O'REILLY®



Advanced SQL

Implementing Modern Data Solutions and ML Applications

Rui Machado, Hélder Russa
& Pedro Esmeriz

Advanced SQL

Implementing Modern Data Solutions and ML Applications

Rui Machado, Hélder Russa, and Pedro Esmeriz



Advanced SQL

by Rui Machado, Hélder Russa, and Pedro Esmeriz

Copyright © 2026 Rui Pedro Machado, Hélder Russa, and Pedro Esmeriz. All rights reserved.

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Andy Kwan

Development Editor: Corbin Collins

Production Editor: Beth Kelly

Copyeditor: Piper Content Partners

Proofreader: Tim Stewart

Indexer: Sue Klefstad

Cover Designer: Susan Brown

Cover Illustrator: Monica Kamsvaag

Interior Designer: David Futato

Interior Illustrator: Kate Dullea

July 2026: First Edition

Revision History for the First Edition

- 2026-07-01: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9798341627482* for release details.

The O’Reilly logo is a registered trademark of O’Reilly Media, Inc. *Advanced SQL*, the cover image, and related trade dress are trademarks of O’Reilly Media, Inc.

The views expressed in this work are those of the authors and do not represent the publisher’s views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

979-8-341-62748-2

[LSI]

Preface

There is a moment, and perhaps you have already lived it, when you ask a large language model to write a SQL query, and it does, without hesitation and with better quality than you were expecting. A window function, a recursive common table expression (CTE), a lateral join that would have taken you twenty minutes of careful thinking: done in seconds. For many practitioners, this moment produces a peculiar mix of relief and unease.

If the machine can do this, you may ask, what exactly am I here for? We wrote much of this book before that question became urgent. But the world shifted under our feet as we wrote, and we owe you an honest account of what that shift means, explaining why it matters more than ever.

Tools like Claude and ChatGPT have done something remarkable: they have made advanced SQL accessible to anyone willing to describe what they need in plain language. The syntax barrier, for so long the wall separating those who could build data systems from those who merely used them, is coming down. Window functions no longer require a ritual of Stack Overflow searches. Complex aggregations no longer demand a specialist. A domain expert who understands the business can now, more than ever before, reach directly into the data.

This is genuinely good and is worth celebrating. But capability without understanding is dangerous. A large language model (LLM) can generate a beautifully structured query that computes the wrong thing, silently, plausibly, and at scale. It can produce a partition strategy that works on your development dataset and destroys performance in production. It can write ML feature pipelines in SQL that look correct and encode subtle data leakage that poisons every model trained on them.

The query runs, the numbers appear, and decisions get made. When we set out to write this book, we kept returning to a single word: trust. Not trust in a narrow technical sense, though constraints, tests, and data contracts matter

enormously, but trust in a human sense. The confidence a chief revenue officer feels when quoting a number in a board meeting. The certainty a data scientist has that the features fed into her model reflect reality rather than an artifact of how the extract, transform, and load (ETL) was written. The moment when data stops being a source of anxiety and becomes a genuine foundation for action. That kind of trust is not a property of the code; it is a property of the practitioner who wrote it.

Every decision in advanced SQL work carries judgment: how you handle NULLs at the boundary of a join, how you define the grain before aggregating, how you account for late-arriving events in a time-series computation, and whether a machine learning feature should be calculated before or after a specific business event. These decisions require context that lives outside any prompt, the organizational history, the known quirks of a source system, the argument last quarter about how refunds should be treated, or the implicit definition of "conversion" that differs between teams.

LLMs are extraordinary at execution. But they are, by their nature, unable to handle context efficiently. You are the one who holds it. That makes you, in the age of generative AI, more important, not less.

Paradoxically, the easier it becomes to generate code, the more critical it becomes to understand what that code does. When writing SQL was slow and deliberate, errors were caught in the effort. Today, when fifty lines can be generated in seconds, the bottleneck has shifted from production to verification. And you cannot verify what you do not understand.

This book covers advanced SQL not as a historical artifact but as the living language of modern data systems, the substrate on which analytical platforms, real-time pipelines, and machine learning applications are built. When you understand how window functions track state across ordered partitions, you will know when the generated code handles ties incorrectly. When you understand the mechanics of incremental processing, you will catch the moment an AI-suggested approach duplicates records under late data. When you understand how SQL integrates with ML workflows,

feature stores, training pipelines, and model scoring, you will identify when a well-intentioned query has introduced a leak between training and serving that will surface only when the model is in production.

Understanding the system is not an alternative to using AI assistance. It is what makes AI assistance safe to use.

The chapters ahead move deliberately from the foundations of advanced query mechanics through the architecture of modern data solutions and into the territory where SQL meets machine learning. Each section builds on the previous one, and we have been intentional about showing you not just how but why: the reasoning behind every pattern and the trade-offs it makes.

We encourage you to read with an LLM by your side. Generate examples, ask for variations, push the model to refactor and then interrogate its choices. Use it as a tireless collaborator, one that is fast, knowledgeable, and completely dependent on you to set direction, define standards, and catch mistakes. That is the appropriate relationship between a skilled practitioner and an AI tool. Not replacement. Not fear. Partnership, with your judgment at the center.

This book does not aim to make you faster at writing SQL. Rather, its goal is to make you the kind of practitioner who can be trusted: with complex systems, with data that drives consequential decisions, with the ML applications that increasingly shape what organizations know about the world. In a moment when code can be written almost on demand, that trust is the rarest and the most valuable thing you can offer. Build systems worth trusting, guide the machines well. That has always been the work, and it still is.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **WARNING**
>
> This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/helder-russa/advanced-sql.git*.

If you have a technical question or a problem using the code examples, please send email to *support@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require

permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Advanced SQL* by Rui Machado, Hélder Russa, and Pedro Esmeriz (O'Reilly). Copyright 2026 Rui Machado, Hélder Russa, and Pedro Esmeriz, 979-8-341-62748-2."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

> **NOTE**
>
> For more than 40 years, O'Reilly Media has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O’Reilly Media, Inc.

141 Stony Circle, Suite 195

Santa Rosa, CA 95401

800-889-8969 (in the United States or Canada)

707-827-7019 (international or local)

707-829-0104 (fax)

support@oreilly.com

https://oreilly.com/about/contact.html

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/advanced-sql*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Acknowledgments

All of the authors would like to thank our technical reviewers Sami Ahlroos, Robert de Graaf, and Sudhanshu Badoni. We are grateful for their time and assistance in ensuring technical accuracy.

Rui Machado

Books are never written alone. Behind every page and every example are people who make the work possible, not by writing a single word but by giving something far more precious: time, patience, and unconditional support.

To my wife, Ana Ferreira, you are my constant. Through late nights and distracted weekends, you never gave up on this. This book exists because you believed it should.

To my daughters, Maria Inês and Margarida, you are the reason any of this matters. You may not yet understand what databases are, or why someone would spend hours thinking about how data moves through a system, but I hope that one day you look at this book and see proof that curiosity and a love for building things are always worth it. I wrote this for you, even when it didn't look that way.

And then there are the people who give you the most valuable thing of all, time. A special thanks to my mother and father for the countless nursery trips. You may not have written a single line, but this book would not exist without you.

Writing a technical book is an act of optimism, a belief that knowledge shared is knowledge multiplied.

Hélder Russa

Writing a book is a journey of self-knowledge. There are moments of pure enthusiasm, and others of real frustration. Writing a book demands consistency, patience, and resilience. I would not have gone through it the same way without you, my wife, Patrícia. Your love, your support, your understanding, and ultimately your ability to lift me up in the toughest moments were my ground when I needed it the most. For that, I am deeply grateful.

To my son, Tomás. You are the purest form of love and joy I know. As I write this, you are still a baby, with those toothless smiles that can light up

an entire room. You may not realize it yet, but you are already one of the strongest forces that keep me going. I love you with all my heart.

To my parents, thank you for every sacrifice you made so I could keep learning, growing, and building my own path. The foundations you gave me made all of this possible. This book is also yours.

Pedro Esmeriz

If someone had told me a few years ago that I would coauthor a book, I would have laughed. And yet, here we are. Writing a book for the first time is equal parts thrilling and terrifying, and I could not have done it without the people who kept me grounded along the way.

To my partner in crime, Raquel—thank you for never letting this be just my project. From the very beginning, you made it ours. Every late night, every "just five more minutes" that turned into two hours, you were there, not waiting on the sidelines but cheering right beside me. This book has my name on it, but it carries your fingerprints all over it.

To my daughter, Júlia—or Ju, as we already call you. You have no idea what a Transformer is—and no, I don't mean the robots—but honestly, I hope you take your time getting there. What I must confess, though, is that you were named after a programming language. One day I stumbled upon Julia Lang and thought, "That's actually a beautiful name." So yes, before you even existed, tech was already shaping your life. I'm sorry. I love you endlessly.

To Rui and Hélder—you are far more than coauthors. You are true friends. Working with people who are positive, inspiring, supportive, and genuinely caring made this process not just bearable but joyful. Thank you for bringing out the best in this book and in me.

One book down, on to the next quest. Onwards!

Part I. The Modern SQL

Chapter 1. Evolution of SQL

SQL has had a remarkable impact in the world of technology as it has continuously adapted and evolved for decades. It has shaped the way we model and query data by introducing a declarative approach, in which we specify *what* data we want, not *how* we retrieve it.

From the first mainframe databases to modern cloud data warehouses, SQL has remained a constant. It combines solid mathematical foundations with practical developments and has evolved from a simple query language to the core of today's data-driven industry. Its past, present, and future are closely connected with the history of data itself, and mastering this language remains essential for any data professional.

The evolution of SQL from its beginnings in the 1970s to the SQL:2023 standard shows how a technology can evolve for a long period without losing its core philosophy. Over the years, it has standardized access to data, enabled countless applications and analytics, and spawned an ecosystem of databases that underpins virtually every industry.

With consistency came innovation as SQL always embraced new ideas such as object-oriented data, online analytical processing, XML/JSON data formats, and geographical data. Few languages remain as relevant after half a century, and the combination of theoretical rigor and practical utility has secured SQL's place at the center of data processing.

Looking to the future, SQL shows no signs of stagnating. Database communities and standards bodies are actively exploring new frontiers such as integrating machine learning and vector search capabilities directly into SQL, expanding analytics capabilities for time series and pattern matching, and making SQL more accessible. For example, the latest SQL:2023 standard introduces formal support for graph queries, and better support for polymorphic storage engines that include row-based, column-based, or document stores is being discussed. Given this track record, it is likely that

these innovations will be adopted in due course to ensure that SQL remains the ideal language for an ever-growing number of data tasks.

Historical Overview: From Relational to Modern SQL

Understanding the future of something often involves learning its past, to envision an informed set of next steps and evolutions. With this in mind, we recall that the SQL language has its roots in the early 1970s with Dr. Edgar F. Codd's relational model.

Inspired by Dr. Codd's relational model, IBM researchers Donald Chamberlin and Raymond Boyce developed Structured English Query Language (SEQUEL), later renamed SQL. Commercial products like SQL/DS (1981) and Oracle V2 (1979) helped establish it as a standard data access language.

As SQL gained popularity, vendors implemented their own dialects with slightly different syntax and capabilities.

Standard SQL-86: Laying the Foundations

In 1986, the American National Standards Institute (ANSI) and International Organization for Standardization (ISO) published the first SQL standard, SQL-86, which defined the core of the language. It was relatively simple and covered basic SQL operations and data types. With it came core capabilities such as table creation and modification, basic filtering and aggregation, and support for primitive types such as `INTEGER`, `SMALLINT`, and `VARCHAR`.

SQL-86 created a common baseline for SQL but was limited by modern standards. Notably, it lacked many features we take for granted today. For example, no `JOIN` operations, combining tables had to be done via Cartesian products and filters, no support for foreign keys or explicit constraints, no

views, and only minimal transaction control. SQL-86 was a basic toolkit reflecting the state of relational databases at that time.

Standard SQL-89: Adding Integrity and Security

Building on the initial standard, SQL-89 introduced essential features for data integrity and access control, including the definition of `PRIMARY KEY` constraints and commands like `GRANT` and `REVOKE` to manage user permissions. These additions strengthened SQL as a consistent and portable query language across systems.

However these features were not enough for the demanding industry that was asking for ways to manipulate data efficiently and at scale, thus allowing for more use cases to be developed in SQL. That's when the real innovation started.

Standard SQL-92: Adding Modeling Capabilities

Officially released in 1992, SQL-92 significantly expanded SQL capabilities and is often referred to as the first modern SQL. SQL-92, sometimes called SQL2, introduced a variety of features that greatly improved the language's expressive power for data modeling and querying.

This standard marked a major step forward by standardizing `JOIN` syntax, which made expressing multi-table relationships more intuitive and portable, and by formalizing subqueries and set operations like `UNION` and `INTERSECT`, which dramatically expanded SQL's expressiveness. It also introduced new data types such as `DATE`, `TIME`, and `TIMESTAMP`. It also reinforced referential integrity with `FOREIGN KEY` and `CHECK` constraints and defined transaction control with `COMMIT`, `ROLLBACK`, and isolation levels such as `READ COMMITTED` and `SERIALIZABLE`.

