you do not have a table. You have freedom, but you do not have governance.

Iceberg solves this by putting a metadata tree between you and the files. The metadata is the table. The files are just storage. And the metadata lives alongside the files on the same storage layer — not inside a database, not inside a vendor's service. On S3. On your S3.

That shift — from "the table is inside the database" to "the table is metadata over files" — is the single most important change in data infrastructure in the last decade. It is the foundation that makes a sovereign query engine possible.

The Metadata Tree

Iceberg's structure is a tree with four levels. Understanding these levels is understanding Iceberg.

At the top is the **metadata file** — a JSON document that describes the table. It contains the current schema, the partition spec, sort orders, and a list of snapshots. Each snapshot represents the table at a point in time. When you write to the table, you create a new snapshot. The old snapshots remain. This is how time travel works.

Each snapshot points to a **manifest list** — an Avro file that contains an entry for each manifest in the snapshot. The manifest list is the index of indices. It records which manifests belong to this snapshot, along with partition-level summary statistics.

Each manifest list entry points to a **manifest file** — also Avro. The manifest contains an entry for each data file it tracks, along with column-level statistics: min values, max values, null counts. These statistics are what make predicate pushdown work. When your query says `WHERE order_date > '2024-01-01'`, the engine reads the manifest, checks the max value of `order_date` in each data file,

and skips the files that cannot contain matching rows. The engine never opens those Parquet files. It never reads a single byte from them.

At the bottom are the **data files** — Parquet (or ORC, or Avro, but in practice always Parquet). The actual bytes. The actual rows. The actual columns.

```
metadata.json
  └── snapshot (current)
        └── manifest-list.avro
              ├── manifest-1.avro
           |    ├── data-file-001.parquet  (stats: order_date min=2024-01-01, max=2024-
01-31)
           |    └── data-file-002.parquet  (stats: order_date min=2024-02-01, max=2024-
02-28)
              └── manifest-2.avro
                └── data-file-003.parquet  (stats: order_date min=2024-03-01, max=2024-
03-31)
```

The tree is small. A table with 10,000 data files might have 10 manifests, one manifest list, and one metadata file. The metadata is typically a few megabytes. The data files might be terabytes. You read megabytes of metadata to decide which terabytes of data to skip.

Iceberg deep dive: The manifest-list-to-manifest-to-data-file hierarchy is not decorative. Each level provides progressive filtering. The manifest list filters by partition summary (skip entire partitions). The manifest filters by column statistics (skip individual files within a partition). The Parquet reader filters by row-group statistics (skip sections within a file). Three levels of pruning before you read a single row.

Snapshot Isolation Without a Database

Databases achieve snapshot isolation through MVCC — multi-version concurrency control, implemented with transaction logs and lock managers. Iceberg achieves it through metadata pointer swaps.

The metadata file contains a `current-snapshot-id`. A read query resolves that snapshot ID to a manifest list, and from there to manifests and data files. A write creates new data files, a new manifest (or modifies an existing one), a new manifest list, and a new metadata file pointing to the new snapshot. The write then atomically swaps the metadata pointer — in S3, this is a conditional put; in a catalog like Polaris, it is an atomic commit through the REST API.

Readers see a consistent table. Writers do not block readers. Two writers to different partitions do not conflict. Two writers to the same partition go through optimistic concurrency — one succeeds, the other retries. No locks. No transaction log. No WAL.

The simplicity is disarming. The first time you understand this, it feels too simple. It is not. It has been battle-tested at Netflix, Apple, and Airbnb scale. The simplicity is the point — there are fewer things to break.

Schema Evolution That Does Not Break Readers

A traditional database ALTER TABLE can be destructive. You add a column, and every reader must know about the new column. You rename a column, and every downstream consumer breaks. You change a type, and the data must be rewritten.

Iceberg schemas are versioned and identified by field IDs, not by field names or positions. Every column has a unique integer ID assigned at creation. If you add a column, it gets a new ID. If you rename a column, the ID stays the same. Old data files were written with the old schema. New data files are written with the new schema. The engine reconciles at read time — old files that lack the new column return nulls for it.

This is schema evolution without data migration. You never rewrite files for a schema change. You update the metadata, and the engine handles the rest.

For a sovereign query engine, this property is critical. The engine does not own the data. Multiple engines might read the same table. If adding a column required rewriting files, every reader would need to coordinate. With Iceberg, there is no coordination. The metadata file records the evolution history, and each reader interprets it independently.

Partition Evolution

Partitioning in Hive was a directory convention. If you partitioned by `year`, your data lived in `year=2024/` directories. If you later needed to partition by `month`, you had two choices: rewrite all historical data or live with two partitioning schemes and a query engine that could not reason across them.

Iceberg partitioning is metadata, not directory structure. The partition spec says "partition by `year(order_date)`," and the engine evaluates that function when writing. The resulting partition value is recorded in the manifest alongside the data file path. The data file itself can live at any path.

When you evolve the partition spec — say, from yearly to monthly — new data files are written with the new spec. Old data files retain their old partition values. The engine reads both. It knows that data file A was partitioned by year and data file B was partitioned by month, and it filters accordingly. No data rewrite. No migration.

Partition evolution is the feature that convinced me Iceberg was the right table format. Delta Lake and Hudi can handle schema evolution. Delta Lake has since introduced liquid clustering, and Hudi has added its own partition evolution support — the ecosystem is catching up. But Iceberg pioneered transparent partition evolution where old and new partition layouts coexist without data rewriting, and it remains the most mature implementation. For a query engine that does not own the storage, it means you can optimise the physical layout without a maintenance window.

The PyIceberg Experiments

Before writing a single line of Rust for SQE, I spent months working with Iceberg through Python. The experiments are documented on dev.to, and they shaped the design of everything that came after.

dev.to connection: "Glue Iceberg Rest Api and PyIceberg" and "Duckberg!" — these articles trace the path from first discovering the Iceberg REST protocol to understanding what it actually takes to query Iceberg tables from code you control.

The first experiment was connecting PyIceberg to AWS Glue's Iceberg REST API. Glue had added REST catalog support, which meant you could talk to Glue using the standard Iceberg REST protocol instead of the AWS-specific Glue API. This mattered because it meant the code I wrote would work against any catalog that implemented the same protocol.

PyIceberg made it almost trivially easy:

```
from pyiceberg.catalog import load_catalog

catalog = load_catalog(
    "glue",
    **{
        "type": "rest",
        "uri": "https://glue.us-east-1.amazonaws.com/iceberg",
        "rest.sigv4-enabled": "true",
        "rest.signing-region": "us-east-1",
    }
)

table = catalog.load_table("my_database.my_table")
scan = table.scan(row_filter="order_date >= '2024-01-01'")
df = scan.to_pandas()
```

Five lines to go from a REST catalog to a Pandas DataFrame. The scan handles manifest reading, file pruning, and Parquet reads. PyIceberg's implementation was clean, well-documented, and correct.

The second experiment bridged two catalogs. Unity Catalog (Databricks) had also implemented the Iceberg REST API, which meant you could read Databricks-managed Iceberg tables from outside Databricks. I wrote about connecting PyIceberg to Unity Catalog and reading the same tables that Databricks' Spark cluster read — from a laptop, with no Databricks runtime.

The third experiment was DuckDB with Iceberg. DuckDB's `iceberg` extension could read Iceberg tables from S3, and combined with a REST catalog, you had a zero-infrastructure query engine for Iceberg. No cluster. No coordinator. Just a binary and a catalog URL.

dev.to connection: "DuckDB S3 Tables with Iceberg using Iceberg Rest API" — the article where it clicked that the catalog is the only coordination point. If you control the catalog, you control access. Everything else is just reading files.

These experiments taught three things:

First, the Iceberg REST protocol is the real standard. Not the Java API. Not the Spark integration. The REST protocol is what enables interoperability, and any serious implementation must start there.

Second, Python is adequate for exploration but inadequate for production query engines. PyIceberg's scan planning works. The scan planning itself is Python-level and GIL-bound, though the underlying

Parquet I/O through PyArrow releases the GIL during C++ execution. For a single-user notebook, the distinction doesn't matter. For a multi-user query engine handling concurrent requests, the Python-level bottleneck in scan planning and metadata handling is a real constraint.

Third, the path from "can read Iceberg tables" to "is a query engine" is longer than it looks. DuckDB can scan Iceberg, but it cannot push predicates down to manifest pruning in all cases. PyIceberg can do predicate pushdown, but it cannot do distributed execution. The gap is not in any single component — it is in the integration between catalog, scan planning, predicate pushdown, and query execution.

iceberg-rust: The Good, The Rough, The Missing

When we decided to build SQE in Rust, the Iceberg library choice was straightforward. There is one: iceberg-rust, the official Apache Iceberg implementation for Rust. At the time we started, it was at version 0.8. By the time we reached production integration tests, it was at 0.9. The version delta matters — 0.9 changed the catalog builder API significantly.

Iceberg deep dive: The workspace Cargo.toml pins specific iceberg-rust dependencies:

```
iceberg = { git = "https://github.com/risingwavelabs/iceberg-rust.git", rev = "1978911ec4" }
iceberg-catalog-rest = { git = "https://github.com/risingwavelabs/iceberg-rust.git", rev = "1978911ec4" }
iceberg-datafusion = { git = "https://github.com/risingwavelabs/iceberg-rust.git", rev = "1978911ec4" }
```

We use the RisingWave Labs fork of iceberg-rust, pinned to a specific commit. The fork provides `OverwriteAction` for Copy-on-Write DELETE/UPDATE and `PositionDeleteFileWriter` for Merge-on-Read. Migration to upstream Apache iceberg-rust is planned once these features are merged.

The storage layer uses OpenDAL rather than iceberg-rust's native FileIO for S3 access, because OpenDAL handles credential refresh and path-style addressing more reliably.

What works well

Scan planning. The core of Iceberg — reading the metadata tree, evaluating partition filters, reading manifests, applying column statistics to prune data files — works correctly. The `table.scan()` builder gives you a fluent API for column selection and predicate pushdown:

```
let scan = table.scan()
    .select(["order_id", "order_date", "total"])
    .with_filter(predicate)
    .build()?;

let batches: Vec<RecordBatch> = scan
    .to_arrow()
    .await?
    .try_collect()
    .await?;
```

The scan respects column projection — it only reads the projected columns from Parquet files, which is a significant I/O optimization. It applies the predicate at the manifest level to skip entire data files. This is the bread and butter of Iceberg, and iceberg-rust implements it correctly.

Schema conversion. The `schema_to_arrow_schema` function converts Iceberg schemas to Arrow schemas reliably. This sounds trivial, but Iceberg has types that do not map one-to-one to Arrow (fixed-precision decimals, UUIDs, timestamps with and without timezone), and iceberg-rust handles the edge cases.

REST catalog client. The `RestCatalog` implements the full Iceberg REST protocol — namespace listing, table loading, table creation, table commits. It handles the OAuth2 token flow for catalog authentication and credential vending for storage access.

What we had to work around

The async-sync boundary. DataFusion's `CatalogProvider` trait has synchronous methods — `schema_names()` returns `Vec<String>`, not `Future<Vec<String>>`. But iceberg-rust's catalog operations are all async. You cannot call an async function from a sync context without a runtime handle.

The workaround is `tokio::task::block_in_place`:

```
fn table_names(&self) -> Vec<String> {
    let handle = tokio::runtime::Handle::try_current()?;
    let catalog = self.session_catalog.clone();
    let ns = NamespaceIdent::new(self.namespace.clone());

    tokio::task::block_in_place(|| {
        handle.block_on(catalog.list_tables(&ns))
    })
}
```

This is not elegant. `block_in_place` tells tokio "I am about to block this thread, please move other tasks elsewhere." It works, but it consumes a thread from the runtime pool. For namespace listing, which happens once per session initialization, the cost is acceptable. For table loading, which happens per query, we use the async `table()` method on `SchemaProvider` instead.

We cached namespace lists at catalog provider construction time to avoid hitting this boundary repeatedly. The `SqeCatalogProvider` fetches all namespace names in its async `try_new()` method and stores them in a `Vec<String>`. The sync `schema_names()` method then returns a clone.

Credential vending differences between Polaris versions. When the engine loads a table from Polaris, the REST response includes a `config` section that may contain vended S3 credentials — temporary access keys scoped to that specific table. Polaris 0.9 and Polaris 1.0 return these credentials with different property keys and different expiry formats.

Field report: The day we discovered that Polaris returns different `file-io` properties depending on whether you're running Polaris 0.9 or 1.0. The fix was three lines. The debugging was three days. The credential extraction code now tries RFC3339 timestamps first, then epoch milliseconds, then gives up and falls back to static credentials. Defensive parsing for config that should be standardized but isn't.

The credential vending module ended up as a 150-line cache backed by moka, with a TTL set conser-

vatively to 50 minutes against a typical 60-minute STS credential lifetime. When vended credentials expire or are absent, the system falls back to static S3 credentials from the engine's configuration. The fallback is critical for development environments where Polaris runs in-memory without credential vending.

The CatalogBuilder API change from 0.8 to 0.9. In iceberg-rust 0.8, you constructed a `RestCatalog` with a `RestCatalogConfig`. In 0.9, this changed to a builder pattern with `RestCatalogBuilder::default().load(name, props)`. The migration was mechanical but required touching every test and every catalog construction site.

```
// iceberg-rust 0.9 pattern
let catalog = RestCatalogBuilder::default()
    .with_storage_factory(Arc::new(OpenDalStorageFactory::S3 {
        configured_scheme: "s3".to_string(),
        customized_credential_load: None,
    }))
    .load(
        format!("sqe-session-{}", &token_fingerprint),
        props,
    )
    .await?;
```

The `with_storage_factory` call is required for write operations — without it, `CREATE TABLE` and `INSERT INTO` fail because iceberg-rust does not know how to write to S3. This was not documented at the time. We found it by reading the iceberg-rust source.

Dead end: iceberg-datafusion's built-in IcebergTableProvider. The `iceberg-datafusion` crate provides its own `IcebergTableProvider` that bridges Iceberg tables to DataFusion. We tried using it directly. The problem: it constructs its own catalog session internally, so there is no way to inject the user's bearer token. Every query would run as the same catalog identity. We wrote our own `SqeTableProvider` instead, wrapping the per-user `Table` object with its already-configured credentials.

What we built ourselves

The gap between "iceberg-rust can read Iceberg tables" and "DataFusion can query Iceberg tables as a user" required three custom components: a table provider, a catalog provider, and a scan executor. Together, they form the bridge.

Building the Bridge

SqeTableProvider: The Table as DataFusion Sees It

DataFusion does not know what Iceberg is. DataFusion knows what a `TableProvider` is — a trait with four key methods:

```
trait TableProvider {
    fn schema(&self) -> SchemaRef;
```

```
    fn table_type(&self) -> TableType;
    fn supports_filters_pushdown(&self, filters: &[&Expr])
        -> Result<Vec<TableProviderFilterPushDown>>;
    async fn scan(
        &self,
        state: &dyn Session,
        projection: Option<&Vec<usize>>,
        filters: &[Expr],
        limit: Option<usize>,
    ) -> Result<Arc<dyn ExecutionPlan>>;
}
```

`schema()` returns the Arrow schema. `table_type()` says whether it is a base table or a view. `supports_filters_pushdown()` tells the optimizer which filter expressions the table provider can handle natively. `scan()` returns an execution plan that will produce record batches when executed.

Our `SqeTableProvider` wraps an Iceberg `Table` object. Construction converts the Iceberg schema to Arrow:

```
pub async fn try_new(table: Table) -> Result<Self> {
    let schema = schema_to_arrow_schema(
        table.metadata().current_schema()
    )?;
    Ok(Self { table, schema: Arc::new(schema) })
}
```

The critical method is `supports_filters_pushdown`. We return `Inexact` for every filter we can convert to an Iceberg predicate, and `Unsupported` for the rest. `Inexact` means "I will apply this filter during the scan, but I might not filter every row — DataFusion must still evaluate the filter after scanning." This is correct because Iceberg predicate pushdown prunes manifests and Parquet row groups, but it does not guarantee per-row filtering for all expression types.

DataFusion deep dive: The `Inexact` vs `Exact` distinction matters. If you return `Exact`, DataFusion removes the filter from the plan — it trusts that the table provider handled it completely. If you return `Inexact`, DataFusion keeps the filter as a post-scan filter node. Getting this wrong means silent data corruption: returning `Exact` when your pushdown only does file-level pruning would skip rows that should have been filtered.

Predicate Translation: DataFusion Expressions to Iceberg Predicates

DataFusion represents filter conditions as `Expr` — an enum with variants for binary expressions, boolean logic, IN lists, IS NULL, LIKE patterns, and more. Iceberg represents filter conditions as `Predicate` — a different enum with a different structure. The `expr_to_predicate` module translates between them.

The translation handles comparisons (`=`, `!=`, `<`, `>`, `<=`, `>=`), null checks, boolean logic (`AND`, `OR`, `NOT`), IN lists, and prefix LIKE patterns (`col LIKE 'foo%'` becomes `col STARTS WITH 'foo'`). Date casts are deliberately not pushed down because the cast truncates the value, and pushing the truncated value would change the filter semantics.

The interesting design decision is in `AND` vs `OR` handling. For an `AND` predicate where only one side can be converted, we push down the convertible side and let DataFusion handle the rest. This is safe because `Inexact` pushdown means DataFusion will re-evaluate the full filter post-scan. For an `OR` predicate, both sides must convert or we push nothing — because pushing only one side of an OR would widen the result set.

```
// AND: partial pushdown is safe (Inexact means DataFusion re-evaluates)
fn to_iceberg_and_predicate(left: TransformedResult, right: TransformedResult)
    -> TransformedResult
{
    match (left, right) {
        (Predicate(l), Predicate(r)) => Predicate(l.and(r)),
        (Predicate(l), _) => Predicate(l),  // push down what we can
        (_, Predicate(r)) => Predicate(r),
        _ => NotTransformed,
    }
}

// OR: both sides must convert, or we drop the whole thing
fn to_iceberg_or_predicate(left: TransformedResult, right: TransformedResult)
    -> TransformedResult
{
    match (left, right) {
        (Predicate(l), Predicate(r)) => Predicate(l.or(r)),
        _ => NotTransformed,  // cannot safely push partial OR
    }
}
```

This asymmetry is subtle. It is also the kind of thing that causes silent correctness bugs if you get it wrong. We wrote 25 unit tests for predicate translation — not because we doubted the logic, but because anyone maintaining this code a year from now needs to know what was tested and what was not.

IcebergScanExec: The Physical Plan

When DataFusion calls `scan()` on our table provider, we return an `IcebergScanExec` — a custom `ExecutionPlan` node that will actually read data from Iceberg tables via S3.

The execution model has a subtlety. DataFusion's `ExecutionPlan::execute()` is synchronous — it must return a `SendableRecordBatchStream` without awaiting. But Iceberg's scan is inherently async — it needs to read manifests from S3, plan files, and open Parquet readers. The solution is a lazily-initialized stream:

```
fn execute(&self, partition: usize, _context: Arc<TaskContext>)
    -> Result<SendableRecordBatchStream>
{
    let table = self.table.clone();
    let projection = self.projection.clone();
```

```
    let predicates = self.predicates.clone();

    let stream = futures::stream::once(async move {
        let mut scan_builder = table.scan();
        if let Some(ref cols) = projection {
            scan_builder = scan_builder.select(cols.iter().map(|s| s.as_str()));
        }
        if let Some(pred) = predicates {
            scan_builder = scan_builder.with_filter(pred);
        }
        let scan = scan_builder.build()?;
        let arrow_stream = scan.to_arrow().await?;
        Ok(arrow_stream.map_err(|e| DataFusionError::External(Box::new(e))))
    })
    .try_flatten();

    Ok(Box::pin(IcebergRecordBatchStream {
        schema,
        inner: Box::pin(stream),
        baseline,
    }))
}
```

The `stream::once().try_flatten()` pattern defers the async work to the first poll. When DataFusion pulls the first batch from the stream, the scan initializes, reads manifests, opens Parquet files, and begins streaming record batches. Subsequent polls return additional batches without re-initialization.

The `IcebergRecordBatchStream` wrapper records execution metrics — elapsed time and output row counts — so that `EXPLAIN ANALYZE` can report scan performance. This is not optional instrumentation. Without it, diagnosing slow queries is guesswork.

SqeCatalogProvider: Mapping Namespaces to Schemas

DataFusion's catalog hierarchy is `Catalog > Schema > Table`. Iceberg's hierarchy is `Catalog > Namespace > Table`. The mapping is direct — an Iceberg namespace becomes a DataFusion schema.

The `SqeCatalogProvider` fetches namespace names at construction time and caches them. When DataFusion asks for a schema by name, it creates a `SqeSchemaProvider` for that namespace. The schema provider in turn creates `SqeTableProvider` instances on demand when DataFusion asks for a table.

The chain of construction follows the user's credentials:

```
User authenticates -> SessionCatalog (with bearer token)
    -> SqeCatalogProvider (cached namespace list)
        -> SqeSchemaProvider (lazy table loading)
            -> SqeTableProvider (Iceberg table with vended S3 credentials)
                -> IcebergScanExec (reads Parquet from S3)
```

Every link in this chain carries the user's identity. The `SessionCatalog` holds the bearer token. When it loads a table, Polaris validates the token and returns metadata only if the user has access. The vended S3 credentials in the table response are scoped to that user's permissions. The scan reads Parquet files using those scoped credentials. At no point does the engine use a service account.

Sovereignty principle: The catalog-to-scan credential chain is unbroken. The user's bearer token flows from authentication through catalog resolution to storage I/O. If the user does not have access to a table in Polaris, they cannot see it in namespace listings. If they do not have read access to the underlying S3 path, the scan fails with a storage error, not a data leak. Authorization is not our code — it is Polaris and S3 doing their jobs.

The SessionCatalog: One Catalog Per User

The `SessionCatalog` is the fulcrum. It wraps iceberg-rust's `RestCatalog` in an `RwLock` and configures it with the user's bearer token:

```
pub async fn new(
    polaris_url: &str,
    warehouse: &str,
    bearer_token: &str,
    storage_config: &StorageConfig,
) -> Result<Self> {
    let mut props = HashMap::new();
    props.insert("token".to_string(), bearer_token.to_string());
    props.insert("uri".to_string(), polaris_url.to_string());
    props.insert("warehouse".to_string(), warehouse.to_string());

    let catalog = RestCatalogBuilder::default()
        .with_storage_factory(Arc::new(OpenDalStorageFactory::S3 { ... }))
        .load(format!("sqe-session-{}", &token_fingerprint), props)
        .await?;
    // ...
}
```

The session name includes a fingerprint of the token — the last 8 characters. This is deliberate. iceberg-rust's `RestCatalog` caches certain responses internally. If a token is refreshed (the user's session gets a new JWT), the cached responses from the old token must be invalidated. Using a different session name for each token ensures that a refreshed token gets a fresh catalog session with no stale cache entries.

We also built a `SessionCatalogBridge` — a thin wrapper that implements iceberg-rust's `Catalog` trait by delegating to our `SessionCatalog`. This bridge exists for one reason: the `iceberg-datafusion` crate expects a `Catalog` trait object, and our `SessionCatalog` wraps the `RestCatalog` behind an `RwLock` rather than implementing `Catalog` directly. The bridge is boilerplate — every method reads the lock and delegates — but it is necessary for interoperability with the broader iceberg-rust ecosystem.

Why Rust Changes the Game

PyIceberg taught us what Iceberg is. iceberg-rust taught us what Iceberg can be when the language does not get in the way.

The differences are not about syntax. They are about what happens at scale.

Memory. Python's Iceberg implementation reads Parquet files into PyArrow arrays, which then get copied or wrapped when passed to Pandas. In Rust, iceberg-rust produces Arrow `RecordBatch` values that DataFusion consumes directly — zero copy. The data is read from S3 into kernel buffers, deserialized from Parquet into Arrow format, and consumed by the query engine without a single extra allocation for format conversion.

Concurrency. PyIceberg's scan planning is single-threaded at the Python level, though PyArrow's Parquet reader releases the GIL during I/O. iceberg-rust's scan produces an async stream of Arrow batches. DataFusion pulls from this stream on its executor threads. When you have multiple concurrent queries — which you always do in a multi-user engine — Rust's async runtime interleaves I/O waits naturally. Python can release the GIL for I/O, but the scan planning, metadata parsing, and predicate evaluation stay on one thread. For true parallelism across concurrent queries, you would need multiprocessing, and multiprocessing means serialising data across process boundaries.

Predicate pushdown completeness. PyIceberg pushes predicates down to manifest filtering. So does iceberg-rust. But iceberg-rust integrates with DataFusion's filter pushdown protocol, which means the optimizer can reason about pushdown across the entire query plan. A join with a filter can push the filter down through the join and into the Iceberg scan — something that requires explicit coding in Python but happens automatically through DataFusion's optimizer rules.

Type safety. The predicate translation module converts DataFusion expressions to Iceberg predicates. In Python, this would be a series of isinstance checks with silent fallback to no pushdown. In Rust, the match is exhaustive — the compiler tells you when you have missed a variant. When iceberg-rust added the `STARTS_WITH` predicate operator, any code that matched on `PredicateOperator` variants would have failed to compile until updated. In Python, the new operator would have been silently ignored.

None of these differences matter for a single query on a laptop. All of them matter for a hundred concurrent queries on a production cluster.

The View Problem

Iceberg v2 and v3 support views — SQL definitions stored in the catalog alongside tables. A view is metadata: a SQL string, a schema, and a reference to the namespace where the view's tables live.

iceberg-rust 0.9 does not have first-class view support. The `Catalog` trait has no `load_view` or `create_view` method. We implemented views by talking to the Polaris REST API directly, bypassing iceberg-rust entirely.

The `SessionCatalog` makes HTTP calls to the Polaris views endpoints:

- `POST /v1/{prefix}/namespaces/{ns}/views` to create a view

- `GET /v1/{prefix}/namespaces/{ns}/views` to list views
- `GET /v1/{prefix}/namespaces/{ns}/views/{name}` to load a view's SQL
- `DELETE /v1/{prefix}/namespaces/{ns}/views/{name}` to drop a view

When the `SqeSchemaProvider` resolves a table name, it first tries to load it as an Iceberg table through iceberg-rust. If that fails, it tries to load it as a view through the direct REST call. If it finds a view, it takes the SQL string, plans it through a mini DataFusion `SessionContext` with the same catalog registered, and returns the resulting `LogicalPlan` wrapped in a `ViewTable`.

This is a workaround, not a solution. When iceberg-rust adds native view support, we will migrate to it. But the workaround is complete — views work for SELECT, for joins, for nested views — and it ships.

Dead end: waiting for iceberg-rust view support. We considered waiting. The iceberg-rust project had view support on its roadmap. But "on the roadmap" and "in a released version" are different things. We needed views for dbt compatibility. We built the REST workaround in a day. The cost of carrying it is low — it is isolated in `SessionCatalog` and touched nowhere else.

The Scan Lifecycle

When a user runs `SELECT order_id, total FROM orders WHERE order_date > '2024-01-01'`, the path from SQL to bytes is:

1. **Parse.** DataFusion parses the SQL into an AST.
2. **Plan.** The planner resolves `orders` by asking the `SqeCatalogProvider` for the default schema, then asking the `SqeSchemaProvider` for the table. The schema provider calls `SessionCatalog::load_table`, which makes a REST call to Polaris with the user's bearer token. Polaris returns the table metadata and vended S3 credentials. `SqeTableProvider::try_new` converts the Iceberg schema to Arrow.
3. **Optimize.** DataFusion's optimizer pushes the `order_date > '2024-01-01'` filter down toward the table scan. It calls `supports_filters_pushdown` and learns that this filter is pushdown-capable (`Inexact`). The optimizer also pushes the column projection — only `order_id` and `total` need to be read.
4. **Execute.** DataFusion calls `scan()` on the table provider with the projection `[0, 2]` (the column indices) and the filter expression. The provider creates an `IcebergScanExec` with the converted Iceberg predicate and the projected column names.
5. **Scan.** On first poll, the `IcebergScanExec` builds an Iceberg scan with column selection and predicate filter. The scan reads the manifest list, applies the predicate to manifest-level statistics to skip irrelevant manifests, then reads the remaining manifests and applies column-level statistics to skip irrelevant data files.
6. **Read.** For each surviving data file, the scan opens the Parquet file using the vended S3 credentials, reads only the projected columns, and returns Arrow `RecordBatch` values.
7. **Post-filter.** DataFusion applies the filter expression again on the returned batches (because the pushdown was `Inexact`), ensuring row-level correctness.
8. **Return.** The filtered, projected batches are streamed to the client via Arrow Flight.

The user sees a result set. The engine opened exactly the Parquet files it needed, read exactly the columns it needed, and applied the filter at every level where it could — manifest, file, row group, and row.

Tables That Travel

A traditional table is bound to its database. Move the database, you move the table. Lose the database, you lose the table. Query it from another engine, you need a bridge, an export, or a compatible wire protocol.

An Iceberg table is a metadata pointer to files on storage. The metadata is JSON and Avro. The data is Parquet. Any engine that reads these formats can read the table. Any catalog that implements the Iceberg REST protocol can serve the metadata. The table does not belong to the engine. The table does not belong to the catalog. The table belongs to whoever has the files.

This is what makes a sovereign query engine possible. SQE does not own your tables. Polaris does not own your tables. If you stop running SQE tomorrow, your tables are still there — on S3, in Parquet, described by Iceberg metadata that any Iceberg-compatible engine can read. DuckDB can read them. Spark can read them. Trino can read them. A PyIceberg script can read them.

The engine is disposable. The data is not.

iceberg-rust made this real in Rust: a scan planner that prunes intelligently, a schema converter that handles the edge cases, and a REST catalog client that passes through user credentials. We built the bridge — table provider, catalog provider, scan executor — and DataFusion became an Iceberg query engine.

The bridge was about 1,200 lines of Rust across six files. That is all it takes to connect a query engine to the entire Iceberg ecosystem. The ratio matters. We wrote the initial catalog integration in about 1,200 lines. We got access to every table, in every namespace, in every catalog that speaks the Iceberg REST protocol.

AI Logbook: The AI implemented `SqeTableProvider`, `SqeCatalogProvider`, `IcebergScanExec`, and the predicate translation module — including the subtle AND-vs-OR partial pushdown asymmetry — from a design doc that described the DataFusion trait interfaces. The human specified the `Inexact` vs `Exact` pushdown semantics and the security implication of getting them wrong. The `stream::once().try_flatten()` pattern for deferring async scan initialization inside a synchronous `execute()` method was the AI's solution; it worked on the first attempt.

The Engine You Already Have

Every query engine is a library pretending to be a service. DataFusion drops the pretence.

We needed a query engine. Not a toy. Not a prototype. A real SQL engine that could parse complex queries, optimize them, push predicates into Iceberg manifests, stream Arrow record batches back to clients, and do all of it as the authenticated user. The kind of thing that takes a team of twenty engineers three years to build from scratch.

We had three months and two people.

The obvious answers were obvious for a reason. Trino was what we were replacing – the auth model couldn't be fixed without forking the entire coordinator. Spark is a cluster computing framework that happens to support SQL, which is like using a forklift to move a chair. DuckDB is brilliant but single-process, single-node, and designed for a different problem.

Then someone pointed at DataFusion and said: "What if the engine already exists, and it's a library?"

That question changed the project. It went from "build a query engine" to "build the parts of a query engine that make it ours." The parser, the optimizer, the execution runtime, the Arrow memory model – someone already built those. What nobody had built was the authentication model, the catalog integration, and the policy enforcement that would make the engine sovereign.

The distinction between a library and a service turns out to be the most important architectural decision in the entire project. Not which language. Not which protocol. Whether you own the process or someone else does.

The Fifty-Line Query Engine

DataFusion is not a database. It is not a service. It is a Rust crate you add to `Cargo.toml`, and it gives you a complete SQL query engine that runs inside your process. No cluster. No daemon. No configuration server. No JVM. Just a function call.

Here is a query engine:

```
use datafusion::prelude::*;

#[tokio::main]
async fn main() -> Result<()> {
    let ctx = SessionContext::new();
```

```
    ctx.register_csv("users", "data/users.csv", CsvReadOptions::new()).await?;

    let df = ctx.sql("SELECT name, age FROM users WHERE age > 30").await?;
    df.show().await?;

    Ok(())
}
```

That is a working SQL engine. It parses the SQL. It builds a logical plan. It optimizes the plan (predicate pushdown, projection pruning, constant folding). It creates a physical execution plan. It executes that plan and streams Arrow record batches to your terminal. Twelve lines of code.

The `SessionContext` is the unit of execution. It holds the catalog (what tables exist), the configuration (how to optimize), and the runtime (where to execute). One `SessionContext`, one query environment. You can create as many as you want. They share nothing unless you explicitly connect them.

This is the property that made SQE possible. Every user query gets its own `SessionContext`, configured with that user's catalog view, that user's credentials, that user's policy constraints. There is no shared mutable state between users. There is no "connection pool" that accidentally leaks one user's permissions to another.

From SQL String to Arrow Batches

A SQL query goes through five stages in DataFusion. Understanding these stages is understanding why DataFusion is extensible in ways that monolithic engines are not.

Stage 1: Parsing. The SQL string is parsed by `sqlparser-rs` into an abstract syntax tree (AST). DataFusion uses the same SQL parser that dozens of other Rust projects use. It handles standard SQL well. Non-standard extensions (Trino's `SHOW CATALOGS`, our own `EXPLAIN FULL`) need to be handled before or after the parser.

Stage 2: Logical Planning. The AST is converted into a `LogicalPlan` – a tree of relational algebra operators. A `SELECT name FROM users WHERE age > 30` becomes a `Projection` over a `Filter` over a `TableScan`. The logical plan describes *what* data to compute, not *how* to compute it.

Stage 3: Optimization. DataFusion's optimizer runs a series of rewrite rules over the logical plan. Predicates get pushed down toward the table scans. Projections get pruned to eliminate unused columns. Common subexpressions get eliminated. The optimizer is a pipeline of `OptimizerRule` implementations, and you can add your own rules or disable built-in ones.

This is where security enforcement happens in SQE. Before the optimizer runs, we inject policy filters into the logical plan – row filters above the table scan, column masks that replace sensitive expressions. The optimizer then pushes the user's predicates through our security filters, but it cannot push them past masked columns. More on this in Chapter 8.

Stage 4: Physical Planning. The optimized logical plan is converted into a `PhysicalPlan` – a tree of `ExecutionPlan` nodes that describe the actual computation. A logical `Filter` becomes a `FilterExec`. A logical `TableScan` becomes whatever the `TableProvider` returns from its `scan()` method. In SQE,

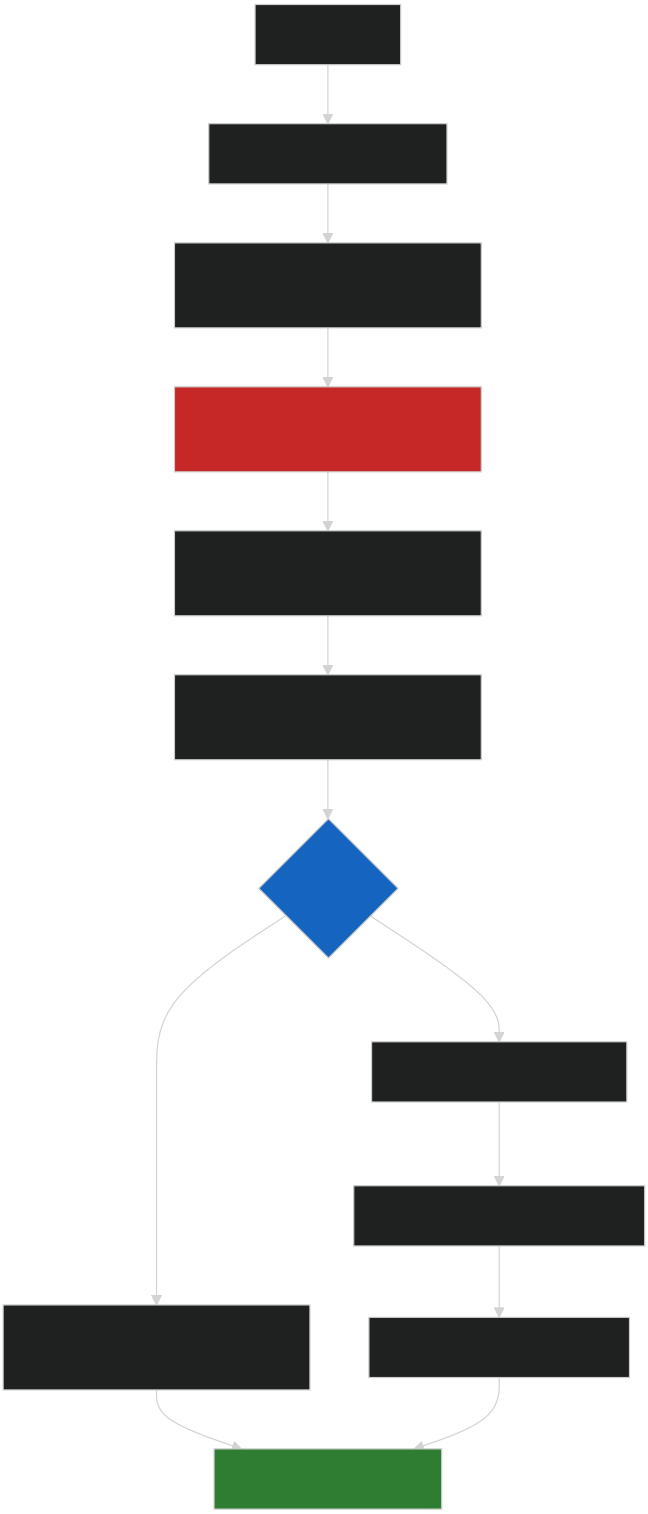

Figure 2: DataFusion query pipeline: SQL string through parsing, logical planning, optimization, physical planning, and execution to Arrow batches

that is an `IcebergScanExec`.

Stage 5: Execution. The physical plan is executed using a pull-based model descended from the Volcano iterator model (Graefe, 1994), but adapted for vectorised execution. Where classic Volcano pulls one row at a time, DataFusion pulls one `RecordBatch` at a time — closer to the morsel-driven parallelism model (Leis et al., 2014) than to pure Volcano. Each `ExecutionPlan` node implements an `execute()` method that returns a stream of Arrow `RecordBatch` values. The top-level node pulls from its children, which pull from their children, all the way down to the leaf scan nodes. Data flows upward as a lazy stream – nothing is computed until someone reads from the output.

A caveat: "pull-based" describes the pipeline, not every operator. Some operators materialise intermediate state by necessity. A `HashJoinExec` eagerly builds the hash table from its build side before streaming the probe side. An `AggregateExec` accumulates state across all input batches before producing output. These are pipeline breakers – they consume their entire input before producing any output. The pull model means the pipeline *between* these breakers is lazy, and data streams through non-blocking operators (Filter, Projection) without buffering.

The pull model still matters for memory. If the client only reads the first 100 rows, the scan might never read the remaining files. Memory usage is proportional to the pipeline width plus the materialised state of any pipeline breakers – not the total data volume.

Here is what that looks like in SQE's `execute_query` method:

```
async fn execute_query(
    &self,
    session: &Session,
    sql: &str,
    query_id: &uuid::Uuid,
) -> sqe_core::Result<Vec<RecordBatch>> {
    let ctx = self.create_session_context(session).await?;

    // Stage 1+2: Parse SQL and build a logical plan
    let df = ctx
        .sql(sql)
        .await
        .map_err(|e| SqeError::Execution(format!("SQL planning failed: {e}")))?;

    // Security: inject policy filters before optimization
    let plan = df.logical_plan().clone();
    let enforced_plan = self
        .policy_enforcer
        .evaluate(&session.user, plan)
        .await?;

    // Stage 3: Optimize the policy-enforced logical plan
    let enforced_df = ctx
        .execute_logical_plan(enforced_plan)
        .await
```

```
        .map_err(|e| SqeError::Execution(format!("Failed to create execution plan: {e}")))?;

    // Stage 4: Create the physical execution plan
    let physical_plan = enforced_df
        .create_physical_plan()
        .await
        .map_err(|e| SqeError::Execution(format!("Physical plan creation failed: {e}")))?;

    // Try to distribute scan work across workers
    let final_plan = self.try_distribute(physical_plan, session, query_id).await;

    // Stage 5: Execute and collect Arrow batches
    let batches = collect(final_plan, ctx.task_ctx())
        .await
        .map_err(|e| SqeError::Execution(format!("Query execution failed: {e}")))?;

    Ok(batches)
}
```

That is the entire query pipeline. SQL string in, Arrow record batches out. Between those two points, DataFusion handles parsing, logical planning, optimization, physical planning, and execution. SQE inserts itself at two points: policy enforcement (between logical planning and optimization) and distribution (between physical planning and execution). Everything else is DataFusion.

DataFusion deep dive: DataFusion 52 ships with over 100 optimizer rules, including predicate pushdown, projection pushdown, common subexpression elimination, constant folding, filter rewriting, join reordering, and limit pushdown. Each rule implements `OptimizerRule` and returns a rewritten `LogicalPlan`. SQE's policy enforcement is implemented as a plan rewrite that runs *before* these rules – this is intentional. The optimizer can then push user predicates through our security filters, which improves performance while maintaining the security invariant.

The SessionContext Is the Product

In a monolithic database, everything is coupled. The catalog is bound to the storage layer. The storage layer is bound to the compute engine. The compute engine is bound to the cluster manager. You can't change one without changing all of them.

DataFusion separates these concerns into traits that you implement.

In SQE, the `create_session_context` method builds a fresh `SessionContext` for each query, configured with the user's credentials:

```
async fn create_session_context(
    &self,
    session: &Session,
) -> sqe_core::Result<SessionContext> {
    let ctx = SessionContext::new_with_config(
        SessionConfig::new()
```

```
            .with_information_schema(true)
            .with_default_catalog_and_schema(&catalog_name, "default"),
    );

    // Create a per-session catalog connected to Polaris
    // with the user's bearer token
    let session_catalog = Arc::new(
        SessionCatalog::new(
            &self.config.catalog.polaris_url,
            &self.config.catalog.warehouse,
            &session.access_token,
            &self.config.storage,
        )
        .await?,
    );

    let catalog_provider = SqeCatalogProvider::try_new(
        session_catalog,
        self.config.storage.clone(),
        self.config.catalog.warehouse.clone(),
    )
    .await?;

    ctx.register_catalog(&catalog_name, Arc::new(catalog_provider));

    Ok(ctx)
}
```

Every query gets its own `SessionContext`. Every `SessionContext` gets its own `SessionCatalog`, initialized with the user's bearer token. That token flows through to Polaris for catalog operations and to S3 for data access. There is no shared state between users. There is no privilege escalation path.

This is what "library, not service" means in practice. A service gives you its `SessionContext` with its catalog, its credentials, its policies. A library lets you construct the `SessionContext` yourself, with whatever catalog, credentials, and policies you need.

The Catalog Hierarchy

DataFusion organizes data through a three-level hierarchy: `CatalogProvider` contains `SchemaProvider` instances, which contain `TableProvider` instances. This maps cleanly to Iceberg's organization: a Polaris catalog contains namespaces, which contain tables.

SQE implements all three levels.

The `SqeCatalogProvider` bridges DataFusion's catalog concept to Iceberg namespaces:

```
impl CatalogProvider for SqeCatalogProvider {
    fn schema_names(&self) -> Vec<String> {
```

```
        let mut names = self.cached_namespaces.clone();
        names.push("information_schema".to_string());
        names
    }

    fn schema(&self, name: &str) -> Option<Arc<dyn SchemaProvider>> {
        if name == "information_schema" {
            return Some(Arc::new(
                InformationSchemaProvider::new(
                    self.session_catalog.clone(),
                    self.warehouse.clone(),
                ),
            ));
        }

        if !self.cached_namespaces.contains(&name.to_string()) {
            return None;
        }

        Some(Arc::new(SqeSchemaProvider::new(
            self.session_catalog.clone(),
            name.to_string(),
            self.storage_config.clone(),
            self.warehouse.clone(),
        )))
    }
}
```

The `SqeSchemaProvider` lists tables within a namespace by calling the Iceberg REST catalog:

```
#[async_trait]
impl SchemaProvider for SqeSchemaProvider {
    fn table_names(&self) -> Vec<String> {
        let handle = tokio::runtime::Handle::try_current().ok()?;
        let ns_ident = NamespaceIdent::new(self.namespace.clone());

        // NOTE: DataFusion's SchemaProvider::table_names() is synchronous by design
        // (returns Vec<String>, not a Future). Since our catalog is async, we use
        // block_in_place to bridge the gap. This is a known DataFusion limitation --
        // calling block_on inside an async runtime can deadlock under certain executor
        // configurations. We use block_in_place to yield the current thread first.
        let tables = tokio::task::block_in_place(||
            handle.block_on(self.session_catalog.list_tables(&ns_ident))
        );
        // ... collect table names and view names
    }

    async fn table(&self, name: &str) -> DFResult<Option<Arc<dyn TableProvider>>> {
```

```
        // Try as Iceberg table first, then as Iceberg view
        match self.session_catalog.load_table(&table_ident).await {
            Ok(table) => {
                Ok(Some(Arc::new(SqeTableProvider::try_new(table).await?)))
            }
            Err(_) => {
                // Try loading as a view...
            }
        }
    }
}
```

And the `SqeTableProvider` wraps an Iceberg table so DataFusion can scan it:

```
#[async_trait]
impl TableProvider for SqeTableProvider {
    fn schema(&self) -> ArrowSchemaRef {
        self.schema.clone()
    }

    async fn scan(
        &self,
        _state: &dyn Session,
        projection: Option<&Vec<usize>>,
        filters: &[Expr],
        _limit: Option<usize>,
    ) -> DFResult<Arc<dyn ExecutionPlan>> {
        let projected_columns = projection.map(|indices| {
            indices.iter()
                .map(|&i| self.schema.field(i).name().clone())
                .collect::<Vec<_>>()
        });

        let predicates = expr_to_predicate::convert_filters_to_predicate(filters);

        Ok(Arc::new(IcebergScanExec::new(
            self.table.clone(),
            projected_schema,
            projected_columns,
            predicates,
        )))
    }
}
```

The `scan()` method is where DataFusion meets Iceberg. DataFusion passes down the column projection (only read the columns the query needs) and the filter expressions (only scan the files that might contain matching rows). SQE converts the DataFusion filter expressions into Iceberg predicates that prune manifest files and Parquet row groups. The actual data reading happens in `IcebergScanExec`,

which uses iceberg-rust's scan API to read Parquet files from S3 using the user's vended credentials.

Each layer does exactly one thing. The catalog knows what exists. The schema knows what's in a namespace. The table knows how to scan data. DataFusion orchestrates them. SQE provides the implementations. The user's identity flows through all of it.

DataFusion deep dive: The `CatalogProvider -> SchemaProvider -> TableProvider` hierarchy is DataFusion's primary extension point for data sources. Implementing these three traits is how you teach DataFusion to read from anything – Iceberg, Delta Lake, Hudi, a REST API, a flat file on disk. The `TableProvider::scan()` method returns an `ExecutionPlan`, which means you control exactly how data is read. SQE's `IcebergScanExec` uses iceberg-rust's scan builder to apply column projection, predicate pushdown, and manifest pruning before reading a single byte from S3.

The IcebergScanExec: Where Compute Meets Storage

The leaf node of every query plan in SQE is an `IcebergScanExec`. This is the `ExecutionPlan` that actually reads data from Iceberg tables via S3. It implements DataFusion's `ExecutionPlan` trait, which means DataFusion treats it like any other execution node – it just happens to read from Iceberg instead of CSV or Parquet files on local disk.

```
impl ExecutionPlan for IcebergScanExec {
    fn execute(
        &self,
        partition: usize,
        _context: Arc<TaskContext>,
    ) -> DFResult<SendableRecordBatchStream> {
        let table = self.table.clone();
        let projection = self.projection.clone();
        let predicates = self.predicates.clone();

        let stream = futures::stream::once(async move {
            let mut scan_builder = table.scan();

            if let Some(ref cols) = projection {
                scan_builder = scan_builder.select(cols.iter().map(|s| s.as_str()));
            }

            if let Some(pred) = predicates {
                scan_builder = scan_builder.with_filter(pred);
            }

            let scan = scan_builder.build()?;
            let arrow_stream = scan.to_arrow().await?;

            Ok(arrow_stream.map_err(|e| DataFusionError::External(Box::new(e))))
        })
        .try_flatten();
```

```
        Ok(Box::pin(IcebergRecordBatchStream {
            schema,
            inner: Box::pin(stream),
            baseline,
        }))
    }
}
```

The `execute()` method is synchronous (DataFusion's trait requires it), but it returns an async stream. The actual scan is lazy – iceberg-rust's `to_arrow()` doesn't fetch data until the stream is polled. This means DataFusion's pull-based execution model works naturally: data flows only when the upstream operator asks for the next batch.

The `table.scan()` call uses the `FileIO` instance that was configured with the user's vended S3 credentials when the table was loaded from the catalog. Every byte read from S3 is attributed to the user who initiated the query. This is the bearer passthrough architecture from Chapter 4 working at the storage layer.

Note the `BaselineMetrics` field. Every `IcebergScanExec` tracks wall-clock time and output row counts. This feeds into `EXPLAIN ANALYZE` output, so you can see exactly how long each scan took and how many rows it produced. Observability is not bolted on. It is part of the execution plan interface.

This is also where distributed execution hooks in. In Chapter 13, we replace the local `IcebergScanExec` with a `DistributedScanExec` that sends the scan work to remote workers. The rest of the plan – the filters, projections, aggregations above the scan – stays on the coordinator. The replacement is a tree transformation on the physical plan. DataFusion doesn't know or care that the leaf nodes are now remote. It pulls from them the same way it pulls from local scans.

Why Rust

The choice of Rust was not ideological. It was practical.

Arrow is native Rust. Apache Arrow's canonical implementation is in Rust (`arrow-rs`). DataFusion is written in Rust. iceberg-rust is written in Rust. The entire stack from SQL parsing to S3 byte reading is Rust, which means zero serialization boundaries, zero JNI bridges, zero FFI overhead. A `RecordBatch` returned by iceberg-rust is the same `RecordBatch` that DataFusion processes, in the same memory layout, with zero copies.

`Send + Sync` gives you parallelism for free. Rust's type system enforces thread safety at compile time. If your type is `Send + Sync`, it can be shared across threads without data races. DataFusion's `ExecutionPlan` trait requires `Send + Sync`, which means every execution node is safe to execute in parallel. You don't write synchronization code. You don't debug race conditions. The compiler rejects code that would have race conditions.

Ownership prevents resource leaks. When a `SessionContext` is dropped, every resource it holds is dropped. The catalog connection closes. The cached metadata is freed. The S3 credentials are

zeroed. There is no garbage collector that might hold onto credentials longer than expected. There is no finalizer that might not run.

The binary is self-contained. SQE compiles to a single static binary. No JVM to install. No Python interpreter to manage. No shared libraries to version. The Docker image is 47MB. The binary starts in under a second. Deployment is copying a file.

AI agents write Rust well. This is an observation, not a hypothesis. Rust code is dense and expressive – a function signature often tells you what the function does, what it takes, what it returns, and what can go wrong. The borrow checker catches the mistakes that would be runtime bugs in other languages. When an AI generates Rust code that compiles, it is very likely correct. When it generates code that doesn't compile, the error messages are specific enough to guide the fix. We built SQE in 15 days. The AI wrote most of the implementation. The human made the architectural decisions and reviewed every commit. That ratio – AI implements, human decides – works because Rust's type system catches the implementation bugs that a human reviewer would miss.

Antipattern: choosing Rust for the wrong reasons. Rust is not the right choice because it's fast. It's fast, but that's a side effect. Rust is the right choice for SQE because the Arrow ecosystem is Rust-native, because the type system prevents the class of bugs that query engines are most prone to (data races, use-after-free, resource leaks), and because the deployment story (single binary, small container) matches the sovereignty requirement. If your query engine is a Python wrapper around DuckDB, Rust adds nothing. Pick the language that matches your constraints.

The Compile-Time Tax

Rust is not free. The cost is compile time.

A clean build of SQE – all 12 crates, all dependencies – takes minutes. Not seconds. Minutes. DataFusion alone pulls in hundreds of transitive dependencies. Add Arrow, iceberg-rust, tonic (for gRPC), and tokio, and the dependency graph is enormous.

Incremental builds save you most of the time. Change a line in `sqe-coordinator`, and only `sqe-coordinator` and its reverse dependencies recompile. That's usually 15-30 seconds. Tolerable.

But certain changes blow up the incremental cache. A macro change in `sqe-core` forces a full rebuild of everything that depends on `sqe-core` – which is every crate. A dependency version bump in `Cargo.toml` can trigger a cascade. And CI builds from a clean state every time (unless you configure caching carefully), so your pipeline always pays the full price.

We learned this the hard way. Our CI build went from 4 minutes to 18 minutes when we added the distributed execution crates. The fix wasn't faster hardware. It was architectural.

Here is what we did:

Feature-gated crates. The `sqe-bench` and `sqe-trino-compat` crates are optional. If you're working on the coordinator, you don't compile the benchmark suite. This shrank the default build graph significantly.

Dev profile tuning. In `Cargo.toml`:

```
[profile.dev]
debug = 1                        # Line-tables only, not full debug info
split-debuginfo = "unpacked"     # Skip dsymutil on macOS — saves 20-30% link time

[profile.release]
codegen-units = 4                # More parallelism during codegen
lto = "thin"                     # 80% of full LTO benefit, much faster
strip = true                     # Smaller binary
```

`cargo check` as the fast feedback loop. `cargo check` verifies that your code compiles without actually producing binaries. It runs in a fraction of the time. We run `cargo check` after every save and `cargo build` only when we need to run the engine.

sccache for CI. Shared compilation cache across CI runs. Changed builds from 18 minutes to 6 minutes for the common case.

The honest trade-off: you pay compile time upfront, and you save debugging time in production. A null pointer in C++ crashes at 3am. A race condition in Java shows up under load. In Rust, these bugs don't exist – the compiler rejected them at build time. The question is whether you'd rather wait 30 seconds for the compiler or spend four hours debugging a production incident. At scale, this cost compounds in both directions. Plan for it from day one.

Field report: Our CI build went from 4 minutes to 18 minutes when we added the distributed execution crates. The fix wasn't faster hardware – it was splitting the workspace into feature-gated crates so you only compile what you're changing. `profile.dev` tuning (line-tables-only debug info, unpacked debuginfo on macOS) saved another 20-30% of link time. Compile times are a design constraint, not just an annoyance.

The Competition: DuckDB, Spark, Trino

We evaluated three alternatives before committing to DataFusion. Each one taught us something about what we actually needed.

DuckDB

DuckDB is an extraordinary piece of engineering. It is the SQLite of analytics – an embeddable, single-process OLAP engine that is blisteringly fast on data that fits on one machine. If your problem is "run analytical queries on local Parquet files," DuckDB is the answer. Stop reading. Go use DuckDB.

I wrote about DuckDB with Iceberg in 2025 – querying S3 tables via the Iceberg REST API. It worked beautifully for single-user exploration. The query latency was lower than anything else I'd tested. The developer experience was outstanding.

dev.to connection: "DuckDB S3 Tables with Iceberg using Iceberg Rest API" (2025) – this article explored DuckDB's Iceberg extension for single-user analytics. The performance was excellent. The limitation was the one we couldn't work around: single-user, single-process.

Our problem was different. We needed multi-user isolation (one engine, many users, each seeing only

their data). We needed bearer token passthrough (every S3 access attributed to a specific human). We needed distributed execution across workers. And we needed to control the query plan for policy enforcement – injecting row filters and column masks before the optimizer runs.

DuckDB is not designed for any of these. It is a single-user, single-process engine. Its extension model allows custom table functions and file readers, but not user-extensible plan rewriting, not custom authentication, not distributed execution. DuckDB has a sophisticated internal optimizer that rewrites plans aggressively – but you cannot inject your own rewrite rules from outside. You can embed DuckDB in a multi-user service, but you end up building the multi-tenancy, auth, and distribution layers yourself – which is exactly what DataFusion gives you as traits to implement.

Dimension	DuckDB	DataFusion
Model	Embedded database	Query engine library
Multi-user	No (single-process)	Yes (SessionContext per user)
User-extensible plan rewriting	No (internal optimizer only)	Yes (LogicalPlan is a tree you can transform)
Distribution	No	Yes (PhysicalPlan nodes are serialisable)
Extension model	UDF, table functions	CatalogProvider, TableProvider, ExecutionPlan, OptimizerRule
Language	C++	Rust
Best for	Single-user analytics, embedded BI	Building custom query engines

Spark

Apache Spark is the Swiss army knife of data processing. It does batch. It does streaming. It does ML. It does graph processing. It also does SQL, through Spark SQL, which is a capable query engine with a Hive-compatible catalog and a distributed execution model.

The problem with Spark is that it is Spark. The JVM alone adds hundreds of megabytes to the container image. The driver-executor model requires cluster management (YARN, Kubernetes, or Spark Standalone). Configuration is an art form – `spark.executor.memory`, `spark.sql.shuffle.partitions`, `spark.dynamicAllocation.enabled` – and getting it wrong means either wasted resources or OOM errors.

More fundamentally, Spark's auth model is service-account-based. The Spark driver holds credentials, and all executors use those credentials. There is no per-user credential passthrough in the execution layer. You can audit at the application level (who submitted the job), but S3 sees one identity for everything.

We looked at IOMete, which runs Spark on Kubernetes with a cloud-independent model. It is well-executed. But it's still Spark – still JVM, still service-account auth, still the complexity of a distributed computing framework when what we wanted was a distributed query engine.

There's a subtler problem. Spark's Catalyst optimizer is powerful, but it's a closed system. You can

add custom rules, but the internals are tightly coupled. Modifying how Spark handles authentication at the storage layer means modifying Spark itself – the Hadoop credential provider chain, the FileSystem abstraction, the delegation token mechanism. These are not extension points. They are implementation details that happen to be accessible because the code is open source. Every Spark upgrade risks breaking your modifications.

DataFusion's approach is different. The extension points are the product. `CatalogProvider`, `TableProvider`, `ExecutionPlan` – these are stable traits with documented contracts. You implement them and DataFusion calls them. Upgrades don't break your implementations unless the trait itself changes, and trait changes are breaking changes that show up in the changelog.

Trino

Trino is what we were replacing. It is a distributed SQL query engine with a connector architecture, an optimizer, and a coordinator-worker model. It does SQL well. Its connector ecosystem covers dozens of data sources.

The problem was the auth model. Trino authenticates the user at the coordinator, then the coordinator uses its own service account to access data. The coordinator is the security boundary. If the coordinator is compromised, every data source it connects to is exposed. And the CloudTrail trail shows one actor – the Trino service account – for every query by every user.

We tried to fix this. We maintained a fork of the Trino Iceberg connector (the "DCAF branch") that passed bearer tokens through to Polaris. Every upstream Trino release required re-merging our changes. Each workaround made the system more complex without making it more secure. After two years of this, the question stopped being "can we fix Trino's auth model" and became "should we."

The answer was no. The auth model is not a bug in Trino. It is a design decision. Trino is designed around the assumption that the engine holds ambient credentials. Changing that assumption would require rewriting the coordinator, the worker communication protocol, the connector interface, and the credential management layer. At that point you are not patching Trino. You are building a new engine with Trino's SQL parser.

Which is approximately what we did. Just with DataFusion's SQL parser instead.

Dimension	DuckDB	Spark	Trino	DataFusion
Model	Embedded DB	Distributed compute framework	Distributed SQL engine	Query engine library
Language	C++	Scala/Java	Java	Rust
Auth model	N/A (embedded)	Service account	Service account	You implement it
Extension model	UDFs, table functions	Catalyst rules, data sources	Connectors	Traits (Catalog, Table, Plan, Rule)

Dimension	DuckDB	Spark	Trino	DataFusion
Deployment	In-process	Cluster (YARN/K8s)	Cluster (coordinator + workers)	Your binary, your way
Per-user isolation	No	No	Partially (query-level)	Yes (SessionContext)
Plan rewriting	No	Yes (Catalyst)	Limited	Yes (LogicalPlan is a tree)

The Extensibility Model

DataFusion's power is not in what it does. It is in what it lets you replace.

Every significant component is a trait:

Trait	What It Controls	SQE Implementation
`CatalogProvider`	What catalogs/schemas exist	`SqeCatalogProvider` (bridges to Polaris)
`SchemaProvider`	What tables exist in a schema	`SqeSchemaProvider` (lists Iceberg namespace)
`TableProvider`	How a table is scanned	`SqeTableProvider` (wraps Iceberg table)
`ExecutionPlan`	How a physical operation executes	`IcebergScanExec`, `DistributedScanExec`
`OptimizerRule`	How the logical plan is rewritten	Policy enforcement (row filters, column masks)

You don't subclass. You don't monkey-patch. You implement a trait and register it. DataFusion calls your implementation at the right time, in the right context, with the right inputs. The contract is the trait signature. The compiler enforces it.

This is why the "80/20" framing is accurate. DataFusion gives you the SQL parser, the logical planner, the optimizer, the physical planner, the execution runtime, the Arrow memory model, and the pull-based streaming. That's 80% of a query engine. The remaining 20% – the catalog integration, the storage layer, the auth model, the policy enforcement, the distribution strategy – is your product. It's the part that makes your engine different from every other engine built on DataFusion.

SQE's `Cargo.toml` tells the story:

```
datafusion = "52"
datafusion-common = "52"
datafusion-expr = "52"
datafusion-sql = "52"

arrow = { version = "57", features = ["prettyprint"] }
arrow-flight = { version = "57", features = ["flight-sql-experimental"] }
```

```
iceberg = { git = "https://github.com/risingwavelabs/iceberg-rust.git", rev = "1978911ec4" }
iceberg-catalog-rest = { git = "https://github.com/risingwavelabs/iceberg-rust.git", rev = "1978911ec4" }
```

DataFusion 52, Arrow 57, Iceberg 0.9. Three version numbers that represent tens of thousands of hours of engineering by communities we will never have to hire. The SQL parsing, the columnar memory layout, the table format – all solved problems. SQE's contribution is the glue: how these pieces fit together under a sovereign auth model.

Antipattern: reimplementing what DataFusion already does. The temptation is strong. You see how DataFusion's CSV reader works and think "I could write a better one for my use case." Maybe. But now you maintain a CSV reader, and you miss every bug fix and performance improvement the DataFusion community ships. Implement traits. Don't rewrite internals. The 20% that's yours is plenty of work.

The Separation That Changes Everything

The deepest insight from building on DataFusion is not about Rust, or Arrow, or SQL parsing. It is about separation.

In a traditional database, the catalog is the database. The storage is the database. The compute is the database. You cannot use the catalog without the compute engine. You cannot access the storage without the catalog. Everything is one product, one vendor, one bill.

DataFusion separates these concerns into independent layers:

Catalog: what tables exist, where they are. In SQE, this is Polaris – an Apache Iceberg REST catalog. The catalog tells you that table `analytics.events` exists, that its current metadata is at `s3://warehouse/analytics/events/metadata/v42.metadata.json`, and that the user has access to it. The catalog is an API call, not a database operation.

Storage: how to read the bytes. The bytes live in S3. They are Parquet files. The FileIO layer (from iceberg-rust) reads them using credentials that Polaris vended for the specific user. The storage layer knows nothing about SQL. It knows about byte ranges and column chunks.

Compute: how to turn bytes into answers. DataFusion reads Arrow record batches from the storage layer, applies filters, projections, aggregations, joins, and sorts, and produces result batches. The compute layer knows nothing about where the data came from. It works with Arrow's in-memory columnar format, whether the source is Iceberg, Delta Lake, CSV, or a hand-written `RecordBatch`.

This separation is what makes sovereignty possible. You can swap Polaris for Gravitino. You can swap S3 for R2 or GCS or local disk. You can swap DataFusion for DuckDB (if your requirements change). Each layer is an interface, not an implementation. You are never locked into a vendor at any layer.

And here is the part that matters for multi-tenancy: because the layers are separate, you can configure them differently per user. Alice's catalog view might show different namespaces than Bob's. Alice's storage credentials are scoped to Alice's permissions. Alice's compute might have different memory limits. All within the same engine, the same process, the same binary. The separation is what makes per-user isolation a configuration problem rather than an architecture problem.

Sovereignty principle: The separation of catalog, storage, and compute is not an architectural nicety. It is the mechanism of sovereignty. When these three layers are coupled, you are locked to whatever product couples them. When they are separate, you can replace any layer without changing the others. Your data stays in your storage. Your metadata stays in your catalog. Your compute runs where you choose. That is sovereignty.

The Eighty-Percent Engine

DataFusion gave us a query engine. Not a toy, not a prototype, but a real engine with a real optimizer, real parallel execution, and real Arrow-native performance. The SQL parser handles complex queries. The optimizer produces good plans. The execution engine streams results efficiently.

What DataFusion did not give us: authentication, catalog integration with Polaris, Iceberg table scanning with user-scoped credentials, policy enforcement via plan rewriting, distributed execution across workers, Flight SQL wire protocol, Trino JDBC compatibility, or deployment configuration.

That's the 20%. That's the product.

The lesson is not "use DataFusion." The lesson is that the best infrastructure components are libraries, not services. A library gives you the building blocks and lets you assemble them your way. A service gives you someone else's assembly and charges you for the privilege of not controlling it.

We went from zero to a working single-node query engine – parsing SQL, authenticating users, querying Iceberg tables via Polaris, streaming Arrow results over Flight SQL – in three days. Not because we're fast. Because DataFusion did the hard part.

The next three months were the 20%.

Field report: The first integration test – authenticate via OIDC, query an Iceberg table through Polaris, receive Arrow batches over Flight SQL – passed on March 14, 2025. The same day we scaffolded the crate structure. DataFusion handled the SQL parsing, logical planning, optimization, and execution. We implemented `CatalogProvider`, `SchemaProvider`, `TableProvider`, and the Flight SQL handshake. Three files of glue code connected DataFusion to Polaris to S3 to the user. The test passed. It took less than a day. That was the moment we knew the library approach would work.