With SQL-92, SQL became much more powerful, allowing developers and analysts to perform complex joins and subqueries in one go instead of multiple passes. Databases could enforce more data quality rules internally with foreign keys and checks, and they could handle dates and times

properly. SQL could now express more sophisticated queries and maintain consistency, paving the way for it to dominate data processing in the enterprise.

SQL-92 also laid the foundation for robust, enterprise-wide database applications. This standard has been widely adopted, and many core features of SQL-92 remain fundamental to all SQL dialects today.

Standard SQL:1999—Handling Industry Needs

In the late 1990s, databases became increasingly complex and new use cases were added, to which the SQL standard responded accordingly. SQL:1999, often referred to as SQL3, was an important further evolution that extended SQL beyond purely relational operations and incorporated concepts from the object-oriented and analytical areas.

This was a turning point that introduced object-relational modeling, `TRIGGERS`, and `STORED PROCEDURES`, allowing databases to encapsulate business rules and reactive logic. Most notably, it added `WITH` and `WITH RECURSIVE` and *common table expressions* (*CTEs*), which are a first-class SQL construct that transformed how complex transformation and preparation logic is written; it also directly influenced modern open source tools like the data build tool (dbt), which rely on CTEs for modular, maintainable data pipelines.

With SQL:1999, the language became far more powerful and expressive. It effectively turned SQL into a richer query and programming language within the database, reducing the need for external logic for many use cases.

From the SQL:2003 Standard to SQL:2008

In the 2000s, the computing landscape saw the rise of XML for data exchange and a push for better database programmability. The SQL standards of the time focused on integrating these trends and adding some long-awaited conveniences.

SQL:2003 introduced major additions including SQL/XML, enabling native storage and querying of XML data, while standardizing `SEQUENCE` generators and `IDENTITY` columns for portable key generation. It also significantly extended window functions—adding `RANK()`, `DENSE_RANK()`, and custom window frames—which transformed SQL into a robust engine for in-database analytics. Minor revisions followed: SQL:2006 improved XML integration, and SQL:2008 added practical enhancements like `INSTEAD OF` triggers for view updates and `TRUNCATE TABLE` for efficient batch deletes in data warehouses.

By 2008, SQL had transformed from a simple query language for basic operations into a robust platform for complex data processing. With XML and later JSON integration, window functions for analytics, and other improvements like sequences and improved triggers, SQL could handle a much wider array of tasks within the database.

From the SQL:2011 Standard to SQL:2016

With the further development of data systems, new requirements for handling time-based data and semi-structured formats such as JSON have emerged. The SQL:2011 and SQL:2016 standards address these areas and ensure that SQL remains equipped for modern use cases:

- SQL:2011 introduced temporal tables via `FOR SYSTEM_TIME`, allowing databases to track and query historical versions of data—crucial for auditing, financial records, and time-based analysis. This unified what had previously been vendor-specific features into a standard approach for handling time-dependent data.
- SQL:2016 brought native support for JSON, including a JSON data type and functions like `JSON_VALUE`, `JSON_QUERY`, and `JSON_OBJECT`, enabling powerful document-style queries directly in SQL. This bridged the gap between relational and semi-structured data, making SQL a first-class option for modern APIs and analytics.

This standardization meant that SQL could natively process JSON documents. A user could store JSON in a column and query fields in that JSON using SQL syntax. This was an important step because it closed the gap between relational storage and NoSQL-style document storage: relational databases could now conveniently process hybrid data.

By 2016, SQL had proven remarkably adaptable. It could manage relational data, time-dimensional historical data, and schema-free JSON data all within one framework. Many databases had already added JSON features on their own, and the standard unified these trends. The combination of temporal and JSON features showed that SQL was responding to modern needs like advanced auditing or semi-structured data without abandoning its relational base foundation.

The Graph Revolution with SQL:2023

The latest standard, SQL:2023, demonstrates the continuous evolution of SQL to meet today's requirements. Published in 2023, the standard introduces features that push SQL into new areas such as graph processing and further improve its usability and compatibility with modern data formats.

SQL:2023 expands into new territory with native property graph queries, allowing users to express patterns like paths and neighborhoods directly on relational data, supporting use cases such as fraud detection and social network analysis. It also deepens JSON integration with a true JSON data type, improved functions like `JSON_SERIALIZE` and `JSON_SCALAR`, and better indexing potential. Alongside these, syntax simplifications and refinements, such as clearer handling of NULLs in constraints, enhance developer productivity and eliminate long-standing inconsistencies.

This 2023 standard shows that even after decades, SQL continues to adapt. By incorporating graph query support and deepening JSON integration, the standard recognizes that modern data platforms often blend paradigms, namely relational, document (NoSQL), or graph. SQL is positioning itself as a universal query interface that can handle diverse data models.

These innovations ensure SQL remains relevant as data evolves—developers can use familiar SQL syntax for new kinds of queries. Importantly, learning SQL's history is also a lesson in how computer science evolves, namely, the language has continually adapted to new technology trends while maintaining backward compatibility and the core relational principles.

Transition to Multipurpose SQL

Over the past decade, SQL has evolved into an indispensable multipurpose language for modern data management. Today's SQL engines seamlessly process structured, semi-structured, and even unstructured data, integrate with machine learning (ML) and large language models (LLMs), and serve as a unified interface for various computing paradigms. In the following sections, we explore how SQL expanded into these new areas.

Semi-Structured Data Handling in SQL

Modern SQL databases can natively handle semi-structured data formats like JSON and XML, which are ubiquitous in web services and APIs. They offer sophisticated functions for parsing, querying, and generating JSON directly within SQL.

By supporting JSON, SQL databases allow the combination of relational and document-style data in one system. For example, SQL implementations now include powerful JSON query functions such as `JSON_VALUE`, `JSON_QUERY`, and even JSON table constructs to project JSON fields as rows. Example 1-1 shows the advanced use of JSON.

Example 1-1. Parsing JSON in SQL

```
SELECT
  account_id,
  JSON_VALUE(profile_data, '$.name') AS name,
  JSON_QUERY(profile_data, '$.preferences') AS prefs
FROM users
WHERE JSON_EXISTS(profile_data, '$.preferences[?(@.category=="music")]');
```

In this example, a user's table has a JSON column `profile_data`. The query uses JSON functions to extract a user's name and preferences, filtering only those users whose preferences include a music category. Such functions eliminate the need for external JSON parsing since the database itself can filter and project JSON content. These functions simplify the interaction with semi-structured data and avoid additional ETL steps and external processing.

Similar to JSON, SQL databases provide robust XML processing capabilities that enable the storage, transformation, and retrieval of XML documents, as shown in Example 1-2.

Example 1-2. Parsing XML in SQL

```
SELECT id,
 EXTRACTVALUE(data, '/customer/name') AS customer_name
FROM orders
WHERE EXISTSNODE(data, '/customer[@status="active"]') = 1;
```

Here the query pulls customer names from an XML document and filters for active customers, all within SQL. This integration of semi-structured data means fewer ETL steps and less external processing—SQL can directly combine relational data with JSON or XML data in queries. This capability has been critical in allowing traditional SQL systems to handle data formats popular in modern applications. The resulting flexibility enables the integration of traditional relational databases with web-based, XML-centric systems.

Temporal Data Management

Managing data changes over time has become increasingly important for auditing and compliance. SQL now includes features for tracking and querying historical data directly. A common approach is the system-versioned temporal table introduced in SQL:2011.

In a system-versioned table, the database automatically preserves old versions of rows, often in a history table with `start_time` and `end_time` timestamps for each version. This allows queries that retrieve a consistent

snapshot of the data as of a past date. For example, one can query "what did this row look like on 2025-01-01?" with a simple time-travel clause instead of maintaining manual history tables. Example 1-3 shows how to implement such behavior.

The temporal capabilities of SQL enable effective handling of historical and changing datasets, greatly enhancing compliance, audit, and analytical insight capabilities.

Example 1-3. A temporal query

```
SELECT *
FROM employees
FOR SYSTEM_TIME AS OF TIMESTAMP '2025-01-01';
```

This SQL, supported in databases like SQL Server, Oracle, etc. with slight variations, would return the contents of the `employees` table as of the start of 2025. Some cloud warehouses like Snowflake offer similar "time travel" query features (often up to a fixed retention window, e.g., 90 days), and BigQuery allows querying earlier versions of data within a 7-day window by specifying a snapshot time.

Temporal querying capabilities make compliance and debugging much easier. If an incorrect data load happens, an engineer can quickly compare yesterday's data to today's to isolate what changed. Auditors can retrieve the exact state of records at quarter-end or any point in time. By keeping a complete history of data changes and exposing them through simple SQL syntax, databases enable treating data as an evolving ledger rather than just the latest state. This has solidified SQL's role not only in managing current data but also in managing its evolution over time.

Extended Analytical and OLAP Functions

SQL today has robust analytical capabilities, largely thanks to window functions (also known as online analytical processing, or OLAP, functions). These allow complex calculations across sets of rows without collapsing results, which is invaluable for business intelligence tasks.

Unlike regular aggregate functions that group rows, window functions produce a value for each row while looking at a "window" of other rows. For example, window functions can compute running totals, moving averages, rankings, and period-over-period comparisons all within a single query, as shown in Example 1-4, eliminating the need for external processing.

Example 1-4. An analytical query

```
SELECT
  department,
  hire_date,
  salary,
  AVG(salary) OVER (
      PARTITION BY department
      ORDER BY hire_date
      ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
  ) AS cumulative_avg_salary
FROM employee;
```

This query adds a column `cumulative_avg_salary` that, for each employee, shows the average salary of all employees hired up to that date in the same department. It demonstrates a window function (`AVG… OVER`) partitioned by department and ordered by `hire_date`, with a frame from the start through the current row. Every row remains in the result along with its computed running average. This kind of analysis would be much harder without window functions—one might resort to self joins or application-side logic.

The introduction of window functions (standardized in SQL:2003) significantly improved the efficiency of SQL reporting and analytics. They enable patterns like year-over-year comparisons using `LAG()` (to get last year's value in the same partition), ranking top performers within categories, and gap-and-island detection in time series. All major SQL databases now support window functions, making SQL a powerful tool for analytical computations directly in the database and reducing the need to export data to spreadsheets or specialized tools for these tasks.

Navigating Graph Structures in SQL

Traditionally, graph databases (with nodes and edges) were separate from SQL databases. However, newer SQL standards and implementations are bringing graph query support into SQL itself. The idea is that users can model data as nodes and edges in their SQL database and query relationships (paths, neighborhoods, subgraphs) using familiar SQL-like syntax, instead of having to use a specialized graph database like Neo4j for every graph problem.

A major development in this area is the SQL:2023 Property Graph Queries (SQL/PGQ) extension. It defines how to represent property graphs within SQL and query them via pattern matching. Essentially, SQL is gaining an embedded graph query language. For example, with a proper graph setup, a user could query a social network graph stored in tables, as in Example 1-5.

Example 1-5. An analytical query of a graph

```
SELECT *
FROM GRAPH contacts_network
MATCH (a:Person)-[:CONNECTED_TO*2..4]->(b:Person)
WHERE a.name = 'Alice';
```

In this hypothetical SQL graph query, we look at a graph `contacts_network` of people and their connections, and find all persons `b` who are within 2 to 4 "hops" from person `a` named *Alice*. The `MATCH` clause syntax is reminiscent of dedicated graph query languages, but here it's part of SQL. The result would list all such people `b` reachable through 2–4 connections from *Alice*.

This graph-querying capability means relational databases can handle certain graph analytics without exporting data to a separate graph database. While a specialized system might still be needed for very complex or deep graph algorithms, for many cases (shortest paths, neighborhood searches, friend-of-friend queries, etc.), SQL databases are now up to the task. As vendors implement the SQL/PGQ standard (Oracle has begun support, with others likely to follow), SQL solidifies its position as a multi-model query language. Fewer data silos are needed because a single SQL database can manage both relational data and graph relationships, accessible through one unified query interface.

If this standard expands to even greater adoption by vendors, SQL will solidify its position as a complete solution for graph analysis.

Dealing with Unstructured Data in SQL

Modern SQL platforms also accommodate unstructured data such as text documents, images, or blobs by allowing storage and indexing of large objects. For text in particular, many SQL databases have integrated full-text search capabilities:

- They let you store large text such as documents in columns, often as `TEXT` or `BLOB` types.
- They provide indexing mechanisms to search this text efficiently. In PostgreSQL, for example, with text search indexes or in SQL Server via Full-Text Search catalogs.

For instance, a database may support a function or predicate `CONTAINS(column, _'phrase'_)` to find whether a text contains certain words near each other, as shown in Example 1-6.

Example 1-6. Text manipulation in SQL

```
SELECT doc_id, content
FROM documents
WHERE CONTAINS(content, 'machine learning NEAR/5 algorithm');
```

In this example, the query searches a `documents` table for content entries that contain the phrase `_"machine learning"_` within 5 words of `_"algorithm"_`. The `CONTAINS` function (syntax here is similar to SQL Server or Oracle Text) uses an internal text index to quickly find matches.

Such full-text search functionalities enable efficient text analysis and information retrieval directly in the database. Rather than extracting text data to an external search engine for every query, simple search needs can be met within SQL. This is particularly useful for applications that need to filter or join unstructured data with structured data (for example, to find all product reviews containing certain keywords and then aggregate by product).

Integrating ML and LLMs in SQL Platforms

SQL is also becoming a gateway to advanced analytics like ML and even interactions with LLMs. Database vendors are extending SQL with capabilities to train models, make predictions, or call external AI services via SQL functions. This convergence allows data teams to leverage ML/AI directly where the data lives, without exporting large datasets to separate platforms.

A prime example is BigQuery ML in Google BigQuery. It allows users to create and apply ML models using SQL commands. Initially, BigQuery ML supported training models for regression, classification, time-series forecasting, etc., with simple `CREATE MODEL` statements. More recently (2023+), Google has even integrated its generative AI models (like Gemini) into BigQuery ML, meaning you can invoke powerful text generation or AI prediction within a SQL query.

For instance, training a logistic regression model to predict customer churn in BigQuery ML might look like Example 1-7.

Example 1-7. Using ML operators in SQL

```
CREATE OR REPLACE MODEL mydataset.customer_churn_model
OPTIONS (model_type='logistic_reg') AS
SELECT
  customer_age, tenure, product_category, churned
FROM mydataset.customer_data;
```

This SQL statement instructs BigQuery to train and save a model called `customer_churn_model` using the data from `customer_data` table (with `churned` as the label). Once the model is trained, you can perform predictions directly via SQL, as shown in Example 1-8.

Example 1-8. Making predictions in SQL

```
SELECT
  customer_id,
  predicted_churn
FROM ML.PREDICT(MODEL mydataset.customer_churn_model,
                TABLE mydataset.new_customers);
```

Here, `ML.PREDICT` applies the model to new customer data and returns predicted churn probabilities or classes. Other databases are also moving in this direction. For instance, PostgreSQL can call out to Python or R via extensions to integrate with external ML libraries, and some cloud databases allow invoking cloud AI services using SQL extensions or functions.

The key trend is bringing ML to the data. By doing so, organizations reduce the friction of moving data between systems and can incorporate predictive analytics into their regular SQL workflows. As an example of LLM integration, imagine a SQL function like `CALL_AI(prompt)` that calls an external API (OpenAI for example). Some experimental integrations exist via database extensions. While this is still emerging, it's clear that SQL's role is expanding from querying data to orchestrating and embedding advanced analytics and AI.

Another great example that leverages advanced AI functions is `ML.GENERATE_TEXT` in BigQuery, which sends a prompt to an LLM hosted via Vertex AI and gets the result as part of a query result. Example 1-9 illustrates how to call this special function.

Example 1-9. Making predictions in SQL

```
SELECT id,
 ML.GENERATE_TEXT('Summarize the following:', content) AS summary
FROM mydataset.documents;
```

Google reports that built-in ML usage in BigQuery has increased by 250% year-over-year, and analysts can now access LLMs with simple SQL statements in the BigQuery console.

In addition, BigQuery has introduced vector search to enable similarity queries over embeddings stored in tables. This is crucial for applications such as semantic text search and retrieval-augmented generation (RAG) with LLMs. BigQuery can also use external AI services to analyze unstructured data. By integrating BigQuery with Vertex AI, for example, you can invoke document processing or speech recognition of files and incorporate the results into SQL queries. All of these developments make

BigQuery an AI-powered analytics platform that allows analysts to combine traditional SQL aggregations with ML predictions and LLM-generated insights in one place.

Snowflake is another example. This cloud data platform has also brought ML to the SQL world. An important new feature is Snowpark, which enables writing Python universal disk formats (UDFs) and external functions, so users can call ML libraries or external APIs during SQL execution. In addition, Snowflake recently introduced Snowflake Cortex, a set of built-in LLM functions that provide SQL users with direct access to a variety of LLMs. Snowflake Cortex includes advanced models from vendors such as Anthropic, Meta, and Google, as well as Snowflake's Arctic model, and makes them accessible via simple SQL functions.

For example, Snowflake provides a `COMPLETE()` function for general text completion and other task-specific functions such as `CLASSIFY_TEXT()` or `SUMMARIZE_TEXT()` as managed UDFs in SQL. Because these models are fully hosted and managed by Snowflake, a user can, for example, perform sentiment analysis for product reviews or create a summary of a customer feedback text by simply calling a function in a `SELECT` query. The data never leaves the storage space defined by its owner, which is important for both management and security.

In addition, Snowflake supports vector embeddings as a data type via its Snowpark ML libraries, enabling similarity search and AI-driven retrieval in SQL. For custom models, Snowflake users can use Python-based ML pipelines with Snowpark or call external ML services via external functions, such as a Hugging Face model or a SageMaker endpoint via a SQL query.

In the open source world, PostgreSQL has gained AI functionality through extensions. PostgreSQL does not contain any ready-made ML algorithms, but projects such as PostgresML (PGML), and the `pgvector` extension make it possible to perform ML tasks within the database. The `pgvector` extension, for example, enables the storage of high-dimensional vectors,

called embeddings, and offers an indexed similarity search that essentially transforms PostgreSQL into a vector database suitable for AI applications.

This is very useful for working with text or image embeddings: one can store document vectors in a Postgres table and efficiently find nearest neighbors for semantic search or recommendations. On the ML side, PostgresML offers an extension that adds SQL functions for training and using models in Postgres. It enables workflows such as training a classification model on data in a SQL table and then executing predictions via a `pgml.predict(model, data)` function in queries.

In addition, developers have built extensions for direct integration with external LLMs and AI APIs. For example, an OpenAI API extension for Postgres provides SQL functions such as `openai_prompt(prompt_text)` to get a completion from an OpenAI GPT model, or `openai_embedding(text)` to retrieve an embedding vector for a piece of text. Under the hood, this leverages the procedural language capabilities of PostgreSQL with an HTTP extension that connects to the OpenAI web service and returns the results in a query.

Similarly, PostgreSQL can be configured to call local open source models; for example, using the Ollama project to host a Llama 2 model on the same server and query it via SQL. These integrations show the power of extensibility in the Postgres ecosystem. By installing an extension, a Postgres database can be directly integrated into ML workflows.

PostgreSQL is becoming increasingly popular for integrating AI capabilities into database applications because it eliminates the need for complex pipelines. You can store data, vectorize, and query models, all in a single SQL environment.

The evolution of SQL has resulted in databases that seamlessly support relational, semi-structured, unstructured, temporal, graph, ML, and generative AI workloads within a single, unified platform. This comprehensive capability greatly simplifies data architectures and reduces complexity and operational overhead.

SQL for Data Transformations

One of the biggest changes in data engineering is the shift from extensive ETL scripts to SQL-centered data transformations in data warehouses. Tools like dbt have changed the game, allowing data teams to fully transform, test, and document data in SQL. Unlike traditional ETL tools, dbt, born in an ETL period where transformation happens after loading raw data, focuses exclusively on the "T" (transform) in these workflows. After data is loaded into a warehouse, analysts/engineers define transformations as SQL models. This approach has quickly become the industry standard for analytics engineering:

- SQL queries are modularized into models (each model is basically a `SELECT` query that creates a derived table).
- These models can be version controlled (stored in Git) and accompanied by tests and documentation.
- dbt then orchestrates these SQL models, materializing them in the correct order (taking care of dependencies).

For example, an analytics engineer might create a model for `daily_sales` as a SQL query selecting and aggregating raw sales data by day, and another model for `monthly_sales` that selects from `daily_sales`. dbt will manage running these in sequence.

Because modern cloud warehouses can handle heavy SQL transformations, this means transformations happen where the data lives, avoiding needless data movement. Analysts can use pure SQL to do tasks that once required specialized ETL tools.

Recent developments have extended these SQL-based transformation tools. In 2023, dbt introduced the concept of a semantic layer by allowing definition of business metrics in one central place and reusing them across models and BI tools. This means, for example, a metric like "monthly recurring revenue" can be defined once in dbt and then any SQL model or even external dashboard can pull that definition to ensure consistency.

Overall, SQL-centric transformation workflows have enabled software engineering best practices such as modularity, testing, and CI/CD to the analytics world. Teams treat their SQL code as first-class code: they review it, test it, and deploy it. This results in data pipelines that are easier to maintain and more reliable. It's not uncommon to see complex data transformation pipelines expressed entirely in SQL and orchestrated by tools like dbt, which speaks to SQL's enduring flexibility and power.

Embedded Analytics with DuckDB

Another innovation in the SQL space is embedded analytical databases like DuckDB. Often called "the SQLite of analytics," DuckDB is an in-process OLAP database that runs inside an application or notebook without needing a server. This design makes it extremely handy for data scientists and analysts working on local data or in development environments. Key features of DuckDB include:

- It runs in your existing runtime. You just import the DuckDB library and start querying—no separate database server needed.
- It's optimized for analytical queries on large datasets, using a columnar storage format and vectorized execution.
- It can query data files, like Parquet or CSV files on disk, directly, as if they were tables, without requiring that they be loaded into a separate database storage. This means you can point DuckDB at a Parquet file and run SQL on it immediately.
- It can even query pandas dataframes in-memory directly, so you could run a SQL query that joins a pandas dataframe with another dataset via DuckDB without converting formats.

Despite its small footprint, DuckDB supports complex SQL (joins, window functions, aggregations) and is ACID-compliant for transactions. It's been shown to handle surprisingly large datasets on a laptop, often achieving performance similar to cloud warehouses for moderate data sizes, thanks to its efficient query engine.

DuckDB's impact is that it brings lightweight SQL analytics to the data scientist's toolkit. Instead of subsetting data or writing complex pandas code for everything, users can leverage the power and familiarity of SQL locally.

Many have found that DuckDB can handle analyses on tens of millions of rows in seconds, making it ideal for exploratory data analysis or prototyping on local data. It has effectively bridged the gap between local analysis and data warehouse analysis. DuckDB's extensibility has even led to experiments like running it in the browser (DuckDB-Wasm) and integrating it with other tools. Developers have also created extensions for DuckDB to connect to external tools or even to call Python functions from SQL. All these developments mean that the distinction between "big data SQL" and "local data analysis" is blurring—SQL is available at all scales.

Comparison of SQL Dialects over Time

While SQL standards unify the core language, different database systems (dialects) historically have had significant differences. Over time, many of these differences have narrowed, but some remain:

Vendor-specific extensions

In the past, each vendor added unique features before they were standardized (or if they never were). For example, Oracle introduced the `DECODE()` function and hierarchical query syntax (`CONNECT BY PRIOR`) long before standard SQL had equivalents. SQL Server's T-SQL provided `IDENTITY` (`auto-increment`) and the `TOP N` clause (versus standard `FETCH FIRST N`). PostgreSQL pioneered features like JSON support (years before SQL:2016) and extensibility such as custom index types and foreign data wrappers. For a long time, MySQL had lenient SQL (implicit conversions, with no need for strict syntax) and was late to implement CTEs and window functions (not until MySQL 8.0 around 2018). These extensions meant that SQL code written for one database might not run on another database without modification.

Convergence in modern SQL

This has occurred in the last 10 to 15 years. Most major databases now implement popular features: window functions, CTEs, `MERGE` statements, JSON functions, etc. For instance, virtually all databases support `ROW_NUMBER()` and other window functions. Recursive CTEs are now widely supported (Oracle was one of the last, added in version 11g). This convergence, driven by user demand and competitive pressure, means switching between modern SQL systems is easier than it was in the 1990s—basic query patterns work similarly, and differences are often in the details (function names, date handling, etc.).

Dialects in newer cloud platforms

Cloud data warehouses introduced new SQL dialects, mostly built on existing ones. Google BigQuery's SQL dialect is based on standard ANSI SQL (they explicitly call it "Standard SQL") and is similar to PostgreSQL in many ways, with a few extensions (e.g., native `ARRAY` data types and some special syntax for items like wildcard table queries). Snowflake's SQL dialect is very close to the Oracle/ANSI standard, with extensions for semi-structured data (it has a `VARIANT` type for JSON and supports standard JSON querying as well as its own functions). Amazon Redshift started as a fork of PostgreSQL, so its SQL feels like PostgreSQL (with some differences and lacking some newer Postgres features). Even analytics engines like HiveQL or Databricks SQL were patterned after familiar languages (MySQL/Oracle).

In summary, the cloud didn't reinvent SQL from scratch. It carried forward the common subset of SQL features and behaviors. One notable quirk: some cloud engines handle case sensitivity of identifiers differently. For example, BigQuery treats unquoted identifiers case-insensitively (like Oracle/SQL Server does) whereas PostgreSQL treats unquoted identifiers as lowercase (and is case-sensitive if quoted). Such differences can surprise users moving between systems.