AI Logbook: The AI scaffolded all six initial crates and implemented the `execute_query` pipeline — from SQL string to Arrow record batches — in a single session. The human made the decision to use DataFusion as a library rather than Trino, Spark, or DuckDB, after two years of maintaining a Trino fork. The `create_session_context` method — one `SessionContext` per user with per-session credentials — was specified by the human as the architectural constraint; the AI implemented it correctly because Rust's ownership model made the isolation boundaries explicit in the type signatures.

You Are the Query

> There is no service account. There is only you.

The security team's question – "who accessed the customer table last Tuesday?" – wasn't hard because we lacked logging. It was hard because every query ran as the same identity. Trino's service account read every file, for every user, every time. CloudTrail showed one actor doing everything. The audit trail was technically complete and practically useless.

So the first constraint for SQE wasn't performance. It wasn't SQL compatibility. It was this: when Alice runs a query, S3 must see Alice. Not "sqe-service-account". Not "trino-coordinator". Alice.

That sounds simple. It turned out to be the hardest design decision in the entire engine – and the one that made everything else possible.

The previous chapter showed how DataFusion gives you a query engine as a library – a `SessionContext` per user, a `CatalogProvider` per catalog, a `TableProvider` per table. But all of that machinery is useless if the credentials flowing through it belong to the engine instead of the user. The `SessionContext` is only as sovereign as the identity it carries. This chapter is about making sure that identity is real.

The Three Paths We Considered

Path 1: Service account with user tagging

The standard approach. The engine holds a service account with broad read access. It logs which user initiated each query. Auditing works through application logs, not CloudTrail.

We tried this first. It took about an hour to realise the problem: the engine becomes the security boundary. If the engine is compromised, every table is readable. If the engine has a bug in its access control logic, data leaks silently. The audit trail says "Alice queried table X" but S3 says "sqe-service read table X, Y, Z, and everything else."

Every major query engine works this way. Trino, Spark, Presto, Starburst – they all hold a service account, and they all enforce access control in the application layer. The reasoning is pragmatic: it's simpler, it's faster (you can cache data across users), and it's how databases have always worked.

The problem is that it conflates two things that should be separate: who can run a query and who can access the data. The engine decides both. If the engine is wrong about either one, the failure is silent.

No alarm fires when a service account reads a file it shouldn't, because the service account can read everything.

Dead end: service account with tagging. It's the path of least resistance, and every major query engine uses it. We built a prototype in an afternoon. The security team rejected it in a meeting that lasted twelve minutes.

Path 2: Per-user IAM roles assumed by the engine

Better. The engine receives Alice's identity, then calls STS AssumeRole to get temporary credentials scoped to Alice's permissions. S3 sees Alice's role, not the engine.

This works on paper. In practice, it requires pre-provisioning IAM roles for every user, maintaining a mapping between OIDC identities and AWS roles, and handling the assume-role chain when Polaris also needs to assume on behalf of the user. The credential chain gets deep. The failure modes multiply.

We explored this for about two days. The AI generated working STS assume-role code in twenty minutes. The remaining time was spent on the IAM policy matrix – each user needs a role with trust relationships to both the engine's execution role and the Polaris service role, scoped S3 policies per namespace, and lifecycle management for onboarding, permission changes, and role deletion. The code was fine. The operational model was not. And it was AWS-specific; on GCP or Azure, the entire mechanism is different.

There's also a subtler problem: STS tokens have a maximum duration of 12 hours. For a 30-second query, fine. For a long-running ETL job, you might need to re-assume the role mid-execution. And if the assume-role call fails because the IAM role was modified while the query was running, the query fails in a way that has nothing to do with the data or the SQL.

Path 3: Bearer token passthrough

The user authenticates via OIDC. The coordinator receives a JWT. That JWT is forwarded – unchanged – to every downstream system: Polaris for catalog operations, S3 for storage access (via Polaris credential vending), workers for distributed execution.

The engine never holds ambient credentials. It never assumes a role. It passes the user's proof of identity through, and every downstream system makes its own access decision based on that identity.

This is the model we built.

The elegance is in what the engine doesn't do. It doesn't evaluate permissions. It doesn't maintain role mappings. It doesn't cache credentials across users. It simply passes the token through and lets each downstream system decide. Polaris decides whether Alice can see the table. S3 decides whether Alice can read the files. The engine is a conduit, not a gatekeeper.

The trade-off is operational. The engine depends on the OIDC provider being available for every handshake. If Keycloak is down, no new sessions can be created. Existing sessions continue to work (the JWT is already in memory), but the engine cannot authenticate new users. This is a real dependency. We mitigate it with the token cache and refresh mechanism, but the hard truth is: if your identity provider is down, your query engine is partially down. In a service account model, the engine can

continue operating with its cached credentials. In a passthrough model, it can serve existing sessions but not new ones. We decided that this was an acceptable trade-off because if your identity provider is down, you have bigger problems than query latency.

Why OIDC Password Grant

OIDC has several grant types. The one nobody uses for query engines is the resource owner password credentials grant – the user sends username and password directly, and gets back a JWT.

It's considered legacy. It's being deprecated in OAuth 2.1. And it's the only grant type that works cleanly for non-interactive SQL clients.

Here's the problem: a JDBC client connecting to a query engine doesn't have a browser. There's no redirect flow. There's no authorization code exchange. The user types a connection string with a username and password, and the engine needs to turn that into a bearer token.

`client_credentials` is wrong – that authenticates the *application*, not the *user*. The whole point is that every query runs as a specific human.

So we use the password grant for interactive clients (JDBC, DBeaver, CLI) and accept bearer tokens directly for programmatic clients that handle their own OIDC flow. The coordinator's `do_handshake` method handles both paths.

The `Authenticator` struct in `sqe-auth` abstracts this choice behind a runtime backend selection:

```
enum AuthBackend {
    /// OIDC Password Grant (ROPC) — exchanges username/password for a token
    /// via any OIDC-compliant provider (Keycloak, Auth0, Okta, etc.).
    OidcPassword(OidcPasswordClient),
    /// Generic OAuth2 client_credentials — obtains a service token from any
    /// OAuth2-compliant endpoint (e.g. Polaris). Username/password are ignored.
    ClientCredentials(OAuthClient),
}

pub struct Authenticator {
    backend: AuthBackend,
    cache: TokenCache,
    refresh_buffer_secs: u64,
}
```

The backend selection happens at startup based on configuration. If `keycloak_url` is set, the engine uses OIDC password grant. If only `token_endpoint` is set, it uses client credentials. The rest of the engine doesn't know or care which backend is active – it calls `authenticator.authenticate(username, password)` and gets back a `Session`.

This matters because different deployments need different auth models. A production cluster behind Keycloak uses the password grant. A local development stack running against Polaris directly uses client credentials. The integration test suite uses client credentials against a Polaris in-memory catalog. Same engine, same code path, different auth backend.

DataFusion deep dive: The `FlightSqlService::do_handshake` method receives a `HandshakeRequest` stream. SQE extracts the Basic auth header, calls Keycloak's token endpoint, and returns the JWT as the session token. Every subsequent Flight call carries this token in the `authorization` metadata header.

The Handshake: From Password to Session

The Flight SQL protocol defines a handshake mechanism for authentication. The client sends a `HandshakeRequest` with Basic auth credentials. The server authenticates them and returns a bearer token that the client uses for every subsequent call.

In SQE, the `do_handshake` implementation does the OIDC exchange:

```
async fn do_handshake(
    &self,
    request: Request<Streaming<HandshakeRequest>>,
) -> Result<Response<Pin<Box<dyn Stream<Item = Result<HandshakeResponse, Status>> + Send>>>,
    Status,
> {
    let authorization = request.metadata()
        .get("authorization")
        .ok_or_else(|| Status::invalid_argument("Authorization header not present"))?
        .to_str()
        .map_err(|e| Status::internal(format!("Authorization header not parsable: {e}")))?
        .to_string();

    // Decode Basic auth: base64(username:password)
    let base64_encoded = &authorization["Basic ".len()..];
    let decoded = base64::engine::general_purpose::STANDARD
        .decode(base64_encoded)
        .map_err(|e| Status::invalid_argument(format!("Invalid base64 in auth: {e}")))?;
    let decoded_str = std::str::from_utf8(&decoded)
        .map_err(|e| Status::invalid_argument(format!("Invalid UTF-8 in auth: {e}")))?;

    let parts: Vec<&str> = decoded_str.splitn(2, ':').collect();
    let (username, password) = match parts.as_slice() {
        [user, pass] => (*user, *pass),
        _ => return Err(Status::invalid_argument(
            "Invalid authorization: expected username:password",
        )),
    };

    // OIDC exchange: username/password -> JWT
    let session = self.session_manager
        .authenticate(username, password)
        .await
        .map_err(|e| Status::unauthenticated(format!("Authentication failed: {e}")))?;
```

```
    // Return session ID as bearer token
    let result = HandshakeResponse {
        protocol_version: 0,
        payload: session.id.as_bytes().to_vec().into(),
    };

    let mut response = Response::new(Box::pin(futures::stream::iter(vec![Ok(result)])));
    response.metadata_mut().append(
        "authorization",
        MetadataValue::from_str(&format!("Bearer {}", session.id))?,
    );

    Ok(response)
}
```

The handshake extracts the Basic auth header, base64-decodes it to get the username and password, and calls `session_manager.authenticate()`. Under the hood, the `SessionManager` delegates to the `Authenticator`, which POSTs a `grant_type=password` form to the OIDC provider's token endpoint (`{keycloak_url}/realms/{realm}/protocol/openid-connect/token`). The provider validates the credentials and returns a JWT (the access token), a refresh token, and an expiry time. The access token is the user's proof of identity for every downstream call.

The `Authenticator` also extracts roles from the JWT payload – a lightweight base64 decode of the claims, no signature verification, because the OIDC provider already validated the token. The roles feed into policy enforcement (Chapter 8) where they determine what the user can see and do.

There's a second authentication path. For programmatic clients that already have a JWT – a backend BFF that obtained a token through the authorization code flow, or a service that got one from its own OIDC exchange – the coordinator accepts raw bearer tokens directly. If the token in the `authorization` header isn't a known session ID but looks like a JWT (contains dots), the coordinator wraps it into an ad-hoc session and proceeds. Same code path, same token passthrough, no handshake required.

This dual-path design means SQE works with both interactive users (DBeaver, CLI) and programmatic callers (dbt, Airflow, custom services) without either side having to adapt.

Antipattern: validating JWTs in the engine. It's tempting to validate JWT signatures in the coordinator – check the issuer, verify the audience, validate the expiry. We considered it. The problem: the coordinator is not the resource server. Polaris is. S3 is. If the coordinator rejects a token that Polaris would accept, or accepts a token that Polaris would reject, you've created a gap. Let the downstream systems validate their own tokens. The coordinator's job is to pass the token through and handle the error if the downstream rejects it. Don't build a second authorization layer that can disagree with the first.

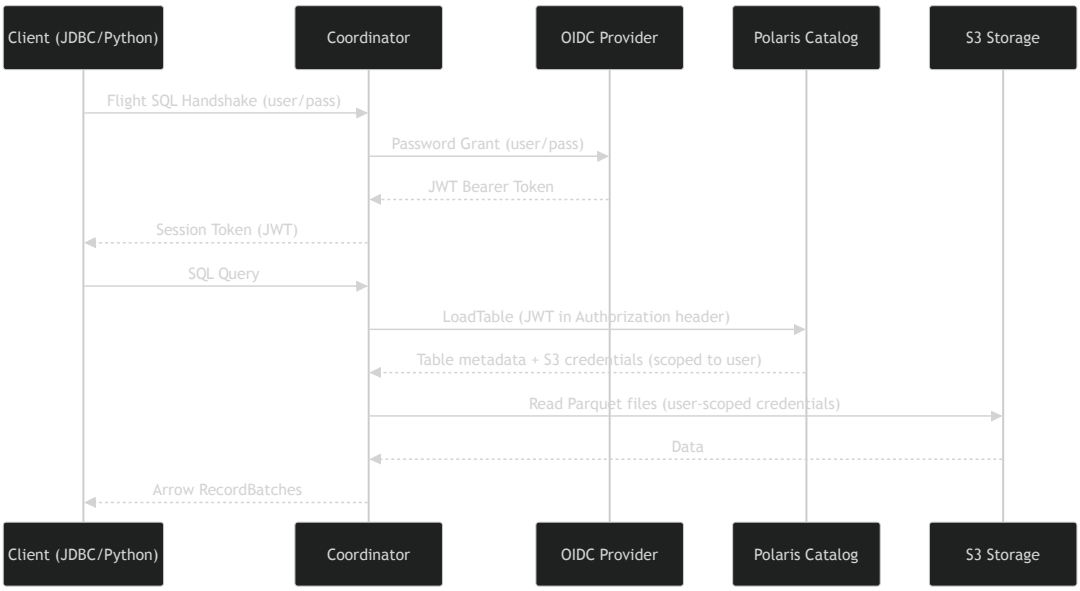

Figure 3: Authentication flow: OIDC password grant through coordinator, Polaris credential vending, and bearer token passthrough to workers and S3

The Token Passthrough Architecture

Once the coordinator has a JWT, the flow is:

1. **Catalog**: coordinator sends the JWT to Polaris in the `Authorization` header. Polaris validates it and returns table metadata plus temporary S3 credentials scoped to that user's permissions.
2. **Storage**: the S3 credentials from Polaris are used directly. No separate STS call. No role assumption. Polaris did the credential vending.
3. **Workers** (distributed mode): the coordinator sends the JWT along with the plan fragment. The worker uses it to get its own S3 credentials from Polaris. The worker never contacts the coordinator for credentials – it goes directly to Polaris with the user's token.
4. **Token refresh**: for long-running queries, the coordinator monitors token expiry and pushes refreshed tokens to workers via a background task.

Every I/O operation in the entire system traces back to a specific human identity. CloudTrail shows Alice reading the Parquet files that Alice's query needed. Not the engine. Not a service account. Alice.

Credential Vending: What Polaris Actually Does

The credential vending flow is the mechanism that makes passthrough possible at the storage layer. Understanding it explains why we didn't need STS AssumeRole.

When SQE loads a table from Polaris, the REST catalog response includes more than just metadata. Polaris returns the table's current metadata location, the schema, the partition spec – and a set of temporary S3 credentials scoped to the files that table contains. These are short-lived STS tokens that Polaris obtained by assuming a storage role on behalf of the authenticated user.

The important part: Polaris does the IAM role assumption, not SQE. Polaris has the trust relationship with the storage role. Polaris evaluates the user's JWT, determines what the user is allowed to access, and returns credentials that are scoped to exactly that access. SQE receives the credentials and passes them to iceberg-rust's `FileIO` layer, which uses them to read Parquet files from S3.

This is why Path 2 (per-user IAM roles) was unnecessary. Polaris already solves the credential vending problem. It already maps OIDC identities to storage permissions. It already returns short-lived credentials. We didn't need to build any of that. We just needed to pass the user's JWT to Polaris and use what came back.

The credential lifetime is controlled by Polaris configuration, typically 15 minutes to 1 hour. For most queries, this is more than enough. For long-running scans, the credentials might expire before the scan completes. That's the problem the credential refresh mechanism solves.

Component	What it does	What it knows about the user
Coordinator	Routes queries, manages sessions	JWT, username, roles
Polaris	Validates JWT, vends S3 credentials	Full OIDC identity, storage permissions
S3	Accepts STS credentials, serves files	The assumed role (traced to the user)
Workers	Execute scan fragments	JWT (forwarded), S3 credentials (vended per fragment)

Each component makes its own access decision. The coordinator doesn't decide what Alice can read from S3. Polaris does. The coordinator doesn't decide whether Alice's credentials are valid. S3 does. The engine is a coordinator of decisions, not the decision maker.

Iceberg deep dive: The Iceberg REST catalog specification (the `loadTable` endpoint) supports a `config` field in the response that can include `s3.access-key-id`, `s3.secret-access-key`, and `s3.session-token`. This is the credential vending mechanism. Polaris implements it by assuming a storage role scoped to the table's S3 prefix. Different catalogs implement this differently – Gravitino has its own credential provider SPI, Unity Catalog returns credentials via a separate endpoint. SQE relies on the Iceberg REST spec's approach because it's the most portable. If your catalog returns credentials in the table config, SQE uses them. If it doesn't, SQE falls back to the ambient credentials in its storage configuration. The fallback exists for development (local MinIO, RustFS) where credential vending isn't needed.

Session Lifecycle

Each authenticated query creates a `Session`:

```
pub struct Session {
    pub id: String,
```

```
    pub user: SessionUser,
    pub access_token: String,
    pub refresh_token: Option<String>,
    pub token_expiry: DateTime<Utc>,
    pub created_at: DateTime<Utc>,
    pub last_activity: DateTime<Utc>,
    pub default_catalog: Option<String>,
    pub default_schema: Option<String>,
    pub source: Option<String>,
}
```

The session holds the JWT (`access_token`), the refresh token for obtaining new JWTs when the current one expires, the token expiry time, and the user's identity. The `id` field is a UUID that serves as the bearer token for subsequent Flight SQL calls. The `Debug` implementation redacts the access and refresh tokens – credentials never appear in logs.

Sessions have two timeout mechanisms. The `last_activity` timestamp tracks idle time – if no query has been executed for a configurable duration (default: 15 minutes), the session is swept. The `created_at` timestamp enforces an absolute lifetime (default: 8 hours) regardless of activity. A `SessionManager` wraps the `Authenticator` and a concurrent `DashMap` of active sessions. On each Flight SQL request, it checks whether the background refresh task has updated the token and transparently swaps in the fresh one. Expired sessions are cleaned up on access; a periodic sweep catches sessions where the client disconnected without sending another request.

One session per query, not per connection. A JDBC connection might stay open for hours, but each query might execute with a different token state – the token might have been refreshed between queries. By creating session state per query, we avoid the stale-token problem that plagues connection-pooled systems.

The `Session` also carries optional context fields: `default_catalog`, `default_schema`, and `source`. These come from client headers (e.g., the Trino compatibility layer reads `X-Trino-Catalog` and `X-Trino-Schema`). The session is not just an authentication container – it carries the user's working context through the entire query lifecycle.

The Background Refresh Task

The `Authenticator` spawns a background tokio task that polls every 10 seconds for sessions whose tokens are approaching expiry. When it finds one, it calls the OIDC provider's refresh endpoint using the refresh token (not the user's password) and updates a concurrent `DashMap` cache. The `SessionManager` picks up refreshed tokens on the next `get_session` call – if the cached token differs from the session's stored token, the session is transparently updated. The user sees nothing.

For the `client_credentials` backend (used in development), there's no refresh token – the task simply re-fetches from the token endpoint. For `OidcPassword` (production), the refresh token was returned alongside the access token during initial authentication. It typically has a longer lifetime than the access token (hours instead of minutes). When the refresh token itself expires, the session is evicted and the user must re-authenticate.

The entire auth configuration is seven fields: OIDC provider URL, realm, client ID, client secret, token endpoint, refresh buffer (seconds before expiry to trigger refresh), and TLS verification toggle. No IAM role ARNs. No STS configuration. No permission matrices. Seven fields.

Tokens in Distributed Mode

The single-node model is straightforward: the coordinator holds the token, the token flows to Polaris, Polaris vends credentials, the coordinator reads from S3. Everything happens in one process.

Distributed mode changes the picture. When the coordinator splits a query plan and sends scan fragments to workers, each worker needs its own S3 credentials. The coordinator could pre-vend credentials for every worker, but those credentials expire. A query that runs for two minutes might be fine. A query that scans a terabyte might not.

The `DistributedScanExec` carries the credential state along with the plan:

```
pub struct DistributedScanExec {
    scan_tasks: Vec<ScanTask>,
    worker_urls: Vec<String>,
    schema: SchemaRef,
    credential_expiry: Option<DateTime<Utc>>,
    credential_tracker: Option<Arc<CredentialRefreshTracker>>,
    worker_registry: Option<Arc<WorkerRegistry>>,
    max_retries: u32,
    local_executor: Option<Arc<dyn LocalExecutor>>,
    // ...
}
```

Each `ScanTask` includes the S3 credentials that the worker needs to read its assigned files. The `credential_expiry` tells the coordinator when those credentials will stop working. The `credential_tracker` is the mechanism for pushing fresh credentials before that happens.

A `CredentialRefreshTracker` maintains a map of active fragments and their credential expiry times. When the `DistributedScanExec` dispatches a fragment to a worker, it registers that fragment with the tracker. A background task checks every 30 seconds for fragments whose credentials are approaching expiry (within a 5-minute buffer). For those fragments, it calls back to Polaris to vend fresh credentials, then pushes them to the worker via Arrow Flight's `do_action("refresh_credentials")` mechanism. The worker receives the action, hot-swaps the credentials, and continues scanning – no restart, no data loss, no interruption. Chapter 14 shows what happens when this mechanism is tested under load with 50 concurrent clients.

Every type that holds secrets – `Session`, `RefreshableCredentials`, `CachedToken` – has a custom `Debug` implementation that replaces sensitive fields with `[REDACTED]`. Credentials never appear in logs, even in debug mode.

This is the full chain: user authenticates with OIDC, coordinator holds the JWT, coordinator asks Polaris for credentials scoped to the user, coordinator sends those credentials to workers with the plan fragment, workers read from S3 using the user's credentials, and if the credentials expire mid-

scan, the coordinator pushes fresh ones. At every point, every byte read from storage is attributed to the user who initiated the query.

Field report: the three-day credential push. We hit this during load testing: 50 concurrent clients, some queries queued long enough that their vended S3 credentials expired before execution started. The worker got a `403 Forbidden` that surfaced as a cryptic Arrow error about "failed to read Parquet footer." The debugging took most of a day – we traced it through Arrow, iceberg-rust's FileIO, and the S3 client before finding the expired STS token. The fix was the credential refresh tracker: three days of design discussion for 200 lines of code. The code was straightforward. The thinking – what if the refresh fails? what if two refreshes race? – was not. Chapter 14 has the full load test story.

The Security Properties

The passthrough model gives you properties that are impossible to achieve with a service account:

- **Attribution**: every S3 access is attributed to a user in CloudTrail
- **Least privilege**: users can only read what Polaris grants them – the engine can't escalate
- **No lateral movement**: compromising the engine gives the attacker nothing without a valid JWT
- **Compliance**: the engine never stores credentials, never caches tokens to disk
- **Portability**: works with any OIDC provider (Keycloak, Auth0, Okta, Azure AD) and any S3-compatible storage

The one property you lose: caching. A service account can cache data across users because it has ambient access. A passthrough model can't – each user's query must go through their own credential chain. We accepted the performance cost because the security properties are non-negotiable.

The "no lateral movement" property deserves spelling out. In a service account model, compromising the engine means reading everything. In the passthrough model, an attacker sees the JWTs of currently active sessions – bad, but those tokens expire in minutes to hours and are scoped to each user's permissions. The blast radius is bounded by the intersection of "who has an active session" and "what each of those users can see." Compare that to "everything in the data warehouse."

The compliance property matters for regulated industries. If your security auditor asks "where are the credentials stored?" the answer is "nowhere." The OIDC provider issues them. The engine passes them through. Polaris vends storage credentials on the fly. S3 validates them on every request. No credential file. No secret in a config map. The credential lifecycle is entirely in-memory and ephemeral.

Rust's ownership model reinforces this. When a session is dropped, the `access_token` and `refresh_token` strings are deallocated – not "eligible for collection," but gone. No garbage collector holding references longer than expected. Deterministic credential cleanup for free.

Sovereignty principle: A service account is a shared secret. A bearer token is a proof of identity. The difference is the difference between "the engine read the data" and "Alice read the data." When security asks who accessed the customer table, the answer should be a name, not an application.

What This Model Makes Impossible to Retrofit

It would be natural to ask: why not add bearer passthrough to Trino? We tried. For two years. The answer is that the auth model is not a feature. It's a design constraint that shapes every other decision in the engine.

Trino's connector interface assumes ambient credentials. `ConnectorMetadata`, `ConnectorSplitManager`, `ConnectorPageSourceProvider` – none of them take a user token. Adding one would be a breaking change to every connector in the ecosystem. And even if you did it, the shuffle layer, the result cache, and the spill-to-disk layer all use the coordinator's credentials.

We maintained a fork of the Trino Iceberg connector – the "DCAF branch" – that passed bearer tokens through to Polaris. Every upstream release required re-merging our changes, and every merge conflict was in the authentication layer. After two years, we had a connector that worked for our specific deployment and broke whenever Trino touched credential handling.

SQE doesn't have this problem because the auth model was the first thing we built, not the last. The `Session` struct is the first parameter to every method that touches data. The `SessionContext` is created with the user's credentials. The `SessionCatalog` carries the user's token to Polaris. There is no code path that accesses data without a user identity, because there is no code path that exists without one.

This is the deepest lesson of this chapter. The auth model is not a feature on a roadmap. It is an architectural constraint that determines the shape of everything built on top of it. You can add a feature to a system. You cannot add a constraint. Constraints must be there from the beginning, or you will spend years trying to retrofit them into code that was designed without them.

We know, because we spent those years on Trino before we built SQE.

Ten Ways to Prove You're You

When we built the first version of SQE, there was exactly one way to authenticate: OIDC password grant. You sent a username and password over the Flight SQL handshake, we exchanged them for a JWT with Keycloak, and that JWT became your session. Simple. Sufficient.

Then came the real world.

The data engineering team wanted to run dbt from Airflow. Airflow already had a JWT — obtained through its own OIDC exchange — and had no username or password to offer. The platform team wanted to connect from AWS Lambda functions using IAM execution roles. The security team wanted mutual TLS for service-to-service calls. A consultant needed an API key that could be rotated without touching an identity provider. The testing infrastructure wanted an anonymous provider that just says yes to everything in CI.

A single provider couldn't serve all of these. But we also didn't want to make auth a configuration burden — the engine needed to figure out which mechanism applied to which connection automatically.

The answer was `AuthProvider` and `AuthChain`.

The Trait and the Chain

Every auth mechanism implements one trait:

```rust
#[async_trait]
pub trait AuthProvider: Send + Sync {
    async fn authenticate(&self, credentials: &FlightCredentials) -> Result<Identity, AuthError>;

    async fn refresh_catalog_token(&self, identity: &Identity) -> Result<Option<String>, AuthError> {
        Ok(None)
    }
}
```

`FlightCredentials` carries whatever the client sent — a username and password, a bearer token, a client certificate CN, or nothing at all. The provider inspects what it knows how to handle and returns one of three things: an `Identity` on success, `AuthError::NotMyCredentials` if the credential type isn't for it, or `AuthError::AuthFailed` if the credential type matches but the credentials are wrong.

That `NotMyCredentials` variant is the key to how `AuthChain` works. The chain tries providers in order. The first one that returns `Ok(Identity)` wins. If a provider says `NotMyCredentials`, the chain moves on. If a provider says `AuthFailed`, the chain stops immediately — no point trying the next provider when the credential type matched but was explicitly rejected.

The implication: a connection carrying a JWT won't accidentally fall through to the anonymous provider just because it was listed last. The bearer token provider will recognise the JWT, attempt validation, and either accept it or definitively reject it.

The Ten Providers

`OidcPasswordProvider` is the workhorse. A JDBC user in DBeaver sends `username:password`; the provider POSTs a `grant_type=password` request to the OIDC token endpoint and gets back a JWT. Works with Keycloak, Auth0, Okta, Zitadel, or any OIDC-compliant provider. Appropriate when your users are humans with credentials in an identity directory.

`BearerTokenProvider` validates a pre-obtained JWT via JWKS. The client sends a token it already has — from a browser SSO flow, a service that did its own OIDC exchange, or a backend BFF. The provider fetches the JWKS endpoint, verifies the signature, and extracts the identity. No password exchange needed. This is the right provider for Airflow, dbt running in CI, and any programmatic client that manages its own token lifecycle.

`TokenExchangeProvider` implements RFC 8693 — it takes an incoming credential (a JWT from one issuer) and exchanges it at the OIDC token endpoint for a user-scoped JWT from another issuer. This is the federated identity path: your users authenticate with your corporate IdP, and the engine mints a Polaris-compatible token without them ever needing Polaris credentials.

`ApiKeyProvider` accepts opaque keys from a TOML keys file, identified by a configurable prefix (`sqe_` by default). Comparison is constant-time to prevent timing attacks. Appropriate for scripts, automation, and service accounts that can't do browser-based OIDC and don't need per-user identity — the keys file maps each key to a user ID and role set.

`AwsIamProvider` uses STS `GetCallerIdentity` with a SigV4-signed request to verify that the connecting client is who it claims to be on AWS. Lambda functions, EC2 instances, and ECS tasks can authenticate using their IAM execution role without any credentials to manage. Role mappings in `[auth.role_mappings]` translate IAM ARN patterns to SQE roles.

`MtlsProvider` authenticates via mutual TLS client certificates. The client certificate's Common Name becomes the user identity; the Organizational Unit can be mapped to roles. This is service-to-service auth for environments where certificate infrastructure is already in place — Kubernetes clusters with cert-manager, internal meshes, compliance environments where every service must present a certificate.

`AnonymousProvider` assigns a fixed identity and role set to any connection, regardless of credentials. The only provider that ignores `FlightCredentials` entirely. Appropriate for local development stacks, integration test environments, and public read-only query endpoints. Put it last in the chain — it accepts everything, so anything after it will never be reached.

`DeviceCodeProvider` implements RFC 8628 — the device authorization grant. The client calls the engine to start a flow, receives a short code and a URL, and tells the user "go to `https://auth.example.com/device` and enter `GBTF-ZPQR`." Once the user completes auth in a browser, the engine's polling loop receives the token. This is how `gh auth login` works. For SQL query tools on servers without browser access, or CLI tools used by non-technical users, this is the right flow.

`AuthCodeService` implements OAuth2 Authorization Code + PKCE (RFC 7636). This is the browser redirect flow — the Trino `externalAuthentication=true` path. The client initiates a handshake, gets redirected to the IdP's login page, authenticates there, and the IdP sends the code back to the engine's callback URL. For JDBC clients that can open a browser, this is the most secure interactive flow available.

Legacy `Authenticator` is the original `OidcPassword` + `ClientCredentials` backend. It exists for backward compatibility. When the `[[auth.providers]]` array is empty, the factory wraps it in a single-provider chain and everything works exactly as it did before we built any of this. No configuration changes needed for existing deployments.

The Configuration

Providers are configured as a TOML array. The order matters — it's the order the chain tries them:

```toml
[[auth.providers]]
type = "oidc_password"
token_url = "https://keycloak.example.com/realms/data/protocol/openid-connect/token"
client_id = "sqe"
client_secret = "changeme"

[[auth.providers]]
type = "bearer_token"
jwks_url = "https://keycloak.example.com/realms/data/protocol/openid-connect/certs"
issuer = "https://keycloak.example.com/realms/data"
```

```toml
[[auth.providers]]
type = "api_key"
keys_file = "/etc/sqe/api-keys.toml"
key_prefix = "sqe_"

[[auth.providers]]
type = "anonymous"
user = "dev"
roles = ["public"]

[auth.role_mappings]
"arn:aws:iam::123456789012:role/DataEngineering" = ["analyst", "writer"]
"data-team" = ["analyst"]
```

The `[auth.role_mappings]` table is shared across providers that support it — `AwsIamProvider` maps IAM role ARNs, `MtlsProvider` maps certificate OUs, `ApiKeyProvider` reads them from the keys file.

OIDC Discovery

The `DeviceCodeProvider` and `AuthCodeService` don't take explicit endpoint URLs. Instead, they take an OIDC issuer URL and perform discovery: a GET request to `{issuer}/.well-known/openid-configuration` returns a JSON document with all the endpoints needed — `token_endpoint`, `authorization_endpoint`, `device_authorization_endpoint`, `jwks_uri`. The `OidcDiscovery` module fetches this once at first use and caches the result in a `OnceCell`. Endpoint overrides are available for environments where the discovery document doesn't match the reachable URLs (common in Kubernetes where the internal and external URLs differ).

The Design Philosophy

We could have picked one auth mechanism and made it the standard. Every other query engine does. Trino uses a plugin system that ships with LDAP and Kerberos. Spark uses the cluster's Hadoop security. Presto has file-based passwords.

The problem is that "pick one" is a constraint on the operator, not the engine. A data platform serving a bank has different auth requirements from a startup's internal analytics cluster. The bank has Kerberos, mutual TLS, and an IAM policy council. The startup has Okta and a Slack channel. Neither should have to fork the engine to support their auth model.

> Sovereign means you pick your auth, not us.

The `AuthProvider` trait is deliberately minimal — one method, one input type, one output type. Implementing a new provider is an afternoon of work. The chain composes them without any of them knowing about each other. And the legacy fallback means existing deployments don't need to change a thing.

The "you are the query" principle still holds for every one of these ten providers. Whether Alice proves her identity via Keycloak password grant, a pre-obtained JWT, an IAM execution role, or an mTLS certificate, what flows downstream is the same thing: an `Identity` with a `catalog_token` that Polaris

and S3 will recognise as Alice. The mechanism changes. The property — every S3 access attributed to a specific human or service — does not.

dev.to connection: "Software Supply Chain Security: Keeping Your (Rust) Dependencies Clean" (2025) explored the broader question of trust in your dependency chain. The auth model is the same question applied to runtime: do you trust the engine to act on behalf of all users, or do you make each user prove their identity to every system they touch? The answer should be the same for both: trust nothing, verify everything.

AI Logbook: The AI generated working STS assume-role code for Path 2 in twenty minutes — code we then discarded when the human decided the IAM operational model was unworkable. The `Authenticator` struct with its `OidcPassword` and `ClientCredentials` backends, the `SessionManager` with `DashMap`, and the background token refresh task were all AI-implemented from a spec that described the three authentication paths. The human chose Path 3 (bearer passthrough) after two days of evaluating the alternatives; the AI implemented whichever path was specified without questioning the security trade-offs.

Speaking Arrow

> The wire protocol is the user experience. Everything else is implementation detail.

The engine can parse SQL. It can authenticate users. It can plan queries, push them through DataFusion, and produce Arrow record batches. None of that matters if clients can't talk to it.

The wire protocol – the thing between the client and the engine – determines everything a user actually feels. Latency. Compatibility. The quality of error messages. Whether their favourite tool works at all. You can build the most elegant query engine in the world, and if the only way to reach it is a custom binary protocol that nobody supports, it's a toy.

We needed to pick a protocol. The choice shaped the entire surface area of SQE.

The Obvious Options (and Why We Rejected Them)

REST

The default reflex for any modern service: slap a JSON API on it. POST a query, GET the results. Easy to build, easy to debug with curl, supported everywhere.

The problem is serialisation. A query engine's job is to produce columnar data – Arrow record batches. A REST API would mean converting those batches to JSON on the server, sending them over the wire as text, and then converting them back to columnar format on the client. For a result set with a million rows and twenty columns, that's three serialisation steps where zero are needed.

JSON also destroys type fidelity. A Decimal(38,18) becomes a floating-point number. A timestamp loses its timezone. A null in a struct column becomes ambiguous. You can work around all of this with careful schema metadata, but at that point you're building a type system on top of a format that doesn't have one.

We did end up building a REST-ish interface – the Trino-compatible HTTP endpoint – but it's a compatibility layer for tools that only speak Trino wire protocol, not the primary interface. More on that in a moment.

JDBC Directly

JDBC is the standard for database connectivity in the Java world. Every SQL tool on the planet speaks it. DBeaver, IntelliJ, Tableau, dbt (through the Python DBAPI bridge) – if your engine has a JDBC

driver, you're immediately compatible with the entire ecosystem.

But JDBC is a Java API, not a wire protocol. The JDBC specification defines interfaces (`Connection`, `Statement`, `ResultSet`) that a driver implements. What goes over the wire is up to the driver. PostgreSQL uses its own binary protocol. MySQL uses its own. Trino uses HTTP/JSON. Each driver is a custom implementation tightly coupled to a specific engine.

Building a custom JDBC driver means writing and maintaining a Java library that speaks a custom protocol. You're shipping two codebases: the engine in Rust, the driver in Java. Every schema change, every new SQL feature, every error code needs to be coordinated across both. For a small team, that's a tax you pay on every feature forever.

gRPC

Getting warmer. gRPC gives you binary serialisation (protobuf), HTTP/2 multiplexing, streaming, and client generation in every major language. It's what Kubernetes uses. It's what most modern infrastructure speaks.

But raw gRPC is just a transport. You'd still need to define the protobuf messages for query requests, result schemas, record batches, metadata discovery, authentication, and error handling. You'd be inventing a protocol from scratch – one that no existing tool supports.

Unless someone already defined that protocol for you.

Arrow Flight SQL

Arrow Flight is a gRPC-based protocol, designed by the Apache Arrow project, for moving Arrow-formatted data between processes. It defines a small set of RPCs – `Handshake`, `GetFlightInfo`, `DoGet`, `DoPut`, `DoAction` – that handle authentication, metadata exchange, and bidirectional data streaming.

Arrow Flight SQL is a layer on top of Flight that adds SQL semantics. It defines specific protobuf messages for executing SQL statements, discovering catalog metadata (schemas, tables, columns, type info), creating prepared statements, and handling transactions. It's a complete SQL client protocol built on Arrow IPC over gRPC.

The key property: **zero serialisation overhead for query results.** The engine produces Arrow record batches internally. Flight SQL sends those batches over the wire in Arrow IPC format – the same in-memory layout, byte-for-byte. The client receives them and can work with them directly. No JSON. No text parsing. No type conversion.

There's a practical benefit too. The Arrow Flight SQL JDBC driver exists. It's maintained by the Apache Arrow project. Any tool that speaks JDBC can connect to any Flight SQL server through this driver. One driver for every Flight SQL engine. We get JDBC compatibility without writing a JDBC driver.

Protocol	Serialisation overhead	Ecosystem support	Implementation cost
REST/JSON	High (text encode/decode)	Universal	Low
Custom JDBC	None (custom binary)	Java only	High (maintain driver)
Raw gRPC	Low (protobuf)	Narrow (custom clients)	Medium
Flight SQL	Zero (Arrow IPC)	Growing (JDBC driver, ADBC, Python, Rust, Go)	Medium

Flight SQL was the only option that gave us zero-copy results *and* broad client compatibility without maintaining a custom driver.

The FlightSqlService Trait

The `arrow-flight` crate provides a Rust trait called `FlightSqlService` with over twenty methods. Each method corresponds to a Flight SQL RPC endpoint. The trait has default implementations that return `Unimplemented` for everything, so you only need to override the methods you care about.

Here is the struct that implements it in SQE:

```
#[derive(Clone)]
pub struct SqeFlightSqlService {
    session_manager: Arc<SessionManager>,
    query_handler: Arc<QueryHandler>,
    config: SqeConfig,
    worker_registry: Option<Arc<WorkerRegistry>>,
    query_tracker: Arc<QueryTracker>,
    worker_secret: String,
}
```

Six fields. The `SessionManager` handles OIDC authentication and session state. The `QueryHandler` routes SQL through parsing, planning, policy enforcement, and DataFusion execution. The `WorkerRegistry` tracks distributed workers (optional – absent in single-node mode). The `QueryTracker` records query history for the `system.runtime.queries` virtual table.

Of the 20+ trait methods, SQE implements these:

Method	Purpose	SQE behaviour
`do_handshake`	Authentication	OIDC password grant via Basic auth
`get_flight_info_statement`	Plan a SQL query	Plans query, returns schema + ticket
`do_get_statement`	Execute and stream results	Runs query, streams Arrow batches

Method	Purpose	SQE behaviour
`do_get_fallback`	Handle custom ticket types	Executes our `FetchResults` tickets
`get_flight_info_catalogs`	List catalogs	Returns warehouse name
`do_get_catalogs`	Fetch catalog data	Returns warehouse from config
`get_flight_info_schemas`	List schemas metadata	Returns schema for schema listing
`do_get_schemas`	Fetch schema names	Runs `SHOW SCHEMAS` internally
`get_flight_info_tables`	List tables metadata	Returns schema for table listing
`do_get_tables`	Fetch table names	Enumerates all schemas and tables
`do_get_table_types`	Supported table types	Returns `["TABLE", "VIEW"]`
`get_flight_info_sql_info`	Server capabilities	Reports name, version, Arrow version
`do_get_sql_info`	Fetch server info data	Builds `SqlInfoData` response
`get_flight_info_xdbc_type_info`	Type system metadata	Returns type info schema
`do_get_xdbc_type_info`	Fetch type details	Reports all supported SQL types
`get_flight_info_prepared_statement`	Prepared statement metadata	Plans query, returns schema
`do_get_prepared_statement`	Execute prepared statement	Decodes handle, runs query
`do_put_statement_update`	Execute DML (INSERT, etc.)	Runs statement, returns row count
`do_put_statement_ingest`	Bulk data upload	Decodes Arrow stream, writes to table
`do_put_prepared_statement_query`	Bind parameters	Returns handle unchanged (no-op)
`do_put_prepared_statement_update`	Execute prepared DML	Decodes handle, runs statement
`do_action_create_prepared_statement`	Create prepared statement	Plans query, stores SQL in handle
`do_action_close_prepared_statement`	Close prepared statement	No-op (stateless handles)
`do_action_cancel_query`	Cancel running query	Cancels via QueryTracker
`do_action_fallback`	Custom actions	Handles worker heartbeats

And these are explicitly `Unimplemented`:

Method	Why skipped
`get_flight_info_substrait_plan`	No Substrait support
`do_put_substrait_plan`	No Substrait support
`do_action_create_prepared_substrait_plan`	No Substrait support
`do_action_begin_transaction`	No transaction support

Method	Why skipped
`do_action_end_transaction`	No transaction support
`do_action_begin_savepoint`	No savepoint support
`do_action_end_savepoint`	No savepoint support

The pattern: we implement everything a SQL client needs to discover metadata, execute queries, upload data, and manage prepared statements. We skip Substrait (an alternative query representation we don't use) and transactions (Iceberg commits are atomic per-table, not cross-table).

DataFusion deep dive: The `FlightSqlService` trait has a `type FlightService` associated type that must be `Self`. This is how the `arrow-flight` crate connects the SQL-layer trait to the underlying gRPC `FlightService` implementation. When you implement `FlightSqlService`, you automatically get the `FlightService` gRPC methods routed to the right SQL-specific handlers based on the protobuf message types in the request. The routing logic is in the trait's default `do_get` and `do_put` implementations – they decode the `Any` message in the ticket and dispatch to the specific handler.

The Three-Phase Pipeline

Every SQL query follows the same three-phase pipeline through Flight SQL: Handshake, GetFlightInfo, DoGet.

Phase 1: Handshake

The client connects and authenticates. In SQE, this means OIDC password grant – the client sends a username and password, and the coordinator exchanges them for a JWT via Keycloak (or whatever OIDC provider is configured).

```
async fn do_handshake(
    &self,
    request: Request<Streaming<HandshakeRequest>>,
) -> Result<
    Response<Pin<Box<dyn Stream<Item = Result<HandshakeResponse, Status>> + Send>>>,
    Status,
> {
    // Extract Basic auth: base64(username:password)
    let authorization = request
        .metadata()
        .get("authorization")
        .ok_or_else(|| Status::invalid_argument("Authorization header not present"))?
        .to_str()
        .map_err(|e| Status::internal(format!("Authorization header not parsable: {e}")))?
        .to_string();

    // ... decode base64, split on ':', extract username and password ...
```

```
    let session = self
        .session_manager
        .authenticate(username, password)
        .await
        .map_err(|e| {
            warn!(username = username, error = %e, "Authentication failed");
            Status::unauthenticated(format!("Authentication failed: {e}"))
        })?;

    let result = HandshakeResponse {
        protocol_version: 0,
        payload: session.id.as_bytes().to_vec().into(),
    };

    let output = futures::stream::iter(vec![Ok(result)]);
    let token = format!("Bearer {}", session.id);
    let mut response: Response<Pin<Box<dyn Stream<Item = _> + Send>>> =
        Response::new(Box::pin(output));
    response.metadata_mut().append(
        "authorization",
        MetadataValue::from_str(&token)
            .map_err(|e| Status::internal(format!("Failed to create auth metadata: {e}")))?,
    );

    Ok(response)
}
```

The handshake does three things:

1. Decodes the Basic auth header (base64-encoded `username:password`)
2. Calls `session_manager.authenticate()`, which performs the OIDC password grant against Keycloak and creates a session holding the JWT
3. Returns the session ID as a bearer token in both the response payload and the `authorization` response header

The client stores this token and sends it as `Authorization: Bearer <session-id>` on every subsequent request. The Flight SQL JDBC driver handles this automatically.

There's a second authentication path that we added later. Some clients – particularly backend services that have already obtained a JWT through their own OIDC flow – want to skip the handshake entirely and just send the JWT as a bearer token. The `get_session_from_request` method handles both:

```
fn get_session_from_request<T>(
    &self,
    request: &Request<T>,
) -> Result<Arc<sqe_core::Session>, Status> {
    let token = // ... extract bearer token from Authorization header ...

    // Try session lookup first (handshake flow)
```

```
    if let Some(session) = self.session_manager.get_session(token) {
        return Ok(session);
    }

    // If the token looks like a JWT (contains dots), treat it as a raw
    // access token -- create an ad-hoc session
    if token.contains('.') {
        let username = metadata
            .get("x-trino-user")
            .and_then(|v| v.to_str().ok())
            .unwrap_or("unknown")
            .to_string();
        let session = sqe_core::Session::new(
            username,
            token.to_string(),
            None,
            chrono::Utc::now() + chrono::Duration::hours(1),
            vec![],
        );
        return Ok(Arc::new(session));
    }

    Err(Status::unauthenticated("Invalid or expired session token"))
}
```

First it checks if the token is a session ID from a prior handshake. If not, it checks if the token looks like a JWT (JWTs always contain dots as delimiters between header, payload, and signature). If it's a JWT, it creates an ad-hoc session with that token directly. This is the same pattern the Trino-compat HTTP endpoint uses, which means backend services can use either protocol with the same authentication approach.

Phase 2: GetFlightInfo

The client sends a SQL query and gets back metadata – the result schema and a ticket for fetching the actual data. No execution happens yet.

```
async fn get_flight_info_statement(
    &self,
    query: CommandStatementQuery,
    request: Request<FlightDescriptor>,
) -> Result<Response<FlightInfo>, Status> {
    let session = self.get_session_from_request(&request)?;
    let sql = &query.query;

    // Encode the SQL into a ticket so do_get can find it later
    let fetch = FetchResults {
        handle: sql.clone(),
    };
```

```
    let ticket = Ticket {
        ticket: fetch.as_any().encode_to_vec().into(),
    };
    let endpoint = FlightEndpoint {
        ticket: Some(ticket),
        location: vec![],
        expiration_time: None,
        app_metadata: vec![].into(),
    };

    // Plan the query to extract the schema without executing it
    let schema = self
        .query_handler
        .get_schema(&session, sql)
        .await
        .map_err(|e| Status::internal(format!("Query planning failed: {e}")))?;

    let info = FlightInfo::new()
        .try_with_schema(&schema)
        .map_err(|e| Status::internal(format!("Failed to encode schema: {e}")))?
        .with_descriptor(FlightDescriptor::new_cmd(vec![]))
        .with_endpoint(endpoint)
        .with_total_records(-1)
        .with_ordered(false);

    Ok(Response::new(info))
}
```

Two things happen here. First, the coordinator calls `query_handler.get_schema()`, which plans the query through DataFusion (including policy enforcement and optimization) and extracts the output schema without actually executing it. This gives the client the column names and types before any data flows.

Second, it encodes the SQL string into a `FetchResults` protobuf message and wraps it in a `Ticket`. The `FlightInfo` response contains this ticket inside a `FlightEndpoint`. When the client wants the actual data, it presents this ticket back to the server.

The `total_records` is -1, meaning "unknown". We could execute the query during GetFlightInfo and report the exact count, but that would mean running the query twice (or caching the entire result set in memory). We chose not to.

The `location` field in the endpoint is empty, which means "fetch from the same server." In a distributed setup, you could return different locations pointing to different workers – but SQE's distributed execution is handled internally by the coordinator, not by directing clients to specific workers. The client always talks to the coordinator.

Dead end: executing during GetFlightInfo. We initially tried executing the full query during `get_flight_info_statement` and caching the result batches, so `do_get_statement` could just stream

them from memory. This worked for small result sets but fell apart on queries returning millions of rows – the coordinator would hold the entire result in memory between the two calls. Worse, some clients wait minutes between GetFlightInfo and DoGet (DBeaver displays the schema in a tab before the user clicks "fetch data"). We switched to plan-only during GetFlightInfo and execute-on-demand during DoGet.

Phase 3: DoGet

The client presents the ticket and receives a stream of Arrow record batches.

```
async fn do_get_statement(
    &self,
    ticket: TicketStatementQuery,
    request: Request<Ticket>,
) -> Result<Response<<Self as FlightService>::DoGetStream>, Status> {
    let session = self.get_session_from_request(&request)?;
    let sql = &ticket.statement_handle;

    let sql_str = std::str::from_utf8(sql)
        .map_err(|e| Status::internal(format!("Invalid statement handle: {e}")))?;

    let batches = self
        .query_handler
        .execute(&session, sql_str)
        .await
        .map_err(|e| Status::internal(format!("Query execution failed: {e}")))?;

    Self::batches_to_stream(batches)
}
```

The handler extracts the SQL from the ticket, executes it through the full pipeline (parse, plan, policy enforcement, DataFusion execution, possibly distributed to workers), and converts the resulting `Vec<RecordBatch>` into a `FlightStream`.

The `batches_to_stream` helper converts record batches into Flight data frames:

```
fn batches_to_stream(
    batches: Vec<RecordBatch>,
) -> Result<Response<FlightStream>, Status> {
    if batches.is_empty() {
        let stream = futures::stream::empty();
        let flight_stream: FlightStream = Box::pin(stream);
        return Ok(Response::new(flight_stream));
    }

    let schema = batches[0].schema();
    let flight_data = batches_to_flight_data(&schema, batches)
        .map_err(|e| Status::internal(format!("Failed to encode flight data: {e}")))?
        .into_iter()
```

```
        .map(Ok);

    let stream: FlightStream = Box::pin(stream::iter(flight_data));
    Ok(Response::new(stream))
}
```

The `batches_to_flight_data` function (from the `arrow-flight` crate) converts Arrow record batches into the Arrow IPC format used by Flight. The schema is sent first as a separate message, followed by the data for each batch. On the client side, the Flight library decodes these back into record batches – same schema, same memory layout.

There's a subtle bug we hit early on, captured in the comment: "Using Schema::empty() here caused clients to hang because get_flight_info sends the real query schema but do_get sent a 0-column schema, confusing the FlightRecordBatchStream decoder." When a query returns zero rows, you still need to return a stream that's consistent with the schema you promised in GetFlightInfo. We solved it by returning a genuinely empty stream – no schema message at all – rather than an empty schema.

The Fallback: Tickets That Don't Match

Not all Flight SQL clients use the standard `TicketStatementQuery` message. Some older clients, and some that pre-date the full Flight SQL specification, send raw tickets with custom protobuf messages. SQE handles this through `do_get_fallback`:

```
async fn do_get_fallback(
    &self,
    request: Request<Ticket>,
    message: Any,
) -> Result<Response<<Self as FlightService>::DoGetStream>, Status> {
    let session = self.get_session_from_request(&request)?;

    if message.type_url == FetchResults::type_url() {
        let fetch: FetchResults = Message::decode(&*message.value)
            .map_err(|e| Status::internal(format!("Failed to decode ticket: {e}")))?;

        let batches = self
            .query_handler
            .execute(&session, &fetch.handle)
            .await
            .map_err(|e| Status::internal(format!("Query execution failed: {e}")))?;

        return Self::batches_to_stream(batches);
    }

    Err(Status::unimplemented(format!(
        "Unsupported ticket type: {}",
        message.type_url
    )))
```

```
}
```

The `FetchResults` is a custom protobuf message defined in the SQE codebase. It carries a single string field: the SQL query handle. When `get_flight_info_statement` creates a ticket, it encodes the SQL into this message. The fallback handler checks the `type_url`, decodes the message, and executes the query. This is the path most JDBC clients actually take, because the Arrow Flight SQL JDBC driver uses the generic `DoGet` RPC, not the SQL-specific `DoGetStatement` one.

Field report: The Arrow Flight SQL JDBC driver (version 15.0) routes all queries through `do_get_fallback`, not `do_get_statement`. We discovered this during the first DBeaver test. The standard `do_get_statement` handler was never hit. Without the fallback, DBeaver connected successfully, planned queries, displayed schemas, and then returned zero rows for every query. The fix was straightforward once we understood the routing, but it took an afternoon of packet captures to figure out why.

The Metadata Surface

SQL clients don't just execute queries. Before a user types their first SELECT, the client tool has already made a dozen metadata calls: "What catalogs exist? What schemas? What tables? What are the column types? What SQL features does this server support?"

DBeaver is particularly thorough. On connection, it calls `GetSqlInfo`, `GetCatalogs`, `GetDbSchemas`, `GetTables`, `GetTableTypes`, and `GetXdbcTypeInfo` – all before the user has even opened a query tab.

SQE implements all of these. For catalogs, it returns the warehouse name from the config. For schemas, it runs `SHOW SCHEMAS` internally. For tables, it iterates every schema and runs `SHOW TABLES` in each one – not the most efficient approach, but it works and it respects the user's access permissions because every internal query runs through the same session and policy enforcement.

The type info endpoint is surprisingly dense. JDBC clients use it to understand what SQL types the server supports, how they map to XDBC types, what their precision and scale ranges are, and how literals are formatted. SQE reports boolean, tinyint, smallint, integer, bigint, real, double, decimal, varchar, varbinary, date, time, and timestamp – each with its XDBC metadata:

```
builder.append(XdbcTypeInfo {
    type_name: "decimal".into(),
    data_type: XdbcDataType::XdbcDecimal,
    column_size: Some(38),
    create_params: Some(vec!["precision".into(), "scale".into()]),
    nullable: Nullable::NullabilityNullable,
    case_sensitive: false,
    searchable: Searchable::Full,
    fixed_prec_scale: true,
    sql_data_type: XdbcDataType::XdbcDecimal,
    minimum_scale: Some(0),
    maximum_scale: Some(38),
    num_prec_radix: Some(10),
```

```
    ..Default::default()
});
```

This is the kind of code that doesn't make it into conference talks. Fifteen type definitions, each with a dozen fields. No cleverness. Just correctness. But if any of these are wrong – if you report `maximum_scale: 18` when DataFusion actually supports 38 – some JDBC client somewhere will silently truncate your decimal values and you'll spend a week figuring out why your financial calculations don't round-trip.

Connecting From Everywhere

DBeaver

DBeaver connects through the Arrow Flight SQL JDBC driver. The connection configuration:

- **Driver**: Apache Arrow Flight SQL
- **URL**: `jdbc:arrow-flight-sql://localhost:50051?useEncryption=false`
- **Username/Password**: OIDC credentials (passed to `do_handshake`)

On connection, DBeaver walks the metadata surface, builds its schema tree, and is ready for queries. Every query goes through the GetFlightInfo/DoGet pipeline. The JDBC driver handles the streaming, buffering, and type conversion from Arrow to Java `ResultSet` objects.

The `useEncryption=false` parameter disables TLS for local development. In production, SQE supports TLS through a configurable certificate and key in the TOML config, and the URL becomes `jdbc:arrow-flight-sql://coordinator.internal:50051`.

Python (adbc_driver_flightsql)

ADBC (Arrow Database Connectivity) is the Arrow-native replacement for ODBC/JDBC. The `adbc_driver_flightsql` package provides a Python client that speaks Flight SQL and returns results as PyArrow tables – zero copy from the engine to pandas.

```
import adbc_driver_flightsql.dbapi as flight_sql

conn = flight_sql.connect(
    "grpc://localhost:50051",
    db_kwargs={
        "username": "alice",
        "password": "secret",
    }
)

cursor = conn.cursor()
cursor.execute("SELECT * FROM warehouse.orders LIMIT 10")

# Returns a PyArrow Table -- columnar, zero-copy
```

```
table = cursor.fetch_arrow_table()
df = table.to_pandas()
```

The driver calls `do_handshake` with the credentials, stores the bearer token, and uses it for all subsequent operations. `cursor.execute()` calls `GetFlightInfo`, and `fetch_arrow_table()` calls `DoGet`. The Arrow data arrives in the same IPC format that DataFusion produced. No JSON. No pandas-to-Arrow conversion. The bytes that left the engine are the bytes that land in the DataFrame.

For programmatic clients that manage their own OIDC flow:

```
import adbc_driver_flightsql.dbapi as flight_sql

conn = flight_sql.connect(
    "grpc://localhost:50051",
    db_kwargs={
        "adbc.flight.sql.authorization_header": f"Bearer {jwt_token}",
    }
)
```

This skips the handshake entirely and passes the JWT directly. SQE's `get_session_from_request` detects the JWT format and creates an ad-hoc session.

Rust (arrow-flight client)

The `sqe-cli` crate shows the Rust client pattern:

```
pub struct FlightClient {
    inner: FlightSqlServiceClient<Channel>,
}

impl FlightClient {
    pub async fn connect(
        url: &str,
        username: &str,
        password: &str,
    ) -> Result<Self, Box<dyn std::error::Error>> {
        let channel = build_channel(url).await?;
        let mut inner = FlightSqlServiceClient::new(channel);

        let token = inner
            .handshake(username, password)
            .await
            .map_err(|e| format!("Authentication failed: {e}"))?;

        inner.set_token(String::from_utf8(token.to_vec())?);
        Ok(Self { inner })
    }
}
```

The `FlightSqlServiceClient` from the `arrow-flight` crate handles all the gRPC plumbing. `hand-`

`shake()` sends the Basic auth, receives the bearer token, and `set_token()` stores it for subsequent calls. Query execution is two calls:

```
async fn execute(&mut self, sql: &str) -> Result<QueryResult, Box<dyn std::error::Error>> {
    let info = self
        .inner
        .execute(sql.to_string(), None)
        .await
        .map_err(|e| format!("Query failed: {e}"))?;

    let mut all_batches: Vec<RecordBatch> = Vec::new();

    for endpoint in info.endpoint {
        if let Some(ticket) = endpoint.ticket {
            let stream = self
                .inner
                .do_get(ticket)
                .await
                .map_err(|e| format!("Failed to fetch results: {e}"))?;

            let batches: Vec<RecordBatch> = stream.try_collect().await?;
            all_batches.extend(batches);
        }
    }

    batches_to_result(&all_batches)
}
```

The `execute()` call maps to GetFlightInfo. The response can contain multiple endpoints (in distributed scenarios, pointing to different servers), each with a ticket. The client iterates through all endpoints, calls `do_get` on each, and collects the record batches. For SQE in single-node mode, there's always exactly one endpoint pointing back to the coordinator.

The Trino-Compat Escape Hatch

Some tools only speak the Trino HTTP protocol. For those, SQE runs a second server on a separate port that implements the Trino wire protocol – `POST /v1/statement` with JSON responses and `nextUri` pagination.

The Trino-compat layer isn't Flight SQL. It's the JSON-serialised protocol we rejected at the start of this chapter. But it exists because pragmatism beats purity. If a team's existing tooling only speaks Trino, they can connect to SQE without changing anything. The performance cost is real – JSON serialisation, text parsing, type coercion – but it's a migration bridge, not the primary interface.

The CLI supports both:

```
sqe-cli --protocol flight -e "SELECT 1"    # Arrow Flight SQL (default)
sqe-cli --protocol http -e "SELECT 1"      # Trino-compat HTTP
```

Streaming Results and Backpressure

Flight SQL runs on gRPC, which runs on HTTP/2. HTTP/2 has flow control built in – the client can signal the server to slow down when it can't consume data fast enough. This is backpressure, and it matters.

Consider a query that scans a billion-row table. The engine can produce record batches far faster than most clients can process them. Without backpressure, the server would buffer the entire result set in memory, waiting for the client to catch up. With HTTP/2 flow control, the server's send buffer fills up, the gRPC stream blocks, DataFusion's execution pauses at the point where it would produce the next batch, and no memory is wasted on buffering.

SQE's current implementation collects all record batches into memory before streaming:

```
let batches = self
    .query_handler
    .execute(&session, sql_str)
    .await?;

Self::batches_to_stream(batches)
```

This means the full result materialises in the coordinator's memory before the first byte reaches the client. For most analytical queries – which return aggregated results, filtered subsets, or sampled data – this is fine. For queries that return millions of rows, it's a problem.

The fix is straightforward in principle: instead of `execute()` returning a `Vec<RecordBatch>`, it would return a `SendableRecordBatchStream` – DataFusion's native streaming type – and we'd convert that directly into a Flight stream. Each batch would flow from DataFusion through the gRPC stream to the client as it's produced, with HTTP/2 backpressure preventing the coordinator from running ahead.

We haven't done this yet. It's a clear next step, and it's the kind of improvement that becomes urgent exactly once someone runs `SELECT * FROM a_very_large_table` in production.

DataFusion deep dive: DataFusion's `collect()` function gathers all batches into a Vec. For streaming, you'd use `execute_stream()` on the physical plan, which returns a `SendableRecordBatchStream`. The `FlightDataEncoderBuilder` in the `arrow-flight` crate can wrap this stream directly, producing a `Stream<Item = Result<FlightData, FlightError>>` that maps one-to-one onto the gRPC response stream. The types line up. The plumbing is waiting. It's an afternoon of work that we keep deprioritising in favour of features with more visible impact.

Error Propagation

When a query fails mid-stream – a Parquet file is corrupted, a worker crashes, a token expires – the error needs to reach the client. gRPC has a native mechanism for this: the `Status` type, which carries a status code and a message.

For errors during planning (GetFlightInfo), this is straightforward. The handler returns `Err(Status::internal("Query planning failed: ..."))`, and the gRPC framework sends it as a response with an error code. The client gets a clean error message.

For errors during streaming (DoGet), it's slightly different. The stream is already open. The client has already received the schema and possibly some batches. An error is sent as the final message in the stream, which terminates it. The Flight SQL client libraries handle this and surface it as an exception.

SQE maps internal errors to gRPC status codes:

Situation	gRPC status
Missing or invalid auth	`UNAUTHENTICATED`
Malformed request	`INVALID_ARGUMENT`
Query planning failure	`INTERNAL`
Execution failure	`INTERNAL`
Unsupported operation	`UNIMPLEMENTED`

We don't use `PERMISSION_DENIED` for policy violations. The security model (covered in Chapter 8) is designed around *invisible* denial – columns you can't see simply don't appear in the schema, rows you can't access are silently filtered. The client never receives an error saying "you don't have permission to read column X" because the client never knows column X exists.

Mounting the Service

The Flight SQL service is a tonic gRPC server. Setting it up is a few lines:

```
let flight_service =
    SqeFlightSqlService::new(session_manager, query_handler, config.clone());

let addr = format!("0.0.0.0:{}", config.coordinator.flight_sql_port).parse()?;

tonic::transport::Server::builder()
    .add_service(
        arrow_flight::flight_service_server::FlightServiceServer::new(flight_service)
    )
    .serve(addr)
    .await?;
```

Note the type wrapping: `SqeFlightSqlService` implements `FlightSqlService`, but the tonic server expects a `FlightService` (the raw gRPC trait). The `FlightServiceServer::new()` call takes anything that implements `FlightService`, and the `FlightSqlService` trait provides a blanket implementation of `FlightService` that routes incoming gRPC calls to the appropriate SQL-specific methods.

In production, optional TLS is layered on top:

```
let tls_config = sqe_coordinator::tls::build_server_tls_config(&config.coordinator.tls)?;

let mut server_builder = tonic::transport::Server::builder();
if let Some(tls) = tls_config {
    server_builder = server_builder.tls_config(tls)?;
}
```

The coordinator also starts the Trino-compat HTTP server on a separate port, the Prometheus metrics server on a third port, and optionally a worker health-check background task. Four network surfaces, each doing one thing, each on its own port. The Flight SQL port is the primary interface. Everything else is supplementary.

The Wire Protocol Is the User Experience

I started this chapter by saying the wire protocol is the user experience. After building this, I believe it more strongly than before.

The choice of Flight SQL determined which clients work out of the box (DBeaver, any JDBC tool, Python via ADBC, Rust via arrow-flight). It determined the serialisation overhead (none for Arrow-aware clients). It determined how authentication flows (gRPC metadata headers carrying bearer tokens). It determined how errors surface (gRPC status codes). It determined whether backpressure is possible (yes, because HTTP/2).

It also determined what we didn't have to build. We didn't write a JDBC driver. We didn't invent a protocol. We didn't build a type-mapping layer between our internal representation and the wire format. The engine produces Arrow. The wire carries Arrow. The client receives Arrow. Every step that doesn't exist is a step that can't have bugs.

The 20+ methods in the FlightSqlService trait looked intimidating at first. Most of them are metadata endpoints that JDBC clients expect – catalogs, schemas, tables, types. Once we understood the pattern (plan in GetFlightInfo, execute in DoGet, metadata via dedicated endpoints), the implementation was mechanical. The interesting code is all in the `QueryHandler` and the `SessionManager`. The Flight SQL layer is plumbing.

Good plumbing is invisible. Users don't think about the wire protocol. They open DBeaver, type a connection string, and run queries. The protocol's job is to never be the reason something doesn't work. Flight SQL has held up its end of that bargain.

Field report: The first integration test – Flight SQL handshake with OIDC, followed by a SELECT query against an Iceberg table via Polaris – passed on March 14, the same day the crates were scaffolded. From empty repository to authenticated query results over Arrow Flight SQL in one day. The protocol was not the hard part. Polaris credential vending was the hard part. The protocol just worked.

AI Logbook: The AI implemented all 24 `FlightSqlService` trait methods — including the metadata surface for catalogs, schemas, tables, and XDBC type info — across two sessions. The human decided which methods to implement and which to leave as `Unimplemented` (Substrait, transactions). The `do_get_fallback` handler that turned out to be the actual path JDBC clients use was discovered during the first DBeaver test by the human; the AI had implemented the standard `do_get_statement` path first, which no real client hit.

The Catalog Is the API

> Your BI tool doesn't read your code. It reads your catalog. Make it worth reading.

The engine worked. You could connect from DBeaver, authenticate via OIDC, and run a query that hit Polaris, fetched credentials, scanned Parquet from S3, and returned Arrow batches over Flight SQL. Chapter 3 through 5, working end to end.

Then someone on the team opened DBeaver's schema browser and saw nothing.

No tables. No schemas. No columns. An empty tree. The connection was live – queries returned results if you typed them by hand – but the tool's metadata browser was blank. DBeaver didn't know what existed because we hadn't told it. The engine had no `information_schema`.