As a result, despite the broad convergence, experienced SQL users still watch out for certain differences when moving between databases:

Date/time functions

Almost every database has its own syntax for current timestamp or date arithmetic. SQL Server uses `GETDATE()`, Oracle uses `SYSDATE`, and PostgreSQL uses `NOW()`. Date add/subtract syntax and functions (e.g., Oracle's date + 1 versus SQL standard `DATE_ADD` or `interval` syntax) can differ, though these are usually minor and easily adjusted.

String concatenation and functions

Concatenation is `||` in standard SQL (and supported by Oracle and Postgres) but in SQL Server it was + (though SQL Server now also supports `||`). Function names like `substring` versus `substr`, `length` versus `len` differ by dialect, as do regex syntax and other text processing quirks.

Identifier case sensitivity

As mentioned, how identifiers (table/column names) are treated if not quoted varies. PostgreSQL lowercases them, Oracle uppercases them, and MySQL and BigQuery don't case-fold but are case-insensitive. This can affect porting code if someone relied on a particular case.

Auto-increment keys

Before standard sequences/identity were added (SQL:2003), each database had its own approach (MySQL `AUTO_INCREMENT`, Oracle sequences + triggers, PostgreSQL `SERIAL`). Today, many databases support the standard `GENERATED AS IDENTITY`, but old habits and slight differences in implementation persist.

Advanced features or areas not fully standardized often diverge

Full-text search is one example: SQL Server's `CONTAINS` versus Postgres text search operators (`to_tsquery with @@`) are completely

different. Geospatial SQL functions follow the Open Geospatial Consortium (OGC) standard somewhat, but implementations (PostGIS versus Oracle Spatial versus others) vary in function names and availability.

In practice, the core SQL a person writes for data retrieval and analysis is highly portable across systems, especially if you stick to ANSI-standard constructs. But when you get into database-specific optimizations or features (such as using `QUALIFY` in Snowflake, or writing a procedure in T-SQL, or using a special index type in Postgres), you're in dialect-specific territory. The trend is that each new SQL standard and each new generation of databases brings more alignment. Thus, for most typical analytical queries, you can expect only minor syntax tweaks moving from, say, Postgres to BigQuery to Snowflake. Understanding the remaining differences helps you quickly adapt queries for the target system without encountering common pitfalls.

Summary

In this chapter we explored how SQL has grown from a simple relational query language into a powerful tool for modern data engineering and analytics. Each new standard brought meaningful improvements. SQL:1999 introduced recursive queries and procedural logic. SQL:2003 expanded analytical capabilities with window functions. SQL:2011 added time-travel queries. SQL:2016 brought native JSON support. Finally, SQL:2023 introduced graph querying and additional developer-friendly improvements. These changes helped make SQL more expressive, portable, and useful across a growing range of use cases.

We also saw how many of the features introduced over the years are now foundational in modern tools and platforms. Concepts like CTEs, window functions, and JSON handling are essential in tools like dbt and in cloud platforms like BigQuery and Snowflake. SQL has become the common

interface for data modeling, transformation, and advanced analytics, even in systems that are no longer purely relational.

In the next chapter we move from the evolution of the language to its core mechanics. We will explore how the `SELECT` statement works; how filtering, joining, and aggregating operate at execution time; and why mastering these fundamentals is critical for writing performant and readable SQL at scale.

Chapter 2. Advanced SQL Techniques

In Chapter 1, we explored SQL's evolution and modern breadth across traditional databases and cloud data platforms. We saw how SQL has grown far beyond simple `SELECT` statements into a rich ecosystem of analytical capabilities. Chapter 2 builds on that foundation by diving into advanced SQL query techniques and practical performance optimization strategies. We will discuss several SQL features like recursive common table expressions (CTEs) and window functions that enable complex data manipulations within SQL. We'll then explore techniques to refactor and optimize queries for better performance. These include indexing in PostgreSQL, understanding how query planners make execution decisions, caching results, and leveraging materialized views in both on-premise and cloud environments. Throughout, we'll compare approaches in PostgreSQL (representative of many traditional RDBMSs) with modern cloud data warehouses like Snowflake and BigQuery, highlighting best practices and real-world scenarios that intermediate and advanced SQL practitioners can apply.

Recursive Queries, Window Functions, and CTEs

Advanced SQL enables you to express complex logic within queries by using CTEs (including recursive CTEs) and window functions. These features let you handle hierarchical or sequential data and perform analytics directly in SQL, capabilities that earlier SQL dialects lacked or made overly difficult. In this section, we explore the syntax and natural usage of nonrecursive and recursive CTEs and window functions. We will look at typical use cases such as hierarchical data traversal (organizational charts,

graph paths), generating sequences, and computing running aggregates. Throughout, examples will be given for PostgreSQL (as a traditional RDBMS) and for modern cloud data warehouses like Snowflake and BigQuery, illustrating that the concepts apply across different SQL implementations.

Common Table Expressions and Subquery Factoring

CTEs, introduced in the SQL:1999 standard, provide a way to define a temporary result set (a subquery) at the top of your query and give it a name, which can then be referenced later in the query. The CTE is defined using a `WITH` clause before the main statement. CTEs improve query readability and structure by allowing complex logic to be broken into named components, rather than nesting subqueries inline.

Consider a scenario where we want to break a complex query into parts. Using a CTE, we can do so as presented in Example 2-1.

Example 2-1. Using a CTE to modularize a query

```
-- Using a CTE to modularize a query
WITH RecentOrders AS
(
  SELECT customer_id,
         amount
  FROM orders
  WHERE order_date >= DATE '2025-01-01'
)

SELECT r.customer_id,
       SUM(r.amount) AS total_2025_spending
FROM RecentOrders AS r
JOIN customers AS c ON c.id = r.customer_id
GROUP BY r.customer_id;
```

In this example, the CTE named `RecentOrders` is defined to select all orders from 2025 on. In the main query, we then refer to `RecentOrders` as if it were a table, joining it with a `customers` table and aggregating. This approach makes the query easier to follow: the filtering logic is

encapsulated in the CTE, and the main query clearly operates on the results of that subquery.

CTEs help modularize SQL queries, especially complex ones with multiple steps. Instead of one huge query, you can build it step-by-step with self-explanatory names for each subquery (beginning with the double hyphen). Most major SQL engines (PostgreSQL, SQL Server, Oracle, etc.) adopted CTEs in the 2000s, so this feature is available in traditional RDBMS engines.

A nonrecursive CTE is essentially a syntax convenience (like a temporary inline view). It doesn't inherently improve performance by itself (and in some cases, breaking a query into many CTEs can inhibit certain optimizations). Nonetheless, for complex analytical queries or multi-step transformations, CTEs are valuable for clarity. They also set up recursive CTEs, which allow a CTE to reference itself.

Recursive CTEs for Hierarchical and Iterative Queries

A recursive CTE is a special kind of CTE that references itself, allowing SQL to perform iterative processing and elegantly handle hierarchical data. Recursive CTEs were a gamechanger for SQL, introduced with the `WITH RECURSIVE` syntax. They consist of two parts: an anchor member (the base query) and a recursive member that references the CTE itself, combined with a `UNION ALL`. The recursive member runs repeatedly, each time using results from the previous iteration, until no new rows are produced (or until an optional termination condition is met).

This construct allows SQL to climb hierarchies (like organizational charts or category trees) or iterate until a condition is reached, all within a single query. Historically, before recursive CTEs, such logic required multiple self joins or writing procedural code outside the database.

A common use case for a recursive CTE is querying employee hierarchical data without knowing in advance how deep the hierarchy goes. For instance, suppose we have an `employees` table with each employee's

`manager_id`. We want to get the management chain of a particular employee or list all subordinate relationships. Using a recursive CTE, this looks like Example 2-2.

Example 2-2. A recursive CTE for a hierarchical query (organization chart)

```
-- Recursive CTE for hierarchical query (organization chart)
WITH RECURSIVE Subordinates AS
(
 /* Anchor member: start with top-level manager(s),
 e.g., CEO (no manager) */
 SELECT id, name, manager_id, 0 AS level
 FROM employees
 WHERE manager_id IS NULL

 UNION ALL

 /* Recursive member: find employees managed by
 those already selected */
 SELECT e.id, e.name, e.manager_id, s.level + 1
 FROM employees e
 JOIN Subordinates s ON e.manager_id = s.id
)

SELECT *
FROM Subordinates
ORDER BY level, manager_id;
```

In this example, the CTE `Subordinates` first selects the top of the hierarchy (employees with no manager, level 0). Then, it recursively joins `employees` (`e`) with `Subordinates` (`s`) to find employees whose manager is someone already in the `Subordinates` result, incrementing the level each time. The recursion continues, adding level 1, 2, 3, … until there are no more employees down the chain. The final `SELECT` then retrieves the assembled hierarchy. Each employee is listed with their hierarchical level (distance from a top manager).

The result of such a query would list the CEO(s) at level 0, their direct reports at level 1, those employees' reports at level 2, and so on. Recursive CTEs make it straightforward to produce this kind of indented hierarchy listing or to find all descendants of a given node.

Common applications of recursive CTEs include organizational charts (in the example), bill-of-materials (BOM) expansions in manufacturing (where a product is composed of subparts, which in turn have subparts, etc.), graph traversals (finding all reachable nodes in a network), and even generating sequences of numbers. For example, recursive CTEs can generate a series of numbers or dates, which in practice becomes the foundation for useful constructs like dynamic calendar tables, missing-date detection, or producing placeholder rows. All of this occurs directly in SQL, without relying on preexisting helper tables.

Recursive CTEs must terminate; otherwise, they can loop indefinitely. In our example, recursion ends naturally when an employee has no further subordinates (the recursive `SELECT` returns no new rows). In most implementations, you don't need an explicit stopping `WHERE` clause in the recursive part: the structure of the query naturally leads to termination. The key requirement is that some condition prevents unbounded growth, such as joining on a column that eventually yields no matches (no more subordinates, no more graph neighbors, etc.).

Modern SQL engines normally have a built-in safeguard like a maximum recursion depth (for example, SQL Server has a default max recursion of 100, which can be overridden). BigQuery also supports recursive CTEs (using the same `WITH RECURSIVE` syntax) and notes that they "run continuously until no new results are found," which is essentially how the termination of recursion is determined. BigQuery's documentation gives an example of finding all reachable graph nodes from a starting node using recursion, demonstrating that modern analytics databases have embraced this SQL feature for graph-like queries as well.

That said, CTE performance can become computationally heavy and expensive, especially when the hierarchy is deep or the dataset is large. Each iteration may scan additional parts of the underlying tables, and without proper indexing or limiting conditions, the recursive step can grow exponentially in cost. However, when used appropriately, for example, on indexed hierarchical keys in PostgreSQL or partitioned datasets in a data

warehouse, recursive CTEs offer an effective solution to problems that would otherwise require complex procedural logic.

Recursive CTEs allow you to express iterative logic (like traversing a tree or graph) declaratively using pure SQL. This was a significant addition to SQL's expressive power, enabling solutions that were previously complex or impractical in set-based SQL. In fact, recursion makes SQL formally Turing complete—capable of expressing any algorithm a general-purpose language can.

> **NOTE**
>
> While few practitioners will push SQL that far, creative examples exist, from a Doom-like multiplayer shooter implemented in pure SQL using recursive CTEs for raycasting logic (DOOMQL), to a full Sudoku solver published on the PostgreSQL wiki.

Advanced Window Functions for Analytic Queries

Another pillar of advanced SQL is the extensive use of window functions for analytic computations. Window functions (also called analytic functions) perform calculations across sets of rows related to the current row, without collapsing the result set the way the `GROUP BY` aggregation does. This means you can compute running totals, moving averages, rankings, and percentiles, and you can compare rows to other rows (lag/lead) all while retaining each individual row in the output.

Formally, a window function operates on a "window" of multiple rows defined relative to the current row, as specified by an `OVER` clause (which can partition the data into groups and order them). Unlike a `GROUP BY`, which reduces each group to one result, window functions output a value for every row.

For example, suppose we want to rank employees by salary within each department. We can use the `RANK()` window function, as presented in Example 2-3.