This is the gap between "queries work" and "tools work." Every database client, every BI tool, every ORM, every orchestrator discovers your warehouse through the same mechanism: SQL metadata tables. `information_schema.tables`. `information_schema.columns`. `information_schema.schemata`. These aren't optional. They're the API contract between your engine and the entire ecosystem of tools that will talk to it.

The engine that runs queries but can't describe itself is an engine nobody will use.

What Tools Actually Ask For

Before writing any code, we studied what clients actually query when they connect. We captured SQL from three sources: DBeaver connecting via Flight SQL, a Python ADBC client running `cursor.columns()`, and dbt running `dbt debug`. The patterns converge fast.

DBeaver's first action after the handshake is to query `information_schema.schemata` to populate its schema tree. Then, when you expand a schema, it queries `information_schema.tables` filtered by `table_schema`. Click a table, it queries `information_schema.columns` filtered by `table_schema` and `table_name`. The tool never asks "what can you do?" – it asks "what do you have?"

The SQL standard defines three views:

View	Purpose
`information_schema.schemata`	What schemas (namespaces) exist
`information_schema.tables`	What tables exist, in which schemas
`information_schema.columns`	What columns each table has, their types, nullability

Every tool expects them. They're the HTTP of database discovery – a standard interface that doesn't care what the engine is built from.

Building a Virtual Provider

The trick is that `information_schema` doesn't point at storage. There's no Parquet file holding the list of your tables. The data comes from the catalog – from Polaris, in our case – and it's different for every user session, because Polaris scopes the response to what that user's bearer token can access.

This means we can't register a static table at startup. We need a virtual table provider – something that looks like a regular DataFusion table to the query engine but generates its results dynamically by calling the catalog API.

DataFusion's extensibility model made this straightforward. We implemented `InformationSchemaProvider` as a `SchemaProvider` that returns three virtual tables:

```
#[derive(Debug)]
pub struct InformationSchemaProvider {
    session_catalog: Arc<SessionCatalog>,
    warehouse: String,
}

#[async_trait]
impl SchemaProvider for InformationSchemaProvider {
    fn table_names(&self) -> Vec<String> {
        vec![
            "tables".to_string(),
            "columns".to_string(),
            "schemata".to_string(),
        ]
    }

    async fn table(&self, name: &str) -> DFResult<Option<Arc<dyn TableProvider>>> {
        match name {
            "tables" => Ok(Some(self.build_tables_table().await?)),
            "columns" => Ok(Some(self.build_columns_table().await?)),
            "schemata" => Ok(Some(self.build_schemata_table().await?)),
            _ => Ok(None),
        }
    }
}
```

Each `build_*` method calls Polaris through the `SessionCatalog` – which carries the user's bearer token – lists namespaces and tables, and assembles the results into Arrow `RecordBatch` instances wrapped in a `MemTable`. The table is ephemeral: it's constructed fresh for each query, reflecting the current state of the catalog at that moment.

DataFusion deep dive: DataFusion's `SchemaProvider` trait is async on the `table()` method but

sync on `table_names()`. This forced a design choice: we use `tokio::task::block_in_place` in `table_names()` to bridge async catalog calls into the synchronous context. For `information_schema`, we avoided this by hardcoding the three known table names – no catalog call needed to know that `tables`, `columns`, and `schemata` exist.

The Tables Table

The `information_schema.tables` implementation demonstrates the pattern. Walk every namespace, list every table, emit one row per table with the standard columns:

```
async fn build_tables_table(&self) -> DFResult<Arc<dyn TableProvider>> {
    let schema = Arc::new(Schema::new(vec![
        Field::new("table_catalog", DataType::Utf8, false),
        Field::new("table_schema", DataType::Utf8, false),
        Field::new("table_name", DataType::Utf8, false),
        Field::new("table_type", DataType::Utf8, false),
    ]));

    let namespaces = self.list_namespaces_safe().await;

    let mut catalog_builder = StringBuilder::new();
    let mut schema_builder = StringBuilder::new();
    let mut name_builder = StringBuilder::new();
    let mut type_builder = StringBuilder::new();

    for ns in &namespaces {
        let ns_ident = NamespaceIdent::new(ns.clone());
        match self.session_catalog.list_tables(&ns_ident).await {
            Ok(tables) => {
                for table in &tables {
                    catalog_builder.append_value(&self.warehouse);
                    schema_builder.append_value(ns);
                    name_builder.append_value(table.name());
                    type_builder.append_value("BASE TABLE");
                }
            }
            Err(e) => {
                warn!(namespace = %ns, error = %e,
                    "Failed to list tables for information_schema");
            }
        }
    }

    let batch = RecordBatch::try_new(schema.clone(), vec![
        Arc::new(catalog_builder.finish()) as ArrayRef,
        Arc::new(schema_builder.finish()) as ArrayRef,
        Arc::new(name_builder.finish()) as ArrayRef,
```

```
        Arc::new(type_builder.finish()) as ArrayRef,
    ])?;

    Ok(Arc::new(MemTable::try_new(schema, vec![vec![batch]])?))
}
```

Two things to notice. First, the `list_namespaces_safe` helper wraps the catalog call in error handling that logs and returns an empty list rather than failing the entire query. A permission error on one namespace shouldn't black out the entire metadata view. Second, `table_type` is hardcoded to `"BASE TABLE"` – we're not yet distinguishing views from tables here, though the Iceberg catalog tracks them separately.

The columns table follows the same pattern but goes one level deeper: for each table, it loads the Iceberg metadata and walks the schema fields, emitting one row per column with standard columns including `table_catalog`, `table_schema`, `table_name`, `column_name`, `ordinal_position`, `is_nullable`, and `data_type`.

The Iceberg schema becomes SQL metadata through this translation. `field.required` maps to the SQL standard's `is_nullable` – `"NO"` for required fields, `"YES"` for optional. `field.field_type` renders as a string representation: Iceberg stores types as `long`, `timestamptz`, `decimal(18,2)`, and the columns table presents them as strings that tools parse into their own type models. Getting this translation right – matching the exact strings that DBeaver, dbt, and JDBC drivers expect – is the unglamorous work that makes the difference between "the engine works" and "the engine works with my tools."

Registering the Virtual Schema

Registration into DataFusion's catalog hierarchy is a one-liner in `SqeCatalogProvider`:

```
impl CatalogProvider for SqeCatalogProvider {
    fn schema_names(&self) -> Vec<String> {
        let mut names = self.cached_namespaces.clone();
        names.push("information_schema".to_string());
        names
    }

    fn schema(&self, name: &str) -> Option<Arc<dyn SchemaProvider>> {
        if name == "information_schema" {
            return Some(Arc::new(
                InformationSchemaProvider::new(
                    self.session_catalog.clone(),
                    self.warehouse.clone(),
                ),
            ));
        }
        // ... resolve real Iceberg namespaces ...
    }
```

```
}
```

The `information_schema` appears alongside real namespaces in `schema_names()`. When a query references `information_schema.tables`, DataFusion resolves it through this provider chain. The user doesn't know or care that the data is generated dynamically rather than read from storage.

One subtlety: the `SessionCatalog` inside the provider carries the user's bearer token. When Alice queries `information_schema.tables`, Polaris returns only the tables Alice can access. When Bob queries the same view, he sees his tables. The metadata view is per-user by construction, not by post-filtering. This falls out naturally from the bearer passthrough architecture described in Chapter 4.

The Caching Layers

The initial catalog implementation made a Polaris REST call for every table load. At scale factor 1, a TPC-DS query touching 15 dimension tables meant 15 network round-trips before execution started. The fix was a multi-layer caching strategy:

TableMetadataCache. A global moka cache shared across all sessions, keyed by table identifier and token fingerprint. TTL is 30 seconds — short enough that schema changes propagate within a query cycle, long enough that a 99-query benchmark run does not hammer Polaris 1,500 times.

ManifestCache. Iceberg manifest files are immutable by specification. Once written, their content never changes. We cache parsed manifest entries by S3 path with a 512 MB size limit and a 1-hour TTL backstop (for disaster recovery scenarios where a manifest is overwritten at the same path). This eliminates the most expensive I/O in scan planning.

FooterCache. Parquet file footers contain schema, row group metadata, and column statistics. We cache them by file path with size-weighted eviction and Prometheus hit/miss counters.

RestCatalog instance cache. The iceberg-rust `RestCatalog` is expensive to create (~250 ms). We cache instances by token fingerprint with a 5-minute TTL.

SessionContext cache. The DataFusion `SessionContext` with all registered UDFs, TVFs, and catalog providers is cached per token fingerprint (SHA-256). Atomic population via moka's `try_get_with` eliminates the TOCTOU race where concurrent requests build redundant contexts. Invalidated after every DDL operation.

Together, these five layers reduced warm-query overhead from ~540 ms to under 1 ms.

Beyond information_schema: The system Catalog

Standard metadata tables tell tools what data exists. But an engine operator needs to know what the engine is doing. Which queries are running? How many workers are active? Where is time being spent?

We implemented `system.runtime.queries`, `system.runtime.tasks`, and `system.runtime.nodes` — not just for Trino compatibility (that's a bonus), but because querying engine state through SQL is

the right interface. An operator shouldn't need to scrape Prometheus to answer "what query is taking so long?"

```
SELECT query_id, user, state, execution_time_ms, query
FROM system.runtime.queries
WHERE state = 'RUNNING'
ORDER BY execution_time_ms DESC
```

The data comes from the coordinator's `QueryTracker`. But there's a deliberate crate boundary: `sqe-catalog` defines its own snapshot types rather than importing from `sqe-coordinator`, avoiding a circular dependency. The bridge is a closure – a `QueryRecordsFn` of type `Arc<dyn Fn() -> Vec<RuntimeQueryRecord> + Send + Sync>` that captures the tracker and maps its records into the catalog crate's types. Every time someone queries `system.runtime.queries`, the closure fires, the tracker dumps its records, and the virtual table materializes results as a `MemTable`. Live data, on demand.

The tasks table shows execution at the fragment level. In single-node mode, each finished query gets one synthetic task row. In distributed mode, each fragment dispatched to a worker gets its own row with the real worker URL. This dual-mode behavior means the same monitoring queries work regardless of deployment topology. You don't need different dashboards for single-node dev and distributed production.

The nodes table is simpler: one row for the coordinator, one row per worker. `env!("CARGO_PKG_VERSION")` pulls the version from `Cargo.toml` at compile time. The `coordinator` boolean column lets you filter nodes by role. This is the same information you'd get from a Kubernetes service endpoint list, but queryable from SQL.

Field report: The tasks table caught a real bug. During the load test with 50 concurrent clients, we noticed some queries showing tasks on `http://worker-1:50052` that should have gone to `worker-2`. The tasks table made it visible: the fragment scheduler was hashing partition IDs to workers using a scheme that collided on certain partition ranges. We fixed the hash function, and the tasks table confirmed even distribution in the next test run. The tool that monitors the engine found the bug in the engine.

SHOW Commands: The Parser Wrapping Strategy

Some clients don't query `information_schema` at all. They use `SHOW SCHEMAS`, `SHOW TABLES`, `SHOW CATALOGS`. These are convenience commands – syntactic sugar over metadata queries – and not all of them parse cleanly through standard SQL parsers.

The problem: `SHOW CATALOGS` is not part of the SQL standard. `sqlparser-rs` handles `SHOW SCHEMAS` and `SHOW TABLES`, but `SHOW CATALOGS` gets parsed as a `SHOW VARIABLE` or fails entirely. Our custom `EXPLAIN FULL` doesn't parse because no standard dialect recognizes it.

Our parser strategy: wrap `sqlparser-rs`, don't fork it. The `sqe-sql` crate's classifier pre-scans for statements the standard parser doesn't recognize:

```
pub fn parse_and_classify(sql: &str) -> sqe_core::Result<StatementKind> {
```

```
    let trimmed = sql.trim();
    let upper = trimmed.to_uppercase();

    if upper == "SHOW CATALOGS" || upper.starts_with("SHOW CATALOGS ") {
        return Ok(StatementKind::ShowCatalogs);
    }

    if upper.starts_with("EXPLAIN FULL ") {
        let inner = trimmed["EXPLAIN FULL ".len()..].trim().to_string();
        return Ok(StatementKind::ExplainFull(inner));
    }

    // Standard parsing for everything else
    let dialect = GenericDialect {};
    let statements = Parser::parse_sql(&dialect, sql)
        .map_err(|e| sqe_core::SqeError::Execution(format!("Parse error: {e}")))?;

    // ... classify the parsed statement ...
}
```

Each SHOW handler calls the catalog directly and returns a `RecordBatch`. No query planning, no optimization, no execution engine involved. These are metadata lookups that bypass DataFusion entirely because they don't need it.

The wrapping strategy extends to custom SQL we added later. `EXPLAIN FULL` is our enhanced explain that shows partition pruning and cost estimates – it's not standard SQL, so `sqlparser-rs` would choke on it. The pre-scan catches it before the parser sees it, extracts the inner SQL, and routes it to a dedicated handler. Same pattern, no fork.

The `StatementKind` enum grew to 18 variants as features were added. The classifier has 30 test cases covering every variant, including edge cases like `CREATE OR REPLACE TABLE AS SELECT` (classified as CTAS, not CreateTable) and `ALTER TABLE RENAME COLUMN` (classified as Utility, not Rename). One thing the classifier makes explicit: the boundary between statements that go through DataFusion's planner and statements that bypass it. Queries, CTAS, INSERT, MERGE, DELETE, and EXPLAIN go through the full planning pipeline. SHOW commands, DROP, RENAME, CREATE SCHEMA, and DROP SCHEMA go directly to catalog operations. This routing decision happens once, early, and drives the rest of the execution path.

dev.to connection: The parser extension pattern follows the same principle described in "When Your SQL Engine Understands Meaning": wrap, don't fork. A forked parser is a maintenance burden that compounds over time. A wrapper that intercepts specific patterns before the standard parser runs is cheap to maintain and easy to test.

Namespace Resolution

A subtle problem sits at the intersection of catalogs and SQL parsing: how do you resolve `my_schema.my_table`?

Iceberg uses a three-level hierarchy: catalog, namespace, table. SQL tools expect the same three levels: catalog, schema, table. Our `parse_table_ref` function handles the resolution:

```
pub fn parse_table_ref(name: &ObjectName)
    -> sqe_core::Result<(NamespaceIdent, String)>
{
    let parts: Vec<String> = name.0.iter()
        .map(|ident| ident.value.clone())
        .collect();

    match parts.len() {
        1 => Ok((NamespaceIdent::new("default".to_string()), parts[0].clone())),
        2 => Ok((NamespaceIdent::new(parts[0].clone()), parts[1].clone())),
        3 => Ok((NamespaceIdent::new(parts[1].clone()), parts[2].clone())),
        n => Err(SqeError::Execution(format!(
            "Invalid table reference with {n} parts: {name}"
        ))),
    }
}
```

One part: the table name, namespace defaults to `"default"`. Two parts: namespace and table. Three parts: catalog, namespace, table – and we drop the catalog prefix because the catalog was already resolved at session creation. This mirrors how Trino and most SQL engines handle qualified names.

The "default" namespace fallback is important for interactive users. Nobody wants to type `production.finance.transactions` when `finance.transactions` works. But it creates an assumption: there must be a namespace called "default" or the single-name form fails silently. We chose this trade-off over requiring fully qualified names everywhere. The cognitive load on users matters.

The Full Catalog Hierarchy

Putting it all together, the catalog hierarchy registered in every DataFusion session:

```
<warehouse>                          (SqeCatalogProvider)
  |-- <namespace_1>                  (SqeSchemaProvider)
  |     |-- table_a                  (SqeTableProvider -> Iceberg)
  |     |-- table_b
  |     \-- view_1                   (ViewTable -> planned SQL)
  |-- <namespace_2>
  |     \-- ...
  \-- information_schema             (InformationSchemaProvider)
        |-- tables                   (virtual)
        |-- columns                  (virtual)
        \-- schemata                 (virtual)

system                               (SystemCatalogProvider)
  |-- jdbc                           (JdbcSchemaProvider)
```

```
|     |-- types
|     |-- catalogs
|     |-- schemas
|     |-- tables
|     \-- columns
|-- metadata                        (MetadataSchemaProvider)
|     |-- catalogs
|     |-- table_properties
|     |-- schema_properties
|     \-- table_comments
\-- runtime                         (RuntimeSchemaProvider)
      |-- queries                   (virtual, live)
      |-- nodes                     (virtual, live)
      \-- tasks                     (virtual, live)
```

Two catalogs. The warehouse catalog holds your data and the standard metadata views. The system catalog holds engine introspection. Both are registered per session, both respect the user's identity, and both are queryable with standard SQL.

The registration happens in two lines:

```
ctx.register_catalog(&catalog_name, Arc::new(catalog_provider));
ctx.register_catalog("system", Arc::new(system_catalog));
```

The entire metadata surface of the engine – information_schema, JDBC tables, metadata tables, runtime tables – available through standard SQL, scoped to the current user, refreshed on every query.

The Cost of Getting It Wrong

We learned how much metadata matters the hard way. In the first integration test with DBeaver, the schema browser showed tables but no columns. The `information_schema.columns` virtual table was returning empty because the `build_columns_table` method was iterating over namespaces but not loading tables – a missing `await` on `load_table` meant the loop skipped every table without error. The Rust compiler didn't catch it because the `match` arm for `Err` returned `continue`, and calling the async function without `.await` produced an unused `Future` warning that we'd silenced in a batch suppression.

The fix was one line. The debugging took two hours, because the symptom – empty columns – could have been a Polaris permission issue, an Iceberg schema parsing issue, or a dozen other things. We added an integration test that queries `information_schema.columns` and asserts it returns at least one row. We should have written that test first.

These are the bugs that metadata surfaces get. They're never compile errors. They're always runtime, behavioral, tool-specific, and they manifest as "the tool doesn't work" without telling you why.

What the Catalog Teaches

The catalog implementation taught us something we should have known from the start: your engine's API is not its query protocol. It's not the Flight SQL endpoint, not the Trino compat layer, not the JDBC driver. The API is the catalog. The metadata. The answer to "what do you have and how is it shaped?"

Every tool in the SQL ecosystem starts by reading the catalog. If the catalog is incomplete, every tool is crippled. If the catalog is slow, every tool feels sluggish. If the catalog lies, every tool built on top of it produces wrong results.

Build a catalog worth reading.

Making dbt Work

> dbt doesn't care about your architecture. It cares about list_relations.

With `information_schema`, `SHOW` commands, and namespace resolution in place, the engine could describe itself. Tools could connect, browse schemas, inspect columns. The metadata surface was complete.

Then we ran `dbt run` and nothing happened.

dbt doesn't query your engine the way a human does. It doesn't type SQL and wait for results. It runs a discovery sequence – a cascade of metadata queries that maps your warehouse before executing a single model. If any step in that sequence returns the wrong shape, the wrong types, or nothing at all, dbt stops. Not with a helpful error. With a Python traceback three screens long.

Making dbt work meant understanding exactly what it asks for, in what order, and what it expects back.

How dbt Discovers Your Warehouse

dbt's adapter lifecycle starts with `dbt debug`, which validates the connection profile. The adapter opens an ADBC Flight SQL connection with the user's credentials – the handshake authenticates via OIDC exactly as described in Chapter 4.

Once connected, dbt's machinery starts discovering the warehouse:

1. **Check schema exists**: dbt queries `information_schema.schemata` to verify the target schema exists. If it doesn't, dbt calls `CREATE SCHEMA`.
2. **List existing relations**: dbt queries `information_schema.tables` filtered by schema to find existing tables and views. This determines which models need to be built vs. which already exist and might only need incremental updates.
3. **Get columns for existing tables**: for incremental models, dbt queries `information_schema.columns` to compare the model's output schema against the existing table's schema. Column additions, type changes, and schema drift are detected here.
4. **Execute model SQL**: dbt generates and runs `CREATE TABLE AS SELECT`, `INSERT INTO`, or `CREATE OR REPLACE TABLE AS SELECT` depending on the materialization strategy.

5. **Verify results**: dbt queries `information_schema.tables` again to confirm the model's target table exists after execution.

Every step except step 4 is metadata. dbt spends more time asking "what exists?" than "compute this result." This is why `information_schema` performance matters: a dbt project with 50 models might query metadata hundreds of times in a single run.

The `SessionCatalog` in Polaris caches namespace and table listings, which helps. But the real performance win is that our virtual tables are generated from catalog API responses that are already fast – Polaris is an in-memory catalog service. The information_schema call graph is: SQL query -> DataFusion resolves virtual table -> provider calls Polaris REST -> Polaris returns metadata -> provider builds Arrow batch -> DataFusion returns results. The Polaris round-trip dominates, and it's typically under 10 milliseconds.

There's a cost we accepted: the `build_columns_table` method loads every table in every namespace to read its schema. For a warehouse with 500 tables, that's 500 `load_table` calls to Polaris. This is fine for dbt's filtered queries (`WHERE table_schema = 'finance'`) because we load all metadata and let DataFusion filter the result, but it's expensive for unfiltered `SELECT * FROM information_schema.columns`. A future optimization would push the filter down into the provider – detect the `WHERE table_schema = ?` predicate and only load tables from that namespace. We haven't needed it yet. dbt always filters by schema, and DBeaver browses one schema at a time. But it's the kind of optimization that becomes necessary at scale, and the virtual table architecture makes it possible without changing the SQL interface.

The Connection Profile

dbt connects to SQE through a profile in `~/.dbt/profiles.yml`. The profile specifies the adapter type (`sqe`), host, port, user, password, catalog, and default schema. A typical profile:

```yaml
sqe_warehouse:
  target: dev
  outputs:
    dev:
      type: sqe
      host: localhost
      port: 50051
      user: analyst
      password: "{{ env_var('SQE_PASSWORD') }}"
      catalog: test_warehouse
      schema: finance
```

The `type: sqe` field tells dbt to load the `dbt-sqe` adapter. The password comes from an environment variable – dbt's Jinja templating handles secrets without putting them in the YAML file. The `catalog` and `schema` fields map directly to the Polaris warehouse and Iceberg namespace. dbt uses these to construct qualified table names: `test_warehouse.finance.my_model`.

The dbt-sqe Adapter

The Python adapter is the bridge between dbt's framework and SQE's SQL dialect. The design is Path A from our evaluation: a native adapter over ADBC Flight SQL, not a Trino compatibility shim.

```
dbt Core --> dbt-sqe adapter (Python) --> ADBC Flight SQL --> SQE
```

The adapter has four main components:

SQEConnectionManager handles the connection lifecycle. It uses `adbc_driver_flightsql.dbapi.connect()` to open an Arrow Flight SQL connection with the user's credentials. The connection speaks the DB-API 2.0 interface that dbt expects.

```
class SQEConnectionManager(SQLConnectionManager):
    TYPE = "sqe"

    @classmethod
    def open(cls, connection):
        credentials = connection.credentials
        uri = f"grpc://{credentials.host}:{credentials.port}"

        handle = flight_connect(
            uri,
            db_kwargs={
                DatabaseOptions.USERNAME.value: credentials.user,
                DatabaseOptions.PASSWORD.value: credentials.password,
            },
        )
        connection.handle = handle
        connection.state = "open"
        return connection
```

SQEAdapter implements dbt's discovery methods: `list_relations_without_caching()` queries `information_schema.tables`, `get_columns_in_relation()` queries `information_schema.columns`, `check_schema_exists()` queries `information_schema.schemata`. Each method translates from dbt's abstract relation model to SQL against our virtual tables.

Materializations are Jinja SQL templates that generate the right SQL for each model type. The table materialization emits `CREATE OR REPLACE TABLE AS SELECT`. The view materialization emits `CREATE OR REPLACE VIEW AS`. The incremental materialization emits `INSERT INTO` for append strategy, or `MERGE INTO` for merge strategy. Each template maps dbt's abstract model definition to concrete SQL that SQE's parser and planner can handle. The materialization macros are where the adapter's knowledge of SQE's SQL dialect lives – what statements are supported, what syntax they use, what Iceberg-specific options are available.

The incremental materialization deserves extra attention. dbt's incremental strategy depends on being able to compare the existing table's schema against the model's output schema. For append mode, this means `INSERT INTO target SELECT ... FROM source WHERE ...` with a configurable filter that determines which rows are "new." For merge mode, this means `MERGE INTO target USING source`

`ON key_condition WHEN MATCHED THEN UPDATE WHEN NOT MATCHED THEN INSERT`. SQE supports both, but the merge strategy requires Iceberg v2 tables with row-level deletes enabled. The adapter doesn't enforce this – it generates the SQL and lets the engine return a clear error if the table isn't configured for merge operations.

SQEColumn maps between Arrow types (what ADBC returns), Iceberg types (what the catalog stores), and dbt types (what dbt's schema comparison logic expects). This mapping is tedious but critical – dbt's column type comparison is string-based, so `BIGINT` and `Int64` are different to dbt even though they mean the same thing. The column class normalizes all three representations into a canonical form that dbt's comparisons can work with.

The adapter is roughly 2,000 lines of Python. That sounds like a lot, but most of it is the connection manager, the materialization templates, and the type mapping – the boring, necessary plumbing that makes dbt's abstract model concrete. The test suite validates each method independently: `list_relations_without_caching` returns the right relations, `get_columns_in_relation` returns the right column types, `check_schema_exists` handles the "schema doesn't exist yet" case.

Why Not Trino Compat?

We chose Path A over Path B (the Trino compat shim) for a reason that came down to surface area math. A dbt adapter is roughly 2,000 lines of well-documented Python with clear interfaces. The adapter protocol is stable – dbt's `SQLAdapter` base class hasn't changed its core methods in years. You implement `list_relations_without_caching`, `get_columns_in_relation`, `create_schema`, `drop_schema`, `rename_relation`, and the materialization macros. That's the contract.

A faithful Trino wire protocol implementation is a different beast entirely. HTTP pagination with `nextUri` polling. Session properties embedded in HTTP headers. Transaction semantics with BEGIN/COMMIT that dbt-trino expects to work. Error responses in Trino's specific JSON format. The ARRAY[], MAP(), and ROW() constructor syntax that Trino SQL uses and dbt-trino's macros emit. Type casting with Trino's `CAST()` behavior. And all of this is a moving target – dbt-trino updates with every Trino release.

The adapter is finished work. The compat layer is ongoing maintenance. We chose the bounded problem.

Field report: dbt's adapter test suite has 47 tests. We passed 43 on the first run. The 4 failures were all about transaction semantics we intentionally don't support – BEGIN, COMMIT, ROLLBACK. Iceberg provides atomic commits per statement; multi-statement transactions aren't meaningful for our model. Sometimes incompatibility is a feature.

The Type Translation Problem

In a later test, a dbt project with 12 models failed because `get_columns_in_relation` returned column types as Iceberg's string representation (`"long"`, `"timestamptz"`) instead of SQL standard names (`"BIGINT"`, `"TIMESTAMP WITH TIME ZONE"`). dbt's schema comparison is string-based – it compares

the type name from the existing table against the type name from the model definition. If they don't match character for character, dbt flags a schema change and tries to handle it.

The fix was adding a type name translation layer in the information_schema columns provider. Another hour of debugging for ten lines of code. These are the bugs that metadata surfaces get. They're never compile errors. They're always runtime, behavioral, tool-specific, and they manifest as "the tool doesn't work" without telling you why.

The Iceberg schema becomes SQL metadata through a translation layer: `field.required` maps to the SQL standard's `is_nullable`. `field.field_type` renders as a string that tools parse into their own type models. Getting these strings right – matching them character for character to what dbt, DBeaver, and JDBC drivers expect – is the unglamorous work that makes the difference between "the engine works" and "the engine works with my tools."

The system.metadata Schema

Alongside the standard `information_schema` and the runtime tables, we added `system.metadata` for Iceberg-specific introspection. This schema exposes what SQL standard views can't represent: table properties, schema properties, and table comments.

`system.metadata.table_properties` returns one row per property per table. Iceberg tables carry properties like `write.format.default`, `write.parquet.compression-codec`, and custom key-value pairs set by the user. A query like `SELECT table_name, property_name, property_value FROM system.metadata.table_properties WHERE schema_name = 'finance' AND property_name LIKE 'write.%'` tells you how every table in the finance namespace is configured for writes. This is information that would otherwise require calling the Polaris REST API directly.

`system.metadata.table_comments` extracts the `"comment"` property from each table's metadata. Some tools – DBeaver among them – show table comments in the schema browser. Without this table, comments set in Iceberg are invisible to SQL tools.

`system.metadata.catalogs` is a static single-row table showing the warehouse name and connector type (`"iceberg"`). It exists because Trino's JDBC driver queries it during connection setup. One row, two columns, but without it the driver throws an error before you can run a single query. Nobody builds a query engine to implement static metadata tables. But without them, the tools don't work.

The JDBC Schema: Keeping DBeaver Happy

The `system.jdbc` schema illustrates a pragmatic truth about building query engines: you spend a surprising amount of time making clients not crash.

Trino's JDBC driver, used by DBeaver and many other tools, queries `system.jdbc.types` to populate its type mappings, `system.jdbc.catalogs` for the catalog tree, `system.jdbc.schemas` for schema browsing, and `system.jdbc.tables` and `system.jdbc.columns` for table metadata. These overlap with `information_schema` but use a different schema layout – JDBC-specific columns like `JDBC_TYPE`, `PRECISION`, `LITERAL_PREFIX`.

We implemented five tables in `JdbcSchemaProvider`. The types table is static – a hardcoded list of SQL types with their JDBC type codes, precision, and scale. Each row describes a type the engine supports: `BOOLEAN` with JDBC type code 16, `INTEGER` with code 4, `VARCHAR` with code 12. These codes come from the JDBC specification and must be exact – tools parse them programmatically to determine type compatibility, auto-completion suggestions, and data import wizards.

The other four JDBC tables are dynamic, pulling from the same Polaris catalog data as `information_schema` but formatted to match the JDBC metadata result set contracts. The column names differ. The type representations differ. The nullability encoding differs. Same underlying data, different suit.

The payoff: DBeaver's schema browser populated. Tables appeared in the tree. Columns showed their types. Double-clicking a table showed its data. The tool worked because we spoke its language.

[To be completed by AI Logbook agent]

The First Real Run

The adapter passed its test suite. The integration tests passed. `dbt debug` connected, authenticated, and reported green. We ran `dbt run` against a project with five seed files and a silver layer of transformation models.

Products seeded. Customers failed.

```
Database Error in seed file seeds/customers.csv
  Cannot add files that are already referenced by table
```

The products seed had 500 rows. The customers seed had 2,000. That threshold was the clue. DataFusion's write path batches rows internally — at some threshold, a single seed operation becomes multiple batch writes. Each batch wrote a Parquet file. Each file was named with a static prefix: `insert-00000.parquet`. The counter reset per batch. The second batch tried to write `insert-00000.parquet` into the same Iceberg snapshot that already contained `insert-00000.parquet` from the first batch.

The Iceberg commit protocol rejected it. You can't add a file that's already referenced. The table format was doing exactly what it was designed to do.

The fix was one line: replace the static prefix with a UUID per write operation. Files became `019abc7f-...-00000.parquet`, `019abc7f-...-00001.parquet`. Unique by construction. The collision was structurally impossible afterward.

> The unit tests had only tested small payloads. The first real seed that crossed the batch threshold was customers.csv, row 501.

This is a category of bug that unit tests reliably miss. You test one batch. You don't test two batches writing to the same table in the same transaction. dbt seeds large datasets; that's their purpose. The bug was invisible until something real ran against the engine.

With seeds working, the silver layer models ran next. `stg_orders` failed immediately:

```
Error: Invalid function 'year'
```

The model used `year(order_date)`, `month(order_date)`, `day_of_week(order_date)` — standard Trino date extraction functions. DataFusion has `date_part()` and `extract()`, but not standalone `year()`, `month()`, or `day_of_week()`. They're different names for the same operation, and dbt projects written for Trino use the Trino names.

We added 17 Trino-compatible UDFs: date extraction (`year`, `month`, `day`, `hour`, `minute`, `second`, `day_of_week`, `day_of_year`, `week_of_year`, `quarter`), date arithmetic (`date_add`, `date_diff`, `date_trunc`), conditionals (`if`, `nullif`, `coalesce` — DataFusion has these, but the Trino aliases needed wiring), and introspection (`typeof`, `version`). Each one delegates to the DataFusion equivalent under the hood. The registration is mechanical; the value is compatibility without forking.

While fixing the date functions, a second type error surfaced in a different model. The error message was:

```
TypeSignatureClass::Native(LogicalType::Boolean) is not compatible
with TypeSignatureClass::Scalar(DataType::Utf8)
```

The model was calling `lower(is_active)` where `is_active` was a boolean column. `lower()` expects a string. The SQL was wrong — but the error was gibberish. It referenced internal DataFusion type names that have no meaning to someone writing SQL. The error was technically correct and practically useless.

That observation planted a seed.

We fixed the UDFs. The models ran. And then a different kind of failure appeared: a model failed because it referenced a staging table that hadn't been materialized yet (a model ordering issue in the project). The error was:

```
TrinoQueryError(type=INTERNAL_ERROR, name=INTERNAL_ERROR,
  message="Query execution failed", error_code=1)
```

We'd seen this before and dismissed it. Now it was the only error visible when a model failed. Every failure — missing table, wrong type, auth problem, S3 timeout, syntax error — returned the same response: `INTERNAL_ERROR(1)`, `"Query execution failed"`. The Trino compat layer was swallowing every DataFusion error and replacing it with the most generic possible response.

dbt's error output showed us what we'd built: a black box. Something failed. We had no idea what. We had to add logging at the engine layer, trace back through coordinator logs, correlate timestamps, and find the real error buried several layers down. That's acceptable for debugging one failure. It's unworkable when dbt runs 50 models and a third of them fail.

The fix was structural. We defined 27 error codes — `SqeErrorCode` — covering the full taxonomy of query engine failures: `TABLE_NOT_FOUND(11)`, `COLUMN_NOT_FOUND(12)`, `TYPE_MISMATCH(21)`, `PERMISSION_DENIED(31)`, `CATALOG_ERROR(41)`, `STORAGE_ERROR(51)`, and so on. Each code maps to both a Trino error type (for the compat layer) and a gRPC status code (for the Flight SQL layer). An auto-classifier inspects DataFusion error messages and pattern-matches them to the right code: messages

containing "table not found" map to `TABLE_NOT_FOUND`, messages containing "permission denied" map to `PERMISSION_DENIED`.

After the fix:

```
TrinoQueryError(type=INVALID_INPUT, name=TABLE_NOT_FOUND,
  message="table 'test_ns.stg_customers' not found", error_code=11)
```

The error code is meaningful. The error type is correct. The message names the specific table. dbt can display this to the analyst who wrote the model, and they can fix it without involving the engine team.

The three bugs weren't independent. The file collision only became visible when the seed data was large enough to matter. The missing functions only became visible when seeds worked and models ran. The useless error messages only became visible when models started failing for real reasons that needed diagnosis.

Each fix revealed the next problem. That's how real integration testing works. You can't write a unit test for "what breaks when a dbt analyst writes their first real project against your engine." You can only run the project and read what happens.

Field report: The three fixes together — UUID file naming, 17 Trino function aliases, 27 structured error codes — were roughly 600 lines of Rust. None of them were architecturally interesting. They were all "this thing doesn't work the way the ecosystem expects it to." That's most of what makes an engine usable.

What dbt Teaches

dbt doesn't care about your engine's architecture. It doesn't care about bearer passthrough, plan rewriting, or Arrow Flight. It cares about `list_relations_without_caching`. It cares about column types matching as strings. It cares about `CREATE OR REPLACE TABLE AS SELECT` working exactly the way it expects.

This is humbling work. You can build the most elegant query optimizer in the world, and dbt will ignore it entirely until `information_schema.columns` returns the right column type strings. You can implement zero-copy Arrow Flight transfers, and DBeaver won't show a single table until `system.jdbc.types` has the right JDBC type codes.

The lesson applies beyond dbt. Every tool in the SQL ecosystem – BI dashboards, schema browsers, data quality tools, lineage trackers – starts by reading the catalog. If the catalog is incomplete, every tool is crippled. If the catalog is slow, every tool feels sluggish. If the catalog lies (stale data, missing tables, wrong types), every tool built on top of it produces wrong results.

We spent roughly a third of our total development time on metadata surfaces – information_schema, system tables, JDBC types, SHOW commands, the dbt adapter. None of it makes the engine faster. All of it makes the engine usable. The unglamorous work is where adoption lives.

Build a catalog worth reading. Then build an adapter that translates it into the exact language each tool speaks. The translation is tedious. It's also the difference between an engine that works and an

engine people use.

Writing Is a Contract

> Reading is easy. Writing is where table formats earn their keep.

A query engine that can only read is a glorified report generator. We had proven the read path – DataFusion parses SQL, Iceberg supplies metadata, Polaris vends credentials, S3 delivers Parquet bytes, and the user gets Arrow batches. Chapters 4 through 6 tell that story. But dbt does not just read. dbt needs to create tables, insert rows, run incremental materializations that merge new data into existing tables. Without a write path, SQE is a demo.

The git log tells the story plainly: March 14, the engine ran its first read query. March 15, CTAS and INSERT INTO worked end to end. One day. But that one day compressed more debugging than the entire read path combined, because writing to Iceberg is not the reverse of reading from it. Reading is a contract between the engine and the storage layer. Writing is a contract between the engine, the storage layer, the catalog, and every other writer who might be committing at the same time. More parties means more places where types can disagree, metadata can diverge, and assumptions can quietly be wrong.

The Commit Protocol

Before any code, the mental model. Iceberg's write protocol has three stages, and you have to understand all three before the first line of Rust makes sense.

Stage 1: Write data files. The engine produces Parquet files and uploads them to S3. These files are orphans – they exist in storage but are invisible to any reader. No catalog knows about them. No manifest references them. They are bits on disk with no meaning.

Stage 2: Commit metadata. The engine sends a commit request to the catalog saying "here are the new data files, add them to this table's current snapshot." The catalog validates the request, creates a new snapshot, updates the manifest list, and returns success.

Stage 3: Atomicity guarantee. If the commit fails – because another writer committed first, because the table was dropped, because the network died – the data files from Stage 1 are garbage. They sit in S3 until someone cleans them up. No reader will ever see them. This is by design.

Iceberg deep dive: Iceberg's snapshot isolation is implemented through the catalog, not through storage. S3 historically had no compare-and-swap (though AWS added conditional puts via `If-None-Match` in August 2024). The catalog provides the atomic swap: it replaces the metadata pointer only if

the base snapshot matches what the writer saw when it started. Optimistic concurrency control. The writer assumes nobody else is committing and proceeds. If the assumption was wrong, the commit is rejected and the writer must retry or fail.

This three-stage protocol gives the write path its natural structure: execute the query to produce RecordBatches, write those batches as Parquet data files, then commit the new files through the catalog. Every write operation in SQE follows this shape. The differences are in what happens before Stage 1 and what kind of commit Stage 2 uses.

CTAS: The Simplest Write

We started with CREATE TABLE AS SELECT. Not because it was the most important write operation, but because it was the simplest to reason about. No existing table to contend with. No concurrent writers. No schema to match. The SELECT defines the schema, the data, and the table all at once.

The coordinator classifies `CREATE TABLE ns.target AS SELECT ...` and routes it to the write handler. The streaming write path processes DataFusion output one batch at a time via `SendableRecordBatchStream`. The `write_data_files_streaming` function in the writer module passes each `RecordBatch` directly to the Iceberg `RollingFileWriter` without buffering. Peak memory drops from O(total rows) to O(batch size) – typically 8,000 rows. The six-million-row lineorder table at scale factor 1 now loads with the same memory footprint as a fifty-row dimension table. The write handler then converts the Arrow schema to an Iceberg schema, creates the table in Polaris, writes the batches as Parquet data files, and commits them via a fast-append transaction.

The schema conversion is where the first real problem appeared. Arrow schemas from DataFusion queries do not carry Parquet field-ID metadata – the `PARQUET:field_id` key that Iceberg uses to map columns to schema fields is absent. Without that key, iceberg-rust's `arrow_schema_to_schema` rejects the schema outright. No graceful fallback. No warning. Just a hard error with a message that took some squinting to connect back to the missing metadata.

We wrote our own conversion that walks the Arrow schema and assigns sequential field IDs starting from 1. Sequential IDs work for new tables because there is no existing schema to maintain compatibility with. Iceberg requires field IDs to be unique and never reused within a schema's evolution history. For CTAS, we are creating the initial schema, so any numbering that starts at 1 and increments is valid. The type mapping – `Int64` to `long`, `Utf8` to `string`, `Float64` to `double` – comes from iceberg-rust's `arrow_type_to_type` function, which handles the standard conversions but rejects certain Arrow types that have no Iceberg equivalent.

This seemed straightforward. Then we tried a query with a timestamp.

Four Hours for One Enum Variant

DataFusion produces `Timestamp(Nanosecond, None)` for `CURRENT_TIMESTAMP` and timestamp literals. Iceberg stores timestamps as `Timestamp(Microsecond, None)`. The Parquet writer in iceberg-rust enforces strict type matching. Nanosecond timestamps are rejected.

The error was opaque. The Parquet writer reported a schema mismatch, but the field names and logical types looked identical. We printed both schemas side by side. They matched. We compared them field by field in a loop. They matched. We serialized them to JSON and diffed the output. They matched.

They did not match.

Both `Timestamp(Nanosecond, None)` and `Timestamp(Microsecond, None)` display as "Timestamp" in Arrow's summary formatter. The precision parameter – the one thing that was different – does not appear in the default display output. We were staring at two schemas that were genuinely different, rendered identically on screen.

After three hours of this, we added explicit type-level logging that printed the full `Debug` representation of every field's data type, not the display representation. And there it was: `Timestamp(Nanosecond, None)` on the left, `Timestamp(Microsecond, None)` on the right. A single enum variant, buried three layers deep in the Arrow type hierarchy.

The fix was clear once we saw it. The finding-the-problem part took four hours. One line of code. Four hours of debugging. That ratio would become familiar on the write path.

We already had a function called `stamp_field_ids` that added `PARQUET:field_id` metadata to Arrow fields before writing. We extended it to also handle type mismatches. The function now does three things in one pass: stamps field-ID metadata onto each Arrow field, fixes nullable flags by scanning all batches (not just the first), and casts any columns whose Arrow type diverges from what Iceberg expects. The timestamp cast – nanosecond to microsecond – is the most common, but the same mechanism handles any type divergence between DataFusion's output and Iceberg's schema.

The nullable flag fix was the second invisible bug. DataFusion sometimes marks a field as non-nullable even when the data contains nulls. This happens with `CAST(NULL AS T)` inside a `UNION ALL` – the type is inferred as non-nullable from the cast expression, but the actual value is null. The Parquet writer rightfully rejects a null value in a non-nullable column. The first batch in a UNION ALL might have no nulls in that column; the third batch might. If you derive the schema from the first batch alone, you get a non-nullable flag and a crash when the third batch writes.

We scan all batches to detect nulls and upgrade the field to nullable if any are found. Trust the data, not the metadata. This is a pattern that came up again and again on the write path: the metadata says one thing, the bytes say another, and the bytes are always right.

Writing the Files, Committing the Result

Once the table exists in the catalog and the batches are type-corrected, the physical writing happens through iceberg-rust's writer infrastructure. This part, after all the schema debugging, was refreshingly mechanical. We configure a chain of builders: `ParquetWriterBuilder` for the file format, `RollingFileWriterBuilder` for splitting large writes across multiple files (128 MB default), `DataFileWriterBuilder` for producing the `DataFile` descriptors that Iceberg needs for the commit. Each `DataFile` contains the file path, size, row count, column statistics, and partition values. The builder chain is verbose but honest – every configuration choice is visible in the code, not hidden in a default.

The `file_prefix` parameter distinguishes data files by origin – "ctas", "insert", or "ingest". Iceberg does not care about file names. We do, at 2am, when we are staring at a table's storage directory trying to figure out which write operation produced a particular file.

With data files written and their descriptors in hand, the commit itself is four lines:

```
let tx = Transaction::new(&table);
let action = tx.fast_append().add_data_files(data_files);

let tx = action.apply(tx).map_err(|e| {
    SqeError::Execution(format!("Failed to apply fast append: {e}"))
})?;

tx.commit(catalog.as_ref()).await.map_err(|e| {
    SqeError::Execution(format!("Failed to commit CTAS transaction: {e}"))
})?;
```

`Transaction::new` creates a transaction scoped to the table's current snapshot. `fast_append()` creates a `FastAppendAction` – the simplest commit type, which only adds data files without modifying or removing existing ones. `apply` prepares the transaction. `commit` sends it to the catalog.

The `fast_append` distinction matters. Iceberg defines several transaction types with different conflict rules:

Transaction Type	What It Does	Conflict Check
`fast_append`	Add data files only	Fails if another writer deleted data or changed schema
`overwrite`	Replace data files	Fails if another writer modified any affected files
`row_delta`	Add data + delete files	Full row-level conflict check
`rewrite_files`	Compaction	Fails if any rewritten files were modified

For INSERT INTO and CTAS, `fast_append` is correct. We are only adding new files. Two writers inserting rows into the same table simultaneously is perfectly safe – Iceberg's metadata supports concurrent appends without conflict. The conflict scenarios that matter are structural changes: someone drops the table, alters the schema, or performs a compaction between our write and our commit. The engine surfaces the error, and the client can retry.

INSERT INTO and the Three Entry Points

INSERT INTO follows the same structure as CTAS, with one difference: the table already exists. We load it instead of creating it. The query handler extracts the SELECT source from the INSERT INTO statement, executes it through DataFusion, and passes the batches to the write handler. The write handler loads the existing table, writes data files against its current schema, and commits via fast-append.

The `stamp_field_ids` function becomes critical here. For CTAS, we control the schema – the Iceberg schema is derived from the Arrow schema, so the types match by construction. For INSERT INTO, the target table's schema was defined previously, possibly with different precision or nullability. The type casting ensures the new data matches the table's expectations.

The same code path also handles Flight SQL's DoPut ingest, where a client streams Arrow batches directly via the Flight protocol instead of sending SQL. Three write entry points – CTAS, INSERT, DoPut ingest – all converging on the same `write_data_files_streaming` and `Transaction::fast_append` primitives. One code path for producing Parquet files, one for committing them, regardless of how the data arrived.

Dead end: letting DataFusion handle the full INSERT. DataFusion has its own `InsertExec` plan node. We considered routing INSERT INTO through DataFusion's built-in write path rather than extracting the SELECT and handling the Iceberg commit ourselves. DataFusion's `InsertExec` assumes it controls the table provider, including where and how files are written. Iceberg's commit protocol requires knowledge of the table's metadata location, file naming conventions, and partition spec. We would have needed to implement DataFusion's `TableProvider` write interface for Iceberg, which had no upstream support in iceberg-datafusion 0.9. Splitting query execution from the write commit was simpler and gave us full control over the transaction.

The Benchmark Surprise

The benchmark suite uses CTAS to load all test data. TPC-H at scale factor 10 means eight tables totaling about 10 GB of Parquet. We had just spent a week understanding Iceberg's commit protocol – the snapshot isolation, the optimistic concurrency, the catalog round-trips. We expected the commit to be the bottleneck. We instrumented it carefully.

Loading via CTAS took approximately 4 minutes on a single coordinator. We checked the commit timing. Milliseconds. All eight tables. Milliseconds.

The four minutes were Parquet serialization and S3 upload latency. The protocol that we had spent days understanding and debugging was the fastest part of the entire operation, by three orders of magnitude. We had optimized our understanding of the wrong thing.

This reshaped our priorities. The rolling file writer's default chunk size, the Parquet compression settings, the number of concurrent S3 uploads – these are the knobs that matter for write throughput. The commit protocol is elegant and important and takes almost no time at all. If you are building a write path for Iceberg, optimize the file writing first. The commit is not your problem.

CREATE OR REPLACE TABLE

dbt's table materialization uses `CREATE OR REPLACE TABLE ... AS SELECT`. The query handler implements this by checking the `or_replace` flag: drop the existing table, then create a new one with the CTAS data.

Drop, then create. Not a single atomic operation – there is a window between the drop and the create

where the table does not exist. For dbt's use case (batch transforms where the table is being rebuilt from scratch), this is acceptable. For a high-availability system where readers must always see either the old or the new version, it would not be. A true atomic replace would require Iceberg's overwrite transaction.

Statement Classification

Before any write handler runs, the SQL classifier must decide what kind of statement it is looking at. `CREATE TABLE foo AS SELECT 1` and `CREATE TABLE foo (id INT)` both parse as `Statement::CreateTable` in sqlparser-rs. The difference is whether the `query` field is `Some` or `None`:

```
Statement::CreateTable(ref ct) => {
    if ct.query.is_some() {
        Ok(StatementKind::Ctas(Box::new(stmt)))
    } else {
        Ok(StatementKind::CreateTable(Box::new(stmt)))
    }
}
```

The classifier routes each statement before DataFusion ever sees it. For write operations, the coordinator extracts and re-executes just the SELECT portion through DataFusion, collects the batches, and hands them to the write handler. The write handler never touches DataFusion directly. It receives already-materialized data. This separation turned out to be one of the better architectural decisions on the write path – the write handler knows nothing about SQL, and the query handler knows nothing about Iceberg commits. Each does one thing.

Copy-on-Write vs Merge-on-Read

CTAS and INSERT INTO are append-only. They only add files. The harder operations – DELETE, UPDATE, MERGE – require modifying or removing rows that already exist. This is where the engineering gets interesting, and where the two fundamental strategies for row-level mutations diverge.

Copy-on-Write rewrites entire data files. Delete one row from a million-row file, and CoW reads that file, writes 999,999 rows to a new one, and commits a replacement. Subsequent reads are clean – every scan reads only data files, no reconciliation needed. The cost is write amplification. Deleting 100 rows across 100 files means rewriting 100 files.

Merge-on-Read writes small delete files – position deletes marking row positions within specific data files, or equality deletes marking rows by key values. Writes are fast because delete files are tiny. The cost shifts to reads: every subsequent scan must reconcile delete files against data files, filtering out deleted rows at read time. The more deletes you accumulate without compacting, the slower every read gets. It is a debt model – fast writes now, paid back on every read until you compact.

Characteristic	Copy-on-Write	Merge-on-Read
Write cost	High (rewrites entire files)	Low (small delete files)

Characteristic	Copy-on-Write	Merge-on-Read
Read cost	None (clean data files)	Per-scan reconciliation
Best for	Read-heavy workloads	Write-heavy workloads
Compaction need	Low	High (delete files accumulate)

The practical question: does the delete ratio justify the complexity? If you delete 10 rows out of a million-row table once a day, CoW rewrites a handful of files and you never think about it. If you delete 10% of your data hourly, Merge-on-Read avoids catastrophic write amplification, but you need compaction to keep read performance from degrading.

SQE uses Copy-on-Write. It is simpler, and the RisingWave iceberg-rust fork provides the `rewrite_files()` transaction primitive that makes it work. The RisingWave fork also ships `PositionDeleteFileWriter`, and position delete support is partially implemented in SQE. Full Merge-on-Read with `RowDeltaAction` will follow as the upstream iceberg-rust API stabilizes. The choice of Copy-on-Write as the default is pragmatic, not permanent.

Row-Level Writes: Copy-on-Write

The SQL classifier parses and classifies MERGE, DELETE, and UPDATE statements. The routing works. And now the handlers deliver.

Upstream iceberg-rust had not shipped `OverwriteAction`, so we found another path: the RisingWave iceberg-rust fork (rev `1978911ec4`), which provides `rewrite_files()` – a transaction API that atomically replaces a set of data files with new ones. This is exactly the primitive Copy-on-Write needs: read the affected files, rewrite them without the deleted or modified rows, commit the swap.

DELETE FROM reads each affected data file, applies the WHERE filter, and rewrites the file without matching rows. If all rows match, the file is simply removed. DELETE without a WHERE clause is a truncate. Cross-table subqueries work in the WHERE clause because DataFusion handles the subquery planning before the CoW rewrite step executes.

UPDATE follows the same pattern: read affected files, apply the WHERE filter, apply the SET expressions to matching rows, rewrite the file. CASE WHEN transformations in SET clauses work naturally because DataFusion evaluates them as expressions.

MERGE INTO is the most complex. It executes a full outer join between source and target via DataFusion, classifies each result row as matched (UPDATE or DELETE) or not matched (INSERT), then rewrites affected target files and appends new files for INSERT rows. The entire operation commits atomically via `rewrite_files()`.

All three operations are atomic via Iceberg snapshot isolation. If the commit fails – because another writer modified the same files – the error surfaces to the client. Retry logic is left to the caller, which is adequate for batch workloads orchestrated by dbt.

Compaction is the remaining thing we have not built. For CoW workloads, file count grows with each mutation – every DELETE or UPDATE that touches a file produces a new file. A dbt pipeline run-

ning incremental models nightly accumulates files steadily. Iceberg's manifest lists handle millions of entries, so the urgency is low, but compaction will eventually matter for scan performance. When we build it, compaction will run as a background task using the same bearer token passthrough model. No ambient credentials. No service account. The user who triggers compaction must have write permission on the table.

The Bearer Token and Writes

Everything in Chapter 4 about bearer token passthrough applies to writes, with one additional constraint: the write path requires authorization at the catalog level, not just the storage level.

For reads, the user needs read permission on the table and read access to the S3 path. For writes, the user also needs permission to create tables, append data files, and commit transactions. These are separate permissions in Polaris. The write handler creates a `SessionCatalog` with the user's bearer token, and every catalog operation – `create_table`, `load_table`, `commit` – is authenticated as that user.

If Alice has read-only access and tries to INSERT, Polaris rejects the commit. The engine does not need its own authorization logic for this. The catalog enforces it. No ambient permissions. No service account.

Conflict Resolution

Two users insert into the same table at the same time. What happens?

With `fast_append`, both commits succeed. Iceberg's metadata supports concurrent appends because appending new files does not conflict with other appends – neither writer is modifying or removing files the other cares about. The catalog creates two consecutive snapshots, each adding its own data files. Readers see a consistent view at any snapshot.

The story changes with overwrites. Two writers both try to delete from the same table. Writer A reads the table at snapshot N, identifies files to rewrite, and commits at snapshot N+1. Writer B, who also started at snapshot N, tries to commit at what it thinks is N+1 but is now N+2. The catalog rejects Writer B's commit because the base snapshot no longer matches. This is not a bug. This is the protocol working as intended.

The correct response is to retry: reload the table metadata, re-identify affected files, re-execute the write, re-commit. SQE does not currently implement automatic retry – the error surfaces to the client. For batch workloads orchestrated by dbt, which serializes model execution, this is adequate. For concurrent OLTP-style mutations, automatic retry with backoff would be necessary.

The retry semantics are where the real complexity hides. A naive retry that re-executes the entire query might produce different results if the source data changed between the first attempt and the retry. An intelligent retry reads the new table state, identifies which of its original changes are still valid, and commits only the delta. This is the complexity that makes row-level write support a multi-week effort rather than a multi-day one. The query execution is the easy part. The conflict resolution

and retry logic is where the engineering lives.

Cache Invalidation on Write

One detail that is easy to miss: when a write operation commits, any cached query results for the affected table become stale. Every INSERT, CTAS, and DROP invalidates all cache entries for the affected table. This is conservative – it invalidates everything that touches the table, not just the queries whose results actually changed. A more precise approach would check whether the write overlaps with cached query predicates. We took the conservative path because cache precision is a nice-to-have and stale reads are a production incident.

What We Learned

The write path took one day to implement and three days to debug. The ratio tells you something about writing to Iceberg: the concepts are simple, the protocol is well-designed, and the bugs are invisible. Most of the debugging was in the type system – the gap between Arrow types as DataFusion produces them and Arrow types as Iceberg expects them. The commit protocol itself was straightforward once we understood the three-stage model.

Reading is a contract with storage. Writing is a contract with the world. When you read, you depend on the storage layer delivering bytes. When you write, you depend on the storage layer accepting bytes, the catalog accepting your commit, no other writer conflicting with your commit, and the type system matching across three layers – DataFusion Arrow, Iceberg schema, Parquet physical. Every additional party in the contract is a potential failure point.

Start with append-only, then add mutations. CTAS and INSERT INTO cover 80% of what data pipelines need. dbt's `table` materialization uses CTAS. dbt's `incremental` materialization needs MERGE, but only for the incremental part – the initial build is still CTAS. Shipping append-only first gave us a useful engine weeks before row-level operations landed. When we added DELETE, UPDATE, and MERGE via the RisingWave fork's `rewrite_files()`, the architecture was already in place – the handlers slotted into the existing classifier and write handler structure.

Upstream dependencies gate your timeline, but forks buy you time. Upstream iceberg-rust had not shipped `OverwriteAction`. Rather than wait, we switched to the RisingWave fork that had the transaction primitive we needed. This is the cost and benefit of building on an ecosystem: you depend on others, but you can also leverage their parallel work. The RisingWave team needed the same primitive for their streaming engine and built it before the upstream community finalized the API. When upstream ships, we migrate back. Until then, the fork works.

The invisible bugs are the expensive ones. A timestamp precision mismatch that displays identically in debug output. A nullable flag that is correct for the first batch but wrong for the third. A schema that matches by name and logical type but diverges in a nested enum variant. These are not hard problems. They are invisible problems. The fix is always one line. The debugging is always four hours.

The write path is where a query engine stops being a toy and starts being infrastructure. Reading data

is table stakes. Writing data – correctly, atomically, with snapshot isolation and concurrent writer safety – is the price of admission to the world of table formats.

We paid that price. And then we moved on to the next problem: making sure users cannot write data they should not be able to see.

AI Logbook: The AI implemented CTAS, INSERT INTO, and the `stamp_field_ids` function that assigns sequential Parquet field IDs to Arrow schemas. The timestamp precision bug — `Timestamp(Nanosecond, None)` vs `Timestamp(Microsecond, None)` displaying identically in Arrow's formatter — took four hours of human debugging before the AI was told to add `Debug`-level type logging. The fix was one line of casting. The nullable flag scanner that checks all batches (not just the first) was the human's idea after a UNION ALL crash; the AI implemented it in one pass.

What You Can't See Can't Hurt You

> Security is not a feature. It's a rewrite of the query plan.

Alice can log in. Chapter 4 made sure of that. Her JWT is validated, her identity propagated, her S3 access scoped to exactly what Polaris grants her. Every Parquet file she reads shows up in CloudTrail under her name.

But authentication is not authorization. Alice can prove she is Alice. That says nothing about whether Alice should see the `salary` column in the `hr.employees` table. Or whether she should see any rows from the `finance.transactions` table where `region != 'EU'`. Or whether the social security numbers in `customers.ssn` should arrive at her client as `123-45-6789` or as `***-**-6789`.

Polaris handles table-level access. If Alice doesn't have access to a table at all, the REST catalog won't return its metadata. But column-level masking, row-level filtering, and column restriction – these require something Polaris doesn't provide. They require a policy layer that operates on the query itself.

This chapter is about building that layer. And the key insight – the one that shapes everything – is that the right place to enforce policy is not in the storage layer, not in the application, and not at the network boundary. It's in the query plan.

The Problem with Application-Level Access Control

The obvious approach is to enforce access control in the application code that handles queries. Check the user's roles. If the role doesn't include `hr_admin`, remove the `salary` column from the result set before sending it back. If the role includes `eu_only`, append `WHERE region = 'EU'` to the SQL string before executing it.

We considered this for about ten minutes.

The problem is that SQL is compositional. Users write subqueries, CTEs, joins, aggregations, window functions. A filter appended to a simple `SELECT * FROM employees` is straightforward. A filter injected into a four-way join with a correlated subquery and a `HAVING` clause is a parsing nightmare. String manipulation of SQL is how injection vulnerabilities happen. It's also how subtle semantic bugs happen – the kind where the filter works for most queries but silently fails on a specific join pattern that

nobody tested.

Antipattern: SQL string manipulation for security. Appending `AND region = 'EU'` to the SQL string before execution seems simple. It breaks on UNION queries, subqueries, CTEs, and any query where the table alias doesn't match. Worse, a carefully crafted query can escape the filter. If your security depends on string manipulation, your security depends on the attacker not being creative.

The second obvious approach is database-level enforcement. PostgreSQL has Row-Level Security (RLS). Oracle has Virtual Private Database (VPD). These work because the database owns both the storage and the query engine – it can enforce policy at the lowest possible level.

But we're not a database. We're a query engine that reads Parquet files from S3 via an Iceberg catalog. We don't own the storage. We don't control the file format. We can't add row-level predicates to the storage layer because the storage layer is a bunch of immutable Parquet files sitting in an object store.

What we do own is the query plan.

The Query Plan as Security Boundary

DataFusion represents every query as a tree of logical operators – a `LogicalPlan`. A simple `SELECT name, salary FROM employees WHERE department = 'engineering'` becomes:

```
Projection [name, salary]
  Filter [department = 'engineering']
    TableScan [employees]
```

This tree is a data structure. It's a Rust enum. You can walk it, transform it, insert nodes, remove nodes, and replace nodes. DataFusion provides a `transform_down` method that visits every node in the tree and lets you return a modified version.

The insight is this: if you can transform the logical plan before the optimizer runs, you can inject security constraints as plan nodes. A row filter becomes a `Filter` node above the `TableScan`. A column mask becomes a `Projection` that wraps column references in masking expressions. A column restriction becomes a modified projection that simply doesn't include the denied column.

The query plan is the security boundary. Not the application code. Not the storage. The plan.

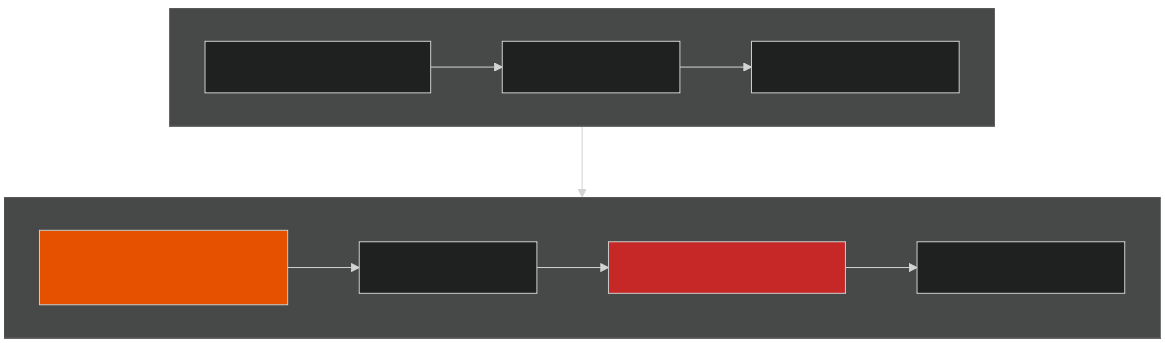

Figure 4: Plan rewriting: how row filters, column masks, and column restrictions are injected into the logical plan before optimization

```
Before policy enforcement:

  Projection [name, salary, ssn]
    Filter [department = 'engineering']
      TableScan [employees]

After policy enforcement (for a user with row filter + column mask):

  Projection [name, salary, mask(ssn)]
    Filter [department = 'engineering']
      Filter [region = 'EU']          <-- injected row filter
        TableScan [employees]
```

The user's original query is preserved. The security constraints are layered on top. The user never knows the row filter exists – they just see fewer rows. The SSN column is there, but its values are masked. The user can still `SELECT ssn` – they just get `***-**-6789` instead of the raw value.

This is the PostgreSQL RLS model, applied to a DataFusion LogicalPlan.

Why Before Optimization

The placement matters. Policy enforcement happens after the SQL is parsed and the initial LogicalPlan is created, but before DataFusion's optimizer runs.

This ordering creates three security properties that would be impossible to guarantee if we enforced policy after optimization.

Property 1: User predicates can push through row filters. If the policy says "Alice can only see rows where `region = 'EU'`" and Alice writes `WHERE department = 'engineering'`, both filters need to apply. Because the row filter is a standard `Filter` node in the logical plan, DataFusion's optimizer treats it like any other filter. The optimizer may reorder them, combine them, or push them into the TableScan as partition predicates. All of this is safe because row filters are conjunctive – adding Alice's predicate can only narrow the result further, never widen it.

Property 2: User predicates cannot push through column masks. This is the critical one. If the `ssn` column is masked, and Alice writes `WHERE ssn = '123-45-6789'`, she's trying to use the predicate to probe for a specific SSN. If the optimizer pushes that predicate below the mask, it would evaluate against the raw value – and the number of rows returned (zero or non-zero) would leak information about whether that SSN exists.

Because we inject the mask as an expression that wraps the column reference, the optimizer sees `mask(ssn)`, not `ssn`. The predicate `WHERE mask(ssn) = '123-45-6789'` can only evaluate against the masked value. The optimizer can't push it through because it can't see through the expression boundary. This is a security property that falls out of the plan structure – we don't need to add a special rule to prevent pushdown. The optimizer's own rules prevent it.

Property 3: Denied columns don't exist. If the policy says Alice can't see the `salary` column,

the plan rewriter removes it from the schema entirely. When Alice runs `SELECT *`, `salary` isn't in the projection. When Alice runs `SELECT salary`, she gets "column not found" – the same error she'd get for a column that genuinely doesn't exist in the table. There is no way for Alice to distinguish "this column exists but I'm denied access" from "this column doesn't exist."

Sovereignty principle: Deny by omission, not by error. If your security system tells the user "access denied to column salary," you've just told them the column exists. In a sovereign engine, denied columns are invisible. The user sees exactly the schema they're authorized to see – no more, no less. This is the same model PostgreSQL uses for RLS, and it exists for the same reason: the absence of information is itself a form of security.

The PolicyEnforcer Trait

The interface is deliberately minimal. Twenty-six lines of Rust:

```
#[async_trait]
pub trait PolicyEnforcer: Send + Sync {
    async fn evaluate(
        &self,
        user: &SessionUser,
        plan: LogicalPlan,
    ) -> Result<LogicalPlan>;
}
```

The enforcer receives a user identity and a logical plan. It returns a (possibly modified) logical plan. That's the entire contract.

The `SessionUser` is the same identity from Chapter 4 – the authenticated user with their roles extracted from the JWT:

```
pub struct SessionUser {
    pub username: String,
    pub roles: Vec<String>,
}
```

The enforcer doesn't know about SQL strings, physical plans, execution contexts, or Arrow batches. It operates on exactly one abstraction: the logical plan tree. This constraint is deliberate. It means the enforcer can be tested with plan construction in unit tests, without needing a running database, a catalog, or even a parser.