Example 2-3. Using a window function to rank employees by salary within each department

```
-- Ranking employees by salary within each department
SELECT
    department_id,
    employee_name,
    salary,
    RANK() OVER (
        PARTITION BY department_id
        ORDER BY salary DESC
    ) AS salary_rank
FROM employees;
```

This query returns each employee's salary with their rank within their department (1 = highest salary in that dept, etc.). The window function `RANK() OVER (PARTITION BY department_id ORDER BY salary DESC)` computes the rank for each row according to the specified partition (department) and order (salary descending). Unlike a `GROUP BY`, all original rows are retained, and each row gets an additional column showing the rank. Figure 2-1 shows a sample output for the preceding query if we had only 6 users and two departments (1 and 2).

department_id	employee_name	salary	salary_rank
1	Helder Russa	90,250	2
1	Thomas Sousa	107,500	1
1	Antonio Maria	83,700	3
2	Peter Esmeriz	93,900	3
2	Patricia Ramalho	101,150	1
2	Rui Machado	95,100	2

Figure 2-1. Window function illustrated

Ranking functions are often used to identify relative positions within a group, but they are not the only way to express ordered analytics. Many SQL engines also support value window functions, which return a value

from a specific position in the ordered window, rather than a rank. These functions are particularly useful when the intent is to retrieve a specific ordered value, such as the second-highest salary (shown in Example 2-4), rather than assign ranks to every row.

Example 2-4. A window function to get the second-highest salary value within each department

```
-- Extracting the second-highest salary value within each department
SELECT
    department_id,
    employee_name,
    salary,
    NTH_VALUE(salary, 2) OVER (
        PARTITION BY department_id
        ORDER BY salary DESC
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS second_highest_salary
FROM employees;
```

Another common analytic task is computing running totals or moving averages over time. Window functions can easily compute a cumulative sum of a value column, ordered by date (or any sequence). For instance, consider computing a running total of sales by date. Using window functions, you could write this as shown in Example 2-5.

Example 2-5. A window function computing a running total of sales by date

```
-- Computing a running total of sales by date
SELECT sale_date,
       sales,
       SUM(sales) OVER (ORDER BY sale_date
            ROWS BETWEEN UNBOUNDED PRECEDING
            AND CURRENT ROW) AS running_total
FROM daily_sales;
```

This query, `SUM(sales) OVER (...)`, computes a running sum up to the current row's date. The window frame clause `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` specifies a cumulative sum from the start through the current row. Each row of output retains its original `sales` value and `sale_date`, along with the `running_total` calculated over the "window" of that row and all prior rows.

Window functions also allow comparing a row to another row in the dataset without self joins. For example, using `LAG()` or `LEAD()`, you can access the previous or next row's value from the current row. Consider computing day-over-day change in sales, as presented in Example 2-6.

Example 2-6. A window function computing day-over-day change in sales

```
-- Calculating day-over-day sales change using LAG
SELECT
  sale_date,
  sales,
  LAG(sales, 1) OVER (ORDER BY sale_date) AS previous_day_sales,
  sales - LAG(sales, 1)
                OVER (ORDER BY sale_date) AS sales_difference
FROM daily_sales;
```

For each `sale_date`, this query shows the `sales` amount, the prior day's sales, and the difference (`sales_difference`). The `LAG(sales, 1)` function looks one row back (according to the date ordering) and fetches that value. This kind of calculation is useful in time-series analysis and financial reporting (e.g., period-over-period change) without requiring a self join of the table to itself.

Common categories of window functions include:

Ranking functions

These functions assign rankings or row numbers within a partition. For example, `ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC)` would enumerate employees in each department by descending salary (useful, for example, for picking top-N earners per department). Other common ranking functions are `RANK()`, `DENSE_RANK()`, and `NTILE()`.

Value functions

These functions return a value from a specific ordered position within the window, rather than assigning a rank or computing an aggregate. Common examples include `FIRST_VALUE`, `LAST_VALUE`, and `NTH_VALUE`. They are useful when the goal is to extract a particular

value from an ordered partition (for example, the second-highest salary in each department) and repeat it across all rows in the window.

Offset functions

The most common are `LAG()` and `LEAD()`, which allow you to access the value from a previous or next row in the partition. For instance, using `LEAD(sales, 1) OVER (ORDER BY date)` would bring the next day's sales value into the current row, enabling easy calculation of day-over-day differences. This is far simpler than self joining a table to itself to align consecutive rows, and enables calculations like differences, ratios, or flags (e.g., flag if the current row's value is greater than the previous row's). Although `LAG()` and `LEAD()` are technically value-returning window functions, they are often treated as a separate category because they express relative access to neighboring rows rather than retrieval of a fixed-position value in the window.

Aggregate window functions

Traditional aggregate functions (`SUM`, `AVG`, `MIN`, `MAX`, `COUNT`) can be used in a window context. They can produce running totals, moving averages, or sliding window calculations. By adjusting the frame (for example, last 7 rows versus all preceding rows), you can compute things like a 7-day moving average of a metric, or a rolling window sum. For example, `AVG(sales) OVER (ORDER BY date ROWS 6 PRECEDING)` gives you the average of the current and previous 6 days' sales (a 7-day moving average).

Percentile and statistical functions

Many SQL implementations include window functions that calculate percentiles, quantiles, and related statistical measures. For example, `PERCENTILE_CONT(0.5)` can return the median value within a moving window, while `NTILE(100)` can assign percentile buckets across rows. Functions like `PERCENT_RANK()` and `CUME_DIST()` provide relative positions within a dataset. Cloud warehouses also offer approximate

versions for large-scale analytics, such as BigQuery's `APPROX_QUANTILES` and Snowflake's `APPROX_PERCENTILE_ACCUMULATE`.

Window functions are particularly useful for time-series analysis (trends, moving averages, cumulative totals) and comparative analytics (comparing each row to other rows, such as for an average or prior period). Window functions enable patterns like year-over-year or month-over-month comparisons (by partitioning by a category and ordering by date, then using `LAG` to get last year's value in the same partition), running totals, cohort analysis, and gap detection (e.g., find if there's a gap between events by comparing current and previous timestamps).

To illustrate an advanced use: suppose we have a table of daily stock prices and want to identify the 5-day moving average as well as flag days when the stock price was higher than the previous day. We could do so using the information shown in Example 2-7.

Example 2-7. Advanced use case of a window function

```
SELECT
    date,
    price,
    /* 5-day moving average of price
    (including current day and 4 prior days) */
    AVG(price)
          OVER (ORDER BY date
              ROWS BETWEEN 4 PRECEDING
              AND CURRENT ROW) AS ma_5,
    -- flag if price increased compared to previous day
    CASE WHEN price > LAG(price, 1) OVER (ORDER BY date)
    THEN 1
    ELSE 0 END AS price_up
FROM stocks
WHERE symbol = 'ACME';
```

This example uses two window functions: one to compute the 5-day moving average of the price, and another (`LAG`) to fetch the prior day's price for comparison. The output retains each day's record, with additional columns for the moving average and the up/down flag. Without window

functions, achieving this would require either self joins (joining the table to itself on date conditions) or subqueries that individually calculate aggregates for each row, both of which are less efficient and harder to read.

All major SQL databases support window functions, though syntax details and supported functions can vary. PostgreSQL has a full implementation of the SQL standard window functions (and even allows custom aggregation in windows). Snowflake also fully supports window functions and provides a convenient `QUALIFY` clause (similar to a `WHERE` but for filtering after window functions are computed). BigQuery also supports window functions and can efficiently execute them on large data due to its massively parallel architecture (for instance, computing a window function over millions of rows is feasible and common in BigQuery).

Minor differences exist: Snowflake's `QUALIFY` clause (e.g., `QUALIFY ROW_NUMBER() OVER(...) = 1` to filter to top-ranked) is very handy; other databases achieve the same functionality via a subquery or CTE. BigQuery and PostgreSQL don't use `QUALIFY`, but you can wrap the window query and then filter. BigQuery also offers nonstandard analytical functions like `APPROX_QUANTILES` (not exactly a window function but often used for large-scale percentile calculations), and Snowflake has specialized `APPROX_PERCENTILE` functions. These differences aside, core window function behavior is consistent.

Mastering window functions is essential for advanced SQL use. It often allows an analyst to perform in-database what otherwise would require exporting data to a spreadsheet or writing a custom Python or R script. As data sizes grow, doing these computations in SQL (leveraging the database's optimizations) is far more scalable. It boosts SQL from a query language that can only group or filter data to one that can perform complex analytics like running totals, rankings, and period-over-period comparisons within a single query, enabling the writing of highly sophisticated data retrieval and analysis logic directly in SQL, typically with better performance (leveraging the database engine) than alternative approaches.

Performance Optimization Strategies

As SQL queries grow in complexity and data volumes increase, understanding how to optimize query performance becomes critical for optimal query execution. In this chapter, we discuss strategies and tools for SQL performance tuning, including how to interpret query plans and leverage cost-based optimizers, the effective use of indexing and data partitioning, and other techniques like query rewriting and caching. We will compare approaches in a traditional RDBMS (using PostgreSQL as an example) with approaches in modern cloud data warehouses like Snowflake and BigQuery, highlighting similarities and differences. The goal is to provide a mental model for how databases execute queries and what we, as SQL practitioners, can do to ensure those queries run efficiently.

Indexing and Efficient Data Access

Indexes are one of the most fundamental tools for improving SQL query performance in traditional databases. An index is a data structure that allows the database to quickly locate specific rows without scanning the entire table. By indexing columns that are used in `JOIN` conditions, `WHERE` filters, or `ORDER BY` clauses, the query planner can check the index instead of performing a full table scan, often making lookups or filtered queries exponentially faster.

> **NOTE**
>
> If you work primarily with systems like Snowflake or BigQuery, you won't be creating traditional indexes at all: these platforms manage data organization automatically. Instead, you'll rely on table partitioning and clustering to achieve similar effects. We'll revisit this in the "Tuning in Cloud Data Warehouses: BigQuery Versus Snowflake", but for context: indexes are still relevant in traditional RDBMSs, whereas cloud native warehouses shift the responsibility to automatic storage management and elastic compute.

For example, suppose we have a table of `orders(order_id, customer_id, order_date, total_amount)` with millions of rows. If we frequently query orders by `customer_id`, as shown in Example 2-8, an index on that column will certainly help.

Example 2-8. Simple query without index

```
-- Without an index on customer_id (potential slow sequential scan):
SELECT *
FROM orders
WHERE customer_id = 12345;
```

Without an index on `customer_id`, the database must perform a sequential scan of the entire `orders` table to find rows for customer `12345`. This means reading every row and checking the condition, which is very slow for millions of rows.

However, we can create an index on `customer_id`, as shown in Example 2-9.

Example 2-9. Index creation

```
-- Creating an index to speed up the query
CREATE INDEX idx_orders_customer ON orders(customer_id);

-- Query uses the index, speeding up retrieval:
EXPLAIN SELECT *
FROM orders
WHERE customer_id = 12345;
```

The query plan (as shown by `EXPLAIN`) will likely indicate an index scan using `idx_orders_customer` instead of a `Seq Scan`. The index acts like a sorted lookup table on `customer_id`, allowing the database to jump directly to entries matching `12345`. This drastically reduces I/O: instead of scanning potentially millions of rows, the database may need to read only a small portion of the index and the matching data pages.

In general, a B-tree index (the default type in most databases) gives about O(log n) search time, meaning even if a table has 1,000,000 rows, an index search might read only dozens of entries to find matching rows, versus potentially all 1,000,000 in a sequential scan. The difference is huge if the

query is filtering to a small subset. For instance, retrieving 100 rows out of a million via an index means the database can fetch those 100 directly, whereas without an index it would still examine all million rows to find those 100.

PostgreSQL and other databases also support specialized index types beyond the default B-tree:

Hash indexes (in Postgres)

> For equality lookups (though in modern Postgres, B-tree often performs as well or better).

Generalized inverted index (GIN) and GiST indexes

> For data like arrays, JSONB or full text—they index elements/keys within a value, which is great for text search or checking existence of a key in a JSON document.

Block range index (BRIN)

> For very large tables that are naturally ordered (e.g., a huge time-series table partitioned by date). These store summaries per block of data and can quickly eliminate large ranges that don't meet a condition.

For example, a GIN index on a JSONB column in Postgres allows querying for records that contain a certain key/value without scanning every row's JSON; a BRIN index on a date column can quickly skip over chunks of a fact table that are out of the date range of interest, at minimal storage cost.

The key point is: choose the right index for the job. For typical queries on individual values or ranges, B-tree is ideal. Ensure that columns that are frequently used to filter or join are indexed. Primary keys are indexed automatically (since they enforce uniqueness), and foreign keys should be indexed (to speed up joins on them).

That said, there is a balance because indexes come with a cost: each index adds overhead on writes (`INSERT`/`UPDATE`/`DELETE`), because the index must

be updated when data changes. Thus, a best practice is to index selectively: create indexes on columns that are frequently filtered or joined but avoid indexing rarely used columns or indexing too many columns on the same table. It's also wise to avoid duplicate indexes or indexes that don't match query patterns.

Here are additional indexing tips:

Multicolumn indexes

> These can speed up queries that filter on multiple columns. For instance, an index on (`customer_id`, `order_date`) can efficiently support a query filtering by customer and date range. However, multicolumn indexes have a defined column order; they work best when the leftmost column in the index is used in the query filter (e.g., the preceding index helps a filter by `customer_id` and `order_date` but wouldn't help a filter by `order_date` alone unless the `customer_id` is also specified).