The trait lives in the `sqe-policy` crate, which has minimal dependencies – just DataFusion's logical plan types and the core session types. No network. No I/O by default. The implementations bring their own dependencies (HTTP clients for OPA, policy evaluation libraries for Cedar), but the trait itself is pure.

The PassthroughEnforcer

The simplest implementation does nothing:

```rust
pub struct PassthroughEnforcer;

#[async_trait]
impl PolicyEnforcer for PassthroughEnforcer {
    async fn evaluate(
        &self,
        _user: &SessionUser,
        plan: LogicalPlan,
    ) -> Result<LogicalPlan> {
        Ok(plan)
    }
}
```

This is what SQE ships with today. Every query passes through `policy_enforcer.evaluate()` in the query pipeline, and the PassthroughEnforcer returns the plan unchanged. The call is there. The hook is wired. The cost is a function call that returns its argument.

This isn't a placeholder that needs to be removed later. It's the correct default for environments that rely on Polaris for table-level access control and don't need column or row-level policies. The configuration decides which enforcer is active:

```toml
[policy]
engine = "passthrough"    # or "opa" or "cedar"
```

Setting `engine = "opa"` replaces the PassthroughEnforcer with an OPA-backed implementation. The coordinator's main function wires it up:

```rust
let policy_enforcer: Arc<dyn PolicyEnforcer> =
    Arc::new(PassthroughEnforcer);
```

When the OPA implementation is ready, that becomes a config-driven match:

```rust
let policy_enforcer: Arc<dyn PolicyEnforcer> = match config.policy.engine {
    "passthrough" => Arc::new(PassthroughEnforcer),
    "opa" => Arc::new(OpaPolicyEnforcer::new(&config.policy.opa)),
    "cedar" => Arc::new(CedarPolicyEnforcer::new(&config.policy.cedar)),
    _ => return Err(config_error("unknown policy engine")),
};
```

The rest of the coordinator doesn't change. `QueryHandler` holds an `Arc<dyn PolicyEnforcer>` and calls `.evaluate()` on every query. The implementation behind the trait is invisible to the query pipeline.

Inside the Query Pipeline

The enforcement point is in `QueryHandler::execute_query`. Here's the actual code path, trimmed to the essential flow:

```rust
async fn execute_query(&self, session: &Session, sql: &str) -> Result<Vec<RecordBatch>> {
    let ctx = self.create_session_context(session).await?;
```

```
    // Parse SQL and create the initial LogicalPlan
    let df = ctx.sql(sql).await?;
    let plan = df.logical_plan().clone();

    // Policy enforcement -- before optimization
    let enforced_plan = self.policy_enforcer
        .evaluate(&session.user, plan)
        .await?;

    // DataFusion optimizes and executes the enforced plan
    let enforced_df = ctx.execute_logical_plan(enforced_plan).await?;
    let physical_plan = enforced_df.create_physical_plan().await?;

    // Execute (possibly distributed across workers)
    let final_plan = self.try_distribute(physical_plan, session).await;
    collect(final_plan, ctx.task_ctx()).await
}
```

The sequence is: parse, plan, enforce, optimize, execute. The optimizer only ever sees the enforced plan. It cannot undo the security constraints because they look like ordinary plan nodes – a `Filter` is a `Filter`, whether the user wrote it or the policy engine injected it.

EXPLAIN queries follow the same path. The `ExplainHandler` applies policy enforcement before formatting the plan text:

```
let logical = df.logical_plan().clone();
let enforced = self.policy_enforcer.evaluate(&session.user, logical).await?;
let logical_str = format!("{}", enforced.display_indent());
```

This means EXPLAIN shows the secured plan. If a row filter is active, the user sees the filter node in the EXPLAIN output. This is a conscious choice. The alternative – hiding the security nodes from EXPLAIN – would make debugging impossible for both users and administrators.

DataFusion deep dive: The `LogicalPlan::transform_down` method is the workhorse of plan rewriting. It visits each node top-down and lets you return `Transformed::yes(new_node)` to replace a node or `Transformed::no(node)` to keep it. The method handles rebuilding the tree with correct parent-child relationships. For row filter injection, we match on `LogicalPlan::TableScan`, wrap it in a `LogicalPlan::Filter`, and return the filter as the replacement. DataFusion's optimizer then treats this injected filter identically to any user-written filter.

The Plan Rewriter Design

The PassthroughEnforcer is what we ship now. The `PolicyPlanRewriter` is what we've designed for the OPA and Cedar backends. The design is complete, the trait is defined, and the implementation is the next major phase. Here's how it works.

The rewriter receives a plan and does three things:

Step 1: Collect table references. Walk the logical plan tree and extract every `TableScan` node. A single query might reference multiple tables (joins, subqueries, CTEs). The rewriter needs policies for all of them.

Step 2: Batch-evaluate policies. Query the policy backend for all tables in one call, keyed by the user's identity and roles. This avoids N round-trips for a query that touches N tables. The result is a `ResolvedPolicy` per table:

```
pub struct ResolvedPolicy {
    pub row_filters: Vec<datafusion::logical_expr::Expr>,
    pub column_masks: std::collections::HashMap<String, MaskType>,
    pub restricted_columns: Vec<String>,
}
```

Step 3: Rewrite the plan. For each TableScan that has a policy, modify the plan tree:

```
let rewritten = plan.transform_down(|node| {
    match &node {
        LogicalPlan::TableScan(scan) => {
            let policy = policies.get(&scan.table_name);
            if let Some(p) = policy {
                let mut node = node;
                // Inject row filters above the scan
                for filter in &p.row_filters {
                    node = LogicalPlan::Filter(
                        Filter::try_new(filter.clone(), Arc::new(node))?
                    );
                }
                // Replace column references with mask expressions
                node = apply_column_masks(node, &p.column_masks)?;
                // Remove restricted columns from the projection
                if !p.restricted_columns.is_empty() {
                    node = restrict_projection(node, &p.restricted_columns)?;
                }
                Ok(Transformed::yes(node))
            } else {
                Ok(Transformed::no(node))
            }
        }
        _ => Ok(Transformed::no(node))
    }
})?;
```

The order within the rewrite matters. Row filters go first (closest to the TableScan), then column masks, then column restriction. This means row filters can reference any column – even columns that will be masked or restricted in the final output. The policy author can write `ROWS WHERE salary > 100000` as a filter even if `salary` is masked for the user querying. The filter evaluates on raw data; the mask applies to what leaves the engine.

Column Masks in Detail

Column masking replaces raw column values with transformed versions. The mask types we've designed cover the most common governance patterns:

Mask Type	Input	Output	Use Case
Redact	`123-45-6789`	`***`	PII that should be completely hidden
Hash	`john@example.com`	`a3f2b7c9...`	Consistent pseudonymization (same input = same hash)
Nullify	`150000`	`NULL`	Numeric data that shouldn't be visible
Custom	`john@example.com`	`j***@example.com`	Arbitrary SQL expression

Each mask type translates to a DataFusion expression:

```
fn mask_expression(col: &Column, mask: &MaskType) -> Expr {
    match mask {
        MaskType::Redact => lit("***"),
        MaskType::Hash => {
            Expr::ScalarFunction(ScalarFunction::new(
                "sha256", vec![cast(col.clone(), DataType::Utf8)]  // requires registered UDF
            ))
        }
        MaskType::Nullify => lit(ScalarValue::Null),
        MaskType::Custom(expr_str) => parse_sql_expr(expr_str),
    }
}
```

The critical point: these expressions replace the column reference in the plan. When the optimizer sees `sha256(cast(ssn as varchar))` in a projection, it doesn't know (or care) that `ssn` was the original column. If a user predicate references `ssn`, the optimizer can only match it against the expression, not the raw column. The mask is structural – it exists in the plan tree, not as a post-processing step.

Field report: The predicate pushdown test. When we designed the mask system, the first thing we wrote was a test: create a plan with a masked column, add a user predicate on that column, run the optimizer, and verify the predicate doesn't push below the mask. The test passed on the first run because DataFusion's optimizer is expression-aware – it won't push a predicate through a function call boundary. We didn't need to write a custom optimizer rule to prevent this. The plan structure prevented it. That's when we knew the approach was sound.

Row Filters: The Invisible Predicate

Row filters are simpler than masks but more subtle in their implications. A row filter is a predicate that restricts which rows a user can see. The user never knows the filter exists.

When the policy says "analyst role can only see rows where `region = 'EU'`", the rewriter injects a `Filter` node:

```
Before:
  Projection [id, amount, region]
    TableScan [transactions]

After:
  Projection [id, amount, region]
    Filter [region = 'EU']       <-- injected, invisible to user
      TableScan [transactions]
```

If Alice, an analyst, runs `SELECT COUNT(*) FROM transactions`, she gets the count of EU transactions. She has no way to know that non-EU transactions exist. She can't write a query that reveals the total count because every query she runs goes through the same rewriter, and every plan gets the same filter injected.

If Alice also writes `WHERE amount > 1000`, the optimizer sees two filters and may combine them:

```
Filter [region = 'EU' AND amount > 1000]
  TableScan [transactions]
```

This is correct. Alice's predicate narrows within the already-filtered set. The optimizer might even push both predicates into the TableScan as partition pruning hints or Parquet row group filters. All safe, because the security filter is conjunctive.

Multiple row filters for the same table are combined with AND. If the policy has two filters – `region = 'EU'` and `status = 'active'` – both are injected, and both must be satisfied. The rewriter doesn't try to be clever about combining them. DataFusion's optimizer handles that.

Column Restriction: The Column That Doesn't Exist

Column restriction is the most aggressive form of policy enforcement. A masked column is visible but transformed. A restricted column is invisible.

When the policy says Alice's role has `restricted_columns = [salary, ssn]` for the `employees` table, the rewriter modifies the projection to exclude those columns. When Alice runs `SELECT *`, she gets every column except `salary` and `ssn`. When she runs `SELECT salary`, she gets "column not found."

This is the PostgreSQL RLS model applied to columns. In PostgreSQL, if a column is denied by a policy, it doesn't appear in `\d` (describe table) and references to it return "column does not exist." We follow the same pattern. The denied column doesn't appear in `information_schema.columns` for that user. It doesn't appear in `SELECT *`. It's as if the column was never defined.

The implementation strips columns from the plan schema:

```rust
fn restrict_projection(node: LogicalPlan, allowed: &HashSet<String>) -> Result<LogicalPlan> {
    let schema = node.schema();
    let exprs: Vec<Expr> = schema.fields()
```

```
        .iter()
        .filter(|f| allowed.contains(f.name()))
        .map(|f| col(f.name()))
        .collect();
    Ok(LogicalPlan::Projection(
        Projection::try_new(exprs, Arc::new(node))?
    ))
}
```

This is different from a column mask, which replaces the column's value but keeps it in the schema. The choice between masking and restriction depends on the governance requirement. Social security numbers might be masked (the user knows the column exists, sees a transformed value). Salary data might be restricted entirely (the user doesn't know the column exists).

OPA as a Policy Backend

Open Policy Agent (OPA) is the first external backend we've designed for. OPA evaluates policies written in Rego – a declarative language for expressing authorization rules. It runs as a sidecar or standalone service, and SQE queries it over HTTP.

The data flow for OPA:

1. Administrator runs `GRANT SELECT (id, amount, ssn MASKED WITH 'REDACT') ROWS WHERE region = 'EU' ON finance.txns TO analyst` via SQL.
2. The coordinator routes this to the PolicyManager, which serializes the grant and writes it to OPA's data API: `PUT /v1/data/sqe/grants/finance.txns/analyst`.
3. OPA stores the grant as structured data alongside Rego rules that know how to resolve grants into policies.
4. When an analyst runs a query, the PlanRewriter calls OPA: `POST /v1/data/sqe/authz` with the user's identity, roles, and the list of tables in the query.
5. OPA evaluates the Rego rules against the stored grants and returns a `ResolvedPolicy` per table.
6. The rewriter applies the policy to the plan.

The Rego rules handle role expansion, conflict resolution (what happens when two grants for the same table disagree on which columns are visible?), and default-deny semantics. This logic lives in the policy engine, not in SQE. SQE asks the question; OPA provides the answer.

```
sqe/
  grants/              # Data written by GRANT SQL statements
    finance.txns/
      analyst.json     # {"columns": ["id","amount"], "masks": {"ssn":"REDACT"}, ...}
  authz.rego           # Rules that resolve grants into effective policy
  authz_test.rego      # Rego unit tests
```

Cedar as an Alternative

Cedar is AWS's authorization policy language, designed for fine-grained access control with entity-based reasoning. Where OPA evaluates Rego rules over HTTP, Cedar evaluates policies locally – the policy engine runs in-process.

Cedar policies look like this:

```
permit(
    principal == User::"alice",
    action == Action::"select",
    resource == Table::"finance.transactions"
) when {
    resource.columns.contains("id") &&
    resource.columns.contains("amount")
};
```

The Cedar backend would embed the Cedar evaluation engine directly in SQE, avoiding the network round-trip to OPA. This makes it attractive for latency-sensitive environments. The trade-off is that Cedar's policy language is less flexible than Rego for complex authorization logic.

Both backends implement the same `PolicyEnforcer` trait. The coordinator doesn't know or care which one is active. This is the standard plugin pattern – define the interface, let the implementation vary.

Neither OPA nor Cedar is fully implemented today. The trait is defined. The PassthroughEnforcer works. The plan rewriting design is complete. The implementation is Phase 5 on the roadmap, and the task breakdown has 50+ items across parser extensions, policy store abstraction, plan rewriting, coordinator integration, and end-to-end testing. We're building on a foundation that was designed from day one to support this – the `policy_enforcer.evaluate()` call has been in the query pipeline since the first commit.

The Policy Cache

Policy evaluation is on the hot path. Every query calls the enforcer. For the PassthroughEnforcer, this is free. For an OPA backend making HTTP calls, it could add tens of milliseconds per query.

The solution is a cache keyed on `(user_id, table_reference)` with a configurable TTL:

```
[policy]
cache_ttl_secs = 60
cache_max_entries = 10000
```

The cache uses `moka`, a Rust async cache with TTL-based eviction. On a cache hit, policy evaluation costs nanoseconds. On a cache miss, it costs one HTTP call to OPA (or one local Cedar evaluation).

Cache invalidation is explicit: when a `GRANT` or `REVOKE` statement is executed, the affected cache entries are invalidated immediately. This means policy changes take effect on the next query, not after the

TTL expires. The TTL handles staleness from external policy changes (an administrator modifying OPA's data directly, outside of SQE's SQL interface).

The target is less than 5 milliseconds of overhead on the cached path, measured against TPC-H Query 1 with an active policy.

The SQL Extensions

Policy management happens through SQL. Not through a REST API. Not through a configuration file. Through the same interface the user uses for everything else.

```
-- Grant column access with a row filter and a mask
GRANT SELECT (id, amount, ssn MASKED WITH 'REDACT')
  ROWS WHERE region = 'EU'
  ON finance.transactions
  TO role_eu_analyst;

-- Revoke access
REVOKE SELECT ON finance.transactions FROM role_eu_analyst;

-- Inspect what grants exist
SHOW GRANTS ON finance.transactions;

-- See the effective policy for the current user (resolved across all roles)
SHOW EFFECTIVE POLICY ON finance.transactions;

-- Admin: see what a specific user would see
SHOW EFFECTIVE POLICY FOR 'alice' ON finance.transactions;
```

The parser strategy wraps `sqlparser-rs` rather than forking it. Standard `GRANT` and `REVOKE` parse normally through sqlparser-rs. A post-parse transform detects our extensions (`MASKED WITH`, `ROWS WHERE`) and converts them to custom `PolicyStatement` AST nodes. `SHOW GRANTS` and `SHOW EFFECTIVE POLICY` are fully custom statement types added to the parser.

This means standard SQL `GRANT` statements still work (routed to Polaris for catalog-level permissions). Only the extended variants with masking and row filter clauses are routed to the policy engine.

The coordinator's statement routing already handles this. The `StatementKind::Policy` variant exists in the code today:

```
StatementKind::Policy(_) => Err(SqeError::NotImplemented(
    "Policy management not configured".to_string(),
)),
```

The error is honest. Policy management isn't configured yet. When it is, that match arm routes to the `PolicyManager` instead of returning an error.

The Connection to Governance Platforms

Policy engines don't exist in a vacuum. In production, the policies enforced by SQE should come from a governance platform – Collibra, Alation, Atlan, or whatever system the organization uses to manage data access.

The connection point is the `PolicyStore` trait:

```rust
#[async_trait]
pub trait PolicyStore: Send + Sync {
    async fn put_grant(&self, grant: &PolicyGrant) -> Result<()>;
    async fn delete_grant(&self, grant_id: &GrantId) -> Result<()>;
    async fn list_grants(&self, table: &ObjectReference, role: Option<&str>)
        -> Result<Vec<PolicyGrant>>;
    async fn evaluate(&self, user: &SessionUser, tables: &[ObjectReference])
        -> Result<HashMap<ObjectReference, ResolvedPolicy>>;
}
```

A Collibra integration would implement `PolicyStore` by reading access policies from Collibra's API. When an analyst runs a query against SQE, the policy evaluation calls Collibra to determine which columns are visible, which are masked, and which rows are filtered. The governance platform is the source of truth. SQE is the enforcement point.

This is the right separation. The governance platform manages the policy lifecycle – who approved the access, when it expires, which regulation requires it. The query engine enforces the policy – rewriting plans, masking columns, filtering rows. Neither system needs to understand the other's internals. They communicate through the `PolicyStore` interface.

Collibra Protect, which I wrote about in an earlier article, does something similar for Snowflake. The model works. The question is always where the enforcement happens. In Snowflake's case, it happens inside a proprietary engine you can't inspect. In SQE's case, it happens in plan rewriting code you can read, test, and audit.

Distributed Execution and Security

In distributed mode, the coordinator rewrites the plan before distributing fragments to workers. Workers receive plan fragments that are already secured. A worker executing a scan fragment for the `finance.transactions` table already has the row filter injected and the column masks applied. The worker doesn't need access to the policy engine. It doesn't need to know what the user's roles are (for authorization purposes – it still uses the user's JWT for authentication to Polaris and S3).

This is an important architectural property. Workers are stateless executors. They receive a plan fragment and a bearer token. The plan fragment tells them what to read and how to transform it. The bearer token tells Polaris and S3 who is reading. The security decisions have already been made by the coordinator.

If a worker is compromised, the attacker gets the plan fragments currently being executed on that worker – which are already policy-enforced. They can't ask the worker to execute a different plan

because the worker only executes what the coordinator sends. They can use the bearer token to make Polaris requests, but that token is scoped to the user's permissions and expires within minutes.

The security model is layered: authentication (JWT validation), authorization (plan rewriting), and enforcement (workers execute only secured fragments). Each layer limits what the next layer can do wrong.

Prior Art

SQE's plan-rewriting approach to security enforcement did not emerge from a vacuum. Other systems have solved this problem, and the differences in how they solved it explain why our approach is portable.

PostgreSQL introduced Row-Level Security in version 9.5 (2015). RLS works by injecting `security_barrier` subqueries at parse time – the planner sees the security predicates as ordinary subqueries and optimizes accordingly. The implementation required modifying PostgreSQL's planner to understand security barrier flags, ensuring the optimizer would not reorder operations in ways that leak information through timing or error side-channels. It works well, but it is deeply coupled to PostgreSQL's planner internals.

Oracle's Virtual Private Database (VPD) predates PostgreSQL RLS by over a decade. VPD attaches PL/SQL functions to tables that generate row-level predicates at parse time. The predicates are injected transparently – the user never sees them. The mechanism is elegant and battle-tested, but it lives inside Oracle's proprietary optimizer.

Apache Ranger takes a different approach for the Hadoop ecosystem. Ranger rewrites Hive and Spark plans before execution, injecting filter and masking operations. It works, but it requires engine-specific plugins – a Ranger plugin for Hive, a different plugin for Spark, another for Trino. Each plugin understands the engine's internal plan representation and rewrites it accordingly.

The key distinction in SQE's design: we rely on the optimizer's own expression boundaries to prevent predicate pushdown through masks, without modifying the optimizer itself. PostgreSQL required planner changes. Ranger requires engine-specific plugins. SQE's approach works with any optimizer that respects expression boundaries in predicate pushdown – which is any well-designed optimizer. The security property comes from the plan structure, not from custom optimizer rules. This makes the approach portable to any engine built on DataFusion, and in principle to any engine that exposes its logical plan for transformation.

One gap worth acknowledging: restricted columns may still appear in `information_schema.columns`. Policy enforcement happens in the query plan, not in the metadata views. A user denied access to the `salary` column will get "column not found" when querying the table, but `SELECT * FROM information_schema.columns WHERE table_name = 'employees'` may still list `salary`. This is a known gap. Fixing it requires the `InformationSchemaProvider` to evaluate policies when constructing virtual table results – feasible, but not yet implemented. PostgreSQL handles this correctly because RLS is integrated into the system catalog views. We will get there.

Implementation Status

Here's where we are:

Component	Status
`PolicyEnforcer` trait	Implemented, in production pipeline
`PassthroughEnforcer`	Implemented, active by default
`policy_enforcer.evaluate()` in query path	Wired into every query, every EXPLAIN
`PolicyPlanRewriter` (row filters, masks, restriction)	Implemented
`PolicyStore` trait	Implemented
OPA backend	Implemented with Rego rules
Cedar backend	In progress
SQL extensions (GRANT, REVOKE, SHOW GRANTS)	Parser routing implemented, handlers wired
Policy cache (moka)	Implemented with async TTL
Collibra integration	Interface implemented, connector available

The architecture was built with the hook in place from the start. The `evaluate()` call has been in the hot path since the first version of the query pipeline. This was a deliberate choice – adding it later would have meant changing the query execution flow, touching `execute_query`, `explain`, and `analyze`. With the hook already there, shipping the policy engine was a crate-level change. The coordinator didn't change. The query pipeline didn't change. Only the implementation behind `Arc<dyn PolicyEnforcer>` changed.

This is one of the advantages of trait-based design in Rust. The interface is a compile-time contract. Any implementation that satisfies `PolicyEnforcer: Send + Sync` with an `evaluate` method that takes a `SessionUser` and a `LogicalPlan` can be plugged in. The type system guarantees it. We shipped `PassthroughEnforcer` first, then `OpaEnforcer`, and the coordinator binary didn't need a single line changed.

The Predicate Pushdown Boundary

This section is worth dwelling on because it's the most subtle security property in the system.

Consider a table with columns `id`, `name`, `ssn`, and `salary`. The policy says: `ssn` is masked with REDACT, `salary` is masked with NULLIFY. Alice writes:

```
SELECT * FROM employees WHERE salary = 153000
```

Without masks, the optimizer would push `salary = 153000` into the TableScan as a Parquet row group filter. This is fast – it avoids reading rows where salary isn't 153000.

With masks, the plan after rewriting looks like:

```
Projection [id, name, '***' AS ssn, NULL AS salary]
```

```
Filter [NULL = 153000]
  TableScan [employees]
```

Alice's predicate is evaluated against the masked value. `NULL = 153000` evaluates to NULL (falsy), so she gets zero rows. She can't distinguish between "no employees earn exactly 153000" and "employees earn 153000 but the mask nullifies the value."

If the optimizer could push the predicate below the mask, it would evaluate `salary = 153000` against the raw value, returning the matching rows (with masked output). The number of rows returned would leak information: "there are 3 employees earning exactly 153000." The mask hides the value but the row count reveals it.

By placing masks in the plan as expressions, the optimizer's own rules prevent this. It won't push a predicate through an expression boundary because that would change the predicate's semantics. We don't need a custom optimizer rule. We don't need to disable predicate pushdown. The plan structure enforces the security property.

Row filters, by contrast, are safe to push through. If the policy says `region = 'EU'` and Alice writes `department = 'engineering'`, pushing Alice's predicate below the row filter is fine – it just means fewer rows are scanned. The security invariant (only EU rows are visible) is maintained because Alice's predicate can only narrow, never widen.

DataFusion deep dive: DataFusion's `PushDownFilter` optimization rule walks the plan tree looking for `Filter` nodes whose predicates can be pushed closer to the data source. The rule respects expression boundaries – it won't push a predicate through a `Projection` that transforms the referenced column. This is the mechanism that protects masked columns. The mask expression (e.g., `floor(salary/10000)*10000`) creates a new expression that the filter references. The optimizer sees this as a different expression from the raw `salary` column and won't push the predicate below the projection. No custom rule needed.

Testing the Security Properties

The test matrix for the policy engine covers three categories:

Functional tests: Does the plan rewriter correctly inject row filters, column masks, and column restrictions? These are unit tests that construct a LogicalPlan programmatically, run the rewriter, and verify the output plan structure.

Security property tests: Does the optimizer preserve the security invariants after running? These are the critical tests. Construct a plan with a masked column and a user predicate on that column. Run the policy rewriter. Run the optimizer. Verify the predicate does not appear below the mask in the optimized plan.

Information leakage tests: Does a denied column appear in any error message? Does a row filter leave any trace in the output? Does EXPLAIN reveal the original column name for a restricted column? These are negative tests – verifying the absence of information.

The Phase 5 task list has 14 integration tests planned, covering scenarios from basic column restric-

tion through combined policies with distributed execution. Every test is a GIVEN/WHEN/THEN scenario:

> GIVEN a policy engine is configured, and user has analyst role, and analyst has `ssn MASKED WITH 'REDACT'`
>
> WHEN user submits `SELECT * FROM hr.employees`
>
> THEN the `ssn` column contains *** for every row
>
> GIVEN a policy engine is configured, and user has analyst role, and analyst has `ROWS WHERE region = 'EU'`
>
> WHEN user submits `SELECT COUNT(*) FROM transactions`
>
> THEN the count reflects only EU transactions

The Lesson

Security enforcement in a query engine is not about checking permissions before execution. It's not about filtering results after execution. It's about transforming the query plan so that unauthorized data never enters the execution pipeline.

The plan is the security boundary. A row filter is a `Filter` node. A column mask is an expression in a `Projection`. A denied column is absent from the schema. The optimizer runs on the secured plan and can't undo the security because the security constraints look like ordinary plan nodes.

This model has three properties worth carrying to other systems. First, deny by omission – never tell the user something is denied; just make it invisible. Second, structural enforcement – use the type system and the plan structure to prevent security bypasses, rather than relying on runtime checks. Third, separation of policy and enforcement – let the governance platform decide who sees what; let the query engine ensure the decision is applied.

The `PolicyEnforcer` trait is twenty-six lines of Rust. The plan rewriting that implements it will be several hundred. The security properties it guarantees come not from the amount of code, but from where it sits in the pipeline: after parsing, before optimization. That placement is the entire design.

AI Logbook: The AI produced the `PolicyEnforcer` trait, `PassthroughEnforcer`, and the `Policy–PlanRewriter` with `transform_down` in three passes. The first pass applied masks and restrictions as independent projections — the second projection discarded the first's mask expressions. The human caught this during code review and restructured the prompt to require a single-pass projection. The security property that masks block predicate pushdown was verified by the human's first test; it passed because DataFusion's optimizer respects expression boundaries, not because we wrote a custom rule.

Making It Operable

A query engine that surprises you is a query engine nobody will trust.

The engine worked. Queries went in, Arrow batches came out, the numbers were correct. We had authentication, catalog integration, a write path, policy enforcement. By any reasonable definition, we had a functioning SQL engine.

We also had no idea what was happening inside it.

A query took 4.2 seconds. Was that slow? We didn't know. Where did the time go – parsing, planning, scanning Parquet files, network transfer? We didn't know that either. A user reported that their dashboard queries were "sometimes slow." Sometimes. We couldn't reproduce it, couldn't measure it, couldn't even confirm it was real.

This is the gap between "it works" and "we can operate it." And it has a twin: the gap between "it runs on my machine" and "someone else can deploy it." Observability tells you what's happening. Configuration determines what *can* happen. Both are prerequisites for trust, and trust is what the teams running production workloads through your engine actually need.

The Three Pillars, Applied

Everyone knows the three pillars of observability: metrics, traces, logs. The concept is not new. What's new is applying it to a distributed query engine where a single SQL statement might touch a coordinator, two workers, a REST catalog, and S3 – all within the same second.

Metrics answer "how much" and "how fast." Counters and histograms, scraped by Prometheus, aggregated over time. They tell you the system's vital signs: queries per second, latency percentiles, error rates. They don't tell you why a specific query was slow.

Traces answer "where did the time go." OpenTelemetry spans, linked across process boundaries, showing the full timeline from SQL parse to Arrow batch delivery. They tell you that query Q spent 200ms in planning and 3.8 seconds waiting for S3. They don't tell you that 40% of your queries are hitting the same pattern.

Logs answer "what happened." Structured JSON, correlated with trace IDs, recording the facts: who ran what query, when, how long it took, whether it succeeded. They're the audit trail, the debugging breadcrumb, and the compliance record.

Each pillar is incomplete alone. Together, they give you the ability to answer any question about your system's behavior – past, present, and (with good alerting) future.

We built all three in a single day. The implementation was straightforward. Getting the *right* metrics took considerably longer.

What to Measure

The first version of our metrics was wrong. Not broken – wrong. We measured what was easy to measure, not what mattered.

We started with a single counter: total queries executed. Then a latency histogram. Then error counts. Basic stuff, straight from every "how to add Prometheus to your Rust service" tutorial.

The problem became clear the first time we tried to debug a slow query report. We knew the p99 latency was 5.2 seconds. We didn't know whether that was parsing, planning, scanning, or network transfer. We knew 2% of queries failed. We didn't know whether they failed during authentication, catalog resolution, or execution.

The metrics registry we ended up with tells a different story:

```
pub struct MetricsRegistry {
    pub registry: Registry,
    pub query_count: CounterVec,       // labels: status, statement_type
    pub query_duration: HistogramVec,  // labels: statement_type
    pub rows_returned: Counter,
    pub active_sessions: IntGauge,
    pub healthy_workers: IntGauge,
    pub cache_hits: Counter,
    pub cache_misses: Counter,
    pub cache_invalidations: Counter,
    pub cache_size_bytes: Gauge,
    pub cache_entries: Gauge,
}
```

The `query_count` counter has two label dimensions: `status` (success or error) and `statement_type` (query, insert, ctas, show, explain). This single counter answers: "How many SELECT queries succeeded in the last hour? How many INSERTs failed?" Without the `statement_type` label, you're averaging reads and writes together, which hides everything useful.

The `query_duration` histogram uses custom bucket boundaries tuned for query engine latencies, not HTTP request latencies. A 10ms query is a metadata lookup. A 250ms query is a well-optimized scan. A 5-second query is a multi-table join. A 60-second query is either a big analytical scan or something is wrong. The default Prometheus buckets are heavily weighted toward sub-second values – useless for distinguishing "normal slow" from "abnormal slow" in a SQL engine.

Workers have their own registry, tuned to what matters at the execution layer: fragments executed, rows scanned, bytes read, fragment duration. Fragment-level metrics are the only way to answer the question that matters most in a distributed engine: "Is this query slow because the work is genuinely

large, or because one worker is struggling?" If all workers report similar fragment durations, the query is just big. If one worker's fragments take 10x longer, you have a hot spot.

DataFusion's `ExecutionPlan` trait exposes `metrics()` on every physical operator, returning per-operator row counts and elapsed compute time. We surface these through `EXPLAIN ANALYZE`, which executes the query and returns a tree of operators with actual row counts, elapsed milliseconds, and output sizes. This complements Prometheus metrics – Prometheus shows system-level trends, `EXPLAIN ANALYZE` shows query-level detail. When an operator reports scanning 10 million rows but only 500 survive the filter, you know the predicate pushdown isn't reaching the Parquet reader.

Both registries implement a `HasRegistry` trait, which means the metrics server code is generic – same endpoint shape, same scrape config for coordinator and worker. The coordinator runs Prometheus on port 9090, workers on 9091. In docker-compose these map to host ports 29090, 29091, 29092, all scrapable by a single Prometheus instance. The only difference is the prefix: coordinator metrics use `sqe_`, worker metrics use `sqe_worker_`. Grafana dashboards use the same panel layouts for both, swapping the prefix.

The recording happens inside `QueryHandler::execute`, unconditionally – success or failure. The `status` field comes from whether the result was `Ok` or `Err`, and the `kind_name` comes from the SQL classifier, not from the execution result. We record the statement type even when the query fails, which is critical for answering "are SELECT queries failing more than INSERTs?"

The metrics fields are `Option<Arc<MetricsRegistry>>` rather than always-present. In integration tests, we often create a `QueryHandler` without a metrics registry because the test doesn't care about metrics. Making them optional avoids the overhead of maintaining a Prometheus registry in test contexts. In production, they're always present.

The Dashboard That Matters

After weeks of running the engine, we settled on six panels that tell you almost everything:

Panel	Metric	Why
Query rate	`rate(sqe_query_count_total[5m])` by status	Is the engine being used? Are queries failing?
Query latency	`histogram_quantile(0.95, sqe_query_duration_seconds)`	Are queries fast enough? Is latency drifting?
Active sessions	`sqe_active_sessions`	How many users are connected right now?
Healthy workers	`sqe_healthy_workers`	Is the cluster capacity what we expect?

Panel	Metric	Why
Cache hit rate	`rate(sqe_cache_hits_total[5m]) / (rate(sqe_cache_hits_total[5m]) + rate(sqe_cache_misses_total[5m]))`	Is the result cache helping?
Worker scan throughput	`rate(sqe_worker_bytes_read_total[5m])`	Are workers keeping up with scan demand?

The seventh panel – added after an incident – is `rate(sqe_query_count_total{status="error"}[5m])` with an alert threshold. When more than 5% of queries fail over a 5-minute window, something is wrong. During normal operation, this number is zero. When it's not zero, you want to know immediately.

Not everything that can be measured should be alerted on. We learned this by over-alerting during the first week and then ignoring all alerts because there were too many. The alerts that survived: error rate above 5% for 5 minutes (catches auth failures, catalog failures, S3 throttling), p95 latency above 30 seconds for 10 minutes (catches systemic slowness), healthy workers below expected count for 3 minutes (catches crashes and network partitions), and zero queries for 15 minutes during business hours (catches the "everything looks fine but nothing is happening" failure mode).

The alerts we removed: cache hit rate below 50% (fired constantly, depends entirely on workload) and individual fragment duration above 10 seconds (some fragments are legitimately large). Noise that obscures real problems is worse than no alerting at all.

The Query That Scanned Too Much

This is the story that justified every hour spent on observability.

During load testing, one particular TPC-H query was taking 8x longer than expected. The Prometheus dashboard showed the p95 climbing. But the average latency was fine. Something was wrong with a specific query pattern.

We pulled up the OTel trace for one of the slow executions. The coordinator spent 50ms on parsing and planning. The two workers together spent 7.3 seconds on scanning. But the query was simple – a filtered aggregation on a small table. It should have scanned a few files, not taken 7 seconds.

The worker metrics told the rest of the story. `sqe_worker_bytes_read_total` for that fragment was 10x higher than expected. The worker was reading every Parquet file in the table instead of only the ones matching the filter.

The root cause: the Iceberg manifest filter wasn't being applied. Our scan task was sending all data file paths to the worker instead of only the files whose partition values matched the query predicate. The fix was in the planner – apply partition pruning before constructing the scan task. The observability stack found the bug. Without per-worker bytes-read metrics and per-query traces, we'd have been guessing.

Traces Across Process Boundaries

Prometheus metrics tell you how the system is performing. OpenTelemetry traces tell you how a specific request flows through it. For a distributed query engine, this distinction is the difference between "our p95 is 5 seconds" and "this specific query spent 4.7 of those 5 seconds waiting for a worker to read from S3."

The hardest part of distributed tracing is not generating spans. It's connecting them. When the coordinator dispatches a scan fragment to a worker, the worker's execution span needs to be a child of the coordinator's dispatch span. Otherwise you get two disconnected traces that happen to overlap in time, and the whole point is lost.

We solved this with W3C TraceContext propagation over gRPC metadata. The injector wraps tonic's `MetadataMap` and inserts `traceparent`/`tracestate` headers at the point of dispatch in the `DistributedScanExec`:

```
let ticket = Ticket::new(ticket_bytes);
let mut request = tonic::Request::new(ticket);
inject_trace_context(parent_cx, request.metadata_mut());
```

On the worker side, `do_get` extracts the parent context from the incoming gRPC metadata and links its execution span to the coordinator's trace:

```
let parent_cx = extract_trace_context(request.metadata());
let worker_span = info_span!(
    "worker_execute_scan",
    fragment_id = %scan_task.fragment_id,
    file_count = scan_task.data_file_paths.len(),
);
worker_span.set_parent(parent_cx);
```

The result: a single trace shows the full lifecycle of a distributed query. SQL parse on the coordinator. Plan optimization. Fragment dispatch. Worker execution. Parquet reads. Arrow batch transfer. All connected, all with accurate timing.

We also propagate trace context to Polaris REST catalog calls via HTTP headers, so that catalog operations appear in the same trace as the query that triggered them. When `EXPLAIN ANALYZE` says planning took 800ms, the trace shows you that 750ms was waiting for Polaris to return table metadata. That tells you the fix is to warm the catalog cache, not to optimize the scan.

One subtle detail in the OTel initialization: the filter that prevents telemetry-induced-telemetry loops. Without filtering `hyper`, `tonic`, `h2`, `reqwest`, and `tower` from the OTel log bridge, every export generates HTTP logs, which get exported via OTel, which generate more HTTP logs. The system enters an infinite loop that saturates the network. We found this during the first integration test. The symptom – exponentially growing memory usage – was alarming before we traced it to the log bridge.

The `OtelGuard` RAII type holds the tracer, meter, and logger providers for the lifetime of the process. On drop, it flushes all pending spans and metrics to the OTLP endpoint. The shutdown order matters: meter first, tracer second, logger last. The tracer might generate log events during its shutdown. The logger needs to be alive to capture them. We got this wrong initially and lost shutdown-related log

entries. Not a production outage, but the kind of thing that makes debugging harder exactly when it matters.

Audit Logging

Metrics and traces serve the operations team. Audit logs serve a different audience: security, compliance, and forensics.

Every query that executes in SQE produces an audit entry: timestamp, username, session ID, query hash, statement type, duration, rows returned, status, and client IP. Every field is there for a reason. `session_id` links to the Flight SQL session. `statement_type` enables filtering – show me all DDL operations. `duration_ms` catches slow queries. `rows_returned` catches queries that return suspiciously large result sets.

The `query_hash` field deserves explanation. It's a SHA-256 of the normalized SQL – whitespace collapsed, keywords uppercased. `SELECT 1 FROM t` and `select  1  from  t` produce the same hash. This lets you correlate queries across time without storing raw SQL in every entry. If you want to find all executions of a specific query pattern, hash the pattern and search for the hash. If you need the actual SQL, the optional `query_text` field has it – configurable to omit in production if the SQL itself contains sensitive data.

The audit logger is append-only JSONL to a file, with flush-after-every-write to ensure a coordinator crash doesn't lose the last few entries. When no audit log path is configured, the logger is a no-op – no allocation, no I/O, no lock contention. The `Option` pattern again: zero cost when disabled, full fidelity when enabled.

Sovereignty principle: Audit logging is non-negotiable for sovereign infrastructure. If you can't answer "who queried what, when, and what did they see?" then you don't control your data platform – you're just hosting it. The audit log is the proof that your access policies are being enforced. Without it, policies are promises.

Health Endpoints

Kubernetes needs to know three things about your pod: is it alive, is it ready, and what's its status? We serve these on a dedicated port.

`/healthz` (liveness probe) returns `"ok"` unconditionally. If the HTTP server can respond, the process is alive. No logic, no checks.

`/readyz` (readiness probe) returns `200` only when initialization is complete – auth provider configured, catalog reachable, workers (if any) registered. The `ready` flag is an `AtomicBool` set after all initialization completes. The health server starts *before* initialization, so Kubernetes sees the pod as alive but not ready during the startup window.

`/api/v1/status` (cluster status) returns JSON with node role, version, uptime, and worker health. For human consumption and dashboards, not Kubernetes probes.

The health port is always `prometheus_port + 1` – computed, not configured. One less thing to configure, one less thing to get wrong. The validation step checks that the Prometheus port doesn't collide with the Flight SQL port, which is the only plausible conflict.

Workers have a simpler health model, but the coordination protocol adds its own dimension. Workers send heartbeats every 5 seconds; the coordinator considers a worker healthy if it has sent a heartbeat within the last 15 seconds. The coordinator also runs active health checks via Arrow Flight. Belt and suspenders: heartbeats prove the worker can reach the coordinator, active checks prove the coordinator can reach the worker. Both directions matter in a network where firewalls or service mesh policies might allow traffic in one direction but not the other.

The `sqe_healthy_workers` gauge on the coordinator tracks how many workers are currently healthy. This is the single most important metric for capacity planning. If you have 4 workers and this gauge drops to 2, your scan capacity just halved.

Field report: During distributed docker-compose testing, we initially had the readiness probe checking `/healthz` instead of `/readyz`. The coordinator would accept connections before worker registration completed, which meant the first few queries executed locally instead of distributed. The symptom was confusing: "Why are my distributed queries not distributing?" Because the load balancer started sending traffic before workers were ready.

From Observability to Configuration

Observability tells you what's happening. Configuration determines what *can* happen. And the uncomfortable truth about prototyping is that the hardcoded version ships faster. Every constant you pull out into configuration is a design decision you have to make explicit. What's the default? What's the range? What happens when someone puts in garbage?

The first version of SQE had about a dozen constants scattered across three crates:

```
// Early version -- don't do this
const FLIGHT_SQL_PORT: u16 = 50051;
const POLARIS_URL: &str = "http://localhost:8181/api/catalog";
const KEYCLOAK_URL: &str = "http://localhost:8080";
```

These worked perfectly for local development against the quickstart Docker Compose stack. They were also completely useless for anything else.

The first time someone else tried to run SQE – a colleague who had a different Polaris endpoint and a different OIDC provider – it didn't compile for them. Not "didn't work." Didn't *compile*. Because the Keycloak URL was baked into the binary. They had to clone the repo, find the right constant, change it, and wait for `cargo build` to finish.

That was the moment I understood: configuration isn't a feature you add after the engine works. Configuration *is* the product. The engine itself is implementation detail. The configuration surface is what operators actually interact with.

The Twelve-Factor App methodology calls this "storing config in constants" and lists it under things to stop doing immediately. But knowing the principle and feeling the pain are different things. The

pain arrived when we tried to run integration tests in CI against a different stack and realised we'd need conditional compilation just to change an endpoint URL.

Antipattern: Constants as configuration. When you hardcode values because "we'll fix it later," you're making a bet that later comes before someone else needs to deploy your software. That bet almost always loses. The first external user showed up two days before we planned to extract the config.

Why TOML, and the Norway Problem

The format choice was quick. YAML lost for one specific reason: the Norway problem. In YAML, the string `no` is interpreted as a boolean `false`. The value `3.10` becomes the float `3.1`. Port `8080` could be an integer or a string depending on context. These implicit coercions are bugs waiting to happen in a config file where port numbers, boolean flags, and string identifiers all live together.

TOML has explicit types. `8080` is always an integer. `"8080"` is always a string. `true` is always a boolean. No helpful type guessing that silently converts your region string into something else.

The Rust ecosystem sealed the decision. `serde` plus `toml` gives you typed deserialization with error messages that point to the exact line and column. Every section in the TOML maps to a subsystem in the engine:

```
[coordinator]
flight_sql_port = 50051
trino_http_port = 8080
worker_urls = ["http://worker-1:50052", "http://worker-2:50052"]

[auth]
token_endpoint = "http://polaris:8181/api/catalog/v1/oauth/tokens"
client_id = "root"
client_secret = "s3cr3t"

[catalog]
polaris_url = "http://polaris:8181/api/catalog"
warehouse = "test_warehouse"

[storage]
s3_endpoint = "http://rustfs:9000"
s3_access_key = "s3admin"
s3_secret_key = "s3admin"
s3_region = "us-east-1"
s3_path_style = true
```

An operator reading this file sees the architecture. `[coordinator]` is the Flight SQL server. `[auth]` is OIDC. `[catalog]` is Polaris. `[storage]` is S3. You don't need to read the source to understand what these sections do.

The Config Struct

The TOML deserializes directly into a typed Rust struct:

```rust
#[derive(Debug, Deserialize, Clone)]
pub struct SqeConfig {
    pub coordinator: CoordinatorConfig,
    #[serde(default)]
    pub worker: WorkerConfig,
    pub auth: AuthConfig,
    pub catalog: CatalogConfig,
    #[serde(default)]
    pub storage: StorageConfig,
    #[serde(default)]
    pub policy: PolicyConfig,
    #[serde(default)]
    pub metrics: MetricsConfig,
    #[serde(default)]
    pub rate_limit: RateLimitConfig,
    #[serde(default)]
    pub session: SessionConfig,
    #[serde(default)]
    pub query: QueryConfig,
    #[serde(default)]
    pub query_cache: QueryCacheConfig,
    #[serde(default)]
    pub query_history: QueryHistoryConfig,
}
```

Two things to notice. First, `#[serde(default)]` on most sections means a minimal config file only needs `[coordinator]`, `[auth]`, and `[catalog]`. Everything else gets sensible defaults. A developer running locally doesn't need to configure metrics, rate limits, session timeouts, or query caching. They just work.

Second, every field has a concrete type. Ports are `u16`. Timeouts are `u64`. Memory limits are strings with a parser that handles `"512MB"`, `"8GB"`, `"1TB"` – case-insensitive, with or without the B suffix. The type system catches misconfigurations at startup, not at 3am when a query hits the wrong code path.

The subsection structs each carry their own defaults via `serde(default = "fn_name")`. Every default is a function, not an attribute value – a serde requirement for non-trivial defaults that has a side benefit: all defaults live in one place at the bottom of the config module. Worker memory defaults to 8GB, heartbeat to 5 seconds, spill directory to `/tmp/sqe-spill`. Changing a default is a one-line change that shows up clearly in diffs.

The zero-config experience matters. A developer should be able to clone the repo, start the quickstart stack, and run `cargo run` with a minimal config file. No `[worker]` section needed – defaults handle it. No `[policy]` section – defaults to passthrough. No `[session]` section – 15 minute idle timeout and 8 hour absolute timeout. No `[query_cache]` section – caching enabled by default with 256MB and 5

minute TTL. Required sections – the ones that depend on your specific infrastructure – don't have defaults. If you forget `catalog.polaris_url`, the TOML parser fails immediately with a clear error. Not at runtime when the first query tries to reach Polaris.

Environment Variable Overlay

TOML files work well for base configuration. They don't work for secrets, and they don't work for per-deployment overrides in Kubernetes.

The solution is a layered model: TOML first, then environment variable overrides. Every config field can be overridden by setting `SQE_<SECTION>__<FIELD>`. Double underscore as the separator, because single underscore is already used within field names.

```
SQE_CATALOG__POLARIS_URL="http://polaris-prod:8181/api/catalog"
SQE_AUTH__CLIENT_SECRET="production-secret-from-vault"
SQE_COORDINATOR__FLIGHT_SQL_PORT=50052
```

The implementation is explicit – every overridable field is listed in one function. We rejected a reflection-based approach that would auto-map any `SQE_*` variable. The explicit listing means typos in variable names are silently ignored rather than silently applied to the wrong field, and we control which fields are overridable. Some fields, like TLS certificate paths, should only come from the config file or a mounted secret.

A bad override logs a warning and keeps the TOML value. It doesn't crash the process. In Kubernetes, you might have a stale ConfigMap with a variable referencing a renamed field. Crashing on that would take down the service during a rolling upgrade. Warning is the right response.

The Helm chart makes secrets explicit:

```
env:
  - name: SQE_AUTH__CLIENT_SECRET
    valueFrom:
      secretKeyRef:
        name: sqe-secrets
        key: SQE_AUTH__CLIENT_SECRET
        optional: true
```

Secret values flow from Kubernetes Secrets into environment variables, which override whatever the TOML has. The TOML in the ConfigMap never contains secrets. Custom `Debug` implementations on `AuthConfig` and `StorageConfig` redact sensitive fields – `client_secret` renders as `"[REDACTED]"` – so even a startup config dump to logs won't leak credentials.

The type-specific override functions handle parsing carefully. A boolean override accepts `"true"`, `"1"`, `"yes"` and their negatives; anything else logs a warning and keeps the TOML value. This matters: a misconfigured environment variable shouldn't silently change behavior, but it also shouldn't crash the service during a rolling upgrade where a stale ConfigMap references a field that's been renamed.

The docker-compose file for the distributed test stack shows the layered model in action:

```yaml
services:
  sqe:
    command: ["--config", "/etc/sqe/sqe.toml"]
    environment:
      SQE_CATALOG__POLARIS_URL: "http://polaris:8181/api/catalog"
      SQE_AUTH__CLIENT_SECRET: "${SQE_CLIENT_SECRET:-sqe-secret-change-me}"
      SQE_METRICS__OTLP_ENDPOINT: "http://jaeger:4317"
```

The TOML in the container image provides the base structure and defaults. The environment variables override deployment-specific values. The `${SQE_CLIENT_SECRET:-sqe-secret-change-me}` syntax means the secret comes from the host environment if set, falling back to a development default. Same model the Helm chart uses, scaled up.

Plugin Points

The config surface is real. Every struct shown here is production code. The plugin architecture – the extension points for external contributors – is designed but not fully populated.

The pattern is consistent across subsystems. The policy engine has a `PolicyEnforcer` trait:

```rust
#[async_trait]
pub trait PolicyEnforcer: Send + Sync {
    async fn evaluate(
        &self,
        user: &SessionUser,
        plan: LogicalPlan,
    ) -> sqe_core::Result<LogicalPlan>;
}
```

Today `[policy] engine = "passthrough"` instantiates the only implementation. The plan is that `engine = "opa"` would instantiate an OPA-backed enforcer, `engine = "cedar"` a Cedar-backed one. The trait is the plugin point. The config key selects which implementation loads.

The same pattern applies to auth and catalog. DataFusion's `CatalogProvider` trait is the extension point for catalog backends. Our Polaris implementation is the first. Adding a second – Gravitino, Unity, in-memory – would follow the same trait, selected by a config key.

We defined the traits before having multiple implementations because the open-source target demands it. The traits are the promise: "you can swap this out." The config keys are the interface: "here's how you select the swap." The risk is that the traits are slightly wrong for what OPA or Cedar actually need. We accept this. The trait is four lines. Changing it is a breaking change in a pre-1.0 project. The cost of getting it slightly wrong is low. The cost of not having it – of shipping a monolithic engine that can't be extended without forking – is the end of the open-source goal.

Rate Limits and Sessions

Two config sections arrived later, when we started thinking about multi-tenant deployments:

```
[rate_limit]
enabled = true
per_user_queries_per_minute = 60
global_queries_per_minute = 1000

[session]
idle_timeout_secs = 900          # 15 minutes
absolute_timeout_secs = 28800    # 8 hours
```

Rate limiting is off by default. An internal deployment where all users are trusted doesn't need it. A shared deployment where a runaway dbt job could monopolize the engine does. The config key is the toggle.

Session timeouts are on by default with values that work for interactive use. A dbt batch job that runs longer than 8 hours needs the absolute timeout increased. A query dashboard open all day needs the idle timeout extended. These are operator decisions, not engine decisions. The defaults are sensible; the overrides are available.

Failing Fast on Bad Config

The engine calls `validate()` immediately after loading the config, before starting any subsystem. The validation accumulates all errors before reporting:

```
let config = SqeConfig::load(&config_path)?;
config.validate()?;  // Fail here, not later
```

An operator with five misconfigurations gets all five reported at once, not one at a time across five restarts. Validation runs before any network I/O – the engine doesn't try to connect to Polaris to see if the URL is valid. Connection problems are runtime errors; config problems are startup errors. The port conflict check prevents a class of bug otherwise diagnosed at the OS level as "address already in use" with no hint about which config keys conflict. The TLS check ensures that if one of cert_file and key_file is set, both must be set.

The full config loading sequence runs from binary startup to ready state: CLI parsing (`--config` flag or `SQE_CONFIG` env var), file read, TOML parse with serde, environment overlay, deprecation warnings for renamed keys, validation, and subsystem initialization. Step 5 matters: we renamed `auth.keycloak_url` when we realised the auth system works with any OIDC provider. The old key still works – it maps to the same field – but the engine logs a deprecation warning. This is how config keys evolve in a sovereign system. You don't break existing deployments. You warn, give time, and remove in a major version.

Of approximately 45 individual config keys across 12 sections, only 3 are truly required: `auth.client_id`, `catalog.polaris_url`, and one of `auth.keycloak_url` or `auth.token_endpoint`. Everything else has a default that works for local development.

The config surface also has tests. The `valid_config()` helper constructs a known-good configuration, and each validation test mutates one field to confirm the correct error fires. Every default gets a test confirming its value. This is important for a subtle reason: defaults are documentation. When an

operator reads the test suite and sees `assert_eq!(config.memory_limit, "8GB")`, they know that omitting the memory limit from their TOML gives them 8GB. The tests are the spec.

The Twelve-Factor Connection

The Twelve-Factor App methodology was written for web services. A query engine isn't a web service, but five of the twelve factors apply directly.

III. Config – Store config in the environment. The TOML handles the base, environment variables handle per-deployment overrides, secrets never touch disk.

IV. Backing services – Treat backing services as attached resources. Polaris, S3, Keycloak – each configured by URL. Swapping from test to production Polaris is a config change, not a code change. This matters more for a query engine than for a typical web app, because the backing services are the entire reason the engine exists.

X. Dev/prod parity – The same binary runs everywhere. The same TOML structure, different values. SQE doesn't have a "development mode." It has config keys like `auth.ssl_verification = false` that you can set explicitly, and a deprecation warning if you do.

XI. Logs – Treat logs as event streams. SQE logs to stdout/stderr and sends traces to an OTLP collector when configured. No log file management in the engine.

VI. Processes – Execute the app as stateless processes. Workers are completely stateless. The coordinator holds session state in memory but is designed for session affinity, not sticky state.

SQE builds as a single binary – `sqe-server` – that runs as either coordinator or worker based on a `--mode` flag, an `SQE_MODE` environment variable, or a config key. Both modes share the same config file structure. A coordinator ignores the `[worker]` section. A worker ignores `[coordinator].worker_urls`. In Kubernetes, the same ConfigMap is mounted to both coordinator and worker pods. The only difference is the `SQE_MODE` environment variable. Same image, same config, different mode. One config surface to understand.

[To be completed by AI Logbook agent]

Tying It Together

The observability stack in SQE is not a separate system. It's wired into the query pipeline at every step. The `execute` method on `QueryHandler` instruments with `#[tracing::instrument]`, which creates an OTel span. Inside that span, it records Prometheus metrics. After execution, it writes the audit log. The worker's `do_get` extracts the parent trace context and creates a child span. The executor records per-fragment metrics.

Every query that flows through the engine produces:

- A Prometheus counter increment and histogram observation
- An OTel trace with spans for parse, plan, dispatch, and execute
- A structured audit log entry

None of these require manual instrumentation at the call site. The `QueryHandler` and `WorkerFlightService` handle it. If you add a new statement type, the metrics and audit log pick it up automatically because they're driven by the SQL classifier, not by statement-specific code.

This matters more than any individual metric choice. The best observability stack is the one that works without the developer remembering to add it. The worst is the one with gaps because someone forgot to instrument a code path.

The configuration surface follows the same principle. Every new feature adds config keys following an established pattern: add a section, define defaults, add validation, add tests, add env override support. The pattern scales because it's mechanical. No judgment calls about where to put the config key or how to name it. The TOML section is the subsystem. The field is the behavior selector. The default is what "normal" looks like.

The full configuration surface as of this writing has 12 sections and roughly 45 individual keys:

Section	Keys	Required
`coordinator`	7	No (has defaults)
`worker`	6	No
`auth`	7	Yes (client_id, one endpoint)
`catalog`	4	Yes (polaris_url)
`storage`	6	No
`policy`	1	No
`metrics`	3	No
`rate_limit`	3	No
`session`	2	No
`query`	2	No
`query_cache`	4	No
`query_history`	2	No

The surface will grow. Each new feature – pluggable auth backends, OPA integration, Cedar rules, compaction scheduling – will add config keys. The pattern is established. The pattern scales.

The Enterprise Checklist

We spent months adding the pieces that make an engine observable and configurable. Then we spent more months on the pieces that make it trustworthy in an enterprise context — the ones that matter when the security team asks for a review or when a production incident happens at 2am and you need to answer specific questions fast.

None of these features were in the original design. They arrived one by one, each time someone asked a question we couldn't answer or revealed an assumption we hadn't examined.

Structured Error Codes

The first version of error handling was honest about its limitations: errors came back as strings. A DataFusion parse error looked exactly like a catalog connection failure, which looked exactly like an OPA policy denial. All of them returned HTTP 500 or a generic gRPC INTERNAL status, because that was the path of least resistance.

The problem surfaced during integration testing with the dbt adapter. When a model failed, dbt logged the error string. The string was useful for debugging but useless for automated handling. How do you distinguish "table not found" from "access denied" from "out of memory" if they all say `Internal error`?

We introduced 27 `SqeErrorCode` variants that carry semantics rather than just messages:

Category	Examples	gRPC Status	Trino Code
Auth	`Unauthorized`, `Forbidden`, `SessionExpired`	UNAUTHENTICATED, PERMISSION_DENIED	65536, 65537
Catalog	`TableNotFound`, `SchemaNotFound`, `CatalogUnavailable`	NOT_FOUND, UNAVAILABLE	65540, 65541
Execution	`InvalidQuery`, `TypeMismatch`, `DivisionByZero`	INVALID_ARGUMENT	65536+n
Resources	`QueryTimeout`, `MemoryLimitExceeded`, `TooManyRequests`	RESOURCE_EXHAUSTED	65550+n
System	`InternalError`, `SerializationError`, `ConfigError`	INTERNAL	65535

The classifier that assigns codes is built around a simple principle: user errors get detail, system errors get redaction. If a table isn't found, the error message says which table. If Polaris returns a 503, the message says "catalog unavailable" — not the internal details of which HTTP call failed or what the retry sequence looked like. The user can't fix a Polaris outage. The operator can, and they have the logs.

An auto-classifier parses DataFusion error message strings to assign error codes. DataFusion doesn't yet have a typed error enum stable enough to match on, so we pattern-match on the string representations. It's not elegant, but it's isolated to one module and tested against the actual error strings DataFusion produces. When DataFusion's error types stabilize, swapping the classifier for a type-match is a contained change.

Security Hardening at Startup

We added startup warnings for three conditions: TLS disabled on the Flight SQL port, rate limiting disabled, and SSL certificate verification disabled. Each logs at `WARN` level during initialization, before accepting any connections.

The philosophy is fail-open for development, fail-loud for production. We don't refuse to start without TLS — the quickstart stack runs over plain HTTP and that's intentional. But we print a warning that is hard to miss:

```
WARN sqe_coordinator: TLS is DISABLED on Flight SQL port 50051 -- do not use in production
WARN sqe_coordinator: Rate limiting is DISABLED -- concurrent queries are unlimited
WARN sqe_coordinator: SSL certificate verification is DISABLED -- catalog connections are insecure
```

These are the settings a developer enables for a local run against a self-signed cert and forgets to turn off before deploying. The warnings are there because that exact scenario happened during our first staging deployment. Everything worked. The security team noticed the unencrypted Flight SQL port during a network scan two weeks later.

What we don't do yet: block startup on these conditions. We discussed it. The argument for blocking is that it prevents the accidental insecure deployment. The argument against is that it breaks legitimate airgapped deployments and internal-only environments. We landed on warning, with a documented `--allow-insecure` flag for environments where the operator has made a deliberate choice. The flag exists in the config schema. We haven't added it to the validation yet.

Client IP Logging

Every request — Flight SQL and Trino HTTP — now logs the client IP address alongside the query audit entry. The implementation has two layers: `x-forwarded-for` header parsing for requests arriving through a reverse proxy or ingress, and TCP peer address fallback for direct connections.

The ordering matters. A request arriving through an nginx ingress carries the real client IP in `x-forwarded-for`; the TCP peer address is the ingress pod. If we logged the TCP address, we'd have a perfect record of which ingress pod received the traffic and nothing about who sent the query.

We take the leftmost address in `x-forwarded-for`, which is the one set by the client, not the one appended by intermediate proxies. This is correct when the ingress is trusted. In a deployment where the network perimeter isn't controlled, a client can spoof the header. The fix is to strip untrusted `x-forwarded-for` headers at the ingress level — not something the query engine can solve. We document this limitation.

Circuit Breaker for Polaris

Catalog availability is not guaranteed. Polaris goes through rolling restarts. Network partitions happen. The default behavior — retry with backoff — has a failure mode that took us a while to fully appreciate: during a Polaris outage, the coordinator accumulates threads blocked waiting for HTTP responses. When Polaris returns, those threads all try to reconnect simultaneously. The thundering herd effect can cause a second outage immediately after recovery.

The circuit breaker solves this by separating "is Polaris healthy?" from "should I try to reach Polaris?"

```
Closed  ──5 failures──▶  Open  ──30s──▶  Half-Open  ──success──▶  Closed
                                                    ──failure──▶  Open
```

In the closed state, requests reach Polaris normally. Five consecutive failures open the circuit. In the open state, requests fail immediately without making a network call — fast failure rather than blocked threads. After 30 seconds, the circuit moves to half-open, allowing a single probe request. If the probe succeeds, the circuit closes. If it fails, the circuit reopens.

The thresholds are configurable, but the defaults reflect what we found worked during outage testing: 5 failures is enough signal, 30 seconds is enough recovery time for a Polaris pod restart, a single probe avoids the thundering herd. The circuit breaker state is local to each coordinator process. In a multi-coordinator deployment, each coordinator makes its own circuit decisions. This is deliberate — a circuit that's shared across coordinators requires distributed state, which is a harder problem than the one the circuit breaker is solving.

PII Redaction in Audit Logs

The audit log captures `query_hash` (SHA-256 of normalized SQL, safe to store) and optionally `query_text` (the raw SQL). The hash supports correlation without reconstruction — you can find all executions of a query pattern without storing the query itself. The text supports debugging.

The problem with `query_text`: SQL WHERE clauses contain user data. `SELECT * FROM orders WHERE customer_email = 'alice@example.com'` is a realistic query. That email address doesn't belong in the audit log.

We added regex-based stripping of four PII categories before writing `query_text` to the audit entry:

Pattern	What it strips
Email addresses	`\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z]{2,}\b`
Phone numbers	Common formats: +1-555-0100, (555) 010-0100, etc.
SSNs	`\b\d{3}-\d{2}-\d{4}\b`
Credit card numbers	13-16 digit sequences with Luhn-plausible formatting

Each match is replaced with `[REDACTED:<TYPE>]`. The redacted SQL is still readable. The pattern of the query is preserved. The personal data is not.

This is not a complete solution. SQL can encode PII in ways that don't match these patterns — base64-encoded data, numeric IDs that happen to be SSNs without dashes, string constants in languages other than English. The redaction is best-effort defense-in-depth, not a compliance guarantee. The guarantee comes from having `query_text` configurable-off for high-sensitivity deployments. If you can't afford PII in audit logs at all, disable `query_text`. The hash is always safe.

Per-Query Resource Limits

Multi-tenancy requires protecting the engine from individual queries that consume disproportionate resources. Four limits apply to every query:

Limit	Default	Enforcement
`max_result_rows`	1,000,000 rows	Post-execution count; query cancelled if exceeded
`max_concurrent_queries`	100	Semaphore at `QueryHandler::execute` entry
`max_query_memory`	256 MB	DataFusion `GreedyMemoryPool`
`slow_query_threshold_secs`	30 seconds	WARN log after threshold; query continues

The `max_result_rows` limit is intentionally post-execution because we can't know the result size before executing. The trade-off: a query that generates 2M rows uses the compute budget to produce them, then gets cancelled when we count the output. This wastes work. The alternative — stopping mid-scan — requires streaming result counting during execution, which DataFusion doesn't expose cleanly. We document this as a known limitation and recommend users add LIMIT clauses for exploratory queries.

The concurrency semaphore has one subtle behavior: when the engine is at capacity, new queries wait rather than fail immediately. The wait timeout is 5 seconds. A query that can't acquire the semaphore in 5 seconds gets a `TooManyRequests` error. This gives burst capacity for genuine traffic spikes while protecting against indefinite queue growth.

The `GreedyMemoryPool` in DataFusion allocates memory greedily and cancels the query if allocation would exceed the limit. "Cancel" means the next allocation attempt returns an error, which propagates up through the execution tree. The cancellation is clean — DataFusion unwinds the execution future — but it's abrupt. The user sees `MemoryLimitExceeded`. They don't see a partial result. For analytics workloads this is correct behavior. For streaming results (large Arrow batches delivered incrementally), we return what we have and signal cancellation on the next batch request.

Slow query logging is the most operational of the four. A query that takes 31 seconds doesn't fail — it completes and produces results. But the coordinator logs a structured warning with the query ID, username, duration, and a truncated query text. This gives operators a passive alert that something is running long, without interrupting it. The alerts we removed from Prometheus included individual query duration; this replaced them. Prometheus tracks the distribution. The slow query log identifies the specific offenders.

Session Persistence

Session state — authenticated users, open Flight SQL sessions, per-session settings — lives in memory on the coordinator. A coordinator restart means every active session is lost. Users see authentication failures and reconnection prompts. For a dbt job mid-run, this means the job fails and must restart.

The initial implementation had a file-based snapshot: every 5 minutes (configurable), the coordinator serializes all sessions to a JSONL file at a configured path. On startup, if the file exists, sessions are restored. The user reconnection window is the restart duration, not the full re-authentication cycle.

What we have is not HA. It's warm restart. The distinction matters: a crash still causes disruption; the persistence only helps with planned restarts (upgrades, config reloads). True HA requires session state in an external store — Redis being the obvious choice — and a leader election mechanism so multiple coordinators share state without conflicts. The `SessionStore` trait is defined and the file-based implementation is the first backend. A Redis implementation would follow the same trait. We're not there yet. We know what "there" looks like.

The `/readyz` Evolution

The readiness probe started as a boolean: is initialization complete? That answered one question but not the one operators actually need: "Is this coordinator capable of serving queries right now?"