Covering indexes

> These include extra columns to serve the query without needing to fetch the table row. PostgreSQL supports the `INCLUDE` clause to add non-key columns to an index. For example, an index on `orders(customer_id) INCLUDE(total_amount)` means queries that filter by `customer_id` and need only `total_amount` can get all necessary data from the index itself, avoiding a table lookup.

Partial indexes

> These can be used to index a subset of data (using a condition) if queries often focus on a particular subset. For instance, indexing only "active" records (`WHERE status = 'active'`) will speed up queries for active records without the bloat of indexing inactive ones.

One more note: Partitioning and clustering can be seen as complements or alternatives to indexing in big data systems. PostgreSQL allows table partitioning (e.g., by date range), and its planner will prune partitions that

aren't needed for a given query. BigQuery and Snowflake, as discussed below, rely heavily on partitioning/clustering rather than user-defined indexes.

In contrast, Snowflake and BigQuery do not require user-managed indexes. Snowflake doesn't support user-defined indexes at all; instead, it organizes data into micropartitions that are automatically sorted and pruned during queries. BigQuery likewise has no B-tree indexes: performance comes from partitioning and clustering, combined with the ability to elastically scale compute. In practice, this means the skill set shifts: instead of designing indexes, you focus on choosing partitioning keys, clustering strategies, and ensuring queries are written to take advantage of them. We'll return to this in "Tuning in Cloud Data Warehouses: BigQuery Versus Snowflake", where we look at how partitioning, clustering, and elastic compute replace indexing as the main tuning levers in these systems.

For traditional databases like PostgreSQL, appropriate indexing is often the single biggest boost you can give to query performance. It turns table scans into index lookups, which are a huge win for selective queries. In analytic databases like BigQuery or Snowflake, you instead ensure your table is partitioned by date or some logical key and possibly clustered so that the storage order supports the kinds of filters you'll run. Under the hood, these approaches are analogous, since they both aim to avoid scanning irrelevant data.

Query Planning and Cost-Based Optimization

When a SQL query is submitted, the database must decide how to execute it: which indexes to use, how to join tables, whether to sort data or use an existing order, and so on. Modern SQL databases use cost-based optimizers to make these decisions. "PostgreSQL's documentation" describes this well: "Choosing the right plan to match the query structure and the properties of the data is absolutely critical for good performance, so the system includes a complex planner that tries to choose good plans."

The database considers many possible ways (plans) to execute a query, estimates the cost of each (based on statistics about table sizes, index selectivity, etc.), and then chooses the lowest-cost plan. In practice, query tuning is largely about understanding what the optimizer is choosing and why. This section intentionally focuses on that high-level reasoning rather than an exhaustive, engine-specific analysis of planner internals or complex execution plans.

Consider Example 2-10, a query written in PostgreSQL.

Example 2-10. Simple introductory query to explain how a planner could decide

```
-- Join query for planner illustration
SELECT *
FROM orders o
JOIN customers c ON o.customer_id = c.id
WHERE o.order_date >= '2025-01-01'
  AND c.region = 'EUROPE';
```

This query joins orders and customers, filtering orders from 2025 onward and customers in Europe. The optimizer must decide the following:

- Do we scan `orders` first and then find the customer for each order (perhaps via index), or scan `customers` first and then find orders?
- Do we use an index on `orders.order_date` if one exists? Or an index on `customers.region`?
- Should we do a hash join: read all qualifying customers, read all qualifying orders, and then hash join them?
- If we have a large portion of `orders` from 2025 onward, would a sequential scan be cheaper than using an index (because reading a huge fraction of a table via index might be slower than just scanning it)?
- If our region filter greatly reduces customers (say only 5% of customers are in Europe), do we start with that filter?

It's up to the cost-based optimizer to make these decisions. It uses statistics to know approximately how many orders are from 2025+ (say 20% of the table) and how many customers are in Europe (say 5%). Using this info, it estimates the number of rows that will result from each step (this is called cardinality estimation). Then it assigns costs to operations: for example, scanning a table of X rows has cost proportional to X; an index scan cost is proportional to rows fetched plus some overhead; a join cost depends on the algorithm and input sizes. The cost-based optimizer effectively builds a plan tree with these nodes and their costs, and then compares different plan trees.

To see what plan the optimizer chooses, we use the `EXPLAIN` command, which shows the planned execution steps without actually running the query. Example 2-11 shows how to use `EXPLAIN`.

Example 2-11. Example of an `EXPLAIN` statement

```
EXPLAIN SELECT *
FROM orders
WHERE customer_id = 12345;
```

Successfully executing the preceding command could output a plan similar to Example 2-12.

Example 2-12. `EXPLAIN` statement output without index

```
-> Seq Scan on orders  (cost=0.00..11578.33 rows=5 width=19)
  Filter: (customer_id = 12345)
```

This indicates a sequential scan was planned. Perhaps no index exists, or the planner thought very few `rows=5` so `Seq Scan` might work here. Normally, however, with 5 rows it would index; this is just an example).

After adding an index as earlier, `EXPLAIN` might show as in Example 2-13.

Example 2-13. `EXPLAIN` statement output with index

```
-> Bitmap Index Scan using idx_orders_customer on orders
(cost=0.00..4.51 rows=11 width=0)
  Index Cond: (customer_id = 12345)
```

This output tells us the planner chose a Bitmap Index Scan using the `idx_orders_customer` index, with some cost estimates. The cost numbers are internal units (not directly timed in milliseconds but relative units where the planner's cost model estimates effort). We see the optimizer expects about 11 rows to match (the `rows=11` estimate).

If we saw a `Seq Scan` instead, as shown in Example 2-14, it means the optimizer did not use the index. A possibility might be because the index was absent, or the optimizer thought the index wouldn't help (for example, if we were filtering a very large fraction of the table, sometimes a sequential scan is actually cheaper than multiple index lookups).

Example 2-14. `EXPLAIN` statement output without index

```
-> Parallel Seq Scan on orders  (cost=0.00..11578.33 rows=5 width=19)
  Filter: (customer_id = 12345)
```

When in textual format, plans are read from top to bottom, where indented lines are inner operations. In the preceding output, only one line is indented under the main operation, which is the index scan performing the filter. In more complex plans with joins, you'd see multiple levels. For example, a join plan might look like Example 2-15.

Example 2-15. `EXPLAIN` statement output with join

```
Hash Join  (cost=... rows=...)
  -> Seq Scan on customers ...
  -> Hash  (cost=...)
       -> Seq Scan on orders ...
```

This indicates the planner chose to sequentially scan orders, build a hash table (for joining on the `join` key), and then scan `customers` and probe the hash for matches. If an index was available on `orders.customer_id`, it might have instead chosen a nested loop with index scans on orders for each customer.

It's best practice to use `EXPLAIN` regularly on complex queries to ensure the plan makes sense (e.g., key filters are using indexes, join order looks reasonable, etc.). If not, consider rewriting the query or adding hints (in PostgreSQL you can sometimes influence via query structure, since direct

hints aren't supported natively). You may also discover that you need to adjust database configuration (for example, increasing `work_mem` for sort/hash operations, enabling parallelism, etc.) for better plans.

Cost-based optimization means the database is making decisions based on its statistics about the data. Key factors include table row counts, index statistics (value distributions, distinct counts), and more. For example, if a table is small, the optimizer might choose a sequential scan (full table scan) because reading the whole table is cheap; for larger tables, an index might be used if the filter is selective. In edge cases, the planner can make mistakes if statistics are outdated or if the query patterns are unusual.

A practical tool is `EXPLAIN ANALYZE` (in Postgres and some others); this tool actually runs the query and shows the runtime and row counts for each step. This is invaluable for troubleshooting because it shows where time is spent and if the estimates were off. For instance, if the planner estimated 1,000 rows but actually 1,000,000 came through, you know the stats might be stale or the estimate was bad, leading to a suboptimal plan.

For most users, it's not necessary to micromanage the plan and know the deep internals of the costing algorithm, but a few things are useful to remember:

- Optimizers rely on statistics (like row counts, histograms of column values, etc.) to make decisions. If those stats are outdated or inaccurate, the optimizer might pick a poor plan. (For example, if PostgreSQL thinks a filter will match 1% of rows but in reality it matches 50%, it might choose an index and nested-loop join strategy that ends up being slow. Keeping statistics fresh with the `ANALYZE` command or auto-analyze is important.)
- Common join strategies include:
    - A nested loop join iterates one table and for each row looks up matching rows in the other (good if one table is very small or the join is highly selective via index).

  - A hash join builds an in-memory hash table of one side (typically the smaller or the side with a filter) and then probes it with the other side (good for large tables with no usable indexes, assuming you have enough memory).
  - A merge join sorts both sides by the join key and then streams through them in order (good if both sides are already sorted on the join key or can leverage an index for sorted output). The optimizer will choose among these based on size estimates and available indexes. You might see these terms in an `EXPLAIN` plan output.

- The order in which tables are joined (join order) can greatly affect performance. The optimizer considers different join orderings: for three tables A, B, C, it could join A-B first and then C, or B-C and then A, etc. The search space grows with more tables, so optimizers use smart algorithms to trim possibilities. Generally, they try to join filters earlier to reduce row counts for subsequent steps.

BigQuery and Snowflake use cost-based optimizers as well, but their behavior differs due to their distributed cloud architectures. BigQuery does not contain user-facing indexes, so the optimizer focuses on minimizing data movement by choosing efficient distributed join strategies. Logical operations such as joins are often implemented using physical techniques like broadcasting a small table to all compute nodes or repartitioning (shuffling) data across nodes by join key. Because shuffles involve network and disk I/O and can dominate query cost, performance tuning in BigQuery is less about manual hints and more about shaping queries well: filtering early, keeping one side of joins small when possible, and taking advantage of partitioning and clustering. The execution plan is visualized as a directed acyclic graph (DAG) in the user interface, showing how data flows through stages such as scans, joins, and aggregations.

Snowflake works similarly but in the context of virtual warehouses: clusters of compute with local memory and disk. Its optimizer also chooses between

broadcast and repartition joins, while additionally leveraging metadata from micropartitions to skip unnecessary data automatically. Snowflake's Query Profile tool shows detailed resource usage and timing for each step. In both systems, traditional principles still apply: reading less data is faster than reading more, and smart schema design makes it easier for the engine to do the right thing.

The phrase "speaking the optimizer's language" means writing SQL that makes it easy for the optimizer to do the right thing. For example, using `EXISTS` versus `IN` versus `JOIN` might yield the same result but one form may help the optimizer more, given its current capabilities.

Always examine the plan for slow queries. It will tell you if it's scanning a huge table when you expected an index, or doing a suboptimal join order. Then you can adjust the query or schema accordingly.

Query Rewriting, Caching, and Materialized Views

Beyond successfully working with indexes or query planning, other optimization techniques are available for optimal query execution. These include query rewriting (either manual or automatic), leveraging caching of query results, and using materialized views to pre-compute expensive operations. Each of these can reduce the work needed for a query, either by doing some of it ahead of time or by reusing previous work.

The way you write a query can affect performance even if the query produces the same result. This is where query rewriting comes in. Query rewriting refers to transformations either done by the database or by developers to make a query faster without changing its output. The database optimizer itself performs many automatic query rewrites: for instance, it might flatten subqueries, combine filters, or transform an `OR` condition into a `UNION` of two queries, if that's expected to be faster. It also may push down predicates (filters) through joins or views. For example, if you define a view as a complex query and you query the view with a `WHERE` clause, the optimizer can often apply that `WHERE` inside the view query for efficiency.

As a developer, you can also rewrite queries in more optimized forms. In some databases, using a CTE versus an inline subquery might affect whether a filter can be pushed down. Or using an `EXISTS` versus an `IN` statement might change the performance, depending on how the query is executed. While an ideal optimizer will find the best plan regardless of query formulation, in practice certain patterns (like unnecessary `DISTINCT`s, or overly complex views on views) can confuse the optimizer or prevent it from utilizing indexes. Efficient query writing also involves avoiding Cartesian joins (also known as `CROSS JOIN`s), applying selective filters early, and reducing unnecessary computations such as redundant sorting or aggregation. A classic example is replacing a correlated subquery with a `JOIN`, as shown in Example 2-16.

Example 2-16. A query that can be optimized through query rewriting

```
SELECT
    e.employee_id,
    e.name,
    (SELECT d.department_name
     FROM departments d
     WHERE d.department_id = e.department_id)
FROM employees e;
```

This example uses a subquery to get the department name for each employee. A better approach is a `JOIN`, as shown in Example 2-17.

Example 2-17. A query optimized by rewriting

```
SELECT
    e.employee_id,
    e.name,
    d.department_name
FROM employees e
JOIN departments d ON e.department_id = d.department_id;
```

A SQL optimizer usually internally transforms the query within Example 2-16 into the query at Example 2-17, but not always: it depends on the database. Writing the query as an explicit join guarantees the most efficient plan. In cases of very complex queries, you might break them into multiple CTEs or temp tables so that each part can be optimized separately. PostgreSQL, for instance, sometimes struggles with extremely large queries

with many joins or subqueries; guiding the planner by using CTEs (especially marking them `MATERIALIZED` where appropriate) is one way to give the optimizer manageable chunks to optimize.

Another important concept to ensure smooth execution of your query is leveraging the cache on queries that run multiple times with the same output. Some systems and query engines have query result caching or automatic result reuse. For instance, Snowflake has a feature called the result cache: if you execute the exact same query twice within a certain time and the underlying data hasn't changed, Snowflake will just return the cached result immediately (this is a form of automatic caching that users benefit from by default). BigQuery similarly caches query results (for up to 24 hours) for identical queries on unchanged data.

From a user perspective, to key to leveraging caching, the key is query consistency: if you want results to be reused, you must issue identical queries. Minor differences in whitespace or using a literal value versus a parameter can sometimes make the query text differ, defeating a naive cache. Snowflake's cache is fairly robust (it even ignores comments and certain formatting differences), while BigQuery's is exact match. Also, any change to underlying data invalidates relevant caches. For BigQuery, if new data is added to the table or you modify it, the cached results of previous queries on that table are not used (to avoid stale data). In Snowflake, if data in the table changes, the result cache is invalidated for queries involving that table (because Snowflake enforces result cache consistency, it won't give you a cached result if the data has changed underneath). Many BI tools exploit this by issuing identical queries for filters that haven't changed. As a developer, be aware that such caching exists. Sometimes you'll notice a query that's normally slow suddenly runs instantaneously; that's likely due to a cached result.

Beyond caching strategies, another advanced approach is the use of materialized views (MVs), available in many databases. A materialized view is essentially a "cached" result of a query that is stored on disk like a table. The database can then use that pre-computed result to answer future queries, avoiding recomputation. For example, if you have a complex

aggregation that is expensive to compute on the fly, you can create a materialized view that holds that aggregation result. The trade-off is that you need to refresh the materialized view when the underlying data changes (either manually or via automation), so that it doesn't become stale.

Modern data warehouses have embraced materialized views for performance. BigQuery, for instance, introduced materialized views and even has the ability to automatically rewrite queries to use them: if a query can be answered by an existing materialized view (that is sufficiently up to date), BigQuery will transparently rewrite the query to use the materialized view's data. This means improved performance and lower cost, without the user having to manually query the materialized view. From a user perspective, you define the materialized view (with a query and refresh policy), and the optimizer does the rest when appropriate.

PostgreSQL supports materialized views, but they are not automatically updated. You must call `REFRESH MATERIALIZED VIEW` to bring them up to date (this can be done periodically or triggered as needed). Even with manual refresh, they can greatly speed up repeated complex queries by shifting expensive computation away from query execution time.

It is important to note, however, that a materialized view does not inherently reduce total compute on its own. In most cases, it redistributes compute, performing the heavy aggregation work during refresh rather than at query time. The overall benefit is lower latency, improved concurrency, and more predictable query performance. A net reduction in compute usage is typically achieved only when materialized views are designed as part of a broader, first-principles data modeling strategy, where expensive transformations are centralized, reused, and amortized across many downstream queries.

Suppose, for example, that you have a very heavy query that joins multiple large tables and aggregates them to produce a report (say, total sales by region by month). If that report is needed frequently, you could create a materialized view `monthly_region_sales` that preaggregates the data. Downstream reporting then becomes a simple `SELECT` from the materialized

view, which is much faster and more stable under load. The materialized view can be refreshed periodically to incorporate new data, aligning compute cost with data freshness requirements rather than with every query execution.

BigQuery and Snowflake both introduced materialized views with automatic maintenance. In BigQuery, you might create a materialized view on a query like `SELECT customer_id, SUM(amount) FROM purchases GROUP BY customer_id`; under the hood, BigQuery will keep this view in sync as new purchases come in (with some latency and certain limitations on what the view query can contain). If you issue a query that could be answered by the materialized view (even if you query the base table in your SQL), BigQuery's optimizer will transparently substitute the materialized view to answer it, improving performance and reducing cost. Similarly, Snowflake's optimizer will automatically rewrite queries to use a materialized view when applicable. For example, if you have a materialized view for "sales by region" and you run a query that asks for sales by region for a particular product, Snowflake might use a combination of the materialized view (to get sales by region overall) and further filter by product if possible, or forgo it if not applicable.

> **WARNING**
>
> Snowflake's materialized views currently have some limitations. They usually can't have joins and nested subqueries, so they're often used for pre-computing aggregations on single tables.

Materialized views trade storage and refresh cost for query speed. They take up space to store the results and time to maintain. If the base data changes frequently, the overhead of constantly refreshing the MV might outweigh the query benefit. But for relatively static or append-mostly datasets with expensive aggregation logic, they can be a game changer. In analytical databases, a common pattern is to materialize daily or monthly rollups of huge fact tables to avoid scanning the raw fact table for every

query. BigQuery even has scheduled queries that some people use similarly (to populate summary tables on a schedule), achieving a result akin to materialized views.

As an example of using a materialized view, imagine an ecommerce database with a transactions table of millions of rows (with columns `date`, `product_id`, `region`, `amount`). Analysts frequently ask: "What is the total sales by region by month?" A query for that might scan the entire transactions table and group it, which can become expensive. To overcome this, we can define a materialized view as shown in Example 2-18.

Example 2-18. Creating a materialized view

```
CREATE MATERIALIZED VIEW mv_sales_region_month AS
SELECT
  DATE_TRUNC(date, MONTH) AS year_month,
  region,
  SUM(amount) AS total_sales
FROM transactions
GROUP BY 1, 2;
```

This example tells the database to maintain a summary of sales per region per month. Suppose an analyst runs a query similar to Example 2-19.

Example 2-19. A query using a materialized view

```
SELECT
    year_month,
    region,
    total_sales
FROM mv_sales_region_month
WHERE region = 'EUROPE'
ORDER BY year_month;
```

It will directly read the pre-computed results, which might be just a few hundred rows from the millions of transactions. Even if the analyst isn't aware of the MV and queries the base table with the same aggregation, BigQuery or Snowflake could transparently use the MV. In PostgreSQL, the analyst would have to query the MV explicitly (since there's no automatic rewrite).

The difference in performance is huge: pre-computing turned a scan of millions of transaction rows into a scan of maybe 100 rows per region in the

MV.

In PostgreSQL, since there's no automatic query rewrite to MVs, you might build that logic into your application or use an intermediate view that points to either the MV or base table.

At the core of the comparison of query result caching versus materialized views, both are forms of reuse, yet:

- Caching is automatic and great for identical repeated queries, but it's short term and exact-match only.
- Materialized views are a deliberate optimization for a class of queries (e.g., any query that can use that aggregated result, even if not identical in text). They persist and can serve many slightly different queries (e.g., any filter on region could use the same MV of sales by region by month).

Using these techniques effectively can make a night-and-day difference in performance, especially in analytical workloads. As an advanced SQL user, consider these tools when you identify repeated heavy computations or patterns in your workload. Sometimes a small change (like adding a materialized view for a key metric, or ensuring a frequent query is reused exactly) can eliminate huge amounts of redundant computation.

Avoiding Common SQL Anti-Patterns

Just as important as knowing what to do for optimization is knowing what not to do. SQL anti-patterns are common query or schema designs that lead to suboptimal performance or maintainability. This selection highlights a few classic issues and discusses how to avoid them. Many of these will echo the strategies we've discussed, but it's useful to frame them as anti-patterns to make them easier to recognize and eliminate.

Selecting all columns (SELECT *) unnecessarily

Using `SELECT *` to retrieve every column can be convenient during development, but it is a bad habit in production queries. It can cause the

database to read a lot more data than needed (especially harmful in a columnar store like BigQuery where each unneeded column still incurs I/O). Also, using PostgreSQL as an example, if an index covers certain columns, a query that selects only those columns can use an index-only scan; `SELECT *` defeats this. Always select only the columns you truly need. This reduces I/O, memory usage, and network transfer. It also protects you from bugs if new columns are added (since your query won't inadvertently start pulling them in).

Not filtering data early

Failing to include a restrictive `WHERE` clause or joining overly broad datasets can lead to huge intermediate results. Always try to filter out irrelevant data as early as possible in the query. For instance, if you need recent orders, include `WHERE order_date >= '2025-01-01'` rather than retrieving all orders and filtering later in the application code. Early filtering means the database processes fewer rows downstream (joins, aggregations, etc., will have less data to handle).

Ignoring indexes (or not having the right index)

As discussed, not having an index on a high-impact filter/join column is a reliable way to get a slow query. But another anti-pattern is writing a query in a form that can't use an index. For example, wrapping a column in a function in the `WHERE` clause (`WHERE DATE(order_date) = '2025-01-01'`) will usually bypass an index on `order_date`, forcing a full table scan. Similarly, using an `OR` across different columns (`WHERE region = 'EUROPE' OR customer_id = 123`) might prevent index use because the condition is not simple. Sometimes rewriting such logic (perhaps as a `UNION` of two queries, one with each condition) can be optimized. Solution: ensure that relevant indexes exist, and write filters that allow those indexes to be used (e.g., use range conditions or equality on raw columns rather than transforming the column). In cloud warehouses, the analog is: use your partition key in filters. Not querying by the partition key (for example, you partitioned by date but your query filters by customer and not date) is an

anti-pattern because it won't prune data; try to structure data and queries such that partitioning/clustering is effective.

Misusing DISTINCT or UNION

Remember that `UNION` by default eliminates duplicates, which means an implicit sort operation to compare results. If you know the result sets are disjointed or you don't care about deduplication, use `UNION ALL` (which just appends results). The same goes for using `DISTINCT` as a temporary measure. Sometimes people add `DISTINCT` to remove duplicates caused by joins without realizing they might be masking an underlying issue (like a missing join condition) and adding a costly sort operation. Use `DISTINCT` only if you truly need to eliminate duplicates; otherwise it's doing unnecessary work.

Wildcard searches without full-text indexing

If a certain search such as `WHERE name LIKE '%Smith%'` is frequent, it shouldn't be used as an index on `name` in traditional B-tree indexing, since it is not recommended to rely on a SQL `LIKE` statement alone. As introduced earlier, a potential solution to overcome this includes using full-text search capabilities (like PostgreSQL's GIN indexes on `to_tsvector(text)`, as shown in Example 2-20 or dedicated search engines) or at least limiting leading wildcards (e.g., `name LIKE 'Smith%'` can use an index by doing a range scan on `name`).

Example 2-20. Using a GIN index

```
-- Use full-text search
CREATE INDEX idx_name_fts ON users USING GIN (to_tsvector('english', name));
SELECT * FROM users WHERE to_tsvector('english', name) @@
plainto_tsquery('Smith');
```

RBAR (row-by-agonizing-row) processing

This is a term for doing iterative row-by-row processing in SQL or in the application, which fails to leverage SQL's set-based power. An example would be a loop in application code that on each iteration runs a single-row

SQL statement (like updating one row at a time, as shown in Example 2-21, or querying one customer at a time).

Example 2-21. RBAR process updating a row at time

```
for id in ids:
    cursor.execute("UPDATE products
                    SET price = price * 1.1
                    WHERE id = %s", (id,))
```

This will execute potentially thousands of individual updates, incurring huge overhead (context switching, repeated parsing, etc.). The set-based approach, as shown in Example 2-22, would be a single statement.

Example 2-22. How to avoid a RBAR process

```
UPDATE products
SET price = price * 1.1
WHERE id IN (<list of ids>);
```

Within SQL, using cursors to loop through results and do logic row by row is similarly an anti-pattern if it can be replaced by a set operation. Aim to let SQL do work in bulk.

Not using prepared statements (when executing many similar queries)

If your application is executing the same type of query repeatedly (just with different values), not using prepared statements or parameterized queries is a performance anti-pattern. Without parameterization, each query is parsed and planned from scratch, and literal values might prevent query result caching or plan caching. Using a prepared statement, as presented in Example 2-23, allows the database to cache the plan.

Example 2-23. How to use the PREPARE statement

```
-- PostgreSQL PREPARE
PREPARE get_customer (INT) AS
SELECT * FROM customers WHERE id = $1;
EXECUTE get_customer(123);

EXECUTE get_customer(123);
EXECUTE get_customer(456);
```

This not only protects against SQL injection but can improve performance by reusing the compiled plan. Most databases, including PostgreSQL, cache the execution plan for a prepared statement and skip replanning on subsequent executions.

Large transactions or improper batching

Doing too much in one transaction (e.g., updating millions of rows in one go) can bloat rollback logs and lock too many rows, causing slowdowns or blocking. It can be more efficient to break large operations into chunks or use bulk loading tools for huge inserts. Similarly, commit regularly during an ETL process rather than as one giant commit at the end, unless you specifically need atomicity for all.