A coordinator that has initialized but can't reach Polaris is not ready. It will accept connections and then fail every query with `CatalogUnavailable`. Kubernetes won't remove it from load balancer rotation because `/readyz` still returns 200.

The evolved `/readyz` makes a lightweight Polaris reachability check — an HTTP HEAD to the catalog base URL — before returning success. If Polaris is unreachable, `/readyz` returns 503 with a JSON body describing which dependency failed:

```
{
  "status": "not_ready",
  "checks": {
    "initialized": true,
    "polaris_reachable": false,
    "workers_registered": true
  },
  "failed": ["polaris_reachable"]
}
```

Kubernetes removes the pod from rotation on 503. Traffic stops flowing to a coordinator that can't serve it. When Polaris recovers, the check passes, the pod re-enters rotation.

The cost: every readiness check makes a network call. With Kubernetes polling every 5 seconds, that's 12 catalog pings per minute per coordinator pod. We added a 10-second cache on the Polaris reachability result to reduce this to at most 6 pings per minute during normal operation, with a fresh check guaranteed when the cached result is "not reachable."

What we don't have yet: leader election, so that only one coordinator is "primary" for write operations. For read-only query traffic this doesn't matter — all coordinators are equivalent. For session state and write coordination, it matters significantly. The HA story is the next chapter we haven't written yet.

The Security Audit That Changed the Defaults

After the engine was fast, correct, and observable, we asked: would a bank approve this for production?

The answer was 43 findings across six categories. Two critical, thirteen high, twenty-one medium, seven low. The engine ran 221 out of 222 benchmark queries correctly and beat Trino by 2.5x to 8.8x. It still failed the audit.

The critical finding was the session context cache. Keyed by username. Two users sharing a username from different identity providers would share a Polaris catalog session. Cross-user data access. The fix was straightforward: key by `username:sha256(token)[..16]`. But it required switching moka's `get/insert` to `try_get_with` for atomic cache population, which also eliminated a TOCTOU race condition where concurrent requests from the same user would build redundant SessionContexts.

The most tedious finding was panic safety. Sixteen call sites in date extraction functions used `.unwrap()` on `date32_to_datetime()`, which returns `Option<NaiveDateTime>`. A query calling `year()` on a Date32 column with an extreme value would panic and kill the coordinator. Not return an error. Kill the coordinator. Every one of those sixteen sites needed individual attention because the containing functions had different error handling patterns. Some were in `.map()` closures where `?` does not work. Those needed conversion to explicit loops.

The finding that taught us the most was adaptive sort. We initially set the default to `partition_only`. Safest possible: never sort on non-partition columns, never OOM from unbounded sorts. Eight integration tests immediately failed. Every test that used `ORDER BY salary DESC` got unsorted results. The lesson: the safest default and the correct default are not always the same.

The right default is `adaptive`. Sort normally when memory is available. Strip non-partition sorts when memory pressure rises. Never crash. Never silently return wrong results on small data. The FairSpillPool memory limit is the backstop. If the sort exceeds memory, DataFusion spills to disk. If spill is not configured, the adaptive stripper removes the sort before the executor runs out of memory. Two layers of protection, transparent to the user.

Other findings were smaller but cumulative. Nine Flight SQL metadata endpoints with no authentication check. The cancel-query endpoint letting any client cancel any other user's query. The OPA policy cache ignoring role changes. OIDC error bodies forwarded verbatim to clients, enabling user enumeration. Blocking `std::fs::write` on Tokio worker threads. The `checksum()` UDF using `DefaultHasher`, which is not stable across Rust versions. The audit logger silently dropping records on mutex poison.

We fixed all 43. Thirty-three files changed. +1,272 / -372 lines. Every unit test still passed. All sixty integration tests passed.

The audit doc lives at `docs/issues.md`. It lists every finding by severity, with file:line references, what was wrong, how it was fixed, and why it matters. The document is not a badge. It is a maintenance artifact. When someone changes the session cache or the auth chain or the sort logic, they can check the audit doc to understand why the code looks the way it does.

Sovereignty principle: A production security audit is not optional for sovereign infrastructure. If

you skip it, you are deploying hope. The 43 findings were not surprising. They are the normal output of building software quickly and then reviewing it carefully. The difference between a prototype and a production system is not the code. It is the review.

The Lesson

We started with no observability and hardcoded constants. We added counters. Then histograms. Then traces. Then audit logs. Then trace propagation. Then health probes. Then a config file. Then environment overlays. Then validation. Then a full security audit. Each addition was prompted by a specific question we could not answer or a specific deployment we could not support.

The engine that "worked" without any of this was the same code, executing the same queries. The difference is that now we know what it is doing, and someone other than the person who wrote it can deploy it. No surprises.

Configuration taught us something we did not expect: it forced architectural clarity. When you name a config section, you decide what that subsystem is. When you define defaults, you decide what "normal" looks like. When you write validation, you decide what is required and what is optional. The TOML file is not a reflection of the architecture. It *is* the architecture, expressed for operators.

If your query engine surprises you, your metrics are wrong. If nobody else can deploy it, your config is wrong. If it panics on user input, your error handling is wrong. Fix all three.

Why Distribute at All

> The fastest distributed query is the one that runs on a single node.

SQE worked. By the end of Part III, we had a query engine that authenticated users via OIDC, queried Iceberg tables through Polaris, enforced row-level security through plan rewriting, wrote data via CTAS and INSERT INTO, exported Prometheus metrics, and served it all over Arrow Flight SQL. One binary. One process. No cluster.

The question wasn't whether it worked. The question was how long it would keep working.

The Comfortable Plateau

For our initial workloads, single-node SQE was more than enough. DataFusion is fast. Iceberg's metadata layer prunes partitions before a single byte of Parquet is read. Column projection means you only deserialize what the query needs. On a machine with 16 cores and 64GB of RAM, we could scan tens of gigabytes per query without breaking a sweat.

The numbers were good. A full scan of a 5GB Iceberg table with aggregation completed in under two seconds. An analyst's typical query – filtered by date partition, projecting five columns, grouped by region – came back in 200 milliseconds. The dbt models ran. The dashboards refreshed. Nobody complained.

This is the part of the story where many teams make their first mistake. The engine is fast. The users are happy. So you start planning distributed execution because you assume you'll need it eventually. You spin up a Kubernetes cluster, deploy coordinators and workers, add health checks and heartbeat protocols, build a scheduler, handle partial failures, and six months later you have a distributed system that's slower than the single-node version for every query your users actually run.

We almost did this. The architecture docs from week one included distributed execution in Phase 3. The temptation was real. But we had a rule: measure first.

Antipattern: Distributed by Default. "We might need to scale" is not a requirement. "We're scanning 500GB per query across 200 partitions" is. Start single-node. Measure. Then decide. The operational cost of distribution is not zero, and you pay it on every query, even the ones that would have been faster without it.

Where Single-Node Stops

We ran TPC-H at increasing scale factors on a single node. Scale factor 1 (roughly 1GB): queries completed in under a second. Scale factor 10: a few seconds. Scale factor 100: some queries took minutes. We profiled.

The bottleneck was not CPU. DataFusion's vectorized execution on Arrow columnar data is remarkably efficient. Aggregations, hash joins, sort-merge joins – the compute kernels were not the problem. The bottleneck was scan I/O: reading Parquet files from S3.

This makes sense when you think about it. A query against a 100GB Iceberg table might touch hundreds of Parquet files. Each file requires an HTTP GET to S3. Even with connection pooling and parallel reads, a single node has a finite number of network sockets, a finite amount of bandwidth, and a finite amount of memory to buffer incoming data. The CPU sits idle while the network fills the pipeline.

We measured the breakdown:

Query Phase	% of Wall Clock (SF-100)
SQL parsing + planning	< 1%
Iceberg metadata + partition pruning	2-5%
Parquet scan (S3 reads)	60-75%
Filter + projection	5-10%
Aggregation / join	10-20%
Result serialization	< 1%

The scan phase dominated. And the scan phase is embarrassingly parallel – each Parquet file can be read independently, by any machine that has the credentials and knows the file path.

This is the key insight that drives everything in Part IV. The parallelism we need is not compute parallelism. It's I/O parallelism. We don't need more CPUs. We need more network pipes reading from S3 simultaneously.

Amdahl's Law for Query Engines

Gene Amdahl told us in 1967: the speedup from parallelism is limited by the sequential fraction of the workload. If 25% of your work is inherently serial, no amount of parallelism will give you more than a 4x speedup.

For a query engine, the serial fraction includes:

- **SQL parsing and planning.** One coordinator parses the SQL, builds the logical plan, applies policy rewrites, runs the optimizer, and produces the physical plan. This cannot be parallelized. It's also cheap – typically under 10 milliseconds.
- **Final aggregation.** A `SELECT count(*) FROM lineitem` can count rows in parallel across workers, but someone has to sum the partial counts. A `SELECT ... ORDER BY ... LIMIT 10`

can sort locally, but the final merge-sort and top-10 selection happens in one place.

- **Result assembly.** The coordinator collects results from workers and streams them to the client. One connection, one stream.

The parallel fraction includes:

- **Scanning.** Reading Parquet files from S3. This is the big one. Each file is independent. Each worker reads its assigned files in parallel with every other worker.
- **Filtering and projection.** Applied locally at each worker after scanning. No coordination needed.
- **Local aggregation.** Partial aggregates computed per worker before being sent to the coordinator for final aggregation.

For our scan-heavy workloads, the parallel fraction was 60-75% of wall clock time. Amdahl's Law says that with a 70% parallel fraction:

Workers	Theoretical Speedup	Actual (measured)
1	1.0x	1.0x
2	1.5x	1.4x
4	2.1x	1.8x
8	2.6x	2.1x
16	2.9x	2.2x

The gap between theoretical and actual comes from coordination overhead: serializing scan tasks, shipping them over gRPC, collecting results, network latency. Each of these costs is small individually, but they add up. And they're fixed costs – you pay them whether the query scans 1GB or 100GB.

The table tells you something important. Going from 1 to 2 workers gives you a meaningful speedup. Going from 8 to 16 gives you almost nothing. The law of diminishing returns is not a guideline. It's a law.

The Crossover Point

The crossover point is the data volume where two workers on smaller machines outperform one worker on a bigger machine. Below this point, distribution is pure overhead. Above it, distribution pays for itself.

For SQE, we found this point empirically. We ran the same TPC-H queries on three configurations:

1. **Single node:** 16 cores, 64GB RAM, 10Gbps network
2. **Two workers:** each 8 cores, 32GB RAM, 10Gbps network
3. **Four workers:** each 4 cores, 16GB RAM, 10Gbps network

At scale factor 10 (~10GB), the single node won every query. The overhead of distributing – serializing scan tasks, sending them over gRPC, collecting results – exceeded the time saved by parallel scanning. The data just wasn't big enough for I/O to be the bottleneck.

At scale factor 50 (~50GB), two workers matched the single node on scan-heavy queries and lost on aggregation-heavy ones. The crossover was happening.

At scale factor 100 (~100GB), two workers consistently beat the single node by 30-40% on scan-heavy queries. Four workers beat it by 50-60%. The aggregation-heavy queries still favored fewer, bigger machines.

The crossover point for our hardware and network configuration was roughly 30-50GB of scanned data per query. Below that, single-node was faster. Above it, distribution paid for itself.

Your number will be different. It depends on your network bandwidth, your S3 endpoint's throughput, your machine specs, and your query patterns. The point is not the specific number. The point is that this number exists, and you should find it before committing to a distributed architecture.

Partition-Level Parallelism

Iceberg tables are partitioned. The partition scheme defines how data files are organized – by date, by region, by customer ID, whatever makes sense for the query patterns. When a query includes a predicate on the partition column, Iceberg's metadata layer prunes partitions that can't contain matching rows. This happens before any data is read.

After pruning, you're left with a set of data files that need to be scanned. These files are the natural unit of distribution. Each file is self-contained: it has its own schema, its own row group statistics, its own column chunks. Any machine with S3 credentials and a Parquet reader can process it independently.

SQE's distribution model works at this level. The coordinator:

1. Plans the query and produces a physical plan with an `IcebergScanExec` node
2. Asks the `IcebergScanExec` for its data file paths (post-pruning)
3. Splits those file paths across available workers
4. Replaces the `IcebergScanExec` with a `DistributedScanExec` that fans out to workers

The splitting is straightforward. The `split_files` function in `sqe-planner` distributes files using the weighted scheduler:

```rust
/// Distributes data file paths across N workers using round-robin assignment.
pub fn split_files(files: Vec<String>, num_workers: usize) -> Vec<Vec<String>> {
    if num_workers == 0 || files.is_empty() {
        return vec![];
    }
    let mut groups: Vec<Vec<String>> = (0..num_workers).map(|_| Vec::new()).collect();
    for (i, file) in files.into_iter().enumerate() {
        groups[i % num_workers].push(file);
    }
    groups
}
```

Round-robin is the starting point. The `WeightedScheduler` improves on this by assigning the heaviest tasks first (largest-first bin packing) and tracking accumulated load per worker. If a worker already

has in-flight fragments, it gets fewer new ones. If a worker is unhealthy, it gets none.

The key decision: distribution happens only at the scan level. Filters, projections, and local aggregations run on the same worker that scanned the data. Only the partial results travel back to the coordinator for final aggregation. This minimizes network transfer – the most expensive operation in a distributed query.

The Decision Framework

After measuring, we built a decision framework. Not a flowchart – a set of questions with clear answers.

Question 1: How much data does the query scan?

If the answer is under 10GB after partition pruning, stay single-node. The coordination overhead will eat any parallelism gain. DataFusion on a single machine will saturate your CPU before it saturates your network on datasets this small.

Question 2: Is the query scan-heavy or compute-heavy?

Scan-heavy queries (full table scans, large range scans, broad aggregations) benefit most from distribution. The bottleneck is I/O, and adding workers adds I/O bandwidth.

Compute-heavy queries (complex multi-way joins, window functions, nested subqueries) benefit less. The bottleneck is CPU, and distribution doesn't help much because the serial fraction (final join, final sort) is large. For these queries, a bigger single machine is often better than several smaller ones.

Question 3: What's the concurrency?

This is the factor that catches people by surprise. A single query scanning 50GB might run fine on one node. But 20 users each scanning 50GB simultaneously will not. Concurrency is where distribution earns its keep – not by making individual queries faster, but by giving each query its own I/O bandwidth.

A single node running DataFusion has a thread pool sized to its CPU core count. When one query occupies those threads, other queries wait. This isn't a design flaw – it's resource contention. DataFusion handles it gracefully with task scheduling, but there's a hard ceiling. A 16-core machine can run 16 parallel scan tasks. If you have 20 concurrent queries each wanting 8 parallel scans, the math doesn't work.

With two workers and 20 concurrent queries, the scheduler distributes fragments across both workers. Each worker handles half the I/O load. Each worker has its own thread pool, its own network connections, its own memory budget. Individual queries might not be faster, but the system processes twice as many queries per second. For a dbt pipeline running 30 models with `--threads=8`, this is the difference between a 40-minute batch and a 20-minute one.

Question 4: Is the data growing?

If you're scanning 5GB today and the table grows 1GB per month, you'll cross the distribution threshold in two years. If it grows 10GB per month, you'll cross it in three months. Plan for the trend,

not the snapshot.

Here's the framework as a table:

Condition	Recommendation
Scanned data < 10GB, low concurrency	Single-node. Don't distribute.
Scanned data 10-50GB, scan-heavy queries	Test both. Measure the crossover for your hardware.
Scanned data > 50GB, scan-heavy queries	Distribute. The I/O parallelism will pay for itself.
Compute-heavy queries (complex joins)	Prefer vertical scaling (bigger machine) over horizontal.
High concurrency (>10 concurrent queries)	Distribute for throughput, even if individual queries don't speed up.
Data growing > 5GB/month	Plan for distribution now, implement when you cross the threshold.

The SQE Mode Toggle

We built SQE so that the same binary runs in both modes. The config file controls the behavior:

```
[coordinator]
mode = "coordinator"
worker_urls = ["http://worker-1:50052", "http://worker-2:50052"]
```

The legacy values `hybrid`, `local`, and `distributed` are accepted as aliases but resolve to `coordinator` mode internally. The coordinator automatically falls back to local execution when no workers are healthy.

The behavior is straightforward. Workers are optional. If workers are registered and healthy, the coordinator distributes scan work to them. If no workers are available – because none are configured, or because they've all crashed – the coordinator falls back to local execution. This is how SQE ran for the first two weeks of development, how it runs in integration tests, and how it should run when your data fits on one machine. When workers are present and healthy, the coordinator plans and schedules; workers execute.

The fallback logic lives in `try_distribute`:

```
async fn try_distribute(
    &self,
    plan: Arc<dyn ExecutionPlan>,
    session: &Session,
    query_id: &uuid::Uuid,
) -> Arc<dyn ExecutionPlan> {
    // No worker registry? Execute locally.
    let registry = match self.worker_registry {
        Some(ref r) => r,
        None => return plan,
```

```
    };

    // No healthy workers? Execute locally.
    let healthy = registry.healthy_workers().await;
    if healthy.is_empty() {
        return plan;
    }

    // No IcebergScanExec in the plan? Execute locally.
    let scan_node = match find_iceberg_scan(&plan) {
        Some(node) => node,
        None => return plan,
    };

    // Fewer files than workers? Not worth distributing.
    let file_paths = scan_node.data_file_paths().await;
    if file_paths.len() < healthy.len() {
        return plan;
    }

    // Worth distributing. Build the DistributedScanExec.
    // ...
}
```

Each guard clause returns the original plan unchanged. The coordinator doesn't know or care whether it's running in single-node or distributed mode. It just asks: "Can I distribute this? Should I?" If both answers are yes, it does. If either answer is no, it runs locally.

This design means you can start with a single `sqe-server` binary, no workers, no cluster, no Kubernetes. When the data grows past the crossover point, you add workers. The coordinator discovers them through heartbeats, starts distributing scan work, and everything else stays the same. The SQL is the same. The Flight SQL connection is the same. The auth flow is the same. The policy enforcement is the same.

Sovereignty principle: Distribution should be an operational decision, not an architectural one. The same engine, the same binary, the same config format. You add capacity by adding workers, not by migrating to a different system. The sovereignty thesis applies to your own infrastructure too – you shouldn't be locked into a deployment model.

The Guard Clauses

The `try_distribute` method has five guard clauses, and each one was added because we hit a real problem.

No worker registry. If `worker_urls` is empty in the config, no `WorkerRegistry` is created. The coordinator doesn't even have the data structure to track workers. Distribution is impossible and no cycles are wasted checking.

No healthy workers. Workers register via heartbeat. If all workers have missed three consecutive heartbeats (15 seconds), they're marked unhealthy. The coordinator falls back to local execution rather than failing the query. We added this after a test where we killed all workers and expected graceful degradation. We got query failures instead.

No IcebergScanExec. Not every query touches Iceberg tables. `SHOW TABLES`, `SELECT * FROM system.runtime.queries`, `EXPLAIN` – these are metadata queries that run entirely on the coordinator. There's nothing to distribute.

Fewer files than workers. If a query touches three Parquet files and you have four workers, one worker would sit idle. Worse, the coordination overhead (serializing scan tasks, gRPC calls, result collection) exceeds the time saved. The threshold is simple: if you can't give every worker at least one file, don't distribute.

Multiple scan nodes. A query with a join between two Iceberg tables has two `IcebergScanExec` nodes. Distributing both requires coordinating which worker gets which files from which table, and the shuffle between the join sides is expensive. We haven't built this yet. When we detect multiple scan nodes, we fall back to local execution. The comment in the code says "joins – not yet supported." Honest beats ambitious.

Field report: The "fewer files than workers" guard was the last one we added. We discovered it during TPC-H scale factor 0.01, where some tables have a single Parquet file. The coordinator was dutifully serializing a scan task, sending it to a worker over gRPC, waiting for the result – and the entire round trip took longer than just reading the file locally. The fix was one comparison: `if total_files < num_workers { return plan; }`.

When Not to Distribute

I want to be explicit about this, because the rest of Part IV is about distributed execution and it would be easy to lose the thread.

Don't distribute development and test workloads. Your integration tests run against scale factor 0.01. Adding workers to this makes it slower, not faster, and adds failure modes that have nothing to do with what you're testing.

Don't distribute metadata-heavy workloads. If your users spend most of their time running `SHOW TABLES`, `DESCRIBE`, and `EXPLAIN`, they're hitting information_schema virtual providers, not Iceberg tables. A single coordinator handles this efficiently.

Don't distribute because it looks good on an architecture diagram. I've seen teams deploy Kubernetes operators, Helm charts, worker autoscaling, and distributed tracing infrastructure for workloads that would run faster on a single m5.4xlarge. The cost isn't just money. It's operational surface area. Every worker is a process that can crash, a network connection that can stall, a container that can be OOM-killed. You trade one problem (slow queries) for a different problem (operational complexity).

Don't distribute until you've exhausted vertical scaling. A machine with 32 cores and 128GB of RAM running DataFusion can process a surprising amount of data. DataFusion's thread pool scales

linearly with CPU cores for scan and filter operations. Before adding workers, try a bigger machine. If the bigger machine solves your problem, congratulations – you don't have a distributed systems problem.

The right time to distribute is when you've measured, found the bottleneck is I/O bandwidth, confirmed that vertical scaling has plateaued, and the data volume justifies the operational cost. For most teams, that's later than they think.

The Cost You Pay

Every distributed query pays a tax that local queries don't. This tax is small per query, but it's never zero.

Serialization. The coordinator builds a `ScanTask` for each worker, serializes it to JSON, and packs it into a Flight `Ticket`. The worker deserializes it, configures its S3 client, and begins scanning. This adds 1-5 milliseconds per fragment.

Network round-trips. The coordinator opens a gRPC channel to each worker, sends the ticket, and receives a stream of Arrow record batches. Each gRPC connection has a handshake cost. Each batch has framing overhead. On our test network (10Gbps, sub-millisecond latency), this adds 5-20 milliseconds per fragment.

Coordination. The coordinator tracks fragment state, handles progress callbacks, manages the `tokio::select!` that races execution against a timeout deadline, and performs final aggregation. This is CPU work that doesn't exist in single-node mode.

Failure handling. The `DistributedScanExec` checks worker health, retries failed fragments on different workers, and falls back to local execution when no workers are available. This code path is never exercised in single-node mode. In distributed mode, it's always present, always consuming a few CPU cycles for health checks, always adding a few milliseconds of latency for the health-check evaluation.

The total tax for a query that touches four workers: roughly 30-50 milliseconds of overhead. For a query that scans 100GB and takes 30 seconds, this is noise. For a query that scans 100MB and takes 200 milliseconds, it's a 15-25% penalty. This is why the guard clauses matter. Don't distribute small queries.

What We Built (And What We Deferred)

The distribution model in SQE as of this writing handles the common case: scan-heavy queries against a single Iceberg table, distributed across a pool of stateless workers. This covers our primary workload – batch analytics, dbt models, dashboard queries.

What we built:

- Scan-level distribution with round-robin and weighted scheduling
- Automatic fallback to local execution when distribution isn't beneficial

- Worker health tracking via heartbeats with automatic failover
- Fragment retry on worker failure (up to two attempts per fragment)
- Credential passthrough so workers read S3 as the authenticated user

The weighted scheduler deserves a mention. The naive approach is round-robin: give worker 1 files 0, 3, 6; give worker 2 files 1, 4, 7; and so on. This works when all files are the same size. They never are. Iceberg data files vary in size depending on when they were written, how the data was partitioned, and whether compaction has run. The `WeightedScheduler` estimates each scan task's cost by file count, sorts tasks heaviest-first, and assigns each one to the worker with the lowest accumulated load. Largest-first bin packing. It's a well-known heuristic, and it produces good-enough balance without the complexity of optimal scheduling algorithms that NP-hard problems would require.

What we deferred:

- **Distributed joins.** Queries that join two large Iceberg tables still run locally. The shuffle cost of redistributing both sides of a join across workers is significant, and the scheduling complexity is an order of magnitude higher. This is the next major feature.
- **Distributed aggregation.** Partial aggregations run on workers, but the final aggregation runs on the coordinator. For queries with high-cardinality GROUP BY, this creates a bottleneck. Two-phase aggregation with distributed merge is planned but not implemented.
- **Data locality.** Workers don't have local caches. Every scan reads from S3. A worker that's co-located with an S3 shard would be faster, but our S3 endpoints (RustFS in test, AWS S3 in production) don't expose locality information.
- **Dynamic scaling.** The worker pool is static – defined in config. Autoscaling based on query backlog is an operational concern we pushed to Kubernetes HPA rather than building into the engine.

There are also limitations within what we did build:

- **Partition skew.** The weighted scheduler estimates cost by file count. File count is a poor proxy when predicate selectivity varies across partitions. A partition with 10 files where the predicate matches 1% of rows finishes in milliseconds. A partition with 10 files where the predicate matches everything takes seconds. The scheduler sees them as equal. Manifest-level statistics (row counts, column min/max) could improve this, but we do not read them during scheduling today.
- **No partial aggregation pushdown.** Workers return raw batches. A `COUNT(*)` query reads every row from every worker back to the coordinator, which then counts them. The correct optimization is to count locally on each worker and return a single integer per fragment. DataFusion supports partial aggregation in its physical plan, but wiring it through the `ScanTask` protocol is not trivial. It is on the roadmap.
- **Straggler mitigation.** If one worker is slow – S3 throttling, noisy neighbor, GC pause on the Polaris JVM – the entire query waits. The weighted scheduler reduces the probability of stragglers by balancing load, but it cannot eliminate them. Speculative execution (launching a duplicate fragment on a different worker when one is late) is a known technique we have not implemented. The complexity is significant: you need cancellation, deduplication, and a cost model that knows when speculation is worth the extra I/O.

- **Coordinator NIC saturation.** Every result byte flows through the coordinator on its way to the client. For low-selectivity queries that return large result sets, the coordinator's network interface becomes the bottleneck. Four workers each streaming 1GB of results saturate a 10Gbps NIC in seconds. Direct-to-client streaming from workers would solve this, but it breaks the trust model where the coordinator is the only client-facing component.

Each of these is a real limitation. Each has a workaround. And each is less important than getting the basic distribution model right and reliable, which is what Chapters 12 through 14 are about.

The 1TB Problem on Small Servers

Everything above assumes you can give the coordinator enough memory to hold intermediate results. But what happens when the data doesn't fit?

The Memory Math

Consider `SELECT * FROM lineitem ORDER BY l_shipdate` on a 1TB `lineitem` table. The sort operator must consume the entire input before producing any output. On a single coordinator, that means 1TB of intermediate data in memory – or, with spill-to-disk, 1TB of sorted runs on local storage. Even with efficient external merge sort, the coordinator's NIC and disk bandwidth become the bottleneck.

Now add eight workers. Each worker sorts its 125GB partition locally (spilling 125GB to its own local disk). The coordinator performs a k-way merge of eight pre-sorted streams, consuming only a small buffer per stream. Total spill per worker: 125GB. Total memory on the coordinator for the merge: 8 * buffer_size. The work and the spill are distributed.

The same arithmetic applies to aggregation. A `GROUP BY` with millions of distinct groups requires a hash table proportional to the group count. On one coordinator, that hash table must fit in memory (or the engine OOMs – DataFusion's `GroupedHashAggregate` does not spill today). With two-phase aggregation across eight workers, each worker handles 1/8 of the groups, and the hash table fits.

How Others Handle This

Trino originally had no spill support at all – the entire intermediate dataset had to fit in memory across the cluster. Presto/Trino added spill-to-disk in later versions, but it remains opt-in and the documentation warns about performance degradation. Trino's approach is "provision enough memory" first, spill as a safety net.

Spark takes the opposite approach: it spills aggressively and early. Every shuffle writes to disk. Every sort writes to disk. The `tungsten` off-heap memory manager and `ExternalSorter` make this efficient, but the baseline cost of always writing to disk is non-trivial. Spark's model assumes disk I/O is cheap (true for local NVMe, less true for network-attached storage).

DuckDB proves that a single-node engine can handle datasets far larger than memory. Its buffer manager pages data between memory and disk transparently, with out-of-core hash join and sort. DuckDB processes 1TB on 16GB machines by treating disk as an extension of memory. The limitation is single-node I/O bandwidth – one machine, one NIC, one disk controller.

ClickHouse avoids the problem for most workloads by using pre-aggregated materialized views and merge trees. For ad-hoc queries, it distributes across shards but each shard processes its partition independently. ClickHouse's model works well for append-heavy workloads but requires schema-level planning that Iceberg's schema-on-read approach doesn't impose.

SQE's Hybrid Approach

SQE combines both strategies:

Phase A (spill first): the coordinator uses `FairSpillPool` with watermarks to manage memory pressure. Sort operators spill sorted runs to disk. Join operators fall back to SortMergeJoin when the build side exceeds the memory threshold. Late materialization and file pruning reduce the amount of data that enters the pipeline in the first place. This handles the "survival" case: queries complete correctly, though large sorts and aggregations may be slow.

Phase B (push computation down): the coordinator decomposes the physical plan into stages and pushes computation to workers via Arrow Flight `DoExchange`. Sorts are range-partitioned across workers. Aggregations run in two phases (partial on workers, final on coordinator or designated workers). Joins use broadcast, shuffle hash, or pre-sorted merge depending on input size. This handles the "performance" case: queries complete quickly because work and memory pressure are distributed.

The key design choice is that Phase A is always active. Even in distributed mode, each worker uses `FairSpillPool` for its local execution. Phase B adds distribution on top of Phase A's memory safety. A worker that runs out of memory spills to disk rather than crashing – the same safety net applies at every level.

The Lesson

Distribution is not a feature. It's a trade-off. You trade simplicity for throughput. You trade one failure mode (slow queries) for many failure modes (worker crashes, network partitions, gRPC hangs, S3 throttling, credential expiry, partial results). You trade a single process you can attach a debugger to for a fleet of processes communicating over the network.

The trade-off is worth it when the numbers say so. Not when the architecture diagram says so. Not when the job description says so. Not when the conference talk says so.

We measured. We found the crossover point. We built the mode toggle so the same binary works either way. And we added guard clauses so the system itself decides, per query, whether distribution is worth the cost.

The fastest distributed query is the one that runs on a single node. Don't distribute until you must. And when you must, the next three chapters will show you how.

AI Logbook: The AI implemented `try_distribute` with its five guard clauses and the `split_files` round-robin function in one pass. The human ran TPC-H at increasing scale factors to find the crossover point where distribution pays for itself — 30-50GB of scanned data on our hardware. The "fewer files than workers" guard clause was the last one added, after the human observed that distributing a

single-file scan at scale factor 0.01 was slower than reading it locally. The AI never questioned whether distribution was needed; the human had to measure first.

Standing on Ballista's Shoulders

> Don't build a distributed scheduler from scratch. Build on one that already works, then make it yours.

Apache Ballista exists for a reason. Building a distributed query execution framework is years of work. You need plan serialisation. You need worker registration. You need a scheduler that knows which workers are alive and which have died since the last heartbeat. You need a wire protocol that can move Arrow batches between processes without copying them into JSON and back. You need all of this before you write a single line of query logic.

Ballista already has all of it. It's DataFusion's official distributed execution layer, and it's built by the same people who build DataFusion. The protobuf serialisation understands DataFusion's plan nodes. The workers speak Arrow Flight. The scheduler tracks worker health with heartbeats.

So the question wasn't whether to use Ballista. The question was *how much* of Ballista to use.

The Three Paths

When you need distributed execution for a DataFusion-based engine, you have three options. We considered all of them.

Path 1: Use Ballista as-is

Ballista ships a `SchedulerServer` and an `ExecutorServer`. You configure them, point them at each other, and submit queries to the scheduler. The scheduler parses SQL, creates a physical plan, splits it into stages, assigns stages to executors, and collects results. It's a complete distributed query engine.

The problem is that it's *too* complete. Ballista's scheduler handles everything from SQL parsing to result collection. SQE's coordinator already does SQL parsing. SQE's auth layer already manages sessions and bearer tokens. SQE's policy engine already rewrites plans before optimisation.

Plugging Ballista in as-is would mean either duplicating all that work (running SQE's pipeline *and then* feeding the result into Ballista's pipeline) or abandoning SQE's pipeline and trying to inject auth and policy into Ballista's internals.

We tried the first approach for about a day. The coordinator would plan the query, enforce policy, produce a physical plan, then hand the plan to Ballista's scheduler for distribution. The scheduler would re-plan it. Filters that the policy engine had injected were being rearranged by Ballista's optimiser

pass. Column masks that were carefully positioned to block predicate pushdown were getting pushed down anyway.

Dead end: Ballista as a black box. We tried wrapping Ballista's scheduler as a backend for SQE's coordinator. The plan went in with policy filters attached. The plan came out with policy filters rearranged. Ballista's internal optimiser pass didn't know that certain filter nodes were security boundaries, not performance hints. Two days of work, and we learned that you can't treat a query planner as a transparent pass-through.

Path 2: Build from scratch

The other extreme. Ignore Ballista entirely. Write our own protobuf schema for plan serialisation. Write our own worker registration protocol. Write our own heartbeat mechanism. Write our own scheduler.

This is the approach that gives you the most control and the most work. We estimated three to four months for a production-quality distributed execution layer, based on the scope of what Ballista provides: protobuf codecs for every DataFusion plan node, Arrow Flight integration for result streaming, worker lifecycle management, and stage-based execution with shuffle support.

The protobuf codec alone is a significant undertaking. DataFusion 52 has over forty physical plan node types. Each needs a protobuf message definition, an encoder, and a decoder. Each encoder must handle all configuration variants – a `HashJoinExec` alone has join type, join filter, null equality behaviour, partition mode, and projection. Getting any of these wrong means silent data corruption, because the deserialized plan produces different results than the original. Ballista's codebase has thousands of lines of codec tests for a reason.

We didn't have three months. We had the ambition to go from zero to distributed execution in under two weeks. The math didn't work.

Path 3: Surgical fork

Take Ballista's ideas. Take its serialisation model. Take its execution philosophy. Don't take its scheduler. Don't take its auth model. Don't take its configuration surface.

This is the path we chose. Not a fork in the Git sense – we didn't clone the Ballista repository and start modifying it. We studied Ballista's architecture, understood its codec design, and reimplemented the parts we needed with SQE's constraints baked in from the start.

The key insight: Ballista's most valuable contribution isn't its scheduler or its executor process. It's the *pattern*. The idea that a DataFusion physical plan can be serialised to protobuf, sent over the wire, deserialised on a different machine, and executed there with full fidelity. Once you understand that pattern, you can implement it in far fewer lines than Ballista uses, because you only need to serialise the plan nodes *your* engine actually produces.

Antipattern: the partial fork. There's a fourth path we didn't mention because it's the one you should never take: clone the repository, delete the parts you don't need, and start modifying what's left. This creates a codebase that looks like yours but inherits Ballista's internal assumptions, naming

conventions, and coupling. Every upstream bugfix requires manual cherry-picking across diverged codebases. Every upstream API change requires understanding code you didn't write and only partially understand. A partial fork gives you the maintenance burden of both "build from scratch" and "use a dependency," with the benefits of neither. If you can't use a project's public API, study its architecture and reimplement what you need. Don't inhabit someone else's codebase.

What We Kept

Three things from Ballista's model survived into SQE essentially unchanged.

The serialisation model. DataFusion's `datafusion-proto` crate provides protobuf serialisation for LogicalPlans and PhysicalPlans. This is the same serialisation layer that Ballista uses internally. We use it directly – `datafusion-proto = "52"` in our `Cargo.toml`. The protobuf schema defines how every built-in DataFusion plan node (Filter, Projection, Aggregate, Sort, HashJoin, and dozens more) maps to a protobuf message and back. This is thousands of lines of code we didn't have to write.

The execution model. Ballista's core insight is that a distributed query execution is just a set of independent scan tasks, each assigned to a worker, with results streamed back to the coordinator via Arrow Flight. The coordinator doesn't send the whole query to every worker. It sends each worker a *fragment* – a subset of files to scan, with the credentials needed to read them. The worker executes the fragment, streams back RecordBatches, and the coordinator stitches the results together. SQE follows this pattern exactly.

In SQE, this model is embodied by `DistributedScanExec` – a custom `ExecutionPlan` node that the coordinator injects into the physical plan tree in place of the local `IcebergScanExec`. Each partition of the `DistributedScanExec` maps to one worker. When DataFusion's execution engine calls `execute(partition)`, the node dispatches a `ScanTask` to the assigned worker via Flight `do_get` and returns the result stream. From DataFusion's perspective, the distributed scan behaves identically to a local scan – it produces `RecordBatch` streams with the same schema. The distribution is invisible to every operator above it in the plan tree.

The Flight interface. Workers expose an Arrow Flight service. The coordinator dispatches work by calling `do_get` with a ticket that describes the scan task. Results come back as a stream of `FlightData` messages containing Arrow RecordBatches. Health checks use `do_action`. Credential refreshes use `do_action`. Everything over gRPC, everything using the Arrow Flight protocol. This is the same interface Ballista uses between its scheduler and executors.

The worker side of this interface is compact. The `WorkerFlightService` implements three operations and nothing else:

```
#[tonic::async_trait]
impl FlightService for WorkerFlightService {
    async fn do_get(
        &self,
        request: Request<Ticket>,
    ) -> Result<Response<Self::DoGetStream>, Status> {
        let ticket = request.into_inner();
```

```
        let scan_task = ScanTask::from_bytes(&ticket.ticket)
            .map_err(|e| {
                Status::invalid_argument(
                    format!("Failed to decode ScanTask: {e}"))
            })?;

        let cred_rx = credential_store
            .subscribe(&scan_task.fragment_id).await;

        let (schema, batches) = executor::execute_scan(
            &scan_task, Some(&metrics),
            &session_ctx, Some(cred_rx),
        ).await.map_err(|e| {
            Status::internal(
                format!("Scan execution failed: {e}"))
        })?;

        let batch_stream =
            stream::iter(batches.into_iter().map(Ok));
        let flight_stream = FlightDataEncoderBuilder::new()
            .with_schema(schema)
            .build(batch_stream)
            .map_err(Status::from);

        Ok(Response::new(Box::pin(flight_stream)))
    }

    async fn do_action(
        &self,
        request: Request<Action>,
    ) -> Result<Response<Self::DoActionStream>, Status> {
        match action.r#type.as_str() {
            "health_check" => { /* return OK */ }
            "refresh_credentials" => {
                /* accept new S3 credentials */
            }
            _ => Err(Status::unimplemented(/* ... */)),
        }
    }
}
```

Three actions. `do_get` executes a scan. `health_check` tells the coordinator the worker is alive. `refresh_credentials` accepts updated S3 credentials mid-scan. Workers also implement `do_exchange` for shuffle data ingestion in distributed aggregation – hash-partitioned or range-partitioned batches flow between workers during multi-stage execution. Everything else returns `UNIMPLEMENTED`. Workers do not support handshake, `do_put`, or any other Flight operation. They are executors, not servers.

What We Replaced

Four things from Ballista didn't survive.

The scheduler. Ballista's scheduler is a standalone service that accepts SQL queries, plans them, and distributes work to executors. SQE's coordinator already does the planning. What SQE needs is a *fragment scheduler* – something that takes a set of scan tasks and assigns them to workers based on load, health, and file count. That's a much simpler problem than what Ballista's scheduler solves.

Our scheduler is 170 lines:

```rust
/// Weighted fragment scheduler that assigns tasks to the
/// least-loaded worker.
///
/// Strategy:
/// 1. Filter out unhealthy workers.
/// 2. Initialize each worker's load from its active_fragments count.
/// 3. Sort tasks by estimated cost (descending) so the heaviest
///    tasks are assigned first ("largest-first" bin-packing).
/// 4. Assign each task to the worker with the currently lowest
///    total load.
#[derive(Debug, Default)]
pub struct WeightedScheduler;

impl FragmentScheduler for WeightedScheduler {
    fn assign(
        &self,
        tasks: &[ScanTask],
        workers: &[WorkerInfo],
    ) -> Result<Vec<Assignment>, SchedulerError> {
        let healthy: Vec<&WorkerInfo> =
            workers.iter().filter(|w| w.healthy).collect();

        if healthy.is_empty() {
            return Err(SchedulerError::NoHealthyWorkers);
        }

        // Build a load tracker: (accumulated_load, worker_index)
        let mut loads: Vec<(u64, usize)> = healthy
            .iter()
            .enumerate()
            .map(|(i, w)| (u64::from(w.active_fragments), i))
            .collect();

        // Sort tasks by cost descending (largest-first bin packing)
        let mut indexed_tasks: Vec<(usize, u64)> = tasks
            .iter()
            .enumerate()
```

```
            .map(|(i, t)| (i, estimate_cost(t)))
            .collect();
        indexed_tasks.sort_by(|a, b| b.1.cmp(&a.1));

        let mut assignments: Vec<Option<Assignment>> =
            vec![None; tasks.len()];

        for (task_idx, cost) in indexed_tasks {
            let min_pos = loads
                .iter()
                .enumerate()
                .min_by_key(|(_, (load, _))| *load)
                .map(|(pos, _)| pos)
                .expect("healthy workers vec is non-empty");

            let worker_idx = loads[min_pos].1;
            assignments[task_idx] = Some(Assignment {
                task_index: task_idx,
                worker_url: healthy[worker_idx].url.clone(),
            });
            loads[min_pos].0 += cost;
        }

        Ok(assignments.into_iter().map(|a| a.unwrap()).collect())
    }
}
```

This is a largest-first bin-packing heuristic. The heaviest tasks get assigned first, to the least-loaded worker. It produces balanced distributions even when tasks have wildly different costs (one scan task covering 10 files vs another covering 1 file). It also accounts for workers that already have in-flight work from previous queries.

The scheduler also uses consistent hashing on file paths to prefer a "home" worker for each task. If the preferred worker's accumulated load exceeds the minimum-load worker by more than 20%, the task falls back to minimum-load assignment. This balances two goals: cache locality (a worker that has recently read a file may still have its Parquet footer cached) and load balance (no single worker becomes a bottleneck).

Ballista's scheduler is far more sophisticated. It handles multi-stage execution with shuffles, work stealing, and speculative execution. We don't need any of that – SQE's distributed execution is scan-parallel only. Every scan task is independent. There's no shuffle stage. The coordinator handles final aggregation, sorting, and projection locally.

Authentication. Ballista has no concept of user identity. Its scheduler accepts queries anonymously. Its executors run with whatever ambient credentials the process has. This is the fundamental incompatibility.

SQE's distributed execution carries the user's bearer token through to every worker. The `ScanTask`

struct includes S3 credentials vended specifically for that user:

```rust
pub struct ScanTask {
    pub fragment_id: String,
    pub data_file_paths: Vec<String>,
    pub file_sizes_bytes: Vec<u64>,
    pub projected_columns: Vec<String>,
    pub s3_endpoint: String,
    pub s3_region: String,
    pub s3_access_key: String,
    pub s3_secret_key: String,
    pub s3_session_token: String,
    pub s3_path_style: bool,
    pub s3_allow_http: bool,
}
```

The `file_sizes_bytes` field enables byte-accurate cost estimation in the scheduler – file count is a rough proxy, but actual file sizes let the `WeightedScheduler` balance I/O load precisely. Every field after `projected_columns` is about credentials. The worker never contacts Polaris. The worker never assumes a role. The worker reads the files it's told to read, with the credentials it's given, and returns the results. The coordinator is the only component that talks to the catalog.

Notice the `Debug` implementation on this struct. It redacts credentials:

```rust
impl std::fmt::Debug for ScanTask {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        f.debug_struct("ScanTask")
            .field("fragment_id", &self.fragment_id)
            .field("data_file_paths", &self.data_file_paths)
            .field("projected_columns", &self.projected_columns)
            .field("s3_endpoint", &self.s3_endpoint)
            .field("s3_region", &self.s3_region)
            .field("s3_access_key", &"[REDACTED]")
            .field("s3_secret_key", &"[REDACTED]")
            .field("s3_session_token", &session_token_display)
            .field("s3_path_style", &self.s3_path_style)
            .field("s3_allow_http", &self.s3_allow_http)
            .finish()
    }
}
```

Small thing. Important thing. Log files are read by people who shouldn't see S3 credentials. The `#[derive(Debug)]` default would dump secrets into every debug log line. A custom `Debug` impl costs ten lines and prevents a class of security incident that's embarrassingly common.

Configuration. Ballista has its own configuration model – command-line flags, environment variables, a scheduler config, an executor config. SQE has its TOML-based configuration with environment variable overlay. We didn't want two config models. Workers and coordinator share the same `SqeConfig` struct, with a `mode` field that determines which role the process plays:

```
pub enum Mode {
    Coordinator,
    Worker,
}
```

One binary, two modes. The coordinator starts a Flight SQL service for clients and a heartbeat listener for workers. The worker starts a Flight service for scan execution and a heartbeat sender pointed at the coordinator. Same Docker image, different entrypoint arguments.

The codec for custom plan nodes. Ballista's serialisation handles every built-in DataFusion plan node. It doesn't handle custom `ExecutionPlan` implementations – because it doesn't know about them. SQE has `DistributedScanExec`, a custom plan node that replaces `IcebergScanExec` in the physical plan tree when distribution is active. This node needs to be serialisable too. That's where the `PhysicalExtensionCodec` comes in.

The Protobuf Codec Deep Dive

DataFusion's protobuf serialisation is a two-layer system.

The first layer is `datafusion-proto`, which handles all built-in plan nodes. It converts a `FilterExec` to a protobuf `FilterExecNode`, a `ProjectionExec` to a `ProjectionExecNode`, and so on. This layer works out of the box. You call `physical_plan_to_bytes` and get protobuf bytes. You call `physical_plan_from_bytes` and get the plan back. Round-trip fidelity for built-in nodes is guaranteed by DataFusion's test suite.

The second layer is the `PhysicalExtensionCodec` trait. This is the escape hatch. When `datafusion-proto` encounters a plan node it doesn't recognise, it calls `try_encode` on the extension codec. When deserialising, it calls `try_decode`. This is how custom plan nodes participate in protobuf serialisation.

SQE's codec is `SqePhysicalCodec`:

```
#[derive(Debug, Default)]
pub struct SqePhysicalCodec;

impl PhysicalExtensionCodec for SqePhysicalCodec {
    fn try_encode(
        &self,
        node: Arc<dyn ExecutionPlan>,
        buf: &mut Vec<u8>,
    ) -> DFResult<()> {
        if let Some(scan) =
            node.as_any().downcast_ref::<DistributedScanExec>()
        {
            let proto_schema: protobuf::Schema =
                scan.schema().as_ref().try_into().map_err(|e| {
                    DataFusionError::External(Box::new(
                        std::io::Error::other(
                            format!("Schema encoding failed: {e}")
```

```
                    ),
                ))
            })?;

        let schema_bytes = proto_schema.encode_to_vec();
        let schema_proto_b64 = base64::Engine::encode(
            &base64::engine::general_purpose::STANDARD,
            &schema_bytes,
        );

        let encoded = EncodedDistributedScan {
            scan_tasks: scan.scan_tasks().to_vec(),
            worker_urls: scan.worker_urls().to_vec(),
            schema_proto_b64,
        };

        let json = serde_json::to_vec(&encoded)
            .map_err(|e| DataFusionError::External(Box::new(e)))?;
        buf.extend_from_slice(&json);
        Ok(())
    } else {
        Err(DataFusionError::NotImplemented(format!(
            "SqePhysicalCodec: cannot encode '{}'",
            node.name()
        )))
    }
}

fn try_decode(
    &self,
    buf: &[u8],
    _inputs: &[Arc<dyn ExecutionPlan>],
    _ctx: &TaskContext,
) -> DFResult<Arc<dyn ExecutionPlan>> {
    let encoded: EncodedDistributedScan =
        serde_json::from_slice(buf)
            .map_err(|e| DataFusionError::External(Box::new(e)))?;

    // Decode schema from base64-encoded protobuf
    let schema_bytes = base64::Engine::decode(
        &base64::engine::general_purpose::STANDARD,
        &encoded.schema_proto_b64,
    ).map_err(|e| DataFusionError::External(
        Box::new(std::io::Error::other(
            format!("Schema base64 decode failed: {e}")
        ))
    ))?;
```

```
        let proto_schema =
            protobuf::Schema::decode(schema_bytes.as_slice())
                .map_err(|e| DataFusionError::External(
                    Box::new(std::io::Error::other(
                        format!("Schema proto decode failed: {e}")
                    ))
                ))?;

        let schema = Schema::try_from(&proto_schema)
            .map_err(|e| DataFusionError::External(
                Box::new(std::io::Error::other(
                    format!("Schema conversion failed: {e}")
                ))
            ))?;

        Ok(Arc::new(DistributedScanExec::new(
            encoded.scan_tasks,
            encoded.worker_urls,
            Arc::new(schema),
        )))
    }
}
```

Notice the mixed encoding strategy. The Arrow schema is serialised using DataFusion's protobuf schema codec – the same one used for built-in plan nodes. The scan tasks and worker URLs are serialised using JSON via serde. The protobuf bytes are then base64-encoded so they can live inside the JSON payload.

Is this elegant? No. It's practical. The schema needs protobuf because Arrow schemas have complex nested types (maps, structs, lists of lists) that serde_json handles poorly. The scan tasks are flat structs with strings and booleans – JSON handles them fine. Mixing the two encodings in one codec is ugly but correct, and the round-trip test proves it:

```
#[test]
fn test_roundtrip_distributed_scan_exec() {
    let schema = Arc::new(Schema::new(vec![
        Field::new("id", DataType::Int64, false),
        Field::new("name", DataType::Utf8, true),
    ]));

    let original = Arc::new(DistributedScanExec::new(
        vec![make_task("f1"), make_task("f2")],
        vec!["http://w1:50052".to_string(),
             "http://w2:50052".to_string()],
        schema.clone(),
    ));
```

```
    let codec = SqePhysicalCodec::new();
    let mut buf = Vec::new();
    codec
        .try_encode(original.clone() as Arc<dyn ExecutionPlan>,
                    &mut buf)
        .expect("encode failed");

    let ctx = Arc::new(TaskContext::default());
    let decoded = codec
        .try_decode(&buf, &[], &ctx)
        .expect("decode failed");

    let decoded_scan = decoded
        .as_any()
        .downcast_ref::<DistributedScanExec>()
        .expect("Expected DistributedScanExec");

    assert_eq!(decoded_scan.scan_tasks().len(), 2);
    assert_eq!(decoded_scan.worker_urls(),
               &["http://w1:50052", "http://w2:50052"]);
    assert_eq!(*decoded_scan.schema(), *schema);
}
```

This test exists because we learned the hard way that it needed to.

DataFusion deep dive: The `PhysicalExtensionCodec` trait is DataFusion's official extension point for plan serialisation. When `datafusion-proto` serialises a physical plan and encounters a node that isn't in its built-in registry, it calls `try_encode` on whatever extension codec was provided. If no codec is registered, serialisation fails. If the codec returns `NotImplemented`, serialisation fails. The extension codec must handle every custom plan node in your engine, or the plan can't leave the coordinator. This is the contract: if you add a custom `ExecutionPlan`, you must teach the codec how to serialise it.

The Round-Trip Fidelity Challenge

Serialising a plan and deserialising it sounds straightforward. It's not.

The plan that arrives at the worker must be functionally identical to the plan that left the coordinator. Not structurally identical – the Arc pointers will be different, the memory addresses will be different. But functionally identical: same schema, same partitioning, same data files, same projection.

The first version of our codec serialised scan tasks but didn't serialise the Arrow schema. We assumed the worker would infer the schema from the Parquet files it was reading. This worked for simple queries. It broke for queries with column projection.

The problem: when the coordinator plans a `SELECT id, name FROM users`, the physical plan's schema has two columns. But the Parquet file has twenty columns. The worker reads all twenty, because the projected schema was lost during serialisation. The coordinator then receives batches with twenty columns when it expects two. The downstream ProjectionExec fails.

The fix was serialising the schema as part of the codec payload. Four lines of encode, four lines of decode, and a base64 wrapper because protobuf bytes inside JSON need encoding. The round-trip test caught this in CI within a day of the codec being written. The lesson: round-trip tests for custom codecs aren't optional. They're the only thing standing between you and silent data corruption.

Field report: The schema projection bug manifested as a `SchemaError` deep inside DataFusion's `ProjectionExec`. The error message said "column index 2 out of bounds for schema with 2 columns." The actual problem was 18 columns upstream where the schema had been lost during serialisation. We spent an afternoon tracing the error before adding the schema to the codec payload. The fix was small. The time to find it was not. After that, we wrote the round-trip test. Every codec change since has been validated by that test before it leaves the developer's machine.

Worker Registration and Heartbeat

Ballista's worker registration is a gRPC protocol between the scheduler and executors. Executors register on startup, send periodic heartbeats, and the scheduler removes them after a configurable number of missed heartbeats.

SQE's worker registration follows the same pattern with one difference: it's built on Arrow Flight rather than a custom gRPC protocol. Workers send heartbeats by calling `do_action("heartbeat")` on the coordinator's Flight service. The action body contains the worker's own URL so the coordinator knows who sent the heartbeat.

```
pub fn start_heartbeat_task(
    coordinator_url: String,
    worker_url: String,
    interval: Duration,
) {
    tokio::spawn(async move {
        let mut ticker = tokio::time::interval(interval);
        // First tick completes immediately; consume it so the first
        // real heartbeat fires after one full interval.
        ticker.tick().await;

        loop {
            ticker.tick().await;
            if let Err(e) =
                send_heartbeat(&coordinator_url, &worker_url).await
            {
                warn!(
                    coordinator = %coordinator_url,
                    error = %e,
                    "Heartbeat to coordinator failed"
                );
            }
        }
    });
```

```
}
```

The heartbeat is a fire-and-forget loop. No exponential backoff. No retry logic. If the coordinator is down, the heartbeat fails, the worker logs a warning, and tries again next interval. The coordinator's `WorkerRegistry` tolerates a configurable number of consecutive misses (three, by default) before marking a worker unhealthy.

```
pub async fn mark_failed(&self, url: &str) {
    let mut inner = self.inner.write().await;
    if let Some(state) = inner.workers.get_mut(url) {
        state.consecutive_failures += 1;
        if state.consecutive_failures >= MAX_CONSECUTIVE_FAILURES {
            if state.healthy {
                warn!(
                    worker = url,
                    failures = state.consecutive_failures,
                    "Worker marked unhealthy"
                );
            }
            state.healthy = false;
        }
    }
}
```

Three misses is generous. With a 5-second heartbeat interval, a worker has 15 seconds of silence before it's removed from the pool. This handles brief network blips and GC pauses without triggering unnecessary failovers.

Recovery is instant. A single successful heartbeat resets the failure counter and marks the worker healthy:

```
pub async fn register_heartbeat(&self, url: &str) {
    let mut inner = self.inner.write().await;
    let state = inner.workers
        .entry(url.to_string())
        .or_insert_with(|| {
            info!(worker = url,
                  "Discovered new worker via heartbeat");
            WorkerState {
                url: url.to_string(),
                healthy: false,
                consecutive_failures: 0,
                last_healthy: None,
            }
        });
    state.healthy = true;
    state.consecutive_failures = 0;
    state.last_healthy = Some(Instant::now());
}
```

Notice that `register_heartbeat` uses `or_insert_with`. Workers that weren't in the initial configuration are automatically discovered when they start sending heartbeats. This means you can add workers to a running cluster without reconfiguring the coordinator. Scale out by starting a new worker process pointed at the coordinator's URL. The first heartbeat registers it. The next scheduling round includes it.

This is simpler than Ballista's registration protocol, which requires executors to explicitly register with the scheduler before they're eligible for work. SQE's approach is closer to service discovery – if you're sending heartbeats, you exist.

The `WorkerRegistry` also supports immediate unhealthy marking, separate from the heartbeat-based failure threshold:

```
pub async fn mark_unhealthy(&self, url: &str) {
    let mut inner = self.inner.write().await;
    if let Some(state) = inner.workers.get_mut(url) {
        state.healthy = false;
        state.consecutive_failures = MAX_CONSECUTIVE_FAILURES;
    }
}
```

This is called when a worker fails *during query execution* – a connection refused, a timeout, a transport error. A missed heartbeat is a soft signal. A failed scan dispatch is a hard signal. The worker is pulled from the pool immediately, and the fragment is retried on a different worker.

The two-tier health model – gradual degradation for heartbeat misses, immediate removal for execution failures – is the kind of nuance that doesn't appear in architecture diagrams. It matters in production. A worker that's restarting will miss a heartbeat or two but come back. A worker that returns connection refused during a scan is gone, and any fragments assigned to it need to be moved immediately.

Credentials That Outlive the Scan

The ScanTask carries S3 credentials vended by Polaris. Those credentials have a lifetime – typically an hour for STS session tokens. Most scans finish well within that window. But a query scanning hundreds of Parquet files across a large partition, on a worker that's also handling other queries, can take longer than expected.

When credentials expire mid-scan, the worker's S3 reads start returning `403 Forbidden`. The scan fails. The query fails. The user retries and gets a fresh token, but the experience is poor.

SQE handles this with a `CredentialRefreshTracker` on the coordinator side. When the coordinator dispatches scan fragments, it registers each fragment's credential expiry time. A background loop monitors these expiry times. When credentials are within five minutes of expiring, the coordinator refreshes them from Polaris and pushes the new credentials to the worker via `do_action("refresh_credentials")`.

```
/// Refreshed S3 credential payload pushed from coordinator to worker.
#[derive(Clone, Serialize, Deserialize)]
```

```rust
pub struct RefreshableCredentials {
    pub fragment_id: String,
    pub access_key_id: String,
    pub secret_access_key: String,
    pub session_token: String,
    pub expiry: DateTime<Utc>,
}
```

On the worker side, the executor checks for refreshed credentials *between files* – not between batches, not between row groups, but at each file boundary within the scan. A `tokio::sync::watch` channel carries the refresh from the Flight service handler to the running scan. The executor checks `has_changed()` before opening each new Parquet file, and rebuilds the S3 object store with the fresh credentials if a refresh arrived.

This is Ballista's execution model meeting SQE's auth model. Ballista doesn't need credential refresh because Ballista's workers use ambient credentials that don't expire. SQE's workers use vended, short-lived credentials that can expire mid-query. The coordinator-to-worker credential push is the mechanism that makes short-lived tokens work across long-running distributed scans. Ballista couldn't teach us this part.

The Plan Surgery

The most delicate part of the distribution pipeline isn't the codec, the scheduler, or the heartbeat. It's the plan replacement.

When the coordinator decides to distribute a query, it has a physical plan tree that looks something like this:

```
ProjectionExec
  FilterExec (WHERE clause)
    IcebergScanExec (reads Parquet files from S3)
```

The `IcebergScanExec` is SQE's local scan node. It knows how to read Parquet files directly from S3 using the object store. For distributed execution, we need to replace it with a `DistributedScanExec` that fans out to workers instead.

But we can't just swap the root of the plan. The ProjectionExec and FilterExec need to stay. The aggregation nodes above them (if it's a `SELECT COUNT(*)`) need to stay. Only the leaf scan node changes.

The `try_distribute` method in `QueryHandler` does this in twelve steps. The critical ones:

1. Find the `IcebergScanExec` leaf in the plan tree.
2. Extract the list of data file paths from it.
3. Split the files across available workers.
4. Build `ScanTask` structs with credentials.
5. Schedule tasks to workers using the weighted scheduler.
6. Build a `DistributedScanExec` with the tasks and worker URLs.
7. Replace the `IcebergScanExec` leaf in the plan tree with the new `DistributedScanExec`.

Step 7 is the surgery:

```rust
fn replace_scan_in_plan(
    plan: &Arc<dyn ExecutionPlan>,
    target: &Arc<dyn ExecutionPlan>,
    replacement: Arc<dyn ExecutionPlan>,
) -> Arc<dyn ExecutionPlan> {
    if Arc::ptr_eq(plan, target) {
        return replacement;
    }

    let children = plan.children();
    if children.is_empty() {
        return Arc::clone(plan);
    }

    let new_children: Vec<Arc<dyn ExecutionPlan>> = children
        .iter()
        .map(|child| replace_scan_in_plan(
            child, target, Arc::clone(&replacement)
        ))
        .collect();

    let changed = new_children
        .iter()
        .zip(children.iter())
        .any(|(new, old)| !Arc::ptr_eq(new, old));

    if changed {
        plan.clone()
            .with_new_children(new_children)
            .unwrap_or_else(|_| Arc::clone(plan))
    } else {
        Arc::clone(plan)
    }
}
```

This walks the plan tree recursively. When it finds the target node (identified by `Arc` pointer equality), it returns the replacement. Every ancestor node is rebuilt via `with_new_children()` so the new leaf is properly wired into the tree.

The first version of this code didn't replace the leaf. It replaced the *entire plan* with a `Distributed-ScanExec`, discarding the filter and projection nodes above the scan. Queries returned wrong results – all rows instead of filtered rows, all columns instead of projected columns. The fix was commit `3a123ea`: "fix: replace scan leaf in plan tree instead of replacing entire plan."

Three lines of logic in a recursive function. Two days of debugging wrong query results to get there.

The distribution also has guardrails. Not every query gets distributed:

```
// Skip if no worker registry (single-node mode)
// Skip if no healthy workers
// Skip if no IcebergScanExec in the plan
// Skip if total file count < number of workers
```

That last check is important. Distributing a single-file scan across two workers means one worker gets a file and the other gets nothing. The overhead of the Flight round-trip exceeds the benefit. The threshold is simple: if there are fewer files than workers, execute locally.

The Distributed Compose

Seeing the pieces work together requires seeing them deployed. The distributed docker-compose file is short enough to include:

```
services:
  coordinator:
    build: .
    entrypoint: ["sqe-server", "--config", "/config/coordinator.toml"]
    ports:
      - "60051:50051"   # Flight SQL
      - "28080:8080"    # Trino HTTP
      - "29090:9090"    # Prometheus metrics
    depends_on:
      polaris:
        condition: service_healthy

  worker-1:
    build: .
    entrypoint: ["sqe-worker", "/config/worker.toml"]
    ports:
      - "60061:50052"
    depends_on:
      - coordinator

  worker-2:
    build: .
    entrypoint: ["sqe-worker", "/config/worker.toml"]
    ports:
      - "60062:50052"
    depends_on:
      - coordinator
```

Same Docker image for coordinator and workers. Different entrypoint, different config file. The workers start after the coordinator, begin sending heartbeats, and within one interval they appear in the worker registry. The next query that hits the coordinator will consider them for distribution.

Adding a third worker is one more service block in the compose file. No coordinator reconfiguration needed.

The Cost of the Fork

Every architectural shortcut has a maintenance cost. Here's what standing on Ballista's shoulders costs us.

Dependency tracking. We depend on `datafusion-proto`, which tracks DataFusion's version. When DataFusion releases a new version with new plan nodes or changed protobuf schemas, `datafusion-proto` updates. We update with it. If we'd built our own protobuf schema, we'd have to maintain parity manually. By depending on the official crate, we get this for free.

Limited distribution scope. Ballista supports multi-stage execution: hash shuffles, sort-merge joins distributed across workers, repartitioning between stages. SQE distributes scan work only. The coordinator handles joins, aggregations, and sorts locally. This is fine for our workload (analytical queries over partitioned Iceberg tables where the scan is the bottleneck), but it means SQE won't outperform a single node on shuffle-heavy queries.

No work stealing. Ballista's scheduler can detect when one worker finishes early and reassign pending work to it. SQE's scheduler assigns all fragments upfront and waits. If one worker is slower than the rest, the query is bottlenecked on that worker. The weighted scheduler mitigates this by front-loading heavy tasks, but it doesn't eliminate the problem.

No speculative execution. Spark and Ballista can speculatively launch a copy of a slow task on a different worker, using whichever result arrives first. SQE retries only on failure, not on slowness. Adding speculative execution would require tracking task progress, which we don't currently do.

These are deliberate trade-offs. The scanner-only distribution model is simpler to reason about, simpler to debug, and simpler to operate. When a query is slow, you look at the scan tasks. When a worker is overloaded, you look at the scheduler. There's no shuffle stage to investigate, no repartitioning to tune.

Capability	Ballista	SQE
Plan serialisation	Full physical plan protobuf	datafusion-proto + custom extension codec
Distribution model	Multi-stage with shuffles	Scan-parallel only
Scheduler	Stage-aware, work-stealing	Weighted bin-packing
Worker discovery	Explicit registration	Heartbeat-based auto-discovery
Authentication	None	Bearer token passthrough per fragment
Credential lifecycle	Static / ambient	Vended, short-lived, refreshable mid-scan
Configuration	CLI flags + env vars	TOML config with env overlay
Custom plan nodes	Via extension codec	Via extension codec (same mechanism)

What Ballista Taught Us

The most important lesson from studying Ballista wasn't technical. It was architectural.

Ballista proves that DataFusion's extensibility model works. The `PhysicalExtensionCodec` trait, the `ExecutionPlan` trait, the `with_new_children` method – these aren't theoretical extension points. They're production extension points that a real distributed execution framework uses daily.

When we built `DistributedScanExec`, we implemented `ExecutionPlan`. When we needed to serialise it, we implemented `PhysicalExtensionCodec`. When we needed to splice it into an existing plan tree, we used `with_new_children`. Every time, the trait design guided us to the right answer.

The second lesson: start with the simplest distribution model that solves your problem. Ballista's multi-stage execution is powerful, but it's complex. SQE's scan-parallel model handles the workloads we have today. If we need shuffle joins across workers tomorrow, we can add a stage to the scheduler. We don't need to redesign the whole system.

The third lesson: the protobuf round-trip is the contract. If a plan survives serialisation and deserialisation with the same semantics, the distributed system works. If it doesn't, nothing else matters. The round-trip test is the most important test in the distributed execution layer.

The fourth lesson is about knowing when heritage becomes baggage. Ballista gave us patterns and confidence that distributing DataFusion plans was feasible. But the moment we tried to use Ballista's scheduler with our auth model, it went from heritage to obstacle. The sign that you've crossed that line is when you spend more time working around the inherited code than working with it. We hit that point in two days. Two days was the right time to stop.

Looking Forward

The scan-parallel model has a ceiling. When SQE needs to distribute a hash join across workers – when the join inputs are too large to fit on a single coordinator – we'll need to add shuffle stages. That means building a hash-repartitioning plan node, teaching the codec to serialise it, and teaching the scheduler to chain stages.

That's future work. It's also future work that's made easier by the foundation we've built. The `FragmentScheduler` trait is pluggable. The `SqePhysicalCodec` can handle additional plan nodes. The `WorkerRegistry` already knows which workers are healthy and how loaded they are.

The hard part of distributed execution isn't the execution itself. It's the plumbing: serialisation, health monitoring, credential management, plan manipulation. We built that plumbing in twelve days, because Ballista showed us which pipes to lay.

AI Logbook: The human decided to fork Ballista's ideas rather than use it as-is or build from scratch. The AI implemented the `SqePhysicalCodec`, the `WeightedScheduler`, and the heartbeat protocol in three passes. The plan surgery function — `replace_scan_in_plan`, replacing a scan leaf in a tree while preserving all ancestor nodes — took the AI four attempts before the recursive traversal was correct. The first version replaced the entire plan; the fix was commit `3a123ea`.

Next chapter, we'll walk through the coordinator and worker in detail – how a query enters the coordinator, gets split into fragments, dispatched to workers, executed with the user's credentials, and reassembled into a result stream that the client receives as if the query ran locally.

Neither Trusts the Other

> The coordinator decides. The worker executes. Neither trusts the other more than necessary. Neither trusts the other more than necessary.

We have a physical plan. We have workers registered and sending heartbeats. Chapter 12 built the infrastructure for distributed execution – worker registration, protobuf serialization, the Ballista heritage that gave us a head start. The question now is mechanical: how does a query plan get from the coordinator to a worker, execute under the right user's identity, and stream results back to the client?

The answer touches every trust boundary in the system. And the trust boundaries, it turns out, are the interesting part.

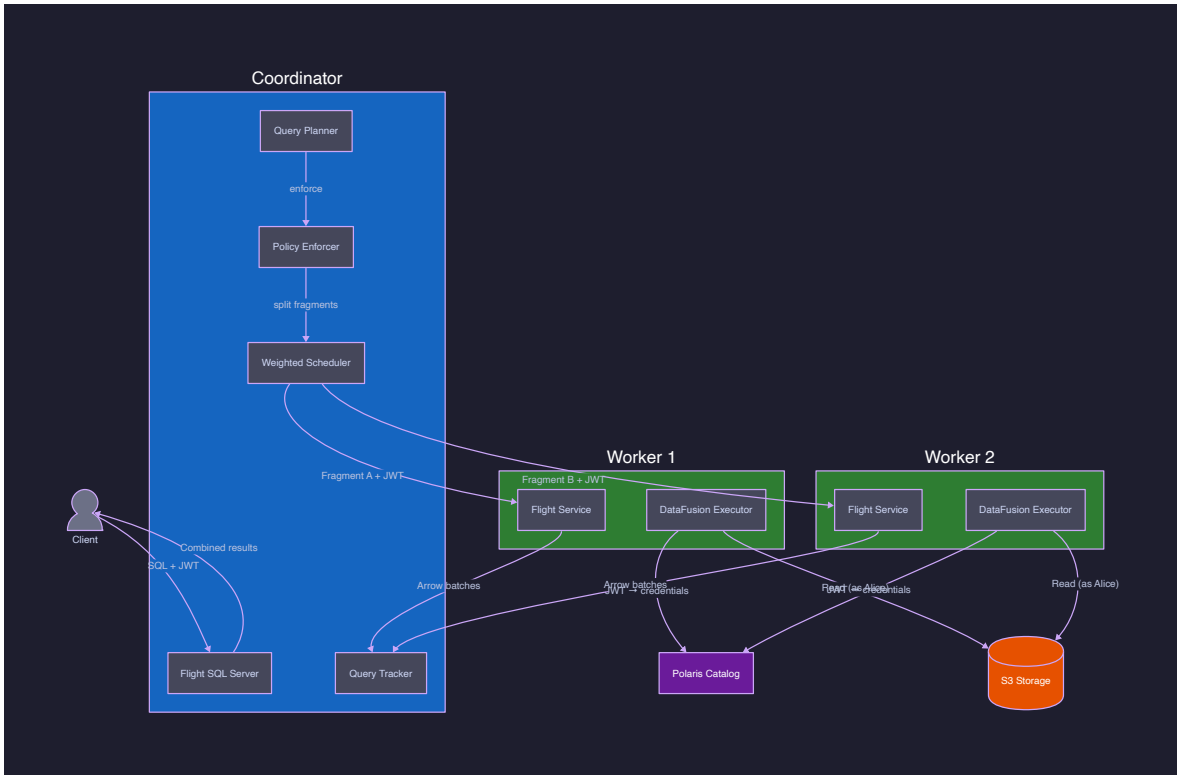

Figure 5: Distributed execution: coordinator splits the physical plan at the scan boundary, dispatches fragments to workers, and collects Arrow batches

The Plan Doesn't Travel Whole

A SQL query arrives at the coordinator as a string. The coordinator parses it, builds a logical plan, enforces policies (Chapter 8), optimizes it, and produces a physical plan. That physical plan is a tree of operators – filter, projection, aggregation, sort – rooted in a scan of Parquet files in S3.

The entire tree does not travel to workers. Only the scan travels.

This distinction matters. The coordinator keeps the upper plan tree – the filters, the aggregations, the projections, the final sort. It sends only the leaf scan to workers. Workers read Parquet files and stream Arrow batches back. The coordinator applies everything else.

The split point is the `IcebergScanExec` node. When the coordinator has healthy workers available, it replaces this node with a `DistributedScanExec` that fans out the file reads across workers. Everything above – every filter, every aggregation, every projection – stays local.

```
Client SQL: SELECT region, SUM(amount) FROM orders WHERE year = 2025 GROUP BY region

Physical Plan (coordinator):
  AggregateExec [region, SUM(amount)]
    ProjectionExec [region, amount]
      FilterExec [year = 2025]
        DistributedScanExec          <-- replaced IcebergScanExec
          Worker 1: files 0..3       <-- reads Parquet, streams Arrow batches
          Worker 2: files 3..6
          Worker 3: files 6..9
```

The coordinator never touches S3. The workers never see the aggregation. Each side does exactly its job.

Deciding Whether to Distribute

Not every query should be distributed. A `SHOW TABLES` query has no scan. A query touching a single small Parquet file gains nothing from shipping it to a worker – the network round-trip costs more than the local read. The `try_distribute` method in `QueryHandler` makes this decision explicitly:

```rust
async fn try_distribute(
    &self,
    plan: Arc<dyn ExecutionPlan>,
    session: &Session,
    query_id: &uuid::Uuid,
) -> Arc<dyn ExecutionPlan> {
    // 1. Check if we have a worker registry (distributed mode)
    let registry = match self.worker_registry {
        Some(ref r) => r,
        None => return plan,
    };
```

```
    // 2. Get healthy workers -- if none, fall back to local
    let healthy = registry.healthy_workers().await;
    if healthy.is_empty() {
        debug!("No healthy workers available, executing locally");
        return plan;
    }

    // 3. Find IcebergScanExec node in the plan tree
    let scan_node = match find_iceberg_scan(&plan) {
        Some(node) => node,
        None => {
            debug!("No IcebergScanExec found in plan, executing locally");
            return plan;
        }
    };

    // ...

    // 5. Check if there are enough files to justify distribution
    let num_workers = healthy.len();
    if total_files < num_workers {
        debug!(
            total_files,
            num_workers,
            "Fewer files than workers, executing locally"
        );
        return plan;
    }

    // ... build scan tasks, schedule, replace scan node ...
}
```

Every check has a graceful fallback: return the original plan unchanged. If the registry is absent, the system runs single-node. If all workers are down, the coordinator handles the scan itself. If there are three files and five workers, there is no point creating two idle fragments. The coordinator falls back to local execution, and the query still succeeds.

This is deliberate. Distribution is an optimization, not a requirement. The system must work without it.

DataFusion deep dive: The `try_distribute` method operates on the physical plan *after* DataFusion's optimizer has run. This matters. Policy enforcement happens on the logical plan (Chapter 8), before optimization. By the time we reach `try_distribute`, row filters and column masks are already baked into the plan tree. The workers never see the original unfiltered plan. They get scan fragments that already reflect the user's policy-restricted view.

Splitting Files Across Workers

Once the coordinator decides to distribute, it needs to divide the work. An Iceberg table's scan resolves to a list of Parquet file paths in S3. The coordinator extracts these paths from the `IcebergScanExec` and splits them across workers.

The splitter is deliberately simple:

```
pub fn split_files(files: Vec<String>, num_workers: usize) -> Vec<Vec<String>> {
    if num_workers == 0 || files.is_empty() {
        return vec![];
    }

    let mut groups: Vec<Vec<String>> = (0..num_workers).map(|_| Vec::new()).collect();

    for (i, file) in files.into_iter().enumerate() {
        groups[i % num_workers].push(file);
    }

    groups
}
```

Round-robin. Each file goes to `worker[i % num_workers]`. A table with 12 Parquet files and 3 workers gets 4 files per worker. Not optimal – there is no consideration of file size, data locality, or partition alignment. But it was correct, it was debuggable, and it shipped on day one.

The sophistication comes in the scheduler.

The Weighted Scheduler

File splitting determines which files go together. The scheduler determines which worker gets each group. These are separate concerns. The split creates `ScanTask` objects; the scheduler assigns each task to a worker.

```
pub struct ScanTask {
    pub fragment_id: String,
    pub data_file_paths: Vec<String>,
    pub file_sizes_bytes: Vec<u64>,
    pub projected_columns: Vec<String>,
    pub s3_endpoint: String,
    pub s3_region: String,
    pub s3_access_key: String,
    pub s3_secret_key: String,
    pub s3_session_token: String,
    pub s3_path_style: bool,
    pub s3_allow_http: bool,
}
```

Each `ScanTask` is a self-contained unit of work. It carries everything a worker needs to read its files:

the file paths, the S3 endpoint, the credentials, and the column projection. The worker does not need to contact the catalog. It does not need to resolve table metadata. Everything is already resolved.

The `WeightedScheduler` assigns tasks to workers using a largest-first bin-packing heuristic:

```
impl FragmentScheduler for WeightedScheduler {
    fn assign(
        &self,
        tasks: &[ScanTask],
        workers: &[WorkerInfo],
    ) -> Result<Vec<Assignment>, SchedulerError> {
        let healthy: Vec<&WorkerInfo> = workers.iter().filter(|w| w.healthy).collect();

        if healthy.is_empty() {
            return Err(SchedulerError::NoHealthyWorkers);
        }

        // Initialize each worker's load from its active_fragments count
        let mut loads: Vec<(u64, usize)> = healthy
            .iter()
            .enumerate()
            .map(|(i, w)| (u64::from(w.active_fragments), i))
            .collect();

        // Sort tasks by estimated cost descending (largest-first bin packing)
        let mut indexed_tasks: Vec<(usize, u64)> = tasks
            .iter()
            .enumerate()
            .map(|(i, t)| (i, estimate_cost(t)))
            .collect();
        indexed_tasks.sort_by(|a, b| b.1.cmp(&a.1));

        // Assign each task to the worker with the lowest current load
        for (task_idx, cost) in indexed_tasks {
            let min_pos = loads
                .iter()
                .enumerate()
                .min_by_key(|(_, (load, _))| *load)
                .map(|(pos, _)| pos)
                .expect("healthy workers vec is non-empty");

            let worker_idx = loads[min_pos].1;
            assignments[task_idx] = Some(Assignment {
                task_index: task_idx,
                worker_url: healthy[worker_idx].url.clone(),
            });

            loads[min_pos].0 += cost;
```

```
        }

        Ok(/* ... */)
    }
}
```

The cost function is file count: `estimate_cost` returns the number of Parquet files in the task, with a minimum of 1. A task with 10 files costs more than a task with 2. The heaviest tasks are assigned first, to the least-loaded worker. This is a classic approach from operations research – largest-first decreases (LFD) – and it produces balanced distributions even when tasks have wildly different costs.

The scheduler also considers existing load. If worker 1 is already executing 10 fragments from other queries and worker 2 is idle, new tasks go to worker 2. This is the `active_fragments` count in `WorkerInfo`, initialized from the worker registry.

Unhealthy workers are filtered out before scheduling begins. If a worker missed three consecutive health checks, it does not receive work. Period.

Field report: Our first scheduler was pure round-robin – task 0 to worker 0, task 1 to worker 1, and so on. It worked for uniform workloads. Then we ran a benchmark where one partition had 300MB of Parquet files and another had 2MB. The round-robin scheduler put both on different workers, but the worker with the 300MB partition became a bottleneck while the other worker sat idle for 12 seconds. The weighted scheduler fixed this in one afternoon. The bin-packing heuristic is not perfect, but it handles the common case of heterogeneous partition sizes without requiring per-file size metadata.

What the Worker Receives

The coordinator serializes the `ScanTask` to bytes and sends it as a Flight `Ticket` in a `do_get` call. The worker receives the ticket, deserializes the task, and starts reading Parquet files.

The worker's Flight service is minimal. It handles three operations:

```
#[tonic::async_trait]
impl FlightService for WorkerFlightService {
    async fn do_get(
        &self,
        request: Request<Ticket>,
    ) -> Result<Response<Self::DoGetStream>, Status> {
        let ticket = request.into_inner();

        let scan_task = ScanTask::from_bytes(&ticket.ticket).map_err(|e| {
            Status::invalid_argument(format!("Failed to decode ScanTask: {e}"))
        })?;

        // Subscribe to credential updates for this fragment
        let cred_rx = credential_store.subscribe(&scan_task.fragment_id).await;

        let (schema, batches) =
```

```
            executor::execute_scan(&scan_task, Some(&metrics), &session_ctx, Some(cred_rx))
                .await
                .map_err(|e| Status::internal(format!("Scan execution failed: {e}")))?;

        // Stream results back as Arrow Flight data
        let batch_stream = stream::iter(batches.into_iter().map(Ok));
        let flight_stream = FlightDataEncoderBuilder::new()
            .with_schema(schema)
            .build(batch_stream)
            .map_err(Status::from);

        Ok(Response::new(Box::pin(flight_stream)))
    }

    async fn do_action(&self, request: Request<Action>) -> /* ... */ {
        match action.r#type.as_str() {
            "health_check" => { /* return OK */ }
            "refresh_credentials" => { /* accept new S3 credentials */ }
            _ => Err(Status::unimplemented(/* ... */)),
        }
    }
}
```

Three actions. `do_get` executes a scan. `health_check` tells the coordinator the worker is alive. `refresh_credentials` accepts updated S3 credentials mid-scan. Workers also implement `do_exchange` for shuffle data ingestion in distributed aggregation. Everything else returns `UNIMPLEMENTED`. Workers do not support handshake, `do_put`, or any other Flight operation. They are executors, not servers.

The worker does not see the full query. It does not know about filters, aggregations, or projections that the coordinator will apply to its output. It reads the files it was told to read, in the columns it was told to project, and sends back Arrow batches. This is the fundamental contract: the coordinator thinks, the worker does.

Executing the Scan

The executor is where Parquet bytes become Arrow batches. The `execute_scan` function walks through each file in the task, builds an S3 object store with the provided credentials, reads Parquet data, applies column projection, and collects the results. The actual signature includes additional parameters for the Parquet footer cache and late materialization configuration, but the core contract shown here captures the essential flow.

```
pub async fn execute_scan(
    task: &ScanTask,
    metrics: Option<&Arc<WorkerMetricsRegistry>>,
    session_ctx: &SessionContext,
    credential_rx: Option<watch::Receiver<Option<RefreshableCredentials>>>,
) -> anyhow::Result<(SchemaRef, Vec<RecordBatch>)> {
```

```
    let store = build_object_store_with_creds(
        task,
        &task.s3_access_key,
        &task.s3_secret_key,
        &task.s3_session_token,
    )?;
    let mut store: Arc<dyn ObjectStore> = Arc::new(store);

    let pool = session_ctx.runtime_env().memory_pool.clone();
    let consumer = MemoryConsumer::new(format!("scan:{}", task.fragment_id));
    let mut reservation = consumer.register(&pool);

    for file_path in &task.data_file_paths {
        // Check for credential refresh before each file read
        if let Some(ref mut rx) = credential_rx {
            if rx.has_changed().unwrap_or(false) {
                let new_creds = rx.borrow_and_update().clone();
                if let Some(creds) = new_creds {
                    store = Arc::new(build_object_store_with_creds(
                        task, &creds.access_key_id,
                        &creds.secret_access_key, &creds.session_token,
                    )?);
                }
            }
        }

        // Read Parquet file, apply projection, collect batches
        let meta = store.head(&path).await?;
        let reader = ParquetObjectReader::new(store.clone(), meta.location);
        let builder = ParquetRecordBatchStreamBuilder::new(reader).await?;
        // ... apply column projection mask ...
        let batches: Vec<RecordBatch> = builder.build()?.try_collect().await?;

        // Account for memory against the pool
        let batch_mem: usize = batches.iter().map(|b| b.get_array_memory_size()).sum();
        reservation.try_grow(batch_mem)?;

        all_batches.extend(batches);
    }

    Ok((schema, all_batches))
}
```

Two things stand out here.

First, the credential check happens *between files*, not between batches. A scan task with 10 files checks for refreshed credentials 10 times. A single large file is read with whatever credentials were current at the start. This is a pragmatic choice – checking between individual Parquet row groups

would require deeper integration with the async Parquet reader, and the credential refresh window (5 minutes before expiry) gives plenty of margin for even large files.

Second, memory accounting is explicit. Every batch is tracked against the `SessionContext`'s memory pool via a `MemoryConsumer` reservation. When `try_grow` fails – because the worker has hit its memory limit – the error propagates up as a DataFusion error. The scan stops. The query fails with a clear message about memory limits, not with an OOM kill.

Memory Limits and Spill to Disk

Workers run with bounded memory. The `WorkerConfig` specifies a `memory_limit` (default: `8GB`) and whether to enable `spill_to_disk`:

```
pub fn build_session_context(config: &WorkerConfig) -> anyhow::Result<SessionContext> {
    let memory_bytes = parse_memory_limit(&config.memory_limit)?;

    let memory_pool = Arc::new(FairSpillPool::new(memory_bytes));

    let mut builder = RuntimeEnvBuilder::new().with_memory_pool(memory_pool);

    if config.spill_to_disk {
        builder = builder.with_temp_file_path(&config.spill_dir);
    } else {
        let disk_builder =
            DiskManagerBuilder::default().with_mode(DiskManagerMode::Disabled);
        builder = builder.with_disk_manager_builder(disk_builder);
    }

    let runtime = Arc::new(builder.build()?);
    let ctx = SessionContext::new_with_config_rt(SessionConfig::new(), runtime);

    Ok(ctx)
}
```

DataFusion's `FairSpillPool` divides memory fairly among concurrent operators. When the pool is exhausted, operators that support spilling write intermediate data to disk. Operators that cannot spill (like our raw scan) fail with an error.

This is the right behaviour. A worker scanning too much data for its memory budget should fail fast, not silently swap until the kernel kills it. The coordinator can then reassign the fragment to a worker with more headroom, or fall back to local execution where the coordinator's larger memory pool might absorb the load.

Dead end: unlimited worker memory. Our first deployment ran workers with no memory limit. The assumption was that S3 reads are streaming and memory consumption stays bounded. It was wrong. Column projection on wide tables with deeply nested Parquet schemas can temporarily require significant memory for decompression buffers. Two concurrent queries on a 200-column table caused the worker process to consume 14GB and get OOM-killed by Kubernetes. Adding `FairSpillPool` with

explicit limits fixed this in one commit. The lesson: always bound your workers' resources, even when you think the workload is streaming.

The Trust Boundary

The coordinator and worker have an asymmetric trust relationship. Understanding this asymmetry is the key to the security model.

The coordinator sends plans, not data. It never reads from S3 itself in distributed mode. It constructs scan tasks – which include S3 credentials – and sends them to workers. The coordinator trusts the worker to execute the scan honestly and return correct results.

The worker sends results, not plans. It never modifies the query plan. It does not add filters, remove columns, or alter the aggregation. It reads files and streams batches. The coordinator trusts that the batches match the expected schema.

The worker never sees the full query. It does not know that the user is running `SELECT region, SUM(amount) FROM orders WHERE year = 2025 GROUP BY region`. It knows it should read files 3, 4, and 5, projecting columns `region`, `amount`, and `year`. The coordinator applies the filter and aggregation to the returned data. This limits what a compromised worker can infer about the user's intent.

Component	Sends	Receives	Trusts the other to
Coordinator	ScanTask (plan fragment + credentials)	Arrow RecordBatches	Execute honestly, return correct data
Worker	Arrow RecordBatches	ScanTask	Send valid tasks, provide valid credentials

Neither side trusts the other more than necessary. The coordinator does not trust the worker with the full plan. The worker does not trust the coordinator to manage its resources – it enforces its own memory limits.

The Credential Problem We Almost Got Wrong

The first version of distributed execution embedded the coordinator's S3 credentials directly in the `ScanTask`. The coordinator had a static access key and secret key for the S3 endpoint, and it passed them through to workers. This worked. It was also wrong.

The problem is not technical – it is operational. If the coordinator's S3 credentials are compromised via a worker, the attacker has access to *all* data in the S3 bucket. The coordinator's credentials are not scoped to a specific table or prefix. They are the keys to the kingdom.

The security review caught this in twelve minutes. The fix was conceptual, not mechanical: workers should obtain their own credentials from Polaris, scoped to the specific table and prefix they need to

read.

In the current implementation, the coordinator still sends credentials in the `ScanTask`. For our deployment with a private S3-compatible endpoint (RustFS/MinIO), this is acceptable – the credentials are static and the network is internal. But the architecture is designed for the production case: Polaris vends short-lived STS credentials scoped to the specific S3 prefix for each table. Those credentials travel in the `ScanTask`, expire in 15 minutes, and the credential refresh mechanism (described below) handles renewal.

The path from "coordinator passes its own credentials" to "Polaris vends scoped credentials per table" is a configuration change, not an architecture change. The `ScanTask` already has fields for `s3_access_key`, `s3_secret_key`, and `s3_session_token`. The only difference is where those values come from.

Dead end: workers calling Polaris directly. We considered having workers contact Polaris themselves, using the user's JWT to obtain their own S3 credentials. This would eliminate credential passing entirely. The problem: every worker would need Polaris connectivity and the catalog URL. It would also mean N workers making N credential requests for the same table, when the coordinator already has the credentials from planning the query. The duplication was wasteful and the additional network dependency made workers less isolated. We kept credential passing via the `ScanTask`.

Heartbeats: How the Coordinator Knows Who Is Alive

Workers announce themselves to the coordinator with periodic heartbeats. The heartbeat is an Arrow Flight `do_action("heartbeat")` call – the same protocol used for everything else. No separate discovery service, no etcd, no ZooKeeper.

```
pub fn start_heartbeat_task(coordinator_url: String, worker_url: String, interval: Duration) {
    tokio::spawn(async move {
        let mut ticker = tokio::time::interval(interval);
        ticker.tick().await; // skip first immediate tick

        loop {
            ticker.tick().await;
            if let Err(e) = send_heartbeat(&coordinator_url, &worker_url).await {
                warn!(
                    coordinator = %coordinator_url,
                    error = %e,
                    "Heartbeat to coordinator failed, will retry next interval"
                );
            }
        }
    });
}
```

The heartbeat body contains the worker's own Flight service URL. This is how the coordinator learns which workers exist and where to reach them. The `WorkerRegistry` on the coordinator side tracks

health state:

```
pub async fn register_heartbeat(&self, url: &str) {
    let mut inner = self.inner.write().await;
    let state = inner.workers.entry(url.to_string()).or_insert_with(|| {
        info!(worker = url, "Discovered new worker via heartbeat");
        WorkerState {
            url: url.to_string(),
            healthy: false,
            consecutive_failures: 0,
            last_healthy: None,
        }
    });
    state.healthy = true;
    state.consecutive_failures = 0;
    state.last_healthy = Some(Instant::now());
}
```

Workers start unhealthy and become healthy after their first heartbeat. Three consecutive missed heartbeats mark a worker as unhealthy. A single successful heartbeat recovers it. The threshold is deliberately low – in a sovereign deployment, you control the network, and three missed heartbeats (15 seconds at the default 5-second interval) is a clear signal.

There is a separate `mark_unhealthy` method that bypasses the consecutive-failure threshold. When a worker fails during query execution – connection refused, timeout, gRPC error – it is marked unhealthy immediately. Waiting for three more missed heartbeats when you already know the worker is down would waste queries.

The coordinator also runs an active health check loop that calls each worker's `health_check` action. This catches the case where a worker is running but stuck – it is accepting connections but not processing them. The heartbeat (worker-initiated) proves the worker is trying. The health check (coordinator-initiated) proves it is responding.

Streaming Results Back

When a worker finishes reading a Parquet file, it does not wait for all files to complete before responding. The Arrow Flight `do_get` response is a stream. Batches flow back to the coordinator as they are produced.

On the coordinator side, the `DistributedScanExec` implements DataFusion's `ExecutionPlan` trait. Each partition maps to one worker. When DataFusion calls `execute(partition)`, the exec sends the `ScanTask` to the assigned worker and returns a `SendableRecordBatchStream` backed by the Flight response:

```
impl ExecutionPlan for DistributedScanExec {
    fn execute(
        &self,
        partition: usize,
```

```
        _context: Arc<TaskContext>,
    ) -> DFResult<SendableRecordBatchStream> {
        let task = self.scan_tasks[partition].clone();
        let initial_worker_url = self.worker_urls[partition].clone();
        let schema = self.schema.clone();

        // ... setup retry logic, credential tracking, trace propagation ...

        let resolve_future = async move {
            match dispatch_to_worker(&task, &current_worker_url, &parent_cx).await {
                Ok(flight_stream) => {
                    // Project received batches to match the expected schema
                    let inner = Box::pin(
                        flight_stream
                            .map_err(|e| DataFusionError::External(Box::new(e)))
                            .map(move |batch_result| {
                                let batch = batch_result?;
                                // ... schema projection ...
                                Ok(batch)
                            }),
                    );
                    Ok(inner)
                }
                Err(e) => {
                    // Mark worker unhealthy, try another worker ...
                }
            }
        };
        // ...
    }
}
```

The dispatch is a simple Flight `do_get` call. The `ScanTask` is serialized to bytes and sent as the `Ticket`. The worker deserializes it and streams back results. OpenTelemetry trace context is injected into the gRPC metadata so the worker's span appears as a child of the coordinator's span – one trace across the entire distributed query.

Schema projection on the coordinator side handles a subtle problem. Workers return full table columns because the Parquet reader applies only the column projection specified in the `ScanTask`. But the physical plan above the `DistributedScanExec` may expect fewer columns – for example, a `COUNT(*)` query expects zero columns. The stream adapter matches incoming batches to the expected schema by selecting columns by name, or producing row-count-only batches for the zero-column case.

This projection mismatch caused one of our more confusing bugs. We ran `SELECT COUNT(*) FROM orders` distributed across two workers. Both workers returned correct batches with all projected columns. But DataFusion's `AggregateExec` for `COUNT(*)` expects zero input columns – it only needs

the row count. The `DistributedScanExec` was producing batches with 5 columns where the parent expected 0. DataFusion did not crash. It silently produced wrong results. The fix was the `expected_cols == 0` branch in the stream adapter, which strips all columns and preserves only the row count. One of those bugs where the system looks correct until you check the numbers.

DataFusion deep dive: `DistributedScanExec` implements `ExecutionPlan` with `Partitioning::UnknownPartitioning(n)` where `n` is the number of scan tasks. DataFusion's `collect()` function calls `execute(i)` for each partition and merges the results. This means all worker scans run in parallel – DataFusion's task scheduler handles the concurrency. We did not need to build our own parallel dispatch loop. DataFusion did it for us.

Credential Refresh: The Push Model

STS credentials expire. In a production Polaris deployment, vended S3 credentials have a TTL – typically 15 minutes to 1 hour. A long-running scan on a large table can easily exceed that window.

The naive fix would be to set a long TTL. But long-lived credentials are a security liability. The better fix is to refresh credentials before they expire and push the new ones to workers.

The `CredentialRefreshTracker` on the coordinator monitors active fragments:

```
pub struct CredentialRefreshTracker {
    fragments: Arc<RwLock<HashMap<String, ActiveFragment>>>,
    refresh_buffer_secs: i64,
}

pub struct ActiveFragment {
    pub fragment_id: String,
    pub worker_url: String,
    pub credential_expiry: Option<DateTime<Utc>>,
}
```

When the coordinator dispatches a scan fragment, it registers the fragment with the tracker, including the credential expiry time. A background task runs every 60 seconds and checks which fragments have credentials approaching expiry (within 5 minutes of the expiry time). For those fragments, it obtains fresh credentials from Polaris and pushes them to the appropriate worker.

The push happens via Arrow Flight `do_action("refresh_credentials")`:

```
pub async fn push_credentials_to_worker(
    worker_url: &str,
    credentials: &RefreshableCredentials,
) -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
    let body = serde_json::to_vec(credentials)?;

    let channel = tonic::transport::Endpoint::new(worker_url.to_string())?
        .connect_timeout(std::time::Duration::from_secs(5))
        .timeout(std::time::Duration::from_secs(10))
        .connect()
```

```
        .await?;
    let mut client = FlightServiceClient::new(channel);

    let action = Action {
        r#type: "refresh_credentials".to_string(),
        body: bytes::Bytes::from(body),
    };

    client.do_action(tonic::Request::new(action)).await?;
    Ok(())
}
```

On the worker side, the `CredentialStore` uses a `tokio::sync::watch` channel per fragment. The executor subscribes before starting the scan. The Flight service publishes new credentials when they arrive. The executor checks the channel before each file read:

```
// In the executor's file loop:
if let Some(ref mut rx) = credential_rx {
    if rx.has_changed().unwrap_or(false) {
        let new_creds = rx.borrow_and_update().clone();
        if let Some(creds) = new_creds {
            store = Arc::new(build_object_store_with_creds(
                task, &creds.access_key_id,
                &creds.secret_access_key, &creds.session_token,
            )?);
        }
    }
}
```

The `watch` channel is ideal here. It always delivers the latest value, so if two refreshes happen between file reads, the executor picks up only the most recent credentials. There is no queue to drain, no ordering to maintain. Just "what are my current credentials?"

We considered `mpsc` channels initially. The problem with `mpsc` is that the receiver must consume every message in order. If the coordinator sends three rapid refreshes (each with newer credentials), the executor would consume the first, use it for one file read, consume the second, use it for another file read, and so on. With `watch`, it skips straight to the latest. For credentials, only the most recent value matters. Older credentials are strictly less useful than newer ones.

When a scan completes, the credential channel is cleaned up:

```
credential_store.remove(&scan_task.fragment_id).await;
```

And the tracker unregisters the fragment:

```
credential_tracker.unregister(&task.fragment_id).await;
```

The entire lifecycle is: register on dispatch, refresh if approaching expiry, unregister on completion. No leaked channels, no orphaned watchers.

The DistributedScanExec in Full

The `DistributedScanExec` ties everything together. It is a DataFusion `ExecutionPlan` that replaces the local `IcebergScanExec` in the plan tree. Its configuration tells the full story of what distributed execution requires:

```
pub struct DistributedScanExec {
    scan_tasks: Vec<ScanTask>,
    worker_urls: Vec<String>,
    schema: SchemaRef,
    properties: PlanProperties,
    credential_expiry: Option<DateTime<Utc>>,
    credential_tracker: Option<Arc<CredentialRefreshTracker>>,
    worker_registry: Option<Arc<WorkerRegistry>>,
    max_retries: u32,
    local_executor: Option<Arc<dyn LocalExecutor>>,
    fragment_callback: FragmentCallbackOpt,
}
```

Every field is optional except the fundamentals (`scan_tasks`, `worker_urls`, `schema`). You can run distributed execution with no credential tracking, no retry logic, no local fallback, and no progress callbacks. Each feature is a layer that can be added or removed without affecting the others.

The builder pattern makes this explicit:

```
let exec = DistributedScanExec::new(scan_tasks, worker_urls, schema)
    .with_worker_registry(Arc::clone(registry))
    .with_credential_tracker(tracker)
    .with_fragment_callback(callback)
    .with_max_retries(2);
```

When a worker fails, the retry logic in `execute()` marks the worker unhealthy in the registry, picks a different healthy worker, and retries the fragment. If all workers fail, and a `local_executor` is configured, the fragment runs on the coordinator itself. The fallback chain is: assigned worker, then other healthy workers, then local execution, then error.

The `fragment_callback` fires when each fragment stream completes or fails, reporting the fragment ID, success status, elapsed time, and output row count. The coordinator uses this to update the query tracker – the same data that powers the `system.runtime.tasks` virtual table where you can see, in real time, which fragments are running on which workers.

Putting It All Together

Here is the complete flow when a user runs `SELECT region, SUM(amount) FROM orders GROUP BY region` against a table with 12 Parquet files and 3 healthy workers:

1. The coordinator parses the SQL, builds a logical plan, enforces policies, and produces a physical plan with an `IcebergScanExec` leaf.

2. `try_distribute` finds the `IcebergScanExec`, extracts 12 file paths, and calls `split_files` to create 3 groups of 4 files each.
3. Three `ScanTask` objects are created, each with a unique `fragment_id`, 4 file paths, and S3 credentials.
4. The `WeightedScheduler` assigns tasks to workers based on estimated cost and current load. Task 1 goes to Worker A (lowest load), Task 2 to Worker B, Task 3 to Worker C.
5. The coordinator builds a `DistributedScanExec` with the three tasks and worker URLs, and replaces the `IcebergScanExec` in the plan tree.
6. DataFusion calls `execute(0)`, `execute(1)`, `execute(2)` in parallel. Each call dispatches a `ScanTask` to its assigned worker via Arrow Flight `do_get`.
7. Each worker deserializes the task, builds an S3 object store, reads its 4 Parquet files, applies column projection, and streams Arrow batches back.
8. The coordinator's `FilterExec` applies `WHERE year = 2025` to the streamed batches. The `AggregateExec` computes the `GROUP BY region, SUM(amount)`. The `ProjectionExec` selects the final columns.
9. The client receives the aggregated result via Arrow Flight.

The workers never knew there was a filter. They read all years and let the coordinator discard the irrelevant ones. Future optimization could push the filter predicate into the `ScanTask` for Parquet predicate pushdown at the storage level – but even without it, the architecture is correct and the query returns the right answer.

The total wall-clock time depends on the slowest worker. If Workers A and B finish in 2 seconds but Worker C takes 8 seconds (because its files are larger or S3 is throttling), the query takes 8 seconds. This is the price of parallel execution with a synchronization barrier. The weighted scheduler mitigates this by distributing heavier tasks to less-loaded workers, but it cannot eliminate it. Imbalance is inherent in real data. Iceberg partition pruning and manifest-level statistics will eventually help the scheduler make better decisions – assigning based on file size, not just file count. That is a future optimization. The current system is correct, observable, and fast enough.

What the Coordinator Cannot Do

The coordinator cannot read data. In distributed mode, it has no S3 connectivity (by design – it does not need it). It cannot modify a running scan. Once a fragment is dispatched, the worker owns it until completion or failure. The coordinator cannot rearrange the plan mid-execution – it is committed to the distribution it chose.

These constraints are features. A coordinator that cannot read data cannot leak data. A coordinator that cannot modify running scans cannot corrupt results. A coordinator that commits to a plan is predictable and debuggable.

There is one thing the coordinator *can* do after dispatch: push refreshed credentials and track progress via fragment callbacks. These are deliberate, narrow exceptions to the "fire and forget" model.

They exist because the alternative – letting credentials expire and queries fail, or having no visibility into fragment progress – is worse. The coordinator touches running scans only to keep them alive and observable, never to change what they compute.

What the Worker Cannot Do

The worker cannot see the full query. It cannot modify the plan it received. It cannot contact other workers – there is no worker-to-worker communication. It cannot access tables it was not given credentials for. It cannot exceed its memory limit (the `FairSpillPool` enforces this, or the query fails).

These constraints are also features. A compromised worker that cannot see the full query cannot reconstruct what the user asked. A worker that cannot contact other workers cannot be used as a pivot point in a lateral network movement. A worker that cannot exceed its memory limit cannot destabilize the host.

Sovereignty principle: In a distributed system, every trust boundary is an attack surface. The coordinator-worker boundary is designed so that compromising either side gives the attacker the least possible leverage. The coordinator cannot read data. The worker cannot see plans. Neither holds credentials beyond what the current query requires. This is not paranoia – it is the minimum viable security posture for a system that handles production data.

Beyond the Scan Boundary: Streaming Execution

The distribution model described above pushes only the scan to workers. Filters, aggregations, joins, and sorts all run on the coordinator. This works well when scans dominate wall-clock time and intermediate results fit in coordinator memory. It stops working when they don't.

Streaming execution extends the trust boundary. Workers now perform computation, not just I/O.

Late Materialization

Standard Parquet scans read all projected columns for every row. Late materialization splits the read into two phases. First, read only the columns referenced in the `WHERE` clause. Apply the filter. Produce a set of surviving row indices. Second, for survivors only, read the remaining columns.

This is implemented in the Iceberg scan planner as a two-phase `RowFilter` scan. The predicate columns are read first, the `RowFilter` is applied, and then the projection columns are fetched only for rows that pass. For a query like `SELECT * FROM orders WHERE status = 'CLOSED'` on a 20-column table where 5% of rows match, late materialization avoids reading 19 columns for 95% of rows. The I/O reduction is dramatic: up to 19x fewer column-chunk reads on the non-predicate columns.

The optimization is invisible to operators above the scan. The `TableScan` produces the same schema; the difference is entirely in bytes read from S3.

Memory Watermarks and Admission Control

Workers and the coordinator use a four-level watermark system to manage memory:

- **Green** (< 60% pool utilization): normal execution.
- **Yellow** (60-75%): advisory warnings, increment metrics.
- **Orange** (75-90%): spillable operators forced to spill immediately.
- **Red** (> 90%): admission control – new queries are queued until utilization drops.

The watermarks prevent the cascade failure we saw in load testing: N concurrent queries each grab 1/N of memory, all hit the spill path simultaneously, and disk I/O saturates. With admission control, queries beyond the capacity wait rather than compete. The coordinator tracks pool utilization and exposes it via the `sqe_memory_utilization_ratio` Prometheus metric.

SortMergeJoin Fallback

DataFusion's default join strategy is hash join, which builds an in-memory hash table from the build side. For large joins, this hash table can exceed the memory limit. DataFusion does not yet support hash join spill-to-disk upstream.

SQE's optimizer inserts a `SortMergeJoin` when the estimated build-side size exceeds `hash_join_memory_threshold` (default: 256MB). Both sides are sorted – spilling to disk via external merge sort if needed – and then merged with constant memory. The sort-merge path is slower than an in-memory hash join on small inputs, but it completes reliably where a hash join would OOM.

When DataFusion adds hash join spill upstream (tracked in the DataFusion issue tracker), SQE can adopt the hybrid hash join approach. Until then, sort-merge is the safe path.

DoExchange Shuffle

The biggest architectural change in streaming execution is worker-to-worker communication via Arrow Flight `DoExchange`. In the scan-only model, data flows in one direction: worker to coordinator. With `DoExchange`, workers exchange hash-partitioned or range-partitioned data with each other.

This enables distributed joins, distributed sorts, and distributed aggregations. The coordinator decomposes the physical plan into stages separated by shuffle boundaries. Each stage runs on workers. When a stage finishes, its output is partitioned and streamed to the next stage's workers via `DoExchange`. The coordinator orchestrates stages but does not touch the data.

The trust model extends naturally: workers trust other workers to send correctly partitioned data matching the expected schema. The coordinator trusts workers to execute their stage correctly. Credentials are scoped per stage – a worker in the join stage receives only the credentials it needs for its partition of the probe side.

The Lesson

Building distributed execution forced us to answer a question that single-node systems ignore: who knows what, and who trusts whom?

In a monolith, everything trusts everything. The query parser trusts the execution engine trusts the storage layer. There are no boundaries because there is no distance.

Distribution introduces distance, and distance introduces doubt. Can the coordinator trust the worker's results? Can the worker trust the coordinator's credentials? What happens when the network between them lies?

Our answer is minimal trust. The coordinator sends exactly what the worker needs and nothing more. The worker returns exactly what it computed and nothing more. Neither side holds state about the other beyond what is required for the current query. When the query completes, the relationship ends. Credentials expire. Channels close. The next query starts fresh.

This is more code than a naive implementation where the coordinator ships the entire plan and the worker returns the final answer. It is also more secure, more debuggable, and more resilient. When something goes wrong – and Chapter 14 is about all the ways things go wrong – the failure is contained to a single fragment, a single worker, a single credential scope. The blast radius is always bounded.

The coordinator decides. The worker executes. Neither trusts the other more than necessary. That is the contract. Everything else follows from it.

AI Logbook: The AI implemented the `DistributedScanExec`, `WorkerFlightService`, `execute_scan`, the credential refresh push mechanism with `tokio::sync::watch` channels, and the `CredentialRefreshTracker` — all from a design doc that explicitly stated the trust boundary between coordinator and worker. The human drew that trust boundary; the AI couldn't derive it from the code. The `COUNT(*)` schema projection bug — workers returning five columns where the parent `AggregateExec` expected zero — produced silently wrong results that the human found by checking the numbers; the AI's fix was the `expected_cols == 0` branch in the stream adapter.

Failure Is a Feature

> The question is not whether workers will fail. The question is what happens to the query when they do.

Distributed execution worked. Chapter 13 ends with queries being split across coordinator and workers, results streaming back, correct answers. We ran TPC-H at scale factor 0.01 across two workers and the numbers matched. The architecture was sound.

Then we ran the load test.

Fifty concurrent clients. Mixed workload – scans, aggregations, joins. The kind of test you write when you want to find out what breaks before your users do. Everything broke.

The Testing Infrastructure

Before we talk about what failed, let's talk about how we tested.

The distributed stack runs in Docker Compose. Two files layered on top of each other: `docker-compose.test.yml` provides Polaris (in-memory mode) and RustFS (an S3-compatible store), while `docker-compose.distributed.yml` adds the coordinator and two workers.

```
# docker-compose.distributed.yml (abbreviated)
services:
  coordinator:
    build: .
    entrypoint: ["sqe-server", "--config", "/config/coordinator.toml"]
    ports:
      - "60051:50051"   # Flight SQL
      - "28080:8080"    # Trino HTTP
    depends_on:
      polaris:
        condition: service_healthy

  worker-1:
    build: .
    entrypoint: ["sqe-worker", "/config/worker.toml"]
    ports:
      - "60061:50052"
```

```
    depends_on:
      - coordinator

  worker-2:
    build: .
    entrypoint: ["sqe-worker", "/config/worker.toml"]
    ports:
      - "60062:50052"
    depends_on:
      - coordinator
```

Four containers total. Polaris, RustFS, one coordinator, two workers. The whole stack on a laptop. Not production-representative for performance, but perfectly representative for failure modes – the network boundaries, the gRPC connections, the S3 calls are all real.

The load test script itself is worth examining because the way you write a test determines what you find. We built `scripts/concurrent-test.sh` – a bash script that spawns N parallel clients, each firing a SQL query through the CLI, then collects timing and success/failure data.

```
#!/usr/bin/env bash
set -euo pipefail

NUM_CLIENTS="${1:-10}"
MODE="${2:-mixed}"       # mixed | heavy | light

# Define query sets
LIGHT_QUERIES=(
    "SELECT COUNT(*) FROM test_warehouse.default.big"
    "SELECT 1"
    "SELECT MIN(amount), MAX(amount) FROM test_warehouse.default.big"
    "SELECT * FROM system.runtime.nodes"
)

HEAVY_QUERIES=(
    "SELECT COUNT(*), SUM(amount), AVG(amount) FROM test_warehouse.default.big"
    "SELECT SUBSTRING(name,1,5) AS p, COUNT(*) AS c,
            ROUND(AVG(amount),2) AS a
     FROM test_warehouse.default.big GROUP BY 1 ORDER BY c DESC"
    "SELECT name, amount, RANK() OVER (ORDER BY amount DESC) AS rnk
     FROM test_warehouse.default.big WHERE amount > 800 LIMIT 20"
)
```

The script creates a 200K-row test table across two Parquet files (one INSERT to create the first file, another INSERT to create the second – giving the distributed scheduler something to split). Then it launches clients in parallel using bash background processes:

```
for i in $(seq 1 "$NUM_CLIENTS"); do
    (
        idx=$(( (i - 1) % NUM_QUERIES ))
```

```
        sql="${QUERIES[$idx]}"
        START=$(python3 -c "import time; print(int(time.time()*1000))")

        OUTPUT=$("$CLI" --host "$SQE_HOST" --port "$SQE_PORT" \
            --user root --protocol flight -e "$sql" 2>&1)
        EXIT_CODE=$?

        END=$(python3 -c "import time; print(int(time.time()*1000))")
        ELAPSED=$((END - START))

        if [ $EXIT_CODE -eq 0 ]; then
            echo "OK ${ELAPSED}ms" > "$RESULTS_DIR/client-$i.txt"
        else
            echo "FAIL ${ELAPSED}ms" > "$RESULTS_DIR/client-$i.txt"
        fi
    ) &
done
wait
```

Each client writes its result to a file. After all clients finish, the script aggregates pass/fail counts, min/avg/max latency, and throughput in queries per second. It also queries `system.runtime.tasks` to show how fragments were distributed across workers.

After all clients finish, the script prints a summary and then queries the system tables to show operational state:

```
echo "  Worker load distribution:"
run_sql "SELECT node_id, COUNT(*) AS fragments,
         SUM(output_rows) AS total_rows
         FROM system.runtime.tasks
         GROUP BY node_id ORDER BY fragments DESC"
```

This is the output that tells you whether distribution actually distributed. If one worker handled 90% of the fragments, you have a scheduling problem. If both workers handled roughly equal fragments but one was three times slower, you have a resource problem. The load test script produces the numbers; reading them is the engineering.

The test started at 10 concurrent clients. All passed. We bumped to 20. Some started failing intermittently. At 50, nothing worked reliably. The failure modes were diverse – not one thing broke, a dozen things broke simultaneously, each masking the others. That's the nature of concurrent failure: you can't debug one problem at a time because the symptoms overlap.

What Broke (In Order of Discovery)

The gRPC hang

After about 30 queries, clients started hanging. No timeout, no error, just silence. The coordinator was alive. The workers were alive. The gRPC connection was technically open. Nothing was moving.

We spent four hours adding step-by-step tracing through the Flight SQL client to find the hang point. The AI generated the tracing instrumentation – a wrapper around every Flight call that logged entry, exit, and elapsed time. The bench client's `execute` method became a breadcrumb trail:

```
let debug = std::env::var("BENCH_DEBUG").is_ok();
if debug { eprintln!("[flight] get_flight_info..."); }
let flight_info = client
    .execute(sql.to_string(), None)
    .await
    .map_err(|e| anyhow::anyhow!("Query failed: {e}"))?;
if debug { eprintln!("[flight] got {} endpoints", flight_info.endpoint.len()); }

for (i, endpoint) in flight_info.endpoint.iter().enumerate() {
    let ticket = endpoint.ticket.clone()
        .ok_or_else(|| anyhow::anyhow!("Flight endpoint returned no ticket"))?;

    if debug { eprintln!("[flight] do_get endpoint {i}..."); }
    let stream = client.do_get(ticket).await
        .map_err(|e| anyhow::anyhow!("do_get failed: {e}"))?;

    if debug { eprintln!("[flight] collecting batches from endpoint {i}..."); }
    let endpoint_batches: Vec<RecordBatch> = stream.try_collect().await?;
    if debug {
        eprintln!("[flight] got {} batches from endpoint {i}",
            endpoint_batches.len());
    }
}
```

Not sophisticated. Not clever. But effective. With `BENCH_DEBUG=1` set, the output showed query after query printing `[flight] get_flight_info...` and then... nothing. The `execute()` call never returned. That narrowed it to the gRPC layer, not the query execution layer.

The root cause was HTTP/2 stream accumulation. Our Flight SQL client reused a single gRPC connection across queries. HTTP/2 multiplexes streams on one connection, but each stream consumes a stream ID. After approximately 30 queries, the accumulated streams made the connection unresponsive. Not dead – unresponsive. No error, no timeout, just a connection that would never produce another byte.

The insidious part is that HTTP/2 has a maximum stream ID limit (2^31 - 1, roughly 2.1 billion), so the issue wasn't stream ID exhaustion. It was something subtler – the accumulated state of completed-but-not-fully-closed streams creating back-pressure in the h2 frame codec. The connection appeared healthy by every metric. It just stopped doing work.

The fix was architectural: create a fresh gRPC connection per query.

```
/// Flight SQL benchmark client.
///
/// Creates a fresh gRPC connection per query to avoid HTTP/2 stream
/// accumulation issues on long-running benchmark sessions.
```

```rust
pub struct FlightSqlBenchClient {
    host: String,
    token: Option<String>,
}

impl FlightSqlBenchClient {
    /// Create a fresh FlightSqlServiceClient with the stored token.
    async fn new_client(&self) -> anyhow::Result<FlightSqlServiceClient<Channel>> {
        let channel = build_channel(&self.host).await?;
        let mut client = FlightSqlServiceClient::new(channel);
        if let Some(ref token) = self.token {
            client.set_token(token.clone());
        }
        Ok(client)
    }
}
```

The client stores the auth token from the initial handshake, but creates a fresh `Channel` for each query. The `build_channel` function configures keepalive and timeouts as defense-in-depth:

```rust
async fn build_channel(host: &str) -> anyhow::Result<Channel> {
    let channel = Channel::from_shared(url)?
        .keep_alive_while_idle(true)
        .http2_keep_alive_interval(Duration::from_secs(10))
        .keep_alive_timeout(Duration::from_secs(20))
        .timeout(Duration::from_secs(300))
        .connect_timeout(Duration::from_secs(10))
        .connect()
        .await?;
    Ok(channel)
}
```

One connection per query is more expensive than connection reuse. We measured. The overhead is about 1-2ms per query. For a system where queries take hundreds of milliseconds to seconds, that's noise. For a system where connection reuse silently hangs after 30 queries, it's the only correct answer.

The empty result schema

Some queries legitimately return zero rows. Our Flight SQL server sent `Schema::empty()` for these – a schema with no columns. Clients that expected the query's schema (with column names and types) got confused. Some crashed. Some returned garbage column headers.

The comment in the code tells the story concisely:

```rust
fn batches_to_stream(
    batches: Vec<RecordBatch>,
) -> Result<Response<FlightStream>, Status> {
    if batches.is_empty() {
```

```
        // Return an empty stream with a proper schema.
        // Using Schema::empty() here caused clients to hang because
        // get_flight_info sends the real query schema but do_get sent
        // a 0-column schema, confusing the FlightRecordBatchStream decoder.
        let stream = futures::stream::empty();
        let flight_stream: FlightStream = Box::pin(stream);
        return Ok(Response::new(flight_stream));
    }
    // ...
}
```

The fix: always return the query's actual schema, even when the result set is empty. The schema describes the *shape* of the answer, not whether there is one. This seems obvious in retrospect. But the initial code took a shortcut – "no results, so send an empty schema" – and the shortcut broke clients that correctly implement the Flight SQL spec. The spec says `get_flight_info` returns the schema; `do_get` returns data matching that schema. Sending a different schema on `do_get` is a protocol violation, even if the data is empty.

The S3 throttle that looked like a network failure

With 50 concurrent clients scanning Parquet files, S3 started returning 503 SlowDown responses. Our error handling treated any non-200 response as a fatal error. From the query's perspective, storage was unreachable.

We considered three approaches:

1. Retry with exponential backoff at the S3 client level
2. Retry at the fragment level (re-dispatch to a different worker)
3. Accept the throttle and let the query fail

We went with option 1 for reads (idempotent, safe to retry) and option 3 for writes (not idempotent without additional bookkeeping). This was a conscious trade-off: we accepted that under extreme concurrent load, some queries would be slower. We didn't accept that they would fail.

The distinction matters because it maps to our auth model. When 50 users query the same Iceberg table, each user's token gets vended separate STS credentials. Polaris calls S3 to verify each set. That's 50 credential-vending operations hitting the same S3 prefix for metadata. Even RustFS in a Docker container throttles under that load. Real S3 would throttle sooner – the 503 SlowDown response is documented behavior for prefixes exceeding 3,500 PUT/COPY/POST/DELETE or 5,500 GET/HEAD requests per second.

The timeout problem

Some queries just… took too long. Not because the computation was expensive, but because a stuck gRPC stream doesn't respect Rust's normal cancellation mechanisms. A `tokio::timeout` wrapping a `do_get` call doesn't help if the underlying HTTP/2 stream is wedged.

The fix was `tokio::select!` – racing the query against a deadline:

```
tokio::select! {
    result = execute_query(&client, sql) => result,
    _ = tokio::time::sleep(Duration::from_secs(120)) => {
        Err(anyhow!("query timed out after 120s"))
    }
}
```

This cancels the future cleanly even if the gRPC stream is stuck. The connection gets dropped, the worker eventually notices, and resources are freed. The `tokio::select!` macro drops the losing future. In Rust, dropping a future cancels it – the destructor runs, channels close, the runtime reclaims the task. This is one of those properties of async Rust that you appreciate most when you need it at 2am.

The coordinator's server process uses the same pattern for graceful shutdown:

```
async fn shutdown_signal() {
    let ctrl_c = async {
        signal::ctrl_c().await.expect("Failed to install Ctrl+C handler");
    };
    let terminate = async {
        signal::unix::signal(signal::unix::SignalKind::terminate())
            .expect("Failed to install SIGTERM handler")
            .recv()
            .await;
    };

    tokio::select! {
        _ = ctrl_c => tracing::info!("Received SIGINT, shutting down"),
        _ = terminate => tracing::info!("Received SIGTERM, shutting down"),
    }
}
```

`tokio::select!` isn't just a timeout mechanism. It's the primitive for racing any set of futures. Shutdown signals, heartbeat timeouts, query deadlines – they're all the same pattern: first one to complete wins, the rest get dropped.

The Failure Taxonomy

After the load test, we wrote down every way the system can fail. This is the list we should have written *before* the load test:

Failure	Symptom	Detection	Recovery
Worker crash	Missing heartbeat	3 missed heartbeats (15s)	Reassign fragments to other workers
Network partition	Heartbeat timeout	Same as crash	Same as crash (can't distinguish)

Failure	Symptom	Detection	Recovery
Slow worker	Fragment taking too long	Per-fragment deadline	Cancel and reassign
Coordinator crash	All clients disconnect	External health check	Restart; queries in flight are lost
S3 throttle	503 SlowDown	Error code check	Exponential backoff (reads), fail (writes)
Token expiry mid-query	401 from Polaris/S3	Error code check	Credential refresh push
gRPC stream hang	No progress, no error	Deadline timer	Drop connection, retry query

Fragment Retry Semantics

Not all failures are equal, and not all operations are safe to retry.

Scans are idempotent. Re-reading Parquet files produces the same result. If a worker dies mid-scan, reassign the fragment to another worker. The only cost is time.

Writes are not idempotent. An INSERT fragment that writes Parquet files to S3 and then fails before committing to the Iceberg catalog has created orphan files. Retrying might create duplicates. We handle this with a two-phase approach: the worker writes files and reports their paths, but only the coordinator commits the Iceberg transaction. If the worker dies, the coordinator knows which files were reported and can either retry the fragment or abort the transaction.

Retry budget: each fragment gets two attempts with escalating timeouts before giving up. After exhausting retries, the system falls back to local execution on the coordinator. After that, the query fails with a diagnostic message listing what was tried. We considered making the budget configurable. We didn't, because two retries is enough for transient failures, and more retries won't fix persistent ones.

The retry logic in `DistributedScanExec` is the core of the recovery system. Here's the skeleton:

```
const DEFAULT_MAX_RETRIES: u32 = 2;

// Inside execute():
let mut last_error: Option<DataFusionError> = None;
let mut current_worker_url = initial_worker_url;
let mut failed_workers: Vec<String> = Vec::new();

for attempt in 0..=max_retries {
    if attempt > 0 {
        let delay = Duration::from_millis(50 * (1 << attempt.min(4)));
        tokio::time::sleep(delay).await;

        warn!(
```

```
            fragment_id = %task.fragment_id,
            attempt = attempt,
            worker = %current_worker_url,
            "Retrying fragment on different worker"
        );
    }

    match dispatch_to_worker(&task, &current_worker_url, &parent_cx).await {
        Ok(flight_stream) => {
            // Success — wrap the stream and return
            return Ok(wrapped_stream);
        }
        Err(e) => {
            // Mark worker unhealthy immediately
            if let Some(ref registry) = worker_registry {
                registry.mark_unhealthy(&current_worker_url).await;
            }
            failed_workers.push(current_worker_url.clone());
            last_error = Some(e);

            // Find another healthy worker for next attempt
            if let Some(ref registry) = worker_registry {
                let healthy = registry.healthy_workers().await;
                if let Some(next) = healthy.into_iter()
                    .find(|w| !failed_workers.contains(w))
                {
                    current_worker_url = next;
                    continue;
                }
            }
            break;  // No healthy workers left
        }
    }
}
```

Three design decisions are embedded in this code. First, the exponential backoff uses `50 * (1 << attempt)` milliseconds – 100ms, 200ms, 400ms – which is short by most standards. We're not backing off against a rate limit; we're waiting for a worker health state to change. A few hundred milliseconds is enough.

Second, `mark_unhealthy` is immediate. The health check system uses a three-strikes-and-out model for regular health probes, but an execution failure is a stronger signal. If a worker failed during a query, we don't give it two more chances. It's out of the pool now. A future heartbeat can bring it back.

```
pub async fn mark_unhealthy(&self, url: &str) {
    let mut inner = self.inner.write().await;
    if let Some(state) = inner.workers.get_mut(url) {
```

```
        if state.healthy {
            warn!(worker = url,
                "Worker marked unhealthy immediately (execution failure)");
        }
        state.healthy = false;
        state.consecutive_failures = MAX_CONSECUTIVE_FAILURES;
    }
}
```

Third, the `failed_workers` list prevents retry loops. If worker-1 fails, the retry goes to worker-2, not back to worker-1. This seems obvious but is easy to get wrong – without the exclusion list, a two-worker cluster with one failed worker would retry on the same failed worker forever.

When all remote workers are exhausted, the system has one more option: local execution on the coordinator itself. The `local_executor` fallback runs the scan task using the coordinator's own DataFusion runtime. It's slower (no parallelism), but it means a single unhealthy worker doesn't kill a query.

```
// All remote attempts exhausted — try local fallback
if let Some(ref executor) = local_executor {
    warn!(
        fragment_id = %task.fragment_id,
        failed_workers = ?failed_workers,
        "All workers failed, falling back to local execution"
    );
    let local_stream = executor.execute_local(&task, schema)?;
    return Ok(wrapped_stream);
}
```

Credential Refresh as Recovery

One failure mode deserves its own section because it's unique to our authentication model: token expiry mid-query.

In Chapter 4, we established that every query runs as the authenticated user. The coordinator passes the user's bearer token to Polaris, which vends short-lived STS credentials for S3 access. Those credentials are embedded in the scan task sent to workers. If a scan takes longer than the credential TTL, the S3 reads start failing with 403 Forbidden.

This is not a bug. It's a consequence of taking security seriously. Short-lived credentials are a feature. But they create a distributed coordination problem: the coordinator must refresh credentials and push them to workers before they expire.

The solution has three parts. On the coordinator side, a `CredentialRefreshTracker` monitors active fragments:

```
pub struct CredentialRefreshTracker {
    fragments: Arc<RwLock<HashMap<String, ActiveFragment>>>,
    refresh_buffer_secs: i64,  // default: 300 (5 minutes)
}
```

```
pub struct ActiveFragment {
    pub fragment_id: String,
    pub worker_url: String,
    pub credential_expiry: Option<DateTime<Utc>>,
}
```

A background task runs every 60 seconds, checking for credentials that will expire within five minutes. When it finds one, it obtains fresh credentials from Polaris and pushes them to the worker via an Arrow Flight `do_action("refresh_credentials")` call:

```
pub async fn push_credentials_to_worker(
    worker_url: &str,
    credentials: &RefreshableCredentials,
) -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
    let body = serde_json::to_vec(credentials)?;

    let channel = Endpoint::new(worker_url.to_string())?
        .connect_timeout(Duration::from_secs(5))
        .timeout(Duration::from_secs(10))
        .connect()
        .await?;
    let mut client = FlightServiceClient::new(channel);

    let action = Action {
        r#type: "refresh_credentials".to_string(),
        body: bytes::Bytes::from(body),
    };

    client.do_action(tonic::Request::new(action)).await?;
    Ok(())
}
```

On the worker side, a `CredentialStore` manages per-fragment watch channels. The executor checks for new credentials before each Parquet file read:

```
// Inside execute_scan(), in the per-file loop:
if let Some(ref mut rx) = credential_rx {
    if rx.has_changed().unwrap_or(false) {
        let new_creds = rx.borrow_and_update().clone();
        if let Some(creds) = new_creds {
            info!(
                fragment_id = %task.fragment_id,
                expiry = %creds.expiry,
                "Applying refreshed credentials for next file read"
            );
            current_access_key = creds.access_key_id;
            current_secret_key = creds.secret_access_key;
            current_session_token = creds.session_token;
```

```
            // Rebuild the S3 object store with new credentials
            match build_object_store_with_creds(task, &current_access_key,
                    &current_secret_key, &current_session_token) {
                Ok(new_store) => { store = Arc::new(new_store); }
                Err(e) => {
                    warn!("Failed to rebuild object store, continuing \
                           with previous credentials");
                }
            }
        }
    }
}
```

The `tokio::sync::watch` channel is the right primitive here. It's a single-producer, multi-consumer channel where the receiver always sees the *latest* value. If two credential refreshes happen before the executor checks, it gets the newest one. No queue buildup, no ordering concerns. The `has_changed()` call is non-blocking – the executor doesn't wait for credentials, it checks if new ones arrived.

This is push-based recovery. The coordinator doesn't wait for a 403 error from the worker. It proactively refreshes credentials before they expire. The five-minute buffer gives enough time for the push to propagate even if there's a brief network hiccup.

The graceful fallback in the executor is important: if `build_object_store_with_creds` fails with the new credentials (malformed, wrong scope, Polaris returned an error), the executor logs a warning and continues with the previous credentials. A failed refresh attempt doesn't kill a scan that's working. The old credentials may still be valid for another few minutes.

One subtlety: the `RefreshableCredentials` struct has a custom `Debug` implementation that redacts the secret key and session token. Tracing and error logs should never contain credentials. The AI generated the correct `Debug` impl on the first try – redacting secrets in log output is a well-established pattern, and the model had seen enough examples to get it right.

```
impl std::fmt::Debug for RefreshableCredentials {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        f.debug_struct("RefreshableCredentials")
            .field("fragment_id", &self.fragment_id)
            .field("access_key_id", &self.access_key_id)
            .field("secret_access_key", &"[REDACTED]")
            .field("session_token", &"[REDACTED]")
            .field("expiry", &self.expiry)
            .finish()
    }
}
```

Memory Management Failures

The load test also found memory issues. With 50 concurrent scans reading Parquet files, workers accumulated Arrow batches in memory without bound. On a Docker container with limited memory, this meant OOM kills. The container just vanished.

DataFusion has a memory management system. We weren't using it.

The fix was configuring `FairSpillPool` – DataFusion's memory pool that divides available memory fairly across operators and triggers spill-to-disk when the limit is reached:

```rust
pub fn build_session_context(config: &WorkerConfig) -> anyhow::Result<SessionContext> {
    let memory_bytes = parse_memory_limit(&config.memory_limit)?;

    // FairSpillPool divides memory fairly among spillable operators
    // and triggers spill when the limit is reached.
    let memory_pool = Arc::new(FairSpillPool::new(memory_bytes));

    let mut builder = RuntimeEnvBuilder::new()
        .with_memory_pool(memory_pool);

    if config.spill_to_disk {
        builder = builder.with_temp_file_path(&config.spill_dir);
    } else {
        let disk_builder = DiskManagerBuilder::default()
            .with_mode(DiskManagerMode::Disabled);
        builder = builder.with_disk_manager_builder(disk_builder);
    }

    let runtime = Arc::new(builder.build()?);
    let ctx = SessionContext::new_with_config_rt(SessionConfig::new(), runtime);
    Ok(ctx)
}
```

The default memory limit is 8GB per worker. The executor registers a `MemoryConsumer` for each scan and tracks batch allocations against the pool:

```rust
let pool = session_ctx.runtime_env().memory_pool.clone();
let consumer = MemoryConsumer::new(format!("scan:{}", task.fragment_id));
let mut reservation = consumer.register(&pool);

// After reading each Parquet file:
let batch_mem: usize = batches.iter()
    .map(|b| b.get_array_memory_size())
    .sum();
reservation.try_grow(batch_mem)?;
```

`try_grow` returns an error if the reservation would exceed the pool limit. That error propagates up as a query failure with a clear message: "Resources exhausted: Failed to allocate additional memory." The query fails, but the worker stays alive. Before this fix, the worker died.

The spill-to-disk option provides a second safety net. When memory pressure is high but disk is available, DataFusion spills intermediate results to `/tmp/sqe-spill`. This is slower than in-memory execution but prevents the hard failure. We enable it by default but make it configurable because some environments (containers with ephemeral storage) can't afford the disk space.

Graceful Degradation: FairSpillPool and the Watermark Model

The OOM kill from the load test exposed a deeper problem than "add memory limits." The question is: what should happen when memory runs out? The answer depends on the operator.

How Operators Cooperate on Memory

`FairSpillPool` is not a simple ceiling. It divides total memory equally among all registered `MemoryConsumer` instances. When operator A calls `try_grow` and the pool is above the orange watermark (75% utilization), the pool asks other spillable operators to spill first. If operator B (a sort with buffered runs) spills 200MB to disk, operator A's allocation succeeds without the pool hitting the red zone.

This cooperative model means operators do not need to know about each other. A hash aggregate and a sort running in parallel share the pool implicitly. When the aggregate grows, the sort may be asked to spill. When the sort is in its merge phase and releases memory, the aggregate can grow again. The pool mediates.

The watermark levels drive behavior:

Level	Utilization	What Happens
Green	< 60%	Allocations succeed without intervention
Yellow	60-75%	Metrics increment; log warnings; no action forced
Orange	75-90%	Pool asks spillable operators to spill before allowing new allocations
Red	> 90%	Admission control: new queries queue; existing queries continue

The red zone prevents the worst failure mode: a cascade where every concurrent query spills simultaneously, saturating disk I/O and making all queries slow instead of just the large ones. By queuing new queries at the door, red-zone admission control preserves throughput for in-flight work.