Ignoring query plan feedback

Always check `EXPLAIN` (and `EXPLAIN ANALYZE` when you can) before guessing at fixes. The plan tells you where time is spent: missing indexes, join order issues, hidden sorts, or filters that apply too late. Use that feedback to rewrite the query or adjust the schema, and then validate that the change actually improved the plan and runtime.

Neglecting to analyze/update statistics

This is more of a maintenance anti-pattern. If you bulk-load or significantly change data and don't update the table statistics (or disable automatic stats updates), the optimizer will be flying blind. Always ensure auto-vacuum/auto-analyze is running (for PostgreSQL) or the equivalent in other systems. In BigQuery and Snowflake, stats are managed for you (Snowflake gathers stats on load, BigQuery doesn't expose much but knows table size, etc.). But in systems where you can, you should manually trigger stat refresh after huge data changes.

By avoiding these anti-patterns, you'll prevent many performance issues before they happen. Efficient SQL comes down to a few repeatable habits: scan only what you need, keep work set-based (not row-by-row), and use the platform's strengths (indexes, partitioning, caching, materialized views)

instead of working around them. When you profile a slow query, check whether any of these anti-patterns are present. The fix might be straightforward and yield a big win.

Tuning in Cloud Data Warehouses: BigQuery Versus Snowflake

Cloud data warehouses like BigQuery and Snowflake operate on different principles from those used in traditional online transaction processing (OLTP) databases, so performance tuning involves a different mindset. You cannot add indexes, and you generally don't micromanage the query plan. Instead, you focus on data partitioning/clustering, distribution of work, and resource provisioning.

Both Snowflake and BigQuery are cloud native, massively parallel processing (MPP) systems, so understanding clustering, partitioning, or even how to leverage caching is only part of the picture. Modern cloud data warehouses introduce additional considerations beyond query optimization. This section focuses on complementary aspects of tuning in cloud environments, specifically concurrency management, workload isolation, cost optimization strategies, and query execution profiling. These topics build on the fundamentals already covered, focusing on how BigQuery and Snowflake handle resource management and performance monitoring at scale.

Concurrency and workload isolation

Achieving high performance isn't just about making individual queries fast; it's also about handling many queries concurrently without them competing destructively for resources. In a multiuser environment, a well-tuned system ensures that one heavy query doesn't stall others and that all users get fair (or appropriately prioritized) access to compute power. BigQuery and Snowflake take fundamentally different approaches to concurrency and workload isolation, reflecting their underlying architectures.

BigQuery operates on a serverless architecture that manages compute resources dynamically through virtual compute units called slots. These slots are automatically allocated to queries based on their complexity, with more intensive queries consuming more slots. Historically, BigQuery imposed a hard concurrency limit of 100 simultaneous queries per project, which posed challenges for high-traffic workloads. However, recent enhancements introduced a query queuing system, allowing excess queries to wait in line rather than fail outright. For flat-rate users, slot reservations can be configured to isolate workloads, ensuring specific teams or job types (e.g., ETL versus dashboards) don't interfere with each other's performance.

To better support performance and workload isolation, BigQuery allows users to define maximum concurrency targets for slot reservations. This ensures that each running query gets sufficient compute capacity by limiting how many queries can run simultaneously. Additionally, slot autoscaling enables the platform to expand or contract the number of slots dynamically, based on demand, reducing the need for constant manual tuning. Through this combination of shared resource management, queuing, and reservation flexibility, BigQuery balances elasticity and predictability in handling concurrent workloads.

In contrast, Snowflake uses dedicated virtual warehouses for compute, offering built-in workload isolation. Each warehouse runs independently, so resource contention is eliminated between workloads. For example, an ETL job can run on a warehouse size configured to large while reporting queries use a medium warehouse size, ensuring mutual performance stability. If a warehouse receives more queries than it can handle concurrently (based on its size and available CPU), Snowflake can automatically scale out using multi-cluster warehouses, launching additional clusters temporarily to handle the load and shutting them down once demand subsides.

Snowflake users are encouraged to match warehouse sizing to their workload type: scale up for heavier queries and scale out for high concurrency. Best practices include assigning dedicated warehouses per workload, enabling auto-suspend/resume to manage idle time cost-

effectively, and monitoring usage to rightsize configurations over time. Ultimately, while BigQuery focuses on dynamic sharing of compute slots with reservation-based isolation, Snowflake provides physical isolation by default and scales through user-defined compute clusters, each model offering different advantages depending on workload patterns and operational goals.

BigQuery concurrency and isolation best practices

To efficiently manage concurrent workloads in BigQuery, it's essential to implement best practices that optimize compute allocation and prevent resource conflicts. These strategies ensure consistent performance and throughput:

Leverage slot reservations

For flat-rate users, dedicate separate slot reservations to different workload types (e.g., ETL, BI) to prevent resource contention and ensure consistent performance for critical tasks.

Tune concurrency for reservations

Limit the maximum concurrent queries within a reservation to guarantee sufficient slots per running query, improving individual query performance. BigQuery will queue excess queries.

Utilize automatic query queuing

Rely on BigQuery's built-in queuing for handling temporary bursts in query volume, reducing the need for manual throttling. Monitor the queue to identify potential capacity bottlenecks.

Monitor slot utilization

Regularly check slot usage via `INFORMATION_SCHEMA` and cloud monitoring to understand capacity and identify the need for more slots or query optimization.

Snowflake concurrency and isolation best practices

For Snowflake environments, concurrency and performance depend on thoughtful warehouse design and workload isolation. These practices will help you get the most out of compute resources while avoiding contention and cost overruns:

Use separate virtual warehouses

> These are the primary method for isolation. Run distinct workloads (e.g., ETL, reporting) on their own, appropriately sized warehouses to eliminate resource contention. Even small, dedicated warehouses can be effective.

Enable multi-cluster for high concurrency

> For workloads with fluctuating user activity, use autoscaling multi-cluster warehouses to automatically add and remove compute resources based on demand. Set a maximum cluster limit to manage costs.

Rightsize warehouses

> Choose warehouse sizes based on workload characteristics (scale up for complex queries, scale out with multi-cluster for concurrency). Monitor performance to adjust sizing; online resizing allows for flexibility.

Isolate critical workloads

> For time-sensitive or high-priority tasks, dedicate specific warehouses to ensure they are not impacted by other processes. Consider scheduling warehouse starts and stops or temporary resizing.

Table 2-1 summarizes BigQuery and Snowflake differences in concurrency and isolation.

Table 2-1. How BigQuery and Snowflake differ in concurrency and isolation

Aspect	BigQuery (Shared Slots Model)	Snowflake (Dedicated Warehouse Model)
Concurrency handling	Queries run on a shared pool of slots (CPU units); dynamic concurrency is managed by BigQuery, which queues excess queries beyond the limit. By default ~100 concurrent queries per project were allowed (now flexible with queuing).	Queries run on a specific warehouse (cluster). Each warehouse can execute multiple queries, but too many concurrent queries cause internal queuing. Solution: add more clusters with multi-cluster warehouses for elasticity.
Workload isolation	Multi-tenant by default: all queries share the same pool of resources. However, flat-rate customers can create slot reservations to allocate separate compute capacity to different workloads, thereby isolating them. On-demand users rely on the global pool (less isolation, but simpler).	Intrinsically isolated: each warehouse is an independent compute cluster. Workloads on different warehouses do not contend with each other at all. This means isolation is achieved by design; you simply direct workloads to different warehouses (e.g., one for ETL, one for BI).

Aspect	BigQuery (Shared Slots Model)	Snowflake (Dedicated Warehouse Model)
Scaling strategy	Automatic, behind the scenes: BigQuery auto-allocates more slots to a query if available. Users can increase overall capacity by purchasing more slots. Concurrency scaling is automatic via queuing; with slot autoscaling, BigQuery will ramp up slots in increments during bursts. You cannot manually tweak how many threads a single query uses—the system decides based on the query plan.	User-controlled, but flexible: You choose warehouse size (XS, S, M, L, etc.) to scale up for more power per query. For scaling out, enable multi-cluster with min/max clusters. Snowflake will automatically start additional clusters when the single cluster is saturated by concurrent queries. Resizing warehouses (scale up/down) can be done at any time with no downtime, allowing manual adjustments as needed.

Cost-optimization strategies

One advantage of cloud data warehouses is the pay-as-you-go model: you pay for the storage and compute you actually use. That flexibility also puts the onus on you to control cost. In this section, we compare BigQuery and Snowflake in terms of pricing models and features that help control costs. You'll see that each platform provides knobs to balance performance and cost, such as auto-suspending resources when idle or choosing the right pricing plan for your usage pattern.

BigQuery: Navigating cost-effective analysis

BigQuery presents users with a fundamental choice in compute pricing: on-demand (pay-as-you-query based on data scanned) and flat-rate (predictable cost via reserved compute slots). Strategically choosing this model is essential, keeping in mind that switching isn't a simple toggle but a commercial decision with potential contractual implications.

On-demand provides advantages for fluctuating or lighter workloads, offering scalability without up-front commitment. However, cost can be unpredictable with large scans or usage spikes. To mitigate this, setting cost quotas at project or user levels is crucial to safeguard against budget overruns.

> **TIP**
>
> Using the dry-run feature feature before query execution not only allows for cost estimation but also helps validate the query structure without incurring any charges.

Flat-rate pricing, which involves purchasing dedicated slots, is ideal for consistent, high-volume workloads. It offers predictable spending and encourages efficient slot utilization. Features like flex slots (short-term commitments) and autoscaling slots (dynamic capacity adjustments) improve cost-effectiveness by minimizing idle resources.

Beyond the pricing model, effective cost control involves:

Minimizing data scanned

: Employing partitioning and clustering (as previously discussed) directly reduces on-demand costs and improves overall efficiency. Materialized views also reduce processing.

Leveraging caching

: BigQuery's automatic result caching prevents redundant scans for identical, unchanged queries, offering cost-free retrieval in on-demand scenarios.

Optimizing query efficiency

Focusing on efficient query design is key. As discussed on "Avoiding Common SQL Anti-Patterns", be cautious of complex joins, poorly written user-defined functions (UDFs), and deeply nested subqueries, as these are common sources of inefficiency.

Proactive monitoring and governance

Utilizing information schema and audit logging helps identify costly queries and usage patterns. Implementing alerts for high-spending queries enables timely intervention.

BigQuery cost optimization best practices

To effectively manage BigQuery expenses without sacrificing performance, you need strategies that reduce compute utilization, storage waste, and unexpected billing. These practices form a structured approach to cost efficiency, covering pricing models, resource safeguards, and query-level optimization:

Strategically choose the pricing plan

Align the pricing model with workload characteristics. Use on-demand for variability and flat-rate for sustained high usage, acknowledging the commercial commitment involved in flat-rate.

Implement robust cost controls

Employ quotas to prevent unexpected spikes and actively monitor spending through BigQuery's tools and billing exports.

Exploit flexible capacity

Utilize flex slots for temporary increases in demand, to avoid permanent over-provisioning.

Optimize data lifecycle

Manage storage costs by expiring or archiving infrequently accessed data. For data not modified in 90 days, consider leveraging long-term storage pricing, which offers reduced rates.

Snowflake: Architecting for cost-aware computation

Snowflake's cost structure builds on compute credits, which are consumed based on virtual warehouse runtime and size, alongside storage. The per-second billing (with a one-minute minimum upon startup) requires careful management, as frequent start-stop cycles from numerous small jobs can unexpectedly increase costs if not monitored closely.

One of the core features of Snowflake cost management is Auto-Suspend, which automatically shuts down inactive warehouses. Aggressively configuring this (often 1–5 minutes) is crucial for preventing unnecessary credit consumption. Auto-Resume ensures a seamless user experience by automatically restarting a warehouse upon new query submission.

Warehouse sizing is another critical point. Matching the warehouse size to the workload prevents overspending on unnecessarily large compute resources. Testing different sizes for performance compared to cost is recommended. For fluctuating concurrency, multi-cluster warehouses offer automatic scaling but also require careful maximum cluster configuration to manage expenses. Scheduled warehouse resizing (which incurs no cost and no downtime) can optimize costs for predictable usage patterns.

Resource monitors provide a governance mechanism by allowing the setting of credit quotas for warehouses or accounts, with options for alerts or automatic suspension upon reaching thresholds. Beyond cost control, multi-warehouse setups also enhance governance and auditability by isolating usage by team, department, or SLA.

Here are a few more cost-optimization ideas:

Strategic caching and result reuse

Snowflake's automatic data caching within warehouses and its cross-warehouse result cache for identical queries significantly reduce

redundant computation.

Efficient data retention

Managing Time Travel retention periods and utilizing zero-copy cloning can optimize storage costs. Regularly removing or tiering older data is also important.

Performance-driven cost reduction

Optimizing query logic to minimize execution time directly translates to lower compute credit consumption.

Snowflake cost optimization best practices

Snowflake bills per-second compute usage and storage retention, making it essential to manage warehouse behavior, data lifecycle, and query logic tightly. Here are best practices to keep spend predictable and aligned with performance needs:

Enforce auto-suspend

Minimize idle warehouse