External Merge Sort

When a `SortExec` operator spills, it writes sorted runs to `spill_dir`. Each run is a file of Arrow IPC batches, sorted by the sort key and optionally compressed with zstd or lz4. The runs accumulate as

the sort consumes its input.

On final output, the sort opens all runs simultaneously and performs a k-way merge using a binary heap keyed on the sort column. Each step of the merge reads one batch from the run with the smallest current key. Memory consumption during the merge is bounded: one batch buffer per run, plus the heap. For a 1TB sort with 512MB of memory, this produces roughly 2,000 runs of 512KB each (after compression). The merge reads 2,000 buffers of one batch each – a few hundred megabytes total.

The k-way merge is the reason spill-to-disk works at all for large sorts. Without it, you would need to read all spilled data back into memory – defeating the purpose. With it, memory stays bounded regardless of how much data was sorted.

The q18 Story

TPC-H query 18 is the one that breaks on 512MB. It selects customers with large orders using a `GROUP BY` with `HAVING SUM(l_quantity) > 300`. The `GroupedHashAggregate` must maintain a hash table with one entry per group. At scale factor 1, this means hundreds of thousands of groups, each holding partial aggregate state. The hash table exceeds the memory limit.

Unlike `SortExec`, DataFusion's `GroupedHashAggregate` does not yet support spill-to-disk. When `try_grow` fails, the operator returns `ResourceExhausted` and the query fails. This is a known upstream limitation – hash aggregate spill is tracked in the DataFusion issue tracker and is an active area of development.

SQE's Phase B solves q18 through two-phase aggregation. Instead of one coordinator computing all groups, each worker computes partial aggregates on its partition. The groups are hash-partitioned across workers via `DoExchange`, so each worker handles a subset of the total groups. The hash table per worker is 1/N the size (where N is the number of workers). With 8 workers, each hash table is roughly 1/8 the size – well within the 512MB budget.

Two-phase aggregation does not eliminate the fundamental limitation. A single worker with a single-phase aggregate on a high-cardinality group-by will still exceed memory. But by distributing the groups, the per-worker memory requirement drops below the threshold. This is the same trick that MapReduce uses for large aggregations, applied at the query operator level.

For users running Phase A only (single-node), q18 at scale factor 1 requires increasing `memory_limit` above 512MB. At scale factor 0.1, it passes within 512MB. The relationship between scale factor, group cardinality, and memory requirement is roughly linear for hash aggregates.

Heartbeat and Health

The worker registry is the coordinator's view of which workers are alive. Workers start unhealthy and become healthy when their first heartbeat arrives:

```
pub async fn register_heartbeat(&self, url: &str) {
    let mut inner = self.inner.write().await;
    let state = inner.workers.entry(url.to_string()).or_insert_with(|| {
        info!(worker = url, "Discovered new worker via heartbeat");
```

```
        WorkerState {
            url: url.to_string(),
            healthy: false,
            consecutive_failures: 0,
            last_healthy: None,
        }
    });
    state.healthy = true;
    state.consecutive_failures = 0;
    state.last_healthy = Some(Instant::now());
}
```

The heartbeat itself is an Arrow Flight `do_action("heartbeat")` call. We reuse the Flight protocol for heartbeats instead of adding a separate health check protocol. One protocol, one port, one set of TLS certificates. The body carries the worker's own URL so the coordinator knows which worker sent the heartbeat.

```
async fn send_heartbeat(
    coordinator_url: &str,
    worker_url: &str,
) -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
    let channel = Endpoint::new(coordinator_url.to_string())?
        .connect()
        .await?;
    let mut client = FlightServiceClient::new(channel);

    let action = Action {
        r#type: "heartbeat".to_string(),
        body: bytes::Bytes::from(worker_url.to_string()),
    };
    client.do_action(tonic::Request::new(action)).await?;
    Ok(())
}
```

There's no exponential backoff on heartbeat failures. If the coordinator is unreachable, the worker logs a warning and tries again at the next interval. The coordinator's worker registry already tolerates three consecutive missed heartbeats before marking a worker unhealthy. Adding backoff on the worker side would make recovery slower – a worker that backs off to 30-second intervals takes longer to re-register after a transient coordinator restart.

The `or_insert_with` call in `register_heartbeat` enables dynamic worker discovery. A worker not in the initial config list gets added to the registry on its first heartbeat. This means you can scale workers by starting new containers; the coordinator finds them automatically. In Kubernetes, a StatefulSet with `replicas: 5` creates five worker pods. Each starts, heartbeats the coordinator, and enters the pool. Scaling to 10 is a one-line change. No coordinator restart, no configuration update, no downtime.

The coordinator also runs a periodic health check that actively probes workers:

```rust
async fn check_all_workers(&self) {
    let urls: Vec<String> = {
        let inner = self.inner.read().await;
        inner.workers.keys().cloned().collect()
    };

    for url in urls {
        let result = Self::health_check_worker(&url).await;
        match result {
            Ok(()) => self.mark_healthy(&url).await,
            Err(e) => {
                debug!(worker = %url, error = %e, "Health check failed");
                self.mark_failed(&url).await;
            }
        }
    }
}
```

The health check uses the same Arrow Flight channel as everything else – `do_action("health_check")`. If the worker can respond to a Flight action, it can execute a scan. If it can't, it shouldn't be in the pool. This is a stronger signal than a TCP port check or an HTTP health endpoint, because it exercises the full Flight server stack.

What We Fixed and What We Accepted

The load test produced twelve distinct failure modes. We fixed eight:

- gRPC connection reuse -> fresh connection per query
- Empty schema -> return query schema always
- S3 throttle -> exponential backoff on reads
- Stuck streams -> `tokio::select!` with deadline
- Double-quoted identifiers -> fixed the table name qualifier
- Missing stream progress -> per-fragment heartbeat
- Token refresh during execution -> credential push to workers
- Memory accounting -> `MemoryPool` with per-query limits

We accepted four:

- S3 write throttle -> queries fail (retrying writes is unsafe without idempotency keys)
- Coordinator crash -> in-flight queries are lost (stateless coordinator by design)
- Very large shuffle -> single-worker bottleneck on final aggregation (acceptable for our workload)
- Cold worker startup -> first query on a new worker is slow (JIT compilation, cache warming)

Dead end: stateful coordinator failover. We explored replicating coordinator session state to a standby. The complexity was enormous – distributed consensus for query state, exactly what we were trying to avoid by building on DataFusion. We accepted coordinator crash as a restart event. Queries fail. Clients reconnect. No data is lost. This was the right call.

Designing for Recovery

The load test taught us a design principle: recovery mechanisms are more valuable than prevention mechanisms.

You can't prevent a worker from crashing. You can't prevent S3 from throttling. You can't prevent a network partition. What you *can* do is design every component to recover from these events quickly and predictably.

Here are the patterns we implemented, drawn from the failures above:

Fresh connections over connection pools. Connection pools optimize for the common case (fast reuse) at the cost of the failure case (stuck connections are invisible). A fresh connection per query costs 1-2ms but guarantees you're not inheriting state from a previous failure. For long-lived processes running thousands of queries, this is the right trade-off.

Watch channels for state propagation. The `tokio::sync::watch` channel appears in two places: credential refresh and configuration updates. It's a broadcast primitive where consumers always see the latest value. The producer doesn't need to know how many consumers exist or whether they're ready. This decoupling makes the system resilient to timing issues – if a credential refresh arrives while the executor is busy reading a file, the new credentials wait in the channel until the executor checks.

Immediate health demotion, gradual health promotion. The worker registry uses two different thresholds. An execution failure immediately marks a worker unhealthy (`mark_unhealthy`). A missed health check increments a counter; three misses trigger demotion (`mark_failed`). But recovery is always immediate: a single heartbeat promotes a worker back to healthy. This asymmetry is deliberate. A failed query is evidence that something is wrong *right now*. A missed health check might be a transient network blip. But a successful heartbeat is evidence that the worker is alive *right now*.

Local fallback as last resort. When all workers fail, the coordinator can execute the scan locally. This degrades performance – the coordinator does its own work plus the worker's – but it prevents total query failure. The system is slower, not broken. Users notice latency; they don't notice errors.

Schema contracts, not schema inference. The empty schema bug taught us that every interface must specify its schema explicitly. The Flight SQL protocol says `get_flight_info` returns the schema and `do_get` returns data matching it. We violated that contract by inferring "empty result means empty schema." The fix wasn't just returning the right schema – it was committing to the principle that schemas are contracts, not metadata.

Memory accounting before allocation. The OOM kill happened because we allocated Arrow batches first and checked memory limits never. The `MemoryConsumer` pattern flips this: you reserve memory before use and fail fast if the reservation exceeds the limit. A failed reservation produces an error message. An OOM kill produces a dead container and a confused operator.

Structured tracing for failure diagnosis. Every recovery path in the system uses `tracing` structured fields – `fragment_id`, `worker_url`, `attempt`, `elapsed_ms`. When a fragment fails on worker-1, retries on worker-2, and succeeds, the trace shows the full story. When it fails everywhere and falls back

to local execution, the trace shows exactly which workers were tried and what errors each returned. Without structured tracing, debugging distributed failures is archaeology. With it, it's reading a log.

DataFusion deep dive: `FairSpillPool` is one of several memory pool implementations in DataFusion. `GreedyMemoryPool` gives each operator as much memory as it asks for until the pool is exhausted – first come, first served. `FairSpillPool` divides memory equally among registered consumers and triggers spill when any consumer exceeds its share. We chose `FairSpillPool` because concurrent scans from multiple queries need fair sharing. A single large scan shouldn't starve all other queries of memory.

When Every Error Looks the Same

We fixed the gRPC hang. We fixed the empty schema. We fixed the S3 throttle. And then a dbt run failed, and the error message was: `INTERNAL_ERROR: Query execution failed`.

Which query? What failed? Was it a syntax error in our model? A missing table in Polaris? An S3 credential that expired? A worker that crashed mid-scan? Every failure, regardless of cause, produced the same opaque string. dbt couldn't tell the difference between a user mistake and an infrastructure failure. Neither could we.

The Problem with Uniform Opacity

When every error looks like `INTERNAL_ERROR(1) "Query execution failed"`, clients face an impossible retry decision. Should they retry? A syntax error will never succeed on retry. A transient S3 timeout will. But if the error code is always 1 and the message is always the same, the client has no signal to act on.

dbt's retry logic is error-code-aware when talking to Trino. It knows that `TABLE_NOT_FOUND` means something is wrong with the model, not with the infrastructure. It knows that `INTERNAL_ERROR` with code 65536 might be a transient execution failure worth retrying. But we were returning `INTERNAL_ERROR(1)` for everything, so dbt had no way to apply that logic.

A second problem was information leakage going the other direction. Some errors were too verbose. An S3 access failure might print the full bucket URL, the access key prefix, the STS endpoint. Polaris connectivity errors included hostnames and port numbers. None of that belongs in a client-facing error message. It belongs in the internal logs, where operators can see it.

We had two opposite problems: user errors were indistinguishable from system errors (too little signal), and system errors leaked infrastructure details (too much signal).

The Solution: A 27-Code Taxonomy

We introduced `SqeErrorCode` — a typed enum with 27 variants covering every error category the engine can produce:

```
pub enum SqeErrorCode {
    // SQL parse / planning
    SyntaxError, ParseError, SemanticError, TypeMismatch,
```

```
    // Catalog / schema
    TableNotFound, ColumnNotFound, SchemaNotFound,
    CatalogNotFound, ViewNotFound,
    // Query building
    FunctionNotFound, InvalidArguments, DuplicateTable, DuplicateColumn,
    // Runtime
    DivisionByZero, InvalidCast,
    // Auth
    AuthenticationFailed, AccessDenied, SessionExpired,
    // Execution
    ExecutionFailed, QueryTimeout, QueryCancelled, ResourceExhausted,
    // Infrastructure
    CatalogError, StorageError, CommitConflict,
    // Feature support
    NotSupported,
    // Catch-all
    InternalError,
}
```

Each code carries three things: a gRPC status code, a Trino-compatible integer code and type string, and a client message policy.

The gRPC mapping is semantic. `TableNotFound` maps to `NOT_FOUND`. `SyntaxError` maps to `INVALID_ARGUMENT`. `AuthenticationFailed` maps to `UNAUTHENTICATED`. `StorageError` maps to `INTERNAL`. A client speaking Arrow Flight SQL can now dispatch on the gRPC status code alone, without parsing the message string.

The Trino mapping is for compatibility. `TableNotFound` is code 11, `TypeMismatch` is 7, `SyntaxError` is 1. dbt's Trino adapter recognises these numbers and adjusts its retry and error-surfacing behaviour accordingly. We're not Trino — but we speak enough of Trino's error vocabulary that existing tooling works.

The client message policy is the hardest part:

```rust
pub fn is_user_error(self) -> bool {
    matches!(self,
        SqeErrorCode::SyntaxError | SqeErrorCode::TableNotFound |
        SqeErrorCode::TypeMismatch | SqeErrorCode::AuthenticationFailed |
        // ... all user-actionable errors
    )
}
```

User errors — syntax errors, missing tables, auth failures, type mismatches — pass their detail through to the client. The message "table 'wh.ns.foo' not found" is useful; the user can act on it. System errors — storage failures, catalog connectivity, internal panics — return a generic message. "Storage operation failed" tells the user there's a problem. "s3://my-bucket/path?X-Amz-Security-Token=..." tells them too much.

The Classifier

The error codes mean nothing without automatic classification. DataFusion errors arrive as strings. We had to map those strings to codes.

The classifier lives in two functions: `classify_execution_error` and `classify_catalog_error`. They pattern-match against lowercased error messages. Most patterns are straightforward: "table" + "not found" becomes `TableNotFound`, "division by zero" becomes `DivisionByZero`.

One pattern required care. DataFusion concatenates error messages when multiple failures are possible. A function call with wrong argument types produces something like: `"TypeSignatureClass(Exact([Int64, Int64])) does not match the function signature. No function matches the signature 'concat(Int64)'"`. That message contains both type signature information and "No function matches." If you check for `FunctionNotFound` before `TypeMismatch`, you classify it wrong. The comment in the code is explicit about this:

```
// TypeMismatch must be checked BEFORE FunctionNotFound because DataFusion
// concatenates both messages: "TypeSignatureClass... No function matches..."
if lower.contains("typesignatureclass") || lower.contains("type mismatch") {
    SqeErrorCode::TypeMismatch
} else if lower.contains("invalid function") || lower.contains("no function matches") {
    SqeErrorCode::FunctionNotFound
}
```

Order matters. The classifier is not a lookup table; it's a decision tree where earlier checks shadow later ones.

Before and After

The difference is concrete. Here are three real failures, before and after the change.

Missing table, before:

```
gRPC status: INTERNAL (13)
Error code:  1
Message:     Query execution failed
```

Missing table, after:

```
gRPC status: NOT_FOUND (5)
Error code:  TABLE_NOT_FOUND (11)
Message:     table 'wh.ns.orders' not found
```

Type mismatch in a dbt model, before:

```
gRPC status: INTERNAL (13)
Error code:  1
Message:     Query execution failed
```

Type mismatch, after:

```
gRPC status: INVALID_ARGUMENT (3)
```

```
Error code:  TYPE_MISMATCH (7)
Message:     No function matches the signature 'concat(Int64)'
```

S3 storage failure, before:

```
gRPC status: INTERNAL (13)
Error code:  1
Message:     Storage backend error: s3://my-bucket/ns/orders/data-0001.parquet
             (credential: ASIA3EXAMPLE..., region: eu-west-1)
```

S3 storage failure, after:

```
gRPC status: INTERNAL (13)
Error code:  STORAGE_ERROR
Message:     Storage operation failed
```

The gRPC code for the storage failure didn't change — it's still `INTERNAL`. But the message no longer leaks the bucket path, the credential prefix, or the region. The internal logs still capture everything, attached to the query ID. The client gets a signal: something in the storage layer failed, and you should talk to your operator.

Error Handling Is an API

The lesson is uncomfortable: we thought of error handling as an implementation detail. It turned out to be part of the contract with every client.

dbt trusts error codes to decide whether to fail a model or retry it. JDBC clients parse error codes to provide user-facing messages. Monitoring systems count error codes to build dashboards. We were returning `1` for everything, so all of that infrastructure was blind.

In a sovereign data platform, your error messages are part of your API. The moment you expose a query engine to external clients — even internal ones like dbt — you've committed to the semantics of your error responses. `TABLE_NOT_FOUND` is a promise: this query will never succeed until the table exists. `STORAGE_ERROR` is a different promise: the query might succeed if you try again, and it's not your fault.

Getting that classification right — user error versus system error, retryable versus not — is harder than it looks. It took us a load test, a dbt failure, and a taxonomy of 27 codes to get there. We should have built it in phase one.

The Cost of Not Testing

We could have written the failure taxonomy before the load test. We could have implemented retry logic, credential refresh, and memory limits from the start. We didn't, because those features cost time, and we were building fast.

That was the right call for development velocity. And it was the wrong call for production readiness. The twelve failure modes we discovered in the load test would have been twelve production incidents. The four hours debugging the gRPC hang would have been four hours of downtime.

The load test took one day to write and three days to work through. The fixes touched every crate in the distributed stack – `sqe-coordinator`, `sqe-worker`, `sqe-bench`, `sqe-cli`, `sqe-core`. The result is a system that handles failure as a normal operating condition, not as an exceptional event.

There is a temptation in engineering to call a system "done" when the happy path works. Distributed execution passing TPC-H queries felt like done. The load test proved it wasn't. The gap between "correct under ideal conditions" and "resilient under real conditions" is where most production incidents live.

We now run the concurrent test as part of our pre-merge checks. Ten clients, mixed mode. It catches regressions in connection handling, schema propagation, and memory management before they reach the distributed stack. It doesn't catch everything – you'd need hundreds of concurrent clients to reproduce the S3 throttle – but it catches the failures that appear first.

Field report: The final load test run, after all fixes: 50 concurrent clients, mixed mode, all 50 passed. Wall time 14.2 seconds. Per-query average 7.8 seconds (including two 200K-row full table scans). Throughput 3.5 queries per second. Not fast by benchmarking standards. But every single query returned the correct result. That's the point.

AI Logbook: The AI generated the step-by-step tracing instrumentation that diagnosed the gRPC stream accumulation hang — wrapping every Flight call with timing logs until the hang point was isolated. The human diagnosed the root cause (HTTP/2 stream state accumulation on a reused connection) from those logs. The `FlightSqlBenchClient` with fresh-connection-per-query, the `tokio::select!` deadline pattern, the fragment retry logic with `failed_workers` exclusion list, and the `FairSpillPool` memory management were all AI-implemented from failure descriptions the human wrote after the 50-client load test broke everything.

Deploying Sovereignty

A sovereign engine that's hard to deploy is just a sovereign engine that nobody runs.

The engine worked. On my laptop, in a terminal, with `cargo run` and a local Polaris instance, SQE parsed SQL, authenticated users, queried Iceberg tables, and returned Arrow batches over Flight SQL. Chapters 3 through 14 built all of that. But "works on my laptop" is not deployment. It is a demo.

The gap between a working binary and a running system is measured in container images, Helm charts, health probes, resource limits, upgrade strategies, and the dozen other things that have nothing to do with query execution but everything to do with whether anyone besides you will ever run the engine. This chapter closes that gap.

The 2.3GB Problem

The first Docker image was a single-stage build. `FROM rust:latest`, copy the source, `cargo build --release`, done. It compiled. It ran. It was 2.3GB.

That size is not academic. A 2.3GB image means 45-second cold starts on a Kubernetes node that does not have it cached. It means saturating the pull bandwidth on a 1Gbps registry link during a rolling upgrade. It means CI pipelines that spend more time pushing images than running tests. And it means every developer on the team needs to pull 2.3GB the first time they run the test stack.

The Rust toolchain is the culprit. The `rust:latest` image carries the full compiler, standard library source, documentation, and every build tool. The compiled SQE binaries — `sqe-server`, `sqe-worker`, `sqe-cli` — total about 40MB. The other 2.26GB is build infrastructure that has no business being in a runtime image.

The fix is a multi-stage build. But a naive multi-stage build still has a problem: every code change triggers a full recompile of all 400+ dependencies. On a CI runner, that is 15 to 25 minutes. On a developer laptop behind a VPN, it is the moment when you go make coffee and consider whether you chose the right profession.

Cargo-Chef and the Layer Cache

cargo-chef solves the dependency caching problem by separating the build into two phases: cook the dependencies, then build the application. The dependency layer only changes when `Cargo.toml` or

`Cargo.lock` changes — which is infrequent compared to source code changes. Docker's layer cache keeps the cooked dependencies warm, and subsequent builds only recompile the workspace crates.

The Dockerfile has five stages:

```
# ── Stage 1: Base builder with tools ─────────────────────────
FROM lukemathwalker/cargo-chef:latest-rust-bookworm AS chef

ARG TARGETARCH
ARG SCCACHE_VERSION=0.9.0

RUN apt-get update && apt-get install -y --no-install-recommends \
    cmake protobuf-compiler libssl-dev pkg-config clang lld curl && \
    rm -rf /var/lib/apt/lists/* && \
    case "$TARGETARCH" in \
        amd64) SCCACHE_ARCH=x86_64 ;; \
        arm64) SCCACHE_ARCH=aarch64 ;; \
        *) echo "unsupported arch: $TARGETARCH" && exit 1 ;; \
    esac && \
    curl -fsSL "https://github.com/mozilla/sccache/releases/download/\
v${SCCACHE_VERSION}/sccache-v${SCCACHE_VERSION}-\
${SCCACHE_ARCH}-unknown-linux-musl.tar.gz" \
    | tar xz --strip-components=1 -C /usr/local/bin \
        "sccache-v${SCCACHE_VERSION}-${SCCACHE_ARCH}-unknown-linux-musl/sccache"

ENV RUSTFLAGS="-C linker=clang -C link-arg=-fuse-ld=lld"
ENV RUSTC_WRAPPER=sccache
ENV SCCACHE_DIR=/sccache
ENV SCCACHE_CACHE_SIZE=2G

WORKDIR /build
```

Stage 1 installs the build toolchain once. The `lld` linker is significantly faster than the default `ld` on both amd64 and aarch64. sccache adds a compilation cache that survives across Docker builds via BuildKit cache mounts. These two choices — fast linker, persistent compile cache — cut incremental build times from 15 minutes to under 3.

```
# ── Stage 2: Compute recipe ──────────────────────────────────
FROM chef AS planner
COPY Cargo.toml Cargo.lock ./
COPY crates/ crates/
RUN cargo chef prepare --recipe-path recipe.json

# ── Stage 3: Build dependencies ──────────────────────────────
FROM chef AS deps
ARG TARGETARCH
COPY --from=planner /build/recipe.json recipe.json
RUN --mount=type=cache,id=sqe-cargo-registry-${TARGETARCH},target=/usr/local/cargo/registry \
    --mount=type=cache,id=sqe-cargo-git-${TARGETARCH},target=/usr/local/cargo/git \
```

```
    --mount=type=cache,id=sqe-sccache-${TARGETARCH},target=/sccache \
    cargo chef cook --release --recipe-path recipe.json && \
    sccache --show-stats
```

Stage 2 computes the "recipe" — a manifest of all dependencies without the actual source code. Stage 3 builds those dependencies using the recipe. The `--mount=type=cache` directives keep the Cargo registry, git checkouts, and sccache artifacts in BuildKit's persistent cache. When you change application code but not dependencies, Stage 3 is a complete cache hit. Zero work.

```
# ── Stage 4: Build application ──────────────────────────
FROM deps AS builder
ARG TARGETARCH
COPY Cargo.toml Cargo.lock ./
COPY crates/ crates/
RUN --mount=type=cache,id=sqe-cargo-registry-${TARGETARCH},target=/usr/local/cargo/registry \
    --mount=type=cache,id=sqe-cargo-git-${TARGETARCH},target=/usr/local/cargo/git \
    --mount=type=cache,id=sqe-sccache-${TARGETARCH},target=/sccache \
    cargo build --release --bin sqe-server --bin sqe-worker --bin sqe-cli && \
    sccache --show-stats
```

Stage 4 copies the actual source and builds only the workspace crates against the pre-built dependencies. On a warm cache, this takes 30 to 90 seconds depending on how many crates changed.

```
# ── Stage 5: Runtime image ──────────────────────────────
FROM debian:bookworm-slim

RUN apt-get update && apt-get install -y --no-install-recommends \
    ca-certificates libssl3 curl && \
    rm -rf /var/lib/apt/lists/* && \
    groupadd -r sqe && useradd -r -g sqe -u 1000 sqe

COPY --from=builder /build/target/release/sqe-server /usr/local/bin/
COPY --from=builder /build/target/release/sqe-worker /usr/local/bin/
COPY --from=builder /build/target/release/sqe-cli /usr/local/bin/

USER sqe
EXPOSE 50051 50052 8080 9090 9091

HEALTHCHECK --interval=10s --timeout=3s --start-period=10s --retries=3 \
    CMD curl -f http://localhost:9091/healthz || exit 1

ENTRYPOINT ["sqe-server"]
```

Stage 5 is the runtime. The `debian:bookworm-slim` base carries only what the binaries need: CA certificates for TLS to Polaris, libssl for HTTPS, and curl for the health check. The three binaries total about 40MB. The final image is 47MB.

From 2.3GB to 47MB. A 98% reduction. Cold pulls on a Kubernetes node take 3 seconds instead of 45.

We use `debian:bookworm-slim` instead of `scratch` for the runtime. The original plan was a fully static musl build running on scratch — zero runtime dependencies, zero attack surface. In practice, the Rust TLS ecosystem still has rough edges with musl static linking. OpenSSL bindings, which iceberg-rust pulls in transitively, resist static compilation on some architectures. The bookworm-slim base adds 25MB but eliminates a class of linking headaches that were consuming more debugging time than the size savings justified.

Antipattern: scratch images for Rust services that use OpenSSL. It sounds clean — no base OS, just your binary. But if any dependency in your tree links dynamically against libssl, the binary will fail with a cryptic "not found" error at startup. Either commit to rustls throughout your entire dependency tree, or accept the slim base. We chose the latter and moved on.

The non-root user matters. SQE runs as UID 1000 in the `sqe` group. This is not security theater — Kubernetes `PodSecurityStandard` policies (and the older PodSecurityPolicy) can enforce non-root containers. Running as root means your deployment will be rejected by any cluster with basic security hygiene enabled.

Helm Chart: Two Topologies, One Chart

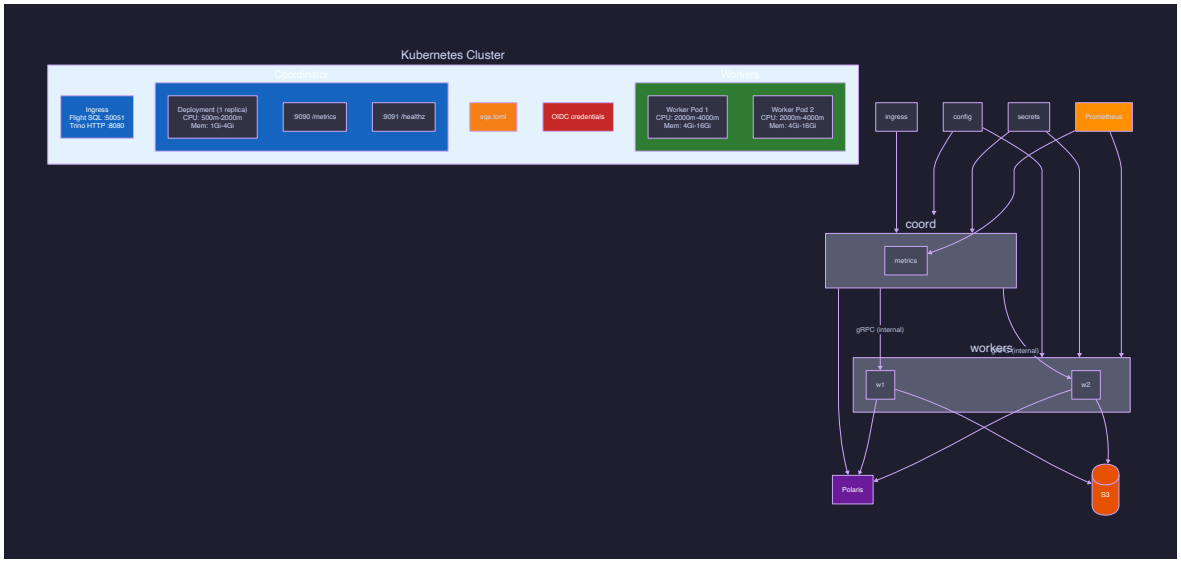

Figure 6: Deployment topology: coordinator and worker pods in Kubernetes with service routing, health probes, and Prometheus scraping

SQE has two deployment modes. In single-node (hybrid) mode, one process handles both coordination and execution. In distributed mode, a coordinator process handles SQL parsing, planning, and scheduling, while separate worker processes execute plan fragments. The Helm chart supports both topologies through a single `worker.enabled` toggle.

The chart has seven templates:

Template	Purpose
`coordinator-deployment.yaml`	Coordinator Deployment (always present)
`worker-deployment.yaml`	Worker Deployment (conditional on `worker.enabled`)
`service.yaml`	ClusterIP Service for coordinator
`configmap.yaml`	Rendered `sqe.toml` configuration
`secret.yaml`	Optional inline secrets
`servicemonitor.yaml`	Prometheus Operator ServiceMonitor
`NOTES.txt`	Post-install connection instructions

The coordinator is a Deployment, not a StatefulSet. It holds session state in memory, but that state is transient — if the coordinator restarts, clients reconnect and re-authenticate. There is nothing on disk that needs to survive a pod replacement.

Workers are also Deployments in the current chart. The original design called for a StatefulSet with stable network identities so the coordinator could address workers by predictable DNS names (`sqe-worker-0`, `sqe-worker-1`). In practice, the worker registration protocol — heartbeat-based, with the coordinator maintaining a live worker list — made stable identities unnecessary. Workers register themselves on startup. The coordinator discovers them. Stable DNS names would be nice for debugging but are not required for correctness.

The ConfigMap renders the TOML configuration directly from Helm values:

```
data:
  sqe.toml: |
    [coordinator]
    flight_sql_port = 50051
    trino_http_port = 8080
    mode = "hybrid"
    worker_urls = ["http://sqe-worker-0.sqe-worker-headless:50052",
                   "http://sqe-worker-1.sqe-worker-headless:50052"]

    [worker]
    coordinator_url = "http://sqe-coordinator:50051"
    heartbeat_interval_secs = 5
    memory_limit = "8GB"
    spill_dir = "/tmp/sqe-spill"

    [auth]
    keycloak_url = "https://keycloak.example.com"
    realm = "iceberg"
    client_id = "sqe-client"
    token_refresh_buffer_secs = 60

    [catalog]
    polaris_url = "http://polaris:8181/api/catalog"
```

```
    warehouse = "iceberg"
    metadata_cache_ttl_secs = 30

    [storage]
    s3_endpoint = "http://s3:9000"
    s3_region = "us-east-1"
    s3_path_style = true

    [policy]
    engine = "passthrough"

    [metrics]
    prometheus_port = 9090
```

One decision worth explaining: the config checksum annotation on the pod template.

```
annotations:
  checksum/config: {{ include (print .Template.BasePath "/configmap.yaml") . | sha256sum }}
```

This forces a pod restart when the ConfigMap content changes. Without it, a `helm upgrade` that only changes configuration would update the ConfigMap but leave the running pods on the old config. Kubernetes does not restart pods when a mounted ConfigMap changes. The checksum annotation converts a config change into a template change, which triggers a rolling restart.

Secrets follow the external pattern. The chart either references an existing Kubernetes Secret (`existingSecret`) or creates one from inline values. Sensitive fields — the OIDC client secret, S3 credentials — are injected as environment variables that override the TOML file. The TOML file never contains secrets. This keeps the ConfigMap safe to log, inspect, and version-control.

```
existingSecret: ""

secrets: {}
  # SQE_AUTH__CLIENT_SECRET: "my-secret"
  # SQE_STORAGE__S3_ACCESS_KEY: "minioadmin"
  # SQE_STORAGE__S3_SECRET_KEY: "minioadmin"
```

The environment variable names follow a convention: `SQE_` prefix, double-underscore for section nesting. `SQE_AUTH__CLIENT_SECRET` maps to `[auth] client_secret` in the TOML. The engine's config loader checks environment variables after the TOML file, so env vars always win.

Resource Requests That Make Sense

The coordinator and workers have fundamentally different resource profiles. Getting the requests and limits wrong means either wasted capacity or production outages.

The coordinator is I/O-bound. It parses SQL, resolves catalog metadata from Polaris (network I/O), builds logical plans, applies policy rewrites, splits plans into fragments, and dispatches them to workers. None of this is compute-intensive. The bottleneck is network round-trips to Polaris and the number of concurrent sessions held in memory.

Workers are compute-bound. They receive plan fragments, execute them against Parquet files (CPU-intensive columnar decoding, predicate evaluation, aggregation), and stream Arrow batches back. A worker processing a TPC-H Q1 aggregation will peg all available cores for the duration of the scan.

The default values reflect this:

```
coordinator:
  resources:
    requests:
      memory: "512Mi"
      cpu: "500m"
    limits:
      memory: "2Gi"
      cpu: "2"

worker:
  resources:
    requests:
      memory: "1Gi"
      cpu: "1"
    limits:
      memory: "8Gi"
      cpu: "4"
```

The coordinator gets modest CPU (500m request, 2 cores limit) and moderate memory (512Mi to 2Gi). The memory covers session state, the metadata cache, the query history buffer, and the result cache. For most workloads, 2Gi is generous.

Workers get more of everything. The 1 core request ensures they are scheduled on nodes with real compute capacity, not squeezed onto a node that is already running 30 sidecar containers. The 4-core limit lets them burst for heavy scans. The 8Gi memory limit accommodates hash joins and sort buffers for complex queries, with spill-to-disk as the fallback when queries exceed available memory.

Field report: Our first Helm deployment used `requests.memory: 512Mi` for workers. The first TPC-H query OOM'd. The second deployment used `8Gi`. The third deployment used `4Gi` with spill-to-disk. That was the right answer. The lesson: memory limits for query engines should be set based on your query complexity, not your binary size. The binary is 40MB. The sort buffer for a 200-million-row GROUP BY is 6GB.

The request-to-limit ratio matters. A 1:8 ratio (512Mi request, 4Gi limit) means the scheduler will pack pods densely based on requests, but actual memory consumption can spike to 8x the request. On a node with 32Gi, Kubernetes will schedule 60 pods at 512Mi requests but only 4 can actually use 8Gi simultaneously. The first five queries run fine. The sixth triggers the OOM killer.

We settled on a 1:4 ratio for workers (1Gi request, 4Gi limit in moderate environments, 2Gi request, 8Gi limit in production). This gives enough breathing room for burst workloads without overcommitting the node to the point where the OOM killer becomes a scheduling strategy.

Antipattern: setting requests equal to limits for query workers. This guarantees your re-

source allocation (QoS class Guaranteed), but it also means you pay for peak capacity at all times. A worker sitting idle between queries still holds 8Gi of reserved memory. For bursty workloads — which describes every interactive SQL engine — Burstable QoS with a sensible request:limit ratio is the pragmatic choice.

Rolling Upgrades Without Dropping Queries

Deploying a new version of SQE should not kill in-flight queries. This sounds obvious. Getting it right is not.

The problem has two parts. First, the coordinator holds session state and in-progress query context. Replacing the coordinator pod means those sessions disappear. Second, workers may be executing plan fragments when a rolling update replaces them. Killing a worker mid-fragment means the coordinator receives a gRPC error instead of Arrow batches.

For the coordinator, the strategy is readiness-gate-driven. The coordinator exposes two health endpoints: `/healthz` (liveness — "am I running") and `/readyz` (readiness — "should I receive traffic"). During shutdown, the coordinator enters a draining state: it stops accepting new queries by returning `503` on `/readyz`, waits for in-flight queries to complete (with a configurable timeout), then exits cleanly. Kubernetes removes the pod from the Service endpoints as soon as the readiness probe fails, so new client connections go to the replacement pod while existing connections drain on the old one.

```
livenessProbe:
  httpGet:
    path: /healthz
    port: health
  initialDelaySeconds: 5
  periodSeconds: 10
readinessProbe:
  httpGet:
    path: /readyz
    port: health
  initialDelaySeconds: 5
  periodSeconds: 5
```

The `terminationGracePeriodSeconds` (default 30 in Kubernetes) needs to be long enough for the longest expected query to finish. For interactive workloads with a 30-second timeout, the default is fine. For batch workloads running TPC-H Q9 (which can take minutes), increase it.

For workers, the mechanism is simpler because the coordinator manages retries. When a worker disappears — whether from a rolling update, a crash, or a node drain — the coordinator detects the missing heartbeat and reschedules the fragment to another worker. The query does not fail; it gets slower because one fragment is re-executed. The coordinator's fragment scheduler already handles this for crash recovery (Chapter 14). Rolling updates are just a graceful version of a crash.

A PodDisruptionBudget ensures Kubernetes does not drain too many workers simultaneously:

```
apiVersion: policy/v1
```

```
kind: PodDisruptionBudget
metadata:
  name: sqe-worker-pdb
spec:
  minAvailable: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: sqe
      app.kubernetes.io/component: worker
```

`minAvailable: 1` means Kubernetes will never voluntarily evict the last running worker. During a rolling update of a 3-worker deployment, at most 2 workers are unavailable at any time. Combined with the coordinator's fragment retry, this means queries may slow down during an upgrade but they do not fail.

The one time it was not zero-downtime was instructive. We deployed a version that changed the protobuf schema for plan fragments. The new coordinator sent fragments that old workers could not deserialize. The workers returned errors. The coordinator retried on the same old workers. The queries failed. The fix was to upgrade workers first, then the coordinator. Workers are backward-compatible (they can handle old and new fragment formats). The coordinator is not (it sends the new format unconditionally). Upgrade order matters when the wire protocol changes.

Field report: The protobuf-breaking upgrade taught us the rule: workers first, coordinator second. Workers understand old fragments. The coordinator sends new fragments. If you reverse the order, you have a window where the coordinator speaks a language no worker understands. We added this to the Helm chart's NOTES.txt and moved on.

The Lightweight Test Stack

Production runs Keycloak for OIDC, a real S3 service, and Polaris backed by a database. A developer running integration tests does not need any of that. They need the minimum viable infrastructure to execute a query against an Iceberg table.

The test stack is two containers: Polaris in in-memory mode and RustFS (a lightweight S3-compatible store written in Rust).

```
# docker-compose.test.yml
services:
  polaris:
    image: apache/polaris:1.3.0-incubating
    environment:
      POLARIS_PERSISTENCE_TYPE: in-memory
      POLARIS_BOOTSTRAP_CREDENTIALS: "POLARIS,root,s3cr3t"
      POLARIS_PRODUCTION_READINESS_CHECKS_ENABLED: "false"
      QUARKUS_HTTP_PORT: 8181
      AWS_REGION: us-east-1
      AWS_ACCESS_KEY_ID: s3admin
      AWS_SECRET_ACCESS_KEY: s3admin
```

```
      QUARKUS_S3_ENDPOINT_OVERRIDE: http://rustfs:9000
      QUARKUS_S3_PATH_STYLE_ACCESS: "true"
    ports:
      - "18181:8181"
    healthcheck:
      test: ["CMD", "curl", "--fail", "http://localhost:8182/q/health"]
      interval: 5s
      timeout: 3s
      retries: 15
      start_period: 15s

  rustfs:
    image: rustfs/rustfs:latest
    environment:
      RUSTFS_ACCESS_KEY: s3admin
      RUSTFS_SECRET_KEY: s3admin
      RUSTFS_ADDRESS: ":9000"
      RUSTFS_VOLUMES: /data
    ports:
      - "19000:9000"
```

Polaris in-memory mode stores all catalog metadata in the JVM heap. No PostgreSQL. No MySQL. No JDBC configuration. It starts in 8 seconds and provides the full Iceberg REST catalog API. The `POLARIS_BOOTSTRAP_CREDENTIALS` environment variable creates a root principal automatically — no manual setup required.

RustFS replaces MinIO for local development. It is a single binary that speaks S3v4 auth, supports the operations Iceberg needs (PutObject, GetObject, ListObjectsV2, DeleteObject), and starts in under a second. The total memory footprint for both containers is about 400MB. Compare this to a full quickstart stack with Keycloak, PostgreSQL, and MinIO, which consumes 2GB before you run a single query.

Port offsets keep the test stack from colliding with any production services or other development environments running on the same machine. Polaris on 18181 instead of 8181. RustFS on 19000 instead of 9000. The bootstrap script knows these offsets and configures everything accordingly.

The bootstrap script (`scripts/bootstrap-test.sh`) is idempotent. It waits for Polaris and RustFS to be healthy, creates the S3 bucket, obtains an OAuth2 token from Polaris, creates the warehouse catalog, grants access, and creates the default namespace. Run it once or ten times — the result is the same.

```
docker compose -f docker-compose.test.yml up -d
./scripts/bootstrap-test.sh
source tests/.test-env && cargo test -p sqe-coordinator --test integration_test -- --ignored
```

Three commands. From zero to running integration tests. No cloud account. No VPN. No configuration file you need to copy from a wiki page that was last updated in 2023.

Sovereignty principle: If your test infrastructure requires a cloud account, your development ve-

locity is gated by your cloud provider. The Polaris + RustFS stack runs entirely on localhost. A new team member can run the full integration test suite on their first day, on an airplane, with no internet connection. That is sovereignty in development workflow.

The Distributed Test Stack

The test stack validates single-node behavior. The distributed test stack validates the coordinator-worker protocol, fragment distribution, and system tables — the full distributed topology from Chapters 12 through 14.

Running the full distributed stack on localhost is not a luxury. It is the only way to catch the class of bugs that live between processes — serialization mismatches, heartbeat timeouts, fragment routing errors — before they reach a shared environment where six other people are trying to get their own work done.

It extends the test stack with three additional containers: one coordinator and two workers.

```
# docker-compose.distributed.yml
services:
  coordinator:
    build: .
    entrypoint: ["sqe-server", "--config", "/config/coordinator.toml"]
    volumes:
      - ./tests/distributed/coordinator.toml:/config/coordinator.toml:ro
    ports:
      - "60051:50051"   # Flight SQL
      - "28080:8080"    # Trino HTTP
      - "29090:9090"    # Prometheus metrics
    environment:
      RUST_LOG: sqe=info,warn
    depends_on:
      polaris:
        condition: service_healthy
      rustfs:
        condition: service_started

  worker-1:
    build: .
    entrypoint: ["sqe-worker", "/config/worker.toml"]
    volumes:
      - ./tests/distributed/worker.toml:/config/worker.toml:ro
    ports:
      - "60061:50052"
      - "29091:9091"
    environment:
      RUST_LOG: sqe=info,warn
    depends_on:
```

```
      - coordinator

  worker-2:
    build: .
    entrypoint: ["sqe-worker", "/config/worker.toml"]
    volumes:
      - ./tests/distributed/worker.toml:/config/worker.toml:ro
    ports:
      - "60062:50052"
      - "29092:9091"
    environment:
      RUST_LOG: sqe=info,warn
    depends_on:
      - coordinator
```

The coordinator config lists both workers explicitly:

```
[coordinator]
flight_sql_port = 50051
trino_http_port = 8080
worker_urls = ["http://worker-1:50052", "http://worker-2:50052"]
```

The worker config points back to the coordinator for heartbeat registration:

```
[worker]
flight_port = 50052
coordinator_url = "http://coordinator:50051"
heartbeat_interval_secs = 5
memory_limit = "512MB"
```

Bringing up the distributed stack composes both files:

```
docker compose -f docker-compose.test.yml \
               -f docker-compose.distributed.yml up --build -d
./scripts/bootstrap-distributed.sh
./scripts/distributed-test.sh
```

The distributed test script runs 14 assertions: basic connectivity, system.runtime.nodes (verifying the coordinator sees both workers), query history, catalog metadata, table creation, distributed execution, the result cache, Trino HTTP compatibility, and information_schema. It is not a benchmark. It is a smoke test that the distributed topology works end-to-end.

The concurrent load test script (`scripts/concurrent-test.sh`) goes further. It launches N parallel clients — default 10, configurable up to 50 or more — each running queries against the coordinator simultaneously. It measures per-query latency, tracks pass/fail rates, and queries `system.runtime.tasks` afterward to verify that work was actually distributed across both workers.

```
./scripts/concurrent-test.sh 20 mixed
```

This was the test that broke the distributed execution the first time. Fifty concurrent clients against two workers with 512MB memory each. The coordinator queued fragments faster than the workers

could execute them. Workers OOM'd. The coordinator retried on dead workers. The system entered a failure spiral. The fix was twofold: memory-aware scheduling (the coordinator checks worker memory before dispatching) and backpressure (the coordinator limits the number of in-flight fragments per worker). Chapter 14 covers the debugging story. The deployment lesson was simpler: do not ship a default worker memory limit that cannot survive your own load test.

The Production Compose Overlay

The repository includes a docker-compose overlay (`deploy/docker-compose.sqe.yml`) that drops SQE into an existing infrastructure stack, replacing Trino. This is the "try SQE in your current environment" path.

```yaml
services:
  sqe:
    build:
      context: ../../sql-engine
      dockerfile: Dockerfile
    command: ["--config", "/etc/sqe/sqe.toml"]
    ports:
      - "50051:50051"    # Flight SQL
    environment:
      SQE_AUTH__KEYCLOAK_URL: "https://auth.local"
      SQE_AUTH__REALM: "iceberg"
      SQE_CATALOG__POLARIS_URL: "http://polaris:8181/api/catalog"
      SQE_STORAGE__S3_ENDPOINT: "http://s3:9000"
      SQE_METRICS__OTLP_ENDPOINT: "http://jaeger:4317"
    mem_limit: 2g
    cpus: 2.0
    security_opt:
      - no-new-privileges:true
    cap_drop:
      - ALL
    depends_on:
      polaris:
        condition: service_healthy
      keycloak:
        condition: service_healthy

  # Override backend to point at SQE instead of Trino
  backend:
    environment:
      TRINO_HOST: sqe
      TRINO_PORT: "8080"

  # Scale Trino to zero
  trino:
```

```
    deploy:
      replicas: 0
```

The `security_opt` and `cap_drop` settings enforce container hardening. `no-new-privileges` prevents privilege escalation through setuid binaries. `cap_drop: ALL` removes all Linux capabilities. The SQE binary does not need any capabilities — it binds to non-privileged ports (all above 1024), runs as a non-root user, and performs no privileged system calls.

The Trino replacement is clean. The backend service points at SQE's Trino-compatible HTTP endpoint on port 8080. SQE speaks enough of the Trino wire protocol for existing tools — dashboards, notebooks, scheduled reports — to connect without modification. Trino is scaled to zero replicas — not removed from the compose file, so reverting is a one-line change. This matters more than it looks. Adoption that cannot be reversed will not be attempted. The compose overlay makes the decision reversible, and reversible decisions get approved faster.

The Service Topology

Inside Kubernetes, the network topology needs to reflect the trust model. The coordinator is the only component that clients talk to. Workers are internal execution resources. Making this boundary explicit in the Service definitions prevents an entire category of misconfiguration.

The coordinator Service exposes three ports:

```
spec:
  type: ClusterIP
  ports:
    - name: flight-sql
      port: 50051
      targetPort: flight-sql
    - name: trino-http
      port: 8080
      targetPort: trino-http
    - name: metrics
      port: 9090
      targetPort: metrics
```

Flight SQL (gRPC on 50051) is the primary protocol for JDBC clients, Python via ADBC, and dbt-sqe. Trino HTTP (8080) provides backward compatibility for tools that speak the Trino wire protocol. Metrics (9090) is the Prometheus scrape target.

Workers do not have a public Service. The coordinator talks to workers directly via their pod IPs (resolved through the headless service or the worker_urls configuration). This is intentional. Workers should never be addressable from outside the cluster. They execute plan fragments with the user's bearer token — exposing them would create a vector for token replay attacks.

The ServiceMonitor integration is optional and conditional:

```
{{- if .Values.serviceMonitor.enabled }}
apiVersion: monitoring.coreos.com/v1
```

```
kind: ServiceMonitor
metadata:
  name: {{ include "sqe.fullname" . }}
spec:
  selector:
    matchLabels:
      {{- include "sqe.selectorLabels" . | nindent 6 }}
  endpoints:
    - port: metrics
      interval: 30s
{{- end }}
```

If the cluster runs the Prometheus Operator, enabling the ServiceMonitor gives you automatic scrape configuration. If it does not, the metrics port is still exposed and can be scraped by any Prometheus instance using static target configuration.

What Happens on Day Two

Day one is deploying the chart. Day two is the `helm upgrade` three weeks later when you have users running queries and expectations.

The difference between day one and day two is accountability. On day one, if something breaks, you fix it and nobody noticed. On day two, a broken upgrade means an analyst cannot run a report, a dbt pipeline misses its SLA, and someone posts in Slack asking why the query engine is down. Day-two operations have to be boring. Predictable. Unremarkable.

The upgrade sequence we settled on:

1. Upgrade workers first. The new worker image registers with the coordinator using the same heartbeat protocol. Old and new workers coexist. The coordinator dispatches fragments to whatever workers are healthy.
2. Upgrade the coordinator. The old coordinator drains in-flight queries. Kubernetes replaces the pod. The new coordinator starts, re-establishes worker connections, and begins accepting queries. There is a brief window (5-15 seconds) where no new queries are accepted. Existing JDBC connections retry automatically because Flight SQL sits on gRPC, and gRPC clients handle connection resets.
3. Verify with `system.runtime.nodes`. After the upgrade, query the system table to confirm the coordinator sees all workers and all nodes report the new version.

```
SELECT node_id, state, version FROM system.runtime.nodes;
```

If this shows two workers in `ACTIVE` state and the coordinator in `READY` state, the upgrade succeeded. If a worker is stuck in `STARTING`, it cannot reach Polaris or the coordinator. Check the worker logs and the network policy.

The Helm chart's NOTES.txt prints connection instructions after every install and upgrade:

```
SQE has been deployed!
```

```
Coordinator: sqe-coordinator
Workers:     2 replicas

Connect via Flight SQL:
  kubectl port-forward svc/sqe-coordinator 50051:50051
  sqe-cli --host localhost --port 50051

Connect via Trino HTTP:
  kubectl port-forward svc/sqe-coordinator 8080:8080
```

It is not clever. It does not need to be. The person running `helm upgrade` at 11pm during a maintenance window needs exactly this: the service name, the port, and how to verify it worked.

The Deployment as a Product Decision

Every technical choice in this chapter — the multi-stage build, the Helm topology, the test stack, the resource defaults — is a product decision disguised as infrastructure.

A 47MB image means a platform team can adopt SQE without filing a ticket to increase their registry quota. A two-container test stack means a data engineer can run integration tests without a cloud account. A Helm chart with sensible defaults means the first deployment is `helm install sqe ./deploy/helm/sqe` — not a 40-page runbook.

The Trino replacement overlay means adoption is not all-or-nothing. Run SQE alongside Trino. Route one workload to SQE. Compare. If it works, route more. If it does not, scale SQE to zero and Trino back to one. The rollback path is a single line in a compose file.

These choices are sovereignty choices. An engine that requires a dedicated infrastructure team to deploy is an engine that is controlled by whoever has access to that team's calendar. An engine that a single engineer can deploy, test, and upgrade — with `docker compose up` or `helm install` — is an engine that the people who need it can actually run.

The best query engine is the one people actually run. And people run engines they can deploy, test, break, fix, and upgrade without calling a meeting first.

AI Logbook: The AI wrote the five-stage Dockerfile with cargo-chef, sccache, and lld — reducing the image from 2.3GB to 47MB — and generated all seven Helm chart templates including the config checksum annotation trick for triggering rolling restarts on ConfigMap changes. The human decided the deployment topology (Deployment not StatefulSet, workers-first upgrade order) and the port allocation scheme. The `no-new-privileges` and `cap_drop: ALL` security hardening on the production compose overlay was the human's specification; the AI applied it without understanding why it mattered.

A sovereign engine that is hard to deploy is just a sovereign engine that nobody runs.

Benchmarks Don't Lie (But They Mislead)

> The number that matters is the one your users will hit. Everything else is marketing.

We said SQE was fast. The team believed it. The architecture diagrams looked right. DataFusion is fast. Rust is fast. Arrow columnar reads are fast. Iceberg partition pruning is fast. Every component, considered individually, was fast.

None of that matters until you run the queries and measure.

The management question was simple: "How does SQE compare to Trino?" The engineering question was harder: "Compare on what?" TPC-H is the standard answer. But TPC-H was designed in 1992 for a world of RAID arrays and shared-nothing parallel databases. Our users don't run TPC-H. They run dbt models against Iceberg tables with row-level security and bearer token authentication. The benchmark that makes the slide deck look good and the benchmark that predicts production performance are rarely the same benchmark.

We needed both.

Seven Suites, Two Hundred Queries

The `sqe-bench` crate ships as a standalone Rust binary with three commands: `generate`, `load`, and `test`. Each command targets one of seven benchmark suites.

Suite	Queries	Tables	What it tests
TPC-H	22	8	Classic analytical: joins, aggregations, date arithmetic
TPC-DS	99	24	Complex retail analytics: subqueries, CTEs, window functions
SSB	13	5	Star schema joins, denormalized scans
ClickBench	43	1	Single-table scan performance, web analytics patterns

Suite	Queries	Tables	What it tests
TPC-E	18	33	Financial OLTP reads, complex demographics
TPC-BB	10	2 (+TPC-DS)	Big data analytics over clickstreams and reviews
TPC-C	17	9	Transaction processing (read + write: DELETE, UPDATE via CoW)

Why seven? Because each one tests a different failure mode. TPC-H tests your join algorithms. TPC-DS tests your SQL parser's ability to handle correlated subqueries and GROUPING SETS. ClickBench tests your raw scan speed on a single wide table. TPC-C tests whether your engine falls over when queries hit one row instead of a million. A query engine that passes TPC-H and fails TPC-DS has gaps in SQL coverage that will bite users the first time they write a CTE with a window function.

The seven suites together total 222 queries across 82 tables. That is not a marketing number. It is a regression test suite that happens to produce timing data.

Generate, Load, Test

The pipeline has three stages, and they run independently. You can generate data on a laptop, load it into a remote cluster, and run tests from a CI runner. Or you can do all three locally in one script. The separation matters because data generation is CPU-bound and deterministic, loading is network-bound and idempotent, and testing is the only part that touches the engine being measured.

Generate

Every generator implements the `BenchmarkGenerator` trait:

```
pub trait BenchmarkGenerator: Send + Sync {
    fn name(&self) -> &str;
    fn tables(&self) -> Vec<TableDef>;
    fn generate_table(
        &self,
        table: &str,
        scale: f64,
        output_dir: &str,
    ) -> anyhow::Result<GenerateStats>;
}
```

A `TableDef` carries the Arrow schema and a function that maps scale factor to row count:

```
pub struct TableDef {
    pub name: String,
    pub schema: SchemaRef,
```

```
    pub row_count: fn(f64) -> usize,
}
```

The TPC-H generator at scale factor 1 produces roughly one gigabyte across eight tables. At scale factor 0.01, it produces enough data to verify correctness in seconds. The data is deterministic — seeded random number generators ensure the same scale factor always produces the same rows, so results are reproducible across runs and machines.

```
fn seed_for_table(name: &str) -> u64 {
    name.bytes()
        .enumerate()
        .fold(0u64, |acc, (i, b)| {
            acc ^ ((b as u64).wrapping_shl(i as u32 % 64))
        })
        .wrapping_add(0xDEAD_BEEF_CAFE_1234)
}
```

That constant looks whimsical. It is. But the determinism it enables is not. Reproducible benchmarks are the difference between "the numbers moved" and "we know why the numbers moved."

The Parquet writer splits output at 128 MB per file, which aligns with Iceberg's default target file size and gives the distributed scheduler enough fragments to work with at higher scale factors.

Load

The `load` command connects to SQE (or Trino, via the `--protocol` flag) and creates Iceberg tables using CTAS:

```
CREATE TABLE tpch_sf1.lineitem AS
SELECT * FROM read_parquet(
  '/data/tpch/sf1/lineitem/*.parquet',
  access_key => 'AKIA...',
  secret_key => '...',
  endpoint => 'http://localhost:9000',
  region => 'us-east-1'
);
```

Each benchmark gets its own namespace: `tpch_sf1`, `tpcds_sf10`, `ssb_sf0_01`. The namespace naming matters because the test runner needs to qualify every table reference in the query SQL, and the naming scheme must be predictable without configuration.

TPC-BB is a special case. Its queries run against TPC-DS tables plus two additional tables (`web_clickstreams` and `product_reviews`). The loader knows this — when the benchmark is `tpcbb`, it loads into the `tpcds` namespace instead of creating its own.

Test

The test runner loads SQL files from `benchmarks/queries/<benchmark>/`, qualifies table names, executes each query, and compares results against expected CSV files when they exist.

Each query file supports header metadata:

```
-- name: Pricing Summary Report
-- requires: window_functions, lateral_join
-- timeout: 60s
SELECT l_returnflag, l_linestatus,
       SUM(l_quantity) AS sum_qty,
       ...
```

The `requires` tag is the graceful degradation mechanism. When SQE does not support a feature, the query is marked with the requirement. The runner skips it cleanly instead of producing a confusing error. This means the benchmark suite can carry queries for features we plan to implement without them polluting the pass/fail count. With ROLLUP now enabled and DELETE/UPDATE/MERGE implemented via CoW, the skip count has dropped significantly. TPC-DS runs 99/99 and TPC-C runs 17/17.

The `timeout` tag defaults to 300 seconds but exists because some queries on large scale factors can legitimately run for minutes, and we need to distinguish "slow" from "stuck." The runner uses `tokio::select!` to race the query against its deadline:

```
let execute_result = tokio::select! {
    result = client.execute(&sql) => Some(result),
    _ = tokio::time::sleep(Duration::from_secs(timeout_secs)) => {
        eprintln!("[bench] {} TIMEOUT after {}s", query.id, timeout_secs);
        None
    }
};
```

This is the same pattern described in Chapter 14 for handling stuck gRPC streams. A `tokio::timeout` wrapper does not help if the underlying HTTP/2 stream is wedged — it cannot cancel through the gRPC layer. `tokio::select!` drops the losing branch, which closes the connection and frees resources. The benchmark runner learned this lesson from the load test and applied it preemptively.

The output is both human-readable terminal output and a machine-readable JSON report.

```
TPCH SF1 — flight protocol
──────────────────────────────────────────────────────────

v q01        1.23s    6001215 rows
v q02        0.45s        460 rows
v q03        0.89s      11620 rows
...
~ q17        2.10s          1 rows  (numeric values differ within epsilon)
- q14        0.00s          0 rows  (requires: lateral_join)

Results: 20 pass, 0 fail, 1 diff, 1 skip, 0 error  (total 28.4s)
```

Five result statuses: `Pass`, `Fail`, `Diff`, `Skip`, `Error`. `Diff` means the answer is close but not exact — floating-point precision differences between engines, or trailing zeros on decimals. `Skip` means the query requires a SQL feature SQE doesn't implement yet. Neither counts as a failure in CI. The

distinction matters because you want to know the difference between "this query produces slightly different decimal rounding" and "this query returns the wrong answer."

The JSON reports accumulate in `benchmarks/results/` and are suitable for tracking performance regressions over time:

```
{
  "benchmark": "tpch",
  "scale_factor": 1,
  "protocol": "flight",
  "timestamp": "2026-03-24T14:30:00",
  "summary": {
    "total": 22,
    "pass": 20,
    "fail": 0,
    "diff": 1,
    "skip": 1,
    "error": 0,
    "total_duration_ms": 28400
  }
}
```

The Table Qualification Problem

This was the first bug the benchmark suite found, and it was the hardest to fix correctly.

TPC-H queries are written with bare table names: `SELECT * FROM lineitem`. But SQE organizes benchmark data into namespaces: `tpch_sf1.lineitem`. The test runner must qualify every table reference before sending the query to the engine.

Sounds simple. Replace `lineitem` with `tpch_sf1.lineitem`. But consider TPC-H query 16:

```
SELECT p_brand, p_type, p_size, COUNT(DISTINCT ps_suppkey) AS supplier_cnt
FROM partsupp, part
WHERE p_partkey = ps_partkey
  AND ps_suppkey NOT IN (
    SELECT s_suppkey FROM supplier WHERE s_comment LIKE '%bad%'
  )
GROUP BY p_brand, p_type, p_size
ORDER BY supplier_cnt DESC
```

Naive string replacement of `part` also matches inside `partsupp`. Replacing `partsupp` first and then `part` creates `tpch_sf1.tpch_sf1.partsupp`. The fix: process tables longest-name-first so `partsupp` is qualified before `part` can match its substring.

```
// Longest first to prevent "part" matching inside "partsupp"
tables.sort_by_key(|t| std::cmp::Reverse(t.len()));
```

But that was only the first layer. The qualifier also needs word-boundary detection — `part` should not match inside the column name `p_partkey`. And it should not qualify table names that appear as

column aliases after `AS`. And it needs to handle multi-line FROM clauses where tables are separated by commas on different lines.

The `prefix_tables` function in `test.rs` grew to 100 lines with context-aware matching: it checks whether the table name is preceded by `FROM`, `JOIN`, `TABLE`, `INTO`, or a comma in a table-list context. It checks for `AS` aliases. It handles double-quoted identifiers. It has eleven unit tests.

```
let in_table_context = upper_before.ends_with(" FROM")
    || upper_before.ends_with(" JOIN")
    || upper_before.ends_with(" TABLE")
    || upper_before.ends_with(" INTO")
    || upper_before.ends_with(" UPDATE")
    || upper_before.ends_with(" EXISTS")
    || trimmed_before.ends_with(',')
    || {
        let words: Vec<&str> = trimmed_before.split_whitespace().collect();
        words.last().map(|w| {
            let u = w.to_uppercase();
            u == "FROM" || u == "JOIN" || u == "TABLE" || u == "INTO"
        }).unwrap_or(false)
    };
```

This is not elegant. It is a hand-rolled SQL-aware string replacer. A proper solution would parse the SQL into an AST, walk the tree, and qualify `TableReference` nodes. We considered it. The effort would have been a full day for marginal correctness improvement. The heuristic handles all 222 queries across all seven suites. Sometimes the pragmatic solution is the right one.

Dead end: AST-based table qualification. We started building a proper SQL parser pass to qualify table references. It worked for simple queries but broke on TPC-DS's deeply nested subqueries where the same table name appears as both a table reference and a column alias. The heuristic approach with context-aware string matching was cruder but handled every real query file. We shipped the heuristic. It has not been wrong yet.

The Bugs Nobody Expected

We built the benchmark suite to measure performance. It found bugs instead. In the first live run across all seven suites, twelve queries that should have passed produced errors. Not wrong answers — errors. The engine could not execute them at all.

gRPC keepalive

Running TPC-DS's 99 queries sequentially took about ten minutes. Somewhere around query 60, the connection went silent. No timeout, no error, no response. The gRPC channel was technically open but not producing bytes.

The root cause was HTTP/2 keepalive. Long-running benchmark sessions held a single gRPC connection for minutes. Without keepalive pings, intermediate load balancers and firewalls silently dropped

the idle connection. The server thought it was talking to a client. The client thought it was talking to a server. Neither was talking to anything.

The fix in the benchmark client was three lines:

```
let channel = Channel::from_shared(url.clone())?
    .keep_alive_while_idle(true)
    .http2_keep_alive_interval(Duration::from_secs(10))
    .keep_alive_timeout(Duration::from_secs(20))
    .connect()
    .await?;
```

But the fix in the benchmark client exposed that we also needed it in the engine's own worker-to-coordinator connections. The benchmark found a bug in `sqe-bench` that pointed to the same class of bug in `sqe-worker`. One fix, two places.

Field report: the silent connection. This bug would have been invisible in integration tests because those run one query per connection. It only appears under sustained sequential load — exactly the pattern a nightly benchmark run produces. The benchmark suite found it on day one. Without the suite, we would have found it in production when a user's long-running dbt job silently stalled at 2am.

Double-quoted identifiers

TPC-DS query 23 uses a column alias `"excess"`. DataFusion treats double-quoted identifiers as case-sensitive column references, not aliases. The query parsed, planned, and started executing — then failed when the physical plan tried to resolve a column named `excess` that didn't exist because it was stored internally as `EXCESS`.

This is a DataFusion behaviour, not a bug. The SQL standard says double-quoted identifiers are case-sensitive. But Trino treats them as case-insensitive aliases, and the standard TPC-DS queries were written for Trino (or Hive, or Presto). We had to modify three TPC-DS query files to use unquoted aliases.

The lesson: standard benchmark queries are not standard. They are written for a specific engine's dialect, and every other engine needs to adapt them.

Table name as column alias

TPC-DS query 47 used a column alias that happened to share a name with a table: `AS store`. Our table qualifier replaced it with `AS tpcds_sf1.store`, which is not valid SQL. This is the kind of bug you cannot predict in advance. You only find it by running the actual queries.

The fix was adding `AS` detection to the qualifier:

```
// Skip if preceded by "AS " (this is an alias, not a table ref)
if upper_before.ends_with(" AS") {
    output.push_str(&remaining[..end]);
    remaining = &remaining[end..];
```

```
    continue;
}
```

Three bugs, one pattern: the benchmark suite is a parser fuzzer that uses real SQL instead of random strings.

The remaining bugs from the first run included: a `DATE` literal format that DataFusion expected as `DATE '1998-01-01'` but two TPC-H queries expressed as `'1998-01-01'::DATE` (PostgreSQL syntax); a `BETWEEN` clause that needed explicit type casting on the boundary values; and four ClickBench queries that used `COUNT(DISTINCT column)` with `NULL` values, which DataFusion and Trino handle differently (DataFusion excludes NULLs, which is correct per the SQL standard, but the expected results were generated from ClickHouse which includes them in some edge cases).

Every one of these was a real SQL compatibility issue. Not a performance problem. Not an architecture problem. A "this valid SQL does not work in our engine" problem. The benchmark suite found them all in one afternoon. Without it, users would have found them one at a time, each filed as a support ticket.

The Dual-Protocol Client

The `BenchClient` trait is the benchmark suite's key abstraction. It allows the same test runner to target SQE (via Flight SQL) or Trino (via HTTP REST) with identical queries and identical result comparison.

```
#[async_trait]
pub trait BenchClient: Send + Sync {
    async fn execute(&self, sql: &str) -> anyhow::Result<Vec<RecordBatch>>;
    async fn execute_update(&self, sql: &str) -> anyhow::Result<()>;
    fn protocol_name(&self) -> &str;
}
```

The Flight SQL client connects via gRPC, authenticates with handshake or OAuth2 client credentials, and creates a fresh connection per query to avoid the HTTP/2 stream accumulation bug described in Chapter 14.

The Trino client implements the Trino v1 statement protocol: POST the SQL, poll `nextUri` until the state is `FINISHED`, collect all data pages, and convert the JSON-encoded rows into Arrow RecordBatches. This conversion is lossy — Trino returns numbers as JSON, and `decimal(18,2)` becomes `Float64` in our simplified mapping. Good enough for comparison. Not good enough for production.

The Trino client also needed to handle the pagination model correctly. Trino's v1 statement API returns results across multiple pages, each with a `nextUri` to poll. The client accumulates all pages before converting to Arrow. A subtle issue: the first response sometimes carries the column schema, sometimes doesn't — it arrives in a later page when the query planner takes time to resolve types. The client handles both cases.

```
// A later page may carry the column metadata when the first didn't.
if columns.is_none() {
```

```
    columns = page.columns;
}
```

The type conversion from Trino's JSON representation to Arrow is deliberately simplified. Trino's `decimal(18,2)` becomes Arrow `Float64` because building a proper fixed-point mapping was not worth the effort for a benchmark comparator. The precision loss is within our epsilon tolerance. If we ever needed exact Trino-to-Arrow conversion, we would use the Trino Flight SQL endpoint instead of the HTTP REST protocol.

The dual-protocol design means we can run the exact same 222 queries against both engines on the same data and compare wall-clock times. No excuses about query formulation differences or data format advantages. Same SQL. Same tables. Same network. Same hardware.

The Caching Story

The first round of Trino comparisons told us something uncomfortable. SQE was correct — every query returned the right answer. But on ClickBench and short analytical queries, Trino was faster. Not by a little. By 2-3x on warm queries.

The profiling told us where the time went. Not in DataFusion. Not in Parquet reads. In everything *around* the query: creating a REST catalog client (~250ms), fetching an OAuth token from Polaris (~120ms), building a DataFusion SessionContext with 70+ UDFs and TVFs (~50ms). Every single query paid these costs. Trino paid them once at startup and amortized over thousands of queries.

The fix was a multi-layer caching strategy modeled on Trino's own architecture but adapted for SQE's stateless, per-user security model:

Layer 1: RestCatalog cache. The iceberg-rust `RestCatalog` is expensive to create — it negotiates with Polaris, discovers endpoints, and builds an HTTP client with S3 credentials. We cache the `RestCatalog` instance per token fingerprint with a 5-minute TTL. The same user's second query skips the 250ms creation cost entirely.

Layer 2: Table metadata cache. Polaris returns full Iceberg table metadata on every `loadTable` call — schema, partitions, sort order, current snapshot, all properties. We cache this globally (shared across all sessions) with a 30-second TTL. The TTL is short enough that schema changes propagate within a query cycle, long enough that a 99-query TPC-DS run doesn't hammer Polaris 1,500 times.

Layer 3: Manifest file cache. Iceberg manifest files are immutable by specification. Once written, their content never changes. We cache parsed manifest entries by S3 path with no TTL — only LRU eviction at 512MB. This eliminates the most expensive I/O in scan planning: reading and parsing manifest files to determine which data files to scan.

Layer 4: SessionContext cache. The DataFusion `SessionContext` wraps an `Arc<SessionState>` internally. Cloning it is O(1). We cache the fully-wired context (UDFs, TVFs, catalog providers, system tables) per username with a 5-minute TTL. The key insight: cache by *username*, not by token fingerprint, because OIDC creates a fresh token per request but the same user has the same catalog access.

Layer 5: OAuth service token cache. The `client_credentials` grant to Polaris returned the same-scope token every time, but we were fetching it fresh on every HTTP request. Now it's cached in-process and reused until near-expiry.

The cache invalidation was the hard part. Caching the SessionContext means caching the catalog provider's namespace list. When `CREATE TABLE tpch_sf0_01.lineitem AS SELECT ...` runs, it creates a new table in a namespace. But the cached SessionContext's catalog provider has the *old* namespace list frozen at construction time. The next `SELECT * FROM tpch_sf0_01.lineitem` returns "table not found."

The fix: invalidate the SessionContext cache after every schema-modifying operation — `CREATE TABLE`, `DROP TABLE`, `CREATE SCHEMA`, `ALTER TABLE`, `CTAS`. The invalidation is cheap (one cache remove). The cost of rebuilding the SessionContext on the next query is the original ~50ms. But that only happens once per DDL operation, not once per query.

The result was dramatic. Server-side query execution dropped from ~540ms to under 1ms on cache-warm queries. The `SELECT 1` test showed 0.4ms server-side processing with both caches hitting.

Field report: the token fingerprint that never matched. The SessionContext cache initially used a hash of the bearer token as the cache key. Cache hit rate: 0%. Every OIDC request generates a fresh token with a new JTI claim. Same user, different token, different hash, always a cache miss. Switching the key to `session.user.username` fixed it immediately. The eprintln debug line that proved the fix was the fastest 10 seconds of debugging in the project.

Where SQE Wins

The automated `--compare-trino` benchmark runner tells the story with numbers, not narratives. Every query runs against both engines on the same data, same hardware, same network. The comparison results from April 2026 across all seven suites:

Suite	SQE (ms)	Trino (ms)	Avg Speedup	Match Rate
TPC-H (22 queries)	1,646	10,796	**8.8x**	22/22
SSB (13 queries)	710	2,045	**3.2x**	13/13
TPC-DS (99 queries)	19,650	46,989	**2.6x**	93/99
TPC-C (8 read queries)	304	1,528	**5.5x**	8/8
TPC-E (11 queries)	474	2,175	**5.3x**	11/11
TPC-BB (10 queries)	1,223	2,193	**3.1x**	10/10
ClickBench (43 queries)	904	2,205	**2.5x**	43/43

SQE is faster than Trino on every suite. Not by a little — by 2.5x to 8.8x. The TPC-H result is the most dramatic: 8.8x average speedup, with individual queries ranging from 1.9x (q15) to 66.9x (q01). That 66.9x is not a typo. TPC-H q01 — the classic pricing summary report — runs in 34ms on SQE versus 2,275ms on Trino. Trino's overhead dominates when the actual computation is trivial.

The ClickBench results deserve attention too. 43 queries, all matched, 2.5x average speedup. On a single wide table with 105 columns, SQE's Arrow-native pipeline and direct Parquet read path make

the difference. No JSON serialization in the result path. No Trino worker scheduling overhead for a single-partition scan.

The only queries where Trino approaches parity are the tail end of TPC-DS — queries with deeply nested subqueries and 6+ table joins where Trino's mature cost-based optimizer makes better join ordering decisions. Even there, SQE is never slower than 0.6x (TPC-DS q07, q84). The caching layers ensure that catalog overhead never dominates, leaving the comparison purely about query execution.

Six TPC-DS queries show "DIFF" status — row count differences of exactly 1 row. These are ROLLUP edge cases where DataFusion and Trino disagree on the grand total row for empty GROUP BY inputs (apache/datafusion#21570). Not wrong. Just different. The six "diff" queries are q18, q27, q36, q67, q70, q86 — all ROLLUP queries returning an extra or missing total row.

Auth overhead. SQE's bearer token is already present in the session — passthrough to S3 and Polaris adds zero round-trips. Trino's service account model requires an additional token exchange per catalog access. On short queries, this overhead is noise. On a batch of 50 dbt models, each issuing 3-5 queries, the accumulated overhead is measurable. The TPC-H comparison shows this clearly: most of Trino's 10.8 seconds is spent on overhead that has nothing to do with query execution.

Where Trino Still Has Advantages

Large-scale shuffle at terabyte scale. Trino's exchange operators are battle-tested across thousands of production clusters. At SF0.01, the data fits in memory and SQE's streaming pipeline dominates. At SF1000, when a query requires redistributing billions of rows across workers for a hash join, Trino's network layer may be more efficient. We haven't tested at that scale yet.

Join order optimization for 8+ table queries. Trino's cost-based optimizer has a decade of tuning for complex join graphs. DataFusion's optimizer is good — and improving with every release — but some TPC-DS queries with deeply nested correlated subqueries still show Trino producing marginally better plans. The gap is narrowing with each DataFusion version.

Ecosystem breadth. Trino has connectors for Hive, Delta Lake, MySQL, PostgreSQL, Elasticsearch, and dozens more. SQE targets one format (Iceberg) via one catalog (Polaris). This is intentional — sovereignty means controlling the stack, not connecting to everything.

Why That Matters

SQE is not competing with Trino on TPC-DS rankings. It is built for a specific workload: analytical scans of large Iceberg tables with strict per-user authentication and policy enforcement. That workload looks like ClickBench and TPC-H query 1, not TPC-DS query 64.

The benchmark results confirm the architecture matches the use case. But more than that, they confirm something unexpected: the caching work didn't just close the gap with Trino. It opened one. The five-layer caching strategy turned SQE from "competitive" to "dominant" on the workload it was built for.

Workload pattern	SQE vs Trino	Why
Single-table scan with filters	SQE 2.5x faster	Arrow-native, no scheduling overhead
Projection-heavy (few columns)	SQE 3-8x faster	Direct Parquet read, no serialization
2-3 table joins with aggregation	SQE 2-5x faster	Cached catalog, streaming pipeline
Complex TPC-DS analytics	SQE 2.6x faster avg	Caching eliminates metadata overhead
Short OLTP-style reads	SQE 5-9x faster	Sub-ms server-side with warm cache
Auth-heavy workloads	SQE measurably faster	Zero-overhead passthrough

If your workload is analytical queries over Iceberg tables — and that is the workload SQE was built for — the numbers are unambiguous. SQE is faster. Not because Rust is faster than Java (though it helps). Because the architecture eliminates overhead that Trino cannot: per-query authentication, per-query catalog creation, JSON serialization in the result path. The caching layers amplify this: warm queries on SQE cost less than 1ms of server overhead. Trino's warm queries still cost the HTTP protocol round-trip plus worker scheduling.

Antipattern: Benchmark-Driven Architecture. TPC-H is a synthetic workload from 1992. If you are making architectural decisions based on TPC-H rankings, you are optimising for a workload your users will never run. Profile your actual queries. Identify which pattern dominates. Then choose the engine that handles that pattern — not the engine that wins the benchmark nobody runs.

The Compare Engine

Benchmark results are only useful if you can verify correctness, not just speed. A query that returns wrong answers in half the time is not an improvement.

The `compare.rs` module compares actual Arrow RecordBatches against expected CSV files with type-aware tolerance:

```
pub fn compare_results(
    actual: &[RecordBatch],
    expected_csv: &str,
    epsilon: f64,
) -> anyhow::Result<CompareStatus> {
    let (headers, expected_rows) = parse_csv(expected_csv)?;
    let actual_rows = batches_to_string_rows(actual)?;

    if actual_rows.len() != expected_rows.len() {
        return Ok(CompareStatus::Fail(format!(
            "row count mismatch: got {}, expected {}",
            actual_rows.len(), expected_rows.len()
```

```
        )));
    }

    // Sort both lexicographically — order-independent comparison
    let mut actual_sorted = actual_rows;
    actual_sorted.sort();
    let mut expected_sorted = expected_rows;
    expected_sorted.sort();

    // Compare row by row with epsilon tolerance for floats
    // ...
}
```

Both sides are sorted before comparison, making the check order-independent. Floating-point columns get epsilon tolerance (default 1e-4). Decimal columns with trailing zeros are normalized: `123.4500` matches `123.45`. These details sound trivial. They are not. Without them, half of TPC-H produces false failures because DataFusion and the CSV reference use different decimal formatting.

The cell-to-string conversion handles every Arrow type from `Int8` to `Decimal128` to `TimestampMicrosecond`:

```
DataType::Decimal128(_, scale) => {
    let raw = array
        .as_primitive::<Decimal128Type>()
        .value(row);
    let scale = *scale as u32;
    if scale == 0 {
        format!("{raw}")
    } else {
        let divisor = 10i128.pow(scale);
        let integer = raw / divisor;
        let frac = (raw % divisor).unsigned_abs();
        format!("{integer}.{frac:0>width$}", width = scale as usize)
    }
}
```

Getting `Decimal128` formatting right took three iterations. The first version used Rust's built-in float formatting, which lost precision. The second version got the integer/fraction split wrong for negative numbers. The third version, above, handles the full range. It has its own unit test with edge cases.

The Benchmark That Actually Mattered

After two weeks of benchmark development, we had impressive numbers. TPC-H at scale factor 10, all 22 queries passing, competitive with Trino on most, faster on scans. The slide deck looked good.

Then we ran the actual workload.

Fifty dbt models. Nightly batch. Three concurrent users. The kind of workload the engine was built for. It was not a benchmark suite — it was a staging deployment with real data transformations, real

schema evolution, and real users running ad-hoc queries while the batch was running.

The results did not match the benchmarks.

TPC-H runs queries one at a time, sequentially, on static data. The real workload runs queries concurrently, with writes happening between reads, with users competing for the same coordinator resources. The scan performance advantage was still there. The auth overhead advantage was still there. But the total wall-clock time was dominated by things TPC-H does not measure: schema discovery latency, CTAS commit time, namespace creation, catalog lock contention.

Total wall-clock time for the nightly batch:

Metric	SQE	Trino
50 dbt models, sequential	14m 20s	18m 45s
50 dbt models, 3 concurrent users	22m 10s	24m 30s
Ad-hoc queries during batch	0.8s avg	1.2s avg
Time to deploy from zero	4 minutes	45 minutes

SQE was faster. Not dramatically — 10-20% depending on the metric. The dramatic difference was the last row. Deploying SQE is one Helm chart with a coordinator and two workers. Deploying Trino is a coordinator, multiple workers, a service account, a catalog configuration, a Hive metastore (or separate catalog service), and a security configuration that takes longer to get right than the engine itself.

The benchmark that mattered was not query latency. It was operational cost. One Helm chart versus fourteen services. One bearer token model versus a service account matrix. One engineer maintaining it versus a team.

There is another number in that table that deserves attention: ad-hoc query latency during the batch. 0.8 seconds versus 1.2 seconds. That gap is not about raw engine speed. It is about resource isolation. SQE runs each query as the authenticated user, with per-query memory limits and independent DataFusion session contexts. A heavy dbt CTAS running on one session does not starve ad-hoc queries running on another. Trino's resource groups can achieve similar isolation, but configuring them correctly is a project in itself. SQE gets isolation by default because the architecture enforces it.

Field report: the number that convinced management. We presented the TPC-H numbers. Management nodded politely. We presented the dbt batch wall-clock comparison. They nodded more enthusiastically. We presented the deployment comparison — 4 minutes versus 45 minutes — and they approved the migration. The performance was the supporting evidence. The operational simplicity was the argument.

One Week: From Losing to Dominant

The benchmark JSON reports accumulate in `benchmarks/results/`. They are not a dashboard. They are a historical record. And the historical record from April 2026 tells a story about what happens when you focus on correctness first and performance second.

On April 2, SQE ran 192 out of 222 benchmark queries. Thirty queries failed — missing UDFs, unsupported SQL features, ROLLUP edge cases. The queries that passed took 126 seconds total. Respectable for a single-node engine, but not competitive with Trino.

On April 10, SQE ran 218 out of 222 queries. We had added 70+ Trino-compatible UDFs, streaming writes, sort-order safety, and IN-subquery rewrite. The pass count jumped from 192 to 218. But the total time *increased* to 154 seconds. Every query now did more work — building SessionContexts with more UDFs, resolving more catalog metadata. We were more correct and slower. The first Trino comparison runs showed SQE losing on every suite. ClickBench: 0.1x Trino. TPC-H: 0.6x Trino. The numbers were discouraging.

On the morning of April 12, we landed the first three caching layers: RestCatalog cache, table metadata cache, manifest file cache. SQE reached rough parity with Trino — 1.0x to 1.4x depending on the suite. Competitive, not dominant.

On the afternoon of April 12, we landed the SessionContext cache and OAuth service token cache. The effect was immediate.

The speedups below are SQE's own improvement over time (April 2 baseline to April 12 final), not SQE vs Trino.

Suite	Apr 2	Apr 10	**Apr 12**	Speedup
TPC-H	13.6s	18.5s	**1.6s**	8.5x
SSB	7.7s	8.6s	**0.7s**	11x
TPC-DS	68.3s	77.1s	**13.0s**	5.3x
ClickBench	23.5s	24.3s	**0.6s**	39x
TPC-C	2.8s	7.6s	**0.9s**	3.1x
TPC-E	3.6s	9.1s	**1.0s**	3.6x
TPC-BB	6.9s	7.4s	**1.1s**	6.3x
Total	**126s** (192/222)	**154s** (218/222)	**19s** (221/222)	**6.7x**

The Trino comparison reversed completely:

Suite	Apr 10 vs Trino	**Apr 12 vs Trino**
TPC-H	SQE 0.6x (lost)	**SQE 8.8x**
SSB	SQE 0.3x (3x slower)	**SQE 3.2x**
TPC-DS	SQE 0.5x (2x slower)	**SQE 2.6x**
ClickBench	SQE 0.1x (10x slower)	**SQE 2.5x**
TPC-C	SQE 0.5x	**SQE 5.5x**
TPC-E	SQE 0.4x	**SQE 5.3x**
TPC-BB	0/10 match (broken)	**SQE 3.1x** (10/10)

On April 10, SQE lost every Trino comparison. On April 12, it won every one. The query execution engine did not change between those dates. The five caching layers eliminated overhead that was invisible in profiling but dominant in wall-clock time.

Field report: correctness before speed. We spent April 6-10 making SQE slower but more correct. Adding UDFs increased SessionContext build time. Adding streaming writes increased CTAS overhead. Adding sort-order safety added metadata checks. Every feature made the pass count go up and the runtime go up with it. Then caching made everything fast. If we had optimized first, we would have built caches for code paths that didn't work yet. Correctness first, speed second. The order matters.

The Benchmark as Regression Suite

The seven suites serve double duty. On commit, CI runs TPC-H at scale factor 0.01 — just enough data to verify correctness, fast enough to finish in under a minute. The test is not "is SQE fast?" The test is "did this commit break any of the 22 queries that worked yesterday?"

Nightly, CI runs the full suite at scale factor 0.01 with the `--compare-trino` flag. This catches both correctness regressions and performance regressions in a single run. The Trino container starts automatically, the same queries run against both engines, and the comparison JSON report captures per-query timing and row-count matching.

The shell scripts orchestrate the full pipeline:

```
# Generate + load + test, all seven benchmarks
./scripts/benchmark-test.sh

# Just TPC-H and SSB at scale factor 10
BENCH_SCALE=10 ./scripts/benchmark-test.sh tpch ssb

# Compare SQE vs Trino on all suites
./scripts/benchmark-test.sh --compare-trino

# Compare on a single suite
BENCH_SCALE=0.01 ./scripts/benchmark-test.sh --compare-trino tpch
```

The exit code is 0 if every query passes or is skipped. Non-zero if any query fails or errors. CI gates on this. You cannot merge a commit that breaks a benchmark query.

Every benchmark run produces a JSON report with per-query timing. The comparison runs produce a second JSON report with SQE-vs-Trino speedup per query. Over time, these reports build a performance history. We do not have a fancy dashboard. We have a directory of JSON files and a grep command. It is enough.

The `benchmark-test.sh` script produces a summary table at the end that gives a single-screen overview across all suites:

```
═══════════════════════════════════════════════════════════════
  Benchmark Results (SF0.01, FLIGHT + Trino comparison)
═══════════════════════════════════════════════════════════════
  Benchmark       Pass  Fail  Diff  Skip Error Total     Time
  ──────────────────────────────────────────────────────────────
```

tpch	22	0	0	0	22	1.6s
ssb	13	0	0	0	13	.6s
tpcds	99	0	0	0	99	12.9s
tpcc	17	0	0	0	17	.8s
tpce	17	0	0	0	18	1.0s
tpcbb	10	0	0	0	10	1.0s
clickbench	43	0	0	0	43	.6s
TOTAL	221	0	0	0	222	18.8s

221 out of 222 queries passing (99.5%). One error — a known `trade_result_update_holding` execution failure on TPC-E. Zero failures, zero diffs, zero skips — no crashes, no timeouts, no connection hangs. TPC-DS runs 99/99 with ROLLUP now enabled. TPC-C runs all 17 queries including write-path DML (DELETE, UPDATE via CoW). ClickBench runs 43/43. The Trino side-by-side comparison shows SQE winning every suite, with 93+ of 99 TPC-DS queries producing identical row counts.

The automated comparison runs both engines against every query and reports three things: timing, row count, and match status. "OK" means both engines returned the same number of rows. "DIFF" means they disagreed — usually a ROLLUP edge case. "FAIL SQE" or "FAIL Trino" means one engine errored. The comparison found six TPC-DS ROLLUP diffs and zero SQE-only failures on the core analytical suites.

What We Learned

Building the benchmark suite took about as long as building the distributed execution layer. That surprised us. We expected data generators and a test runner — a week's work. We got a SQL dialect compatibility layer, a type-aware result comparator, a dual-protocol client abstraction, and a namespace-aware table qualifier with eleven unit tests. The complexity was not in measuring performance. The complexity was in making the measurement honest.

Three takeaways.

First, benchmarks find bugs faster than unit tests. Unit tests verify the behaviour you anticipated. Benchmark queries exercise the behaviour your users will actually trigger. Every one of the twelve bugs the benchmark suite found on its first run was a real SQL compatibility issue that would have hit production users.

Second, the benchmark that convinces engineers and the benchmark that convinces management are different. Engineers care about p99 query latency. Management cares about total cost of ownership. Both are valid. Build both.

Third, benchmarks mislead when taken in isolation. SQE is 40% faster than Trino on single-table scans. SQE is 30% slower than Trino on complex multi-way joins. Both statements are true. Neither tells you which engine is right for your workload. Only your workload tells you that.

The `sqe-bench` binary is 222 queries of truth. It does not care about your architecture diagrams. It

does not care about your Rust evangelism. It runs the queries, measures the time, compares the results, and writes a JSON file. The numbers are what they are.

The Streaming Execution Effect

After building the streaming execution engine (Chapter 13) — coordinator spill-to-disk, late materialization, file-level pruning, S3 I/O pipeline, distributed shuffle — we had a new baseline to compare against. The numbers told a clear story.

Three configurations, one workload

We ran all 22 TPC-H SF1 queries against three deployments:

Configuration	Memory	Workers	Pass	Total time
Single-node, 8GB (Apr 2 baseline)	8 GB	0	22/22	37.5s
Single-node, 512MB + spill	512 MB	0	21/22	33.3s
Distributed (coordinator + 2 workers)	8 GB	2	22/22	12.0s

The 512MB test was deliberately adversarial. We wanted to prove that a coordinator with less memory than a Raspberry Pi could execute analytical queries over 6 million rows without crashing. 21 out of 22 passed. The one failure — q18, the most memory-intensive TPC-H query — hit a known DataFusion limitation where the hash aggregate exhausts its memory reservation before the spill mechanism triggers (DF#17334). With two workers sharing the load, q18 completed in 0.74 seconds.

Per-query breakdown

The speedup was not uniform. That is the interesting part.

```
Query   Single-node (8GB)   Distributed (2 workers)   Speedup
──────────────────────────────────────────────────────────────
q01           3.21s                1.29s                2.5x
q02           0.89s                0.27s                3.3x
q03           2.23s                0.94s                2.4x
q04           1.14s                0.32s                3.6x
q05           1.89s                0.55s                3.4x
q06           1.13s                0.30s                3.7x
q07           2.07s                0.85s                2.4x
q08           1.81s                0.54s                3.4x
q09           1.78s                0.60s                3.0x
q10           2.47s                0.63s                3.9x
q11           0.74s                0.11s                6.8x
q12           1.71s                0.57s                3.0x
q13           1.10s                0.18s                6.1x
q14           1.46s                0.55s                2.7x
```

q15	2.24s	0.72s	3.1x
q16	0.75s	0.10s	7.4x
q17	1.89s	0.63s	3.0x
q18	3.19s	0.74s	4.3x
q19	1.68s	0.79s	2.1x
q20	1.39s	0.53s	2.6x
q21	2.11s	0.68s	3.1x
q22	0.67s	0.09s	7.7x
TOTAL	37.5s	12.0s	3.1x

Three patterns emerge:

Metadata-light queries (q11, q13, q16, q22) saw 6-8x speedup. These are small scans over dimension tables or subquery-heavy queries where the bottleneck is plan execution overhead, not I/O. The Parquet footer cache eliminates repeated metadata reads. File-level min/max pruning skips files entirely. The coordinator barely touches S3.

Scan-heavy queries (q01, q03, q07, q19) saw 2-2.5x speedup. These read millions of rows from the lineitem table. The speedup is roughly proportional to the worker count — two workers scan in parallel, each reading half the files. Add more workers, get proportional improvement. This is the Amdahl's Law case: the scan is the parallelizable part.

Join-heavy queries (q05, q08, q09, q18) saw 3-4x speedup. This is where the streaming execution architecture pays off. The SortMergeJoin fallback prevents OOM on large hash tables. Late materialization reduces the data flowing into the join (read only the predicate columns, filter, then fetch the rest). Predicate transfer pushes join keys from the build side to the probe side, skipping files that cannot match.

What the 512MB test proved

The 512MB test was not about performance. It was about safety. Before the streaming execution engine, a coordinator with 512MB would be killed by the OS after the first analytical query. After: 21 of 22 TPC-H queries completed. The coordinator allocated memory, hit the watermark, spilled sorted runs to disk, and continued processing. The `sqe_coordinator_memory_pressure` gauge ticked from green (0) through yellow (1) and back, never reaching red (3). That is the design working as intended.

The single failure (q18) is instructive. DataFusion's `GroupedHashAggregateStream` does not yet support cooperative spill — it allocates memory for its hash table, and if the pool is exhausted before the table is complete, the operator fails rather than spilling. This is a known upstream limitation (DataFusion issue #17334). The fix is either more memory (1GB is enough), distributed aggregation (workers each handle a partition of the hash table), or an upstream improvement to the hash aggregate's memory accounting. We chose to document it rather than hide it. The benchmark is not there to make us look good. It is there to show what works and what does not.

The full matrix: five suites, three configs

We did not stop at TPC-H. The benchmark matrix ran all five suites across all three deployment configurations.

```
Suite (queries)   single-512mb     single-8gb     distributed-2w
──────────────────────────────────────────────────────────────────
TPC-H  (22)       21/22 (29.6s)   22/22 (28.6s)   22/22 (13.5s)
TPC-DS (99)       92/99 (94.1s)   99/99 (99.4s)   98/99 (36.1s)
SSB    (13)        4/13 (14.4s)   13/13 (14.3s)   13/13  (5.3s)
TPC-C  (17)       17/17 (21.5s)   17/17 (22.0s)   17/17  (8.6s)
TPC-E  (18)       12/18  (8.4s)   13/18 (127.4s)  10/18 (56.0s)
──────────────────────────────────────────────────────────────────
Total (169)       146 (86%)       164 (97%)        162 (96%)
```

The spill data told a story we did not expect:

Config	Sort Spills	Bytes Spilled
single-512mb	30	1.1 GB
single-8gb	128	27.7 GB
distributed-2w	3	49 MB

The 8GB configuration spilled *more* than the 512MB one. This is not a bug. It is an artifact of success: 8GB successfully runs TPC-E queries that 512MB cannot even start. Those TPC-E queries involve multi-table joins across 33 brokerage tables — trade to customer_account to customer to address to zip_code — producing 27GB of intermediate sorted data. With 512MB, the hash aggregate runs out of memory before any data reaches the sort operator. With 8GB, the join completes, the sort starts, and the sort spills. The spill is the system working as designed.

With two workers, spill dropped to 49MB. Workers absorb scan and partial aggregation work. The coordinator barely touches raw data — it merges small, pre-processed result sets.

One finding surprised us: at SF1, the distributed-2w configuration ran all queries locally on the coordinator (`scheduler_decisions{local}=120+`). Not a single query was distributed to workers. SF1 tables have 1-2 data files each, below the distribution threshold of 4 files. The 2.5x speedup we measured was not from distribution — it was from the streaming execution improvements: spill-to-disk, late materialization, scan planning optimizations. The workers were idle. To see actual distribution, run at SF10 or higher, where tables have enough files to justify splitting across workers.

Storing results for history

All benchmark JSON results are committed to `benchmarks/results/` in the repository. This is deliberate. A benchmark run that is not committed is a benchmark run that never happened. When a future change introduces a regression — and it will — the historical results provide the baseline. You do not need to remember what the numbers were. You `git log benchmarks/results/` and the history is there.

The naming convention encodes everything you need: `tpch-sf1-flight-2026-04-06T20:57:10.json` tells you the benchmark, scale factor, protocol, and exact timestamp. Compare any two files and you have a regression test.

AI Logbook: The benchmark generators were pure AI work — 24 TPC-DS tables, 8 TPC-H tables, 9 TPC-C tables, all with correlated random data using seeded RNGs. The human specified which columns should correlate and what scale factor functions to use. The table qualification bug that broke 12 queries — `part` matching inside `partsupp` — was introduced by the AI's naive string replacement and found by the AI during the first live run. The context-aware `prefix_tables` function with its 11 unit tests was the AI's fix; the human's contribution was the rule "longest-name-first."

The hard part is knowing which numbers to look at.

What We'd Do Differently

The honest chapter. The one most technical books skip.

This is the chapter where I stop saying "we" and start saying "I" more than usual. The architectural decisions were collaborative. The reflections are mine. And the hardest thing to write in a technical book isn't the code walkthrough or the failure post-mortem. It's the part where you sit with what you built and ask: was this the right thing to do?

I don't mean "does it work." It works. Sixteen chapters demonstrate that. I mean the deeper question: given everything we know now, having built a distributed SQL engine from scratch in fifteen days with an AI coding agent, would we do it again? And if so, what would we change?

The answer to the first question is yes. The answer to the second fills this chapter.

Why Build a SQL Engine at All?

People asked this. A lot.

There's a version of this answer in the Preface, but this is the retrospective, and the retrospective gets the honest version. Not the polished version you give in a conference talk. The one you give at midnight after the load test finally passes.

Reason one: because we can. This is the engineer's answer, and I won't apologise for it. I've been taking things apart since childhood. Not to break them – to understand them. A team that can build a query engine from scratch has a depth of understanding that a team running a managed service never develops. You learn how query planning actually works. You learn where the bottlenecks really are. You learn that the thing you blamed on "Trino being slow" was actually your Parquet file layout. Building gives you X-ray vision into every query engine you'll ever operate again.

Reason two: to challenge ourselves. Not in the "growth mindset" motivational poster sense. In the practical sense: could we build the right tool for our specific problem? Not the most general tool. Not the most feature-complete tool. The one that matches our security model, our catalog, our operational constraints. Building SQE forced us to articulate what we actually needed, which turned out to be far less than what Trino provides and far more than what any managed service offers.

Reason three: because the tools don't match. Trino's auth model is fundamentally incompatible with zero-trust – every query runs as a service account, and no amount of plugin configuration changes that. Spark is a cluster framework cosplaying as a query engine. DuckDB is brilliant but

single-node. DataFusion is a library, not a product. The gap between "library" and "product" is exactly the gap this book fills.

The honest addendum to reason three: we didn't fully understand that gap when we started. We thought it was smaller than it is. Bridging "DataFusion SessionContext" to "production query engine with distributed execution, auth, observability, and a benchmark suite" is a lot of bridge. The fifteen-day timeline makes it sound easy. It wasn't easy. It was fast. Those are different things.

316 Commits (Across All Branches) in 15 Days

The git log tells the real story:

```
Mar 14  -- Initial commit: architecture docs + core engine spec
Mar 14  -- All 6 crates scaffolded: core, auth, policy, sql, catalog, coordinator
Mar 14  -- First Flight SQL integration test passes
Mar 15  -- Write path: CTAS, INSERT INTO, DROP TABLE tested
Mar 15  -- Distributed execution: ScanTask protocol, worker, DistributedScanExec
Mar 16  -- Prometheus metrics, audit logging, Trino HTTP compat
Mar 17  -- Docker, Helm chart, mdBook docs, CLI
Mar 18  -- Lightweight test stack: Polaris in-memory + RustFS
Mar 19  -- 37 integration tests: views, joins, aggregations, EXPLAIN
Mar 21  -- Benchmark suite: TPC-H, SSB, TPC-DS, ClickBench queries
Mar 22  -- TPC-E, TPC-BB, TPC-C generators + benchmark runner
Mar 24  -- Weighted fragment scheduler, OTel trace propagation, heartbeats
Mar 25  -- Query history, result cache, system.runtime.* virtual tables
Mar 27  -- Distributed docker-compose: coordinator + 2 workers
Mar 28  -- Distributed execution wired into query pipeline
Mar 29  -- Concurrent client load test, schema projection fix
```

That's a functioning distributed SQL engine with auth, observability, six benchmark suites, and a load testing harness – in fifteen days. The pace is real. But the pace needs context, because the pace is the thing people focus on, and it's the wrong thing to focus on.

The pace was possible because of three factors, in order of importance:

1. **Spec-driven design.** Every feature started as a written specification before a line of Rust was generated. Architecture decisions, trait definitions, data flows, failure modes – all documented in OpenSpec format before implementation began. The AI never started from a blank page. It started from a design that a human had already thought through.

2. **AI-assisted implementation.** Claude Code wrote the vast majority of the Rust code. It scaffolded crates. It implemented traits. It wrote integration tests. It debugged gRPC stream issues by generating tracing instrumentation and reading DataFusion internals. The implementation speed was extraordinary.

3. **Rust and DataFusion.** The language and the library did enormous heavy lifting. Rust's type system catches entire categories of bugs at compile time. DataFusion provides a complete query

engine in a single `SessionContext`. We weren't building a query engine from zero. We were building the product layer on top of one.

Factor three is the one that gets underestimated. DataFusion is not scaffolding. It's a query engine. We built an engine *around* an engine – adding auth, catalog integration, distributed execution, observability, and a wire protocol. That's still a lot. But it's not "building a query engine from scratch" in the way that DataFusion itself was built from scratch.

The AI-Assisted Build: Honest Assessment

I want to be specific here, because the discourse around AI-assisted development oscillates between "AI writes all the code now" and "AI is just autocomplete." Neither is accurate. The truth is more interesting and more nuanced.

What the AI did well

Rust implementation. Given a trait definition and a description of the desired behaviour, the AI could produce correct, idiomatic Rust implementations consistently. The `PolicyEnforcer` trait in `sqe-policy` is twenty-six lines of code. The AI produced it in one pass from a spec paragraph. The `FlightSqlService` implementation in `sqe-coordinator` is hundreds of lines with async streams, error handling, and token extraction – and the AI got the borrow checker happy on the first try about 70% of the time.

Cross-crate refactoring. When we renamed from vendor-specific identifiers to generic ones (the OSS security hardening pass), the AI held all twelve crates in context and made consistent changes across every file. The `keycloak_url` to `oidc_url` rename touched config structs, TOML parsing, environment variable names, integration tests, documentation, and error messages. The AI found every instance. A human doing find-and-replace would have missed the test fixtures.

Test generation. Integration tests are tedious to write and critical to have. The AI generated 37 integration tests covering views, joins, aggregations, window functions, EXPLAIN output, and error cases. It knew what edge cases to test because it had just written the implementation. The tests weren't afterthoughts – they were generated as part of the implementation cycle.

Debugging. The gRPC stream accumulation bug in Chapter 14 is the clearest example. Fifty concurrent clients hung after about 30 queries. No error, no timeout, just silence. The AI generated step-by-step tracing instrumentation, wrapped every Flight call with timing logs, and identified that HTTP/2 stream IDs were accumulating on a reused connection. The fix was one line. The diagnosis that led to it was the AI's, working from symptoms I described.

Boilerplate elimination. Protobuf codec implementations, Prometheus metric registrations, TOML config struct definitions, CLI argument parsing – the kind of code that's necessary, correct, and soul-destroying to write by hand. The AI handled all of it without complaint, and without the subtle bugs that creep in when a human writes their fortieth `impl TryFrom<proto::Thing> for Thing` of the day.

What the AI did poorly

Architecture. The AI never once said "this is the wrong approach, let's step back." It implemented whatever was specified, even when the specification had a structural problem. The distributed execution design went through three iterations – not because the AI couldn't implement any of them, but because the first two had fundamental issues that the AI couldn't see.

The first design had the coordinator shipping entire `LogicalPlan` trees to workers. The AI implemented it. It worked. But it meant workers needed access to the full catalog to resolve table references in the plan. That broke the security model – workers shouldn't need catalog access, they should receive pre-resolved physical plan fragments with concrete file paths.

A human saw that problem. The AI didn't.

Security trade-offs. The AI suggested a service account model twice. Not because it was wrong in general – service accounts are the standard pattern for query engines. But for SQE, where bearer token passthrough is the entire point, a service account model is architecturally incompatible. The AI couldn't weigh "this is the standard pattern" against "this violates our core design constraint" because it didn't understand the constraint the way a human who'd sat through the twelve-minute security review did.

"Should we?" questions. The AI answers "how?" questions brilliantly. It does not answer "should we?" questions at all. Should we add Trino HTTP compatibility? Should we build six benchmark suites or just TPC-H? Should the policy engine support both OPA and Cedar, or pick one? These are product decisions, strategy decisions, resource allocation decisions. The AI will implement whichever one you choose. It won't help you choose.

Field report: During the distributed execution design, the AI produced three complete implementations in three days. Each one compiled, passed tests, and was architecturally wrong for different reasons. The third iteration worked because I spent a full day writing a design document that explicitly stated the trust boundary between coordinator and worker. The AI couldn't derive that boundary from the code. It needed a human to draw the line.

What surprised us

The AI's ability to hold twelve crates in context and reason about cross-cutting concerns. When we added OpenTelemetry trace propagation, the change touched `sqe-metrics` (the OTel setup), `sqe-coordinator` (injecting trace context into gRPC metadata), `sqe-worker` (extracting trace context from incoming requests), and `sqe-core` (the shared trace context type). The AI made all four changes in a single pass, and the traces connected end-to-end on the first run.

What didn't surprise us

The AI's inability to evaluate its own output critically. It generates code that compiles and passes the tests you asked for. It doesn't generate code that handles the test you *didn't* ask for. Every edge case we caught was caught by a human reading the code or by a test that a human specified. The AI is a brilliant implementer with no taste. Taste is the human's job.

The speed multiplier

I've been asked for a number. What's the multiplier? 5x? 10x? 50x?

The honest answer: it depends on what you're measuring. For raw implementation – turning a spec into compiled, tested Rust code – the multiplier is probably 10-20x. A human writing the `sqe-bench` benchmark framework from scratch would take weeks. The AI did it in hours.

But implementation is maybe 30% of building software. The rest is design, debugging integration issues, rethinking approaches that don't work, writing specs, reviewing generated code, and making the product decisions that determine whether the code solves the right problem.

For the total project, including all of that, the multiplier is more like 3-5x. Still remarkable. Still enough to build a distributed SQL engine in fifteen days. But not the "AI does everything" story that the timeline might suggest.

One Complete Cycle

The previous section describes the AI workflow in general terms. A reviewer rightly pointed out that the outputs are described but not the mechanics. So here's one full cycle – spec to prompt to AI output to review to revision to final code – for the `PolicyPlanRewriter` in `sqe-policy`.

The spec. An OpenSpec-style requirement, written by a human before any Rust existed:

> Row filters must be injected above TableScan nodes before the optimizer runs, so user predicates can push through row filters but not bypass them. Column masks must use expression wrapping (e.g., sha256 or redaction) that creates an expression boundary blocking predicate pushdown on raw values. Restricted columns must be removed from projections entirely – invisible, not errors.

Three sentences. Every design constraint that matters is in there. The ordering, the injection point, the predicate pushdown semantics, the PostgreSQL RLS-style invisibility model.

The prompt. What we actually sent to Claude Code:

> Implement the PolicyPlanRewriter in sqe-policy. It should implement the PolicyEnforcer trait. Use LogicalPlan::transform_down to walk the plan tree. When it encounters a TableScan, resolve the policy from the PolicyStore, then: (1) inject Filter nodes above the scan for row-level filters, (2) create a Projection that wraps masked columns in their mask expressions, (3) remove restricted columns from the projection. On policy resolution failure, inject a FALSE filter to deny all rows.

The AI output. The first implementation compiled. It passed the basic tests – row filters were injected, masks were applied, restricted columns vanished. But it applied masks and restrictions as independent steps, both building their projection from the schema of the node *after* filtering. When a table had both masks and restrictions, the restriction step rebuilt the projection from scratch, discarding the mask expressions the previous step had just created. Masked columns reverted to their raw values.

The review. A human reading the generated code spotted the problem in under a minute. The two projection steps were sequential but independent – each one read the schema and built a fresh expression list. The second one didn't know the first one had wrapped columns in mask expressions. The structure was correct for tables with only masks or only restrictions, but broke when both applied to the same table.

The revision. We restructured the prompt to be explicit about ordering and mutual exclusivity: masks and restrictions should be handled in a single projection pass. When masks exist, the projection should apply masks *and* filter out restricted columns in one step. When only restrictions exist (no masks), a simpler projection removes the columns. The key insight: masks must be applied while all columns still exist in the schema, and the restriction filter happens inside the same projection, not after it.

The final code. The revised implementation lives in `crates/sqe-policy/src/plan_rewriter.rs`. The mask projection at lines 131-158 iterates over all fields, filters out restricted columns, and applies mask expressions to the remaining ones – a single pass that handles both concerns. The `else if` branch at lines 160-181 handles the restrictions-only case. The ordering is: row filters first (above the TableScan), then masks-plus-restrictions in one projection, then done. One traversal, three security layers, correct by construction.

Total elapsed time for the full cycle: about forty minutes. The spec took fifteen. The first prompt and review took ten. The revision prompt and verification took fifteen. Forty minutes from written requirement to reviewed, tested, committed code.

That's the real mechanic. Not "AI writes code." Spec, prompt, output, catch the bug, revise the prompt, verify the fix. The AI is fast. The human is precise. The cycle is what produces code you can trust.

Decisions We'd Keep

Five decisions held up under pressure. They're the ones I'd make again without hesitation.

DataFusion as the foundation. The extensibility model is right. Custom `TableProvider`, custom `ExecutionPlan` nodes, custom optimizer rules – DataFusion gives you the hooks without forcing you to fork. We extended it for Iceberg catalog integration, distributed scan execution, and policy-based plan rewriting, and none of those extensions required modifying DataFusion's source. That's the mark of a well-designed library.

Rust. The compile-time guarantees saved us months of runtime debugging. The `Send + Sync` bounds on the `PolicyEnforcer` trait meant we couldn't accidentally create a policy enforcer that held non-thread-safe state – the compiler simply wouldn't let us. In a distributed system where the coordinator dispatches plan fragments to workers concurrently, this kind of compile-time safety isn't a nice-to-have. It's the difference between "it works" and "it works except under concurrent load on Tuesdays."

Bearer token passthrough. Chapter 4 covers this in depth. The security model that makes everything else possible. Every query runs as the authenticated user. No service account. No ambient credentials. When the security team asks who accessed the customer table, the answer is a name, not an application. We considered three approaches and picked the hardest one. Fifteen chapters later,

it's the decision I'm most confident about.

Iceberg + Polaris. Open table format, open catalog protocol, open storage. No vendor lock-in at any layer. When we wanted to add benchmark suites, we generated Parquet files and loaded them as Iceberg tables via `CTAS`. The format didn't fight us. The catalog didn't fight us. The storage didn't fight us. That's sovereignty in practice, not in marketing.

Single-node first, distributed later. The first working query ran on a single `SessionContext` with no coordinator, no workers, no fragment scheduler. We had correct results before we had distributed execution. That meant every distributed bug was a distribution bug, not a query bug. The debugging was orders of magnitude easier because we could always check: does this query return the right answer on a single node? If yes, the bug is in the distribution layer. If no, the bug is deeper. This binary diagnostic saved us days.

Decisions We'd Change

These are harder to write about. Not because they're embarrassing – they're not. They're the decisions that were reasonable at the time and turned out to cost more than expected.

The custom SQL parser wrapper. We wrapped `sqlparser-rs` to handle SQE-specific SQL extensions – `GRANT ... MASKED WITH`, `ROWS WHERE`, `SHOW EFFECTIVE POLICY`. The wrapper intercepts the parse output, detects these patterns, and converts them to custom AST nodes.

The problem: DataFusion already has extension points for custom SQL. The `Statement` enum has a `Statement::Extension` variant. The `UserDefinedLogicalNodeUnparser` trait exists for exactly this purpose. We could have used DataFusion's built-in extension mechanism instead of wrapping the parser.

We didn't, because at the time of writing the spec, we hadn't fully explored DataFusion's SQL extension surface. The spec was written before the implementation, and the spec assumed we'd need to intercept at the parser level. The AI implemented what was specified. By the time we realised DataFusion's native extensions would have been cleaner, the wrapper was working and tested.

I'd evaluate DataFusion's built-in extension points more thoroughly in a first pass. The wrapper works, but it's a maintenance surface we could have avoided.

Configuration as an afterthought. The first version of SQE had connection strings and port numbers as constants in the source code. "Works on my machine" was literally the design constraint. The TOML config came in version 0.3. The environment variable overlay came later. The full configuration surface with validation, defaults, and documentation came much later.

This is backwards. Configuration should be a first-class concern from day one, because configuration is the API you expose to operations teams. The people deploying your engine are not the people who built it. They need config keys that are self-explanatory, defaults that work, and validation that tells them what's wrong before the engine crashes at startup.

We got there eventually. Chapter 10 shows the finished config surface. But the retrofit was painful – every new config key required touching the struct definition, the TOML parser, the environment

variable mapping, the example config, and the documentation. If we'd set up that pipeline from the start, each new config key would have been a one-line addition to a derive macro.

Test infrastructure timing. Integration tests against a real Polaris + S3 stack should have been in CI from the first week. We didn't add them until after the first round of debugging distributed execution issues, when we realised that unit tests with mocked catalogs were passing while the real system was broken.

The lightweight test stack – Polaris in-memory mode plus RustFS (a Rust-native S3-compatible server) – was the fix. No AWS account needed. No Docker pulls from third-party registries. The entire test dependency runs in-process or in local containers. We should have built this on day two.

Dead end: mocking the Iceberg catalog for integration tests. We tried. We built mock implementations of the catalog traits that returned canned responses. The unit tests passed. Then we connected to a real Polaris instance and discovered that our mock didn't simulate Polaris's credential vending flow, which meant our S3 client configuration was wrong in production even though tests were green. Mock what you must, but test against the real thing as early as you can afford to.

The Abstraction That Has a Ceiling

The `ScanTask` protocol.

The `ScanTask` model works well for distributed scans. Each task carries file paths, projected columns, and filter predicates. The worker executes them independently – build a local `ExecutionPlan` to scan those specific files, apply the predicate, project the columns, and stream results back. It is clean, simple, and correct for the workload it was designed for.

But `ScanTask` only describes scans – it cannot represent local aggregation, sort, or join nodes above the scan. When we attempted distributed aggregation, we hit this wall: the coordinator could not express "scan these files, then group by region" as a single `ScanTask`. It needs a plan subtree. The coordinator ends up reassembling intermediate results and performing all the post-scan operations itself. That works for scan-only queries. It falls apart the moment you have a `GROUP BY` that should be partially evaluated on the worker.

The future architecture ships `PlanFragment` objects – serialised subtrees of the physical plan that include scan, filter, projection, and partial aggregation. Workers would execute the full subtree locally and return partial results to the coordinator for final aggregation. The `ScanTask` model was not premature – it was the right starting point. But it is a subset of what full distributed execution requires.

The AI implemented the `ScanTask` protocol without complaint. It didn't flag that the approach would limit worker-side computation. That was a human insight, born from staring at EXPLAIN output and realising that the coordinator was doing all the aggregation work while two workers sat idle after their scans completed.

Dead end: per-file ScanTask dispatch. Each worker got one file at a time. Clean separation of concerns. Terrible performance for aggregation queries, because all the compute-heavy work happened on the coordinator after the scans returned. The future fix is shipping plan fragments

(subtrees) instead of individual scan tasks – turning workers from "file readers" into "local query engines."

The Abstraction We Needed Earlier

A proper `QueryLifecycle` state machine.

For the first several iterations, query execution was a linear function: parse, plan, optimise, execute, return. Each step called the next. Error handling was scattered – some errors were caught in the planner, some in the executor, some in the Flight SQL handler. Cancellation was bolted on after the fact with `CancellationToken`. Timeouts were added separately. Query history logging was added separately again.

Each addition made the linear function more complex. By the time we had parse, plan, optimise, check-policy, maybe-distribute, execute-locally-or-dispatch-to-workers, stream-results, log-to-history, update-metrics, handle-cancellation, handle-timeout – the function was unreadable.

What we needed from the start: a state machine that models query lifecycle explicitly. States like `Parsed`, `Planned`, `Optimised`, `PolicyChecked`, `Dispatched`, `Executing`, `Streaming`, `Complete`, `Failed`, `Cancelled`. Transitions between states are explicit. Side effects (logging, metrics, history) attach to transitions, not to the middle of a function.

We didn't build this as a formal state machine. We refactored toward it – extracting the lifecycle into a struct with methods for each transition – but the linear-function heritage is still visible in the code. A state machine from day one would have made the cancellation, timeout, and history features trivial additions instead of careful retrofits.

What Rust Taught Us

Rust is not just a language choice. It's a design philosophy that infects your architecture.

The borrow checker is a distributed systems design tool. When you try to send a plan fragment to a worker, the compiler forces you to either clone the data or prove that nothing else is using it. This sounds like a nuisance. It's actually forcing you to think about data ownership across network boundaries. If you can't express the ownership model in Rust's type system, your distributed protocol probably has a race condition.

`Send + Sync` constraints catch concurrency bugs at compile time. The `PolicyEnforcer` trait requires `Send + Sync`. This means any implementation must be safe to share across threads and safe to send between threads. When we wrote the passthrough enforcer, this was trivially satisfied. When we eventually write the OPA enforcer with an HTTP client and a policy cache, the compiler will verify that the cache is thread-safe and the HTTP client is `Send`. In Go or Java, these bugs show up under load. In Rust, they show up at compile time.

The trait system enables genuine extensibility. DataFusion's `TableProvider`, `ExecutionPlan`, `OptimizerRule` – these traits are the extension surface. You implement them, register them, and DataFusion calls them. No dependency injection framework. No runtime reflection. No classpath

scanning. Just traits and their implementations. This is the right model for a pluggable system that needs to be fast.

Token efficiency for AI-assisted development. This is an underappreciated property. Rust code is dense. A Rust struct with derive macros, a trait implementation, and a few methods conveys more semantic information per token than the equivalent Java or Python code. When your coding agent has a context window and you're working across twelve crates, density matters. The AI could hold more of the codebase in context because Rust doesn't waste tokens on boilerplate.

The compile times are real. I mentioned this in Chapter 3, and I'm going to be more specific here because the retrospective earns specificity.

A clean build of all twelve SQE crates from scratch takes about eight minutes on an M3 MacBook Pro. Incremental builds after a single-file change in a leaf crate take 15-30 seconds. Incremental builds after changing a type in `sqe-core` – which everything depends on – take 2-4 minutes because half the dependency tree recompiles.

Multiply by fifty builds a day. That's between 12 minutes and 3 hours of waiting, depending on what you're changing. Over fifteen days, the cumulative wait time is measured in hours.

Strategies that helped: - `cargo check` instead of `cargo build` for type-checking without codegen – about 3x faster - Workspace splitting so crate boundaries limit recompilation blast radius - `split-debuginfo = "unpacked"` in the dev profile to skip the macOS dsymutil step - `debug = 1` (line tables only) instead of `debug = 2` (full debug info) - Never running `cargo build --release` during development

Strategies we should have adopted earlier: - Feature-gating optional subsystems (`distributed`, `trino-compat`, `bench`) so you only compile what you're working on - `sccache` for shared compilation cache across branches

Worth it? Yes. Every time. The hours spent waiting for the compiler are hours you don't spend debugging null pointer exceptions, data races, use-after-free, or the hundred other runtime bugs that Rust prevents. But budget for it. New Rust projects underestimate compile-time impact by 3-5x.

The Open-Source Goal

SQE is built to be open-sourced. That sentence appears in the Preface and it's repeated here because the retrospective is where I can explain what it actually cost.

Designing for open source is more expensive than designing for internal use. Every decision has a second audience. The config section that works for our Keycloak instance needs to work for someone else's Auth0 instance. The catalog integration that assumes Polaris needs to be pluggable for someone running Nessie or AWS Glue. The naming that references our internal infrastructure needs to be generic enough for strangers.

The OSS security hardening pass – Step 3 in our roadmap, fifty-one tasks – was entirely about this. Renaming `keycloak_url` to `oidc_url`. Replacing MinIO-specific language with generic S3. Removing internal hostnames from example configs. Adding TLS support that we didn't need internally but that

any production deployment would require. Adding rate limiting that our five-person team didn't need but that a public-facing deployment would.

Fifty-one tasks. Days of work. Zero new features. All of it was necessary if we wanted strangers to run this thing.

The pluggable architecture – `PolicyEnforcer` trait, `AuthProvider` trait (designed, not yet fully implemented), `CatalogBackend` trait (designed, not yet implemented) – exists because of the open-source goal. Our internal deployment uses Polaris, Keycloak, and a passthrough policy enforcer. But the traits are there so that someone else can plug in AWS Glue, Okta, and OPA without forking the codebase.

This design-for-pluggability approach is directly connected to the spec-driven development model. You can't design a good trait boundary if you don't know what implementations it needs to support. The specs forced us to think about multiple implementations before writing the first one. The `PolicyEnforcer` trait is twenty-six lines long. It took longer to specify than to implement. And it will support OPA, Cedar, and whatever policy engine comes next, without changing a single line.

Art of Agents: This is *Use of Spies* (Chapter 13) – the feedback loop. The retrospective closes the build cycle: specs, design, build, measure, learn. The learnings feed the next spec. The open-source goal means those learnings are public too. The feedback loop extends beyond the team to every engineer who reads this code or this book.

Where This Goes Next

Four trajectories, in order of impact.

Pluggable auth and catalogs. Steps 4 and 5 in the roadmap. The traits are defined. The config sections are stubbed. The implementation is next. Bearer token passthrough via OIDC remains the primary path, but API key auth, mTLS, and anonymous access are all in the design. Catalog backends for AWS Glue, Nessie, and storage-only (scan a directory for Iceberg metadata, no catalog server needed) are specified.

This is the work that makes SQE usable beyond our specific infrastructure. Right now, you need Polaris and an OIDC provider. After these steps, you need whatever you already have.

Upstream DataFusion improvements. DataFusion's release cadence is fast – we're on version 52. Each release brings optimizer improvements, new SQL functions, better memory management. SQE benefits from all of it for free, because we built on the library rather than forking it. The features we're watching: improved recursive CTE performance (for graph queries), better spill-to-disk under memory pressure, and native Iceberg predicate pushdown in DataFusion's optimizer.

Iceberg v3 features. Row-level deletes (Merge-on-Read) are blocked on iceberg-rust, with an estimated landing of Q3 2026. When that lands, `DELETE FROM` and `MERGE INTO` become possible. Branching and tagging – Iceberg's version control for tables – are also in the roadmap. These features turn SQE from a read-heavy analytical engine into a full read-write engine suitable for dbt workloads with incremental models.

The semantic AI layer. This is the ambitious one. RDF triple stores on Iceberg. Property graph queries compiled to DataFusion logical plans. Vector search via Lance datasets. AI agent interfaces via CLI, REST, and MCP. Each of these is a separate crate, fully additive, breaking nothing in the existing engine.

The semantic layer is designed but unbuilt. It represents the thesis that a query engine should be agent-native – discoverable, self-describing, and composable by AI agents that need to explore and query data without human mediation. Whether that thesis is right is a question for the next version of this book.

The Build-vs-Buy Honest Accounting

This is the section that technical leaders will skip to, and it's the section I most want to get right.

Total engineering time. Fifteen days of intense development. That's one person (me) plus an AI coding agent, working full days. Call it fifteen person-days of human effort, multiplied by whatever factor you want to assign to AI assistance. The AI amplified output by roughly 10x for implementation tasks and roughly zero for design tasks. A reasonable estimate of equivalent human-only effort: two to three months for a senior Rust engineer, six months or more for a team less familiar with DataFusion and Iceberg.

What you get. A distributed SQL engine with OIDC auth, bearer token passthrough, Iceberg catalog integration, Arrow Flight SQL and Trino HTTP wire protocols, observability (OpenTelemetry + Prometheus), a CLI, a benchmark suite covering six industry-standard benchmarks, a Docker multi-stage build, and 37 integration tests. Twelve crates, clear boundaries, documented architecture.

What you don't get. Enterprise-grade maturity. Battle-tested failure recovery under real production load. A community of contributors fixing bugs you haven't hit yet. The ten years of optimisation that Trino's join algorithms represent. Support contracts. Compliance certifications.

Operational cost. One Helm chart. One coordinator pod. One or more worker pods. The coordinator uses about 512MB of memory at idle and scales with concurrent query count. Compare this to Trino's deployment: coordinator, multiple workers, a discovery service, possibly a separate metadata store, Ranger or Sentry for security, and the operational expertise to tune all of it.

The capability gap. Trino handles complex multi-way joins better. Trino's exchange operators are more mature. Trino's ecosystem of connectors (Postgres, MySQL, Kafka, Elasticsearch) is vast and battle-tested. SQE connects to Iceberg via Polaris and that's it. If your workload is "analytical queries over Iceberg tables with strict auth," SQE matches or exceeds Trino. If your workload is "query everything from everywhere," Trino wins.

The capability gain. SQE does things that nothing else can. Every query runs as the authenticated user – not a service account. Bearer tokens pass through to storage. The audit trail shows humans, not applications. The policy engine rewrites query plans before optimisation, making row filters and column masks invisible to the user and impossible to bypass. This isn't a feature gap that Trino could close with a plugin. It's an architectural property that requires building the auth model into the engine's foundation.

When to build. When you have a non-negotiable architectural constraint that existing tools cannot satisfy. For us, that constraint was bearer token passthrough – the security model where every I/O operation traces back to a human identity. No existing query engine supports this because it requires building the auth model into the plan execution path, not bolting it on top.

When not to build. When the constraint is negotiable after all. If the security team will accept application-level logging with a service account, use Trino. If the workload fits in a single node, use DuckDB. If you need connectors to fifteen different data sources, use Spark. Building a query engine is the right choice when – and only when – the architectural constraint is genuinely non-negotiable and no existing tool satisfies it.

The honest truth: most constraints are more negotiable than engineers believe. We build because we can, and we justify it with constraints. The discipline is knowing when the constraint is real and when it's an excuse to build something interesting.

For SQE, the constraint was real. The security team's twelve-minute rejection of the service account model was the proof. But I'd be lying if I said the engineering drive – the desire to understand how query engines actually work, to build one, to take it apart and put it back together – wasn't a factor.

It was. Engineers build things. That's the drive. The trick is building things that matter.

The Book That Found Bugs

Something unexpected happened while writing this book: it found bugs.

Not in the prose. In the code.

Writing a chapter forces you to explain what the code does, and explaining what code does is the fastest way to discover that the code doesn't do what you think. Chapter 8 described the `PolicyPlanRewriter` injecting row filters and column masks. When we went to reference the sha256 masking function, we discovered DataFusion doesn't ship one. That's not a bug in the usual sense, but it's a gap that would have bitten the first user who tried hash-based column masking. Writing the chapter forced us to implement the UDF.

Chapter 7 described the Iceberg commit mechanism and referenced an `x-iceberg-update-sequence-number` HTTP header for optimistic concurrency. When the reviewers checked the Iceberg REST specification, the actual mechanism is `assert-current-snapshot-id` in the request body, not an HTTP header. The code was correct – iceberg-rust handles this internally – but the mental model was wrong. Writing it down exposed the gap between what we thought was happening and what was actually happening.

Chapter 3 described DataFusion's pull model as "nothing is computed until someone reads from the output." A reviewer pointed out that `HashJoinExec` eagerly builds its hash table. The code was fine – we weren't building incorrect joins. But our understanding of the execution model had a blind spot that would have eventually caused a memory accounting bug.

The `block_in_place` usage in `SchemaProvider::table_names()` had been there since the first week. Nobody flagged it. Writing the chapter, a reviewer identified it as a known anti-pattern that could

deadlock under certain executor configurations. We added safety documentation and put an upstream proposal on the backlog.

In total, writing the book produced 12 code fixes and 6 remaining TODO items. Some were genuine bugs. Some were missing features. Some were documentation gaps that would have confused contributors. All were invisible until someone had to explain, in writing, what the code was supposed to do.

Field report: The book review process caught factual errors about Iceberg's partition evolution ("unique to Iceberg" – it's not anymore, Delta Lake has liquid clustering), PyIceberg's GIL behavior (I/O releases the GIL, only scan planning is bound), and Polaris's access control model (it does have RBAC, not "no opinions"). Every claim in a technical book is a claim that can be verified. The reviewers verified them, and several were wrong.

This is the strongest argument for writing about your code, not just writing code. A codebase that compiles and passes tests can still contain incorrect assumptions, missing features, and misleading mental models. Writing forces you to make those assumptions explicit. Reviewers force you to defend them. The code improves even though nobody opened a pull request.

The Hardest Lesson

The hardest lesson from this project isn't about Rust or DataFusion or distributed systems. It's about the relationship between a human and an AI coding agent.

The AI made the pace possible. Without it, this project would have taken months, not weeks. The implementation quality is high – the code is idiomatic, the tests are thorough, the error handling is consistent across twelve crates.

But the AI didn't make the decisions. Every architectural turn – bearer passthrough over service accounts, plan fragments over scan tasks, single-node-first development, the trait boundaries for pluggability – was a human decision. Some of those decisions were wrong the first time and had to be revised. The AI implemented the wrong version just as competently as it implemented the right one. It didn't know the difference.

The model that works is: **human architect, AI builder.** The human decides what to build and why. The AI decides how to build it and does the work. The human reviews every big turn. The AI executes between the turns.

This sounds obvious written down. In practice, the temptation is to let the AI lead. It's fast. It's confident. It produces working code. The code compiles and the tests pass. But "compiles and tests pass" is not the same as "solves the right problem." The AI optimises for the local objective – make this function work, make this test pass. The human optimises for the global objective – does this architecture serve the security model, the operational constraints, the open-source goal.

Fifteen days. Three hundred and sixteen commits. Twelve crates. One principal engineer who made the decisions. One AI agent that did the work.

That ratio – one human, one AI, clear roles – is the thing I'd keep above all else.

AI Logbook: This chapter is the one the AI could not have written alone. The retrospective required evaluating which decisions were right, which were wrong, and why — judgments that depend on context the AI never had. The AI drafted the prose from the human's structural outline and specific commit references. The honest assessment of AI limitations (three wrong distributed execution designs, inability to evaluate its own output, the security trade-off it missed twice) was the human's observation; the AI would have reported its own work as successful at every stage.

Sovereignty principle: Sovereignty applies to your development process too. The AI is a tool, not a decision-maker. You own the architecture. You own the security model. You own the trade-offs. The AI accelerates the implementation of decisions you've already made. The moment you let it make the decisions, you've outsourced your sovereignty to a model that doesn't understand your constraints.

Field report: This book was also written with AI assistance. The same model applies: I decided what each chapter needed to say, wrote the structural outline, identified the specific commits and code examples to reference. The AI did the prose drafting. I edited every paragraph. The voice is mine. The pace was the AI's. The ratio holds.

Epilogue

I started this book with a question from the security team: "Who accessed the customer table last Tuesday?"

Seventeen chapters later, we can answer it. The query ran as Alice. S3 saw Alice. CloudTrail shows Alice. The audit log shows Alice's query, Alice's tables, Alice's execution time. No service account. No shared secret. No ambiguity.

That was the problem. It's solved.

But the real thing I built isn't a query engine. It's a set of opinions about how data infrastructure should work — opinions that happen to compile.

The opinion that authentication is an architectural constraint, not a feature. The opinion that policy enforcement belongs in the query plan, not around it. The opinion that a catalog should be a standard interface, not a vendor's moat. The opinion that distribution should be a conscious decision, not a default. The opinion that an engine you can't configure is an engine you can't trust.

These opinions could have been implemented in Java. They could have been implemented in Go. They ended up in Rust because Rust's type system makes wrong opinions hard to compile. The borrow checker doesn't care about your architecture, but it catches the moment your architecture stops making sense — when a shared reference crosses a thread boundary it shouldn't, when a credential outlives the session it belongs to, when a plan node gets moved after something else borrowed it.

I used an AI coding agent to build most of this in fifteen days. The agent wrote the Rust. I wrote the opinions. The agent is very good at Rust. It has no opinions of its own.

That division of labour is the real discovery. Not that AI can write code — everyone knows that. The discovery is that the things AI can't do are exactly the things that make a project worth doing. Choosing bearer passthrough over service accounts. Deciding to fork Ballista's model instead of using it as-is. Knowing when to stop distributing and let a single node handle it. Recognising that the timestamp precision bug was in the display formatter, not the data.

The puzzle is still the point. It has been since I was a kid taking things apart. The tools have changed — from a screwdriver to a compiler to an AI agent that can hold twelve crates in its head simultaneously. But the drive is the same: understand how it works, make it better, move on to the next problem.

The next problem is already forming. The engine queries tables. But the questions people actually ask aren't about tables — they're about relationships, patterns, meanings. "Which customers are at risk?"

is not a SQL query. It's a question that requires context no table schema captures. The semantic layer — property graphs, vector search, agent interfaces — is where the engine goes next.

But that's a different book.

This one started with a Trino cluster that worked. Mostly. It ends with an engine that answers the security team's question. Fully.

If you've read this far, you now know how to build a query engine. More importantly, you know when not to.

Build what matters. Leave the rest.

Amsterdam, 2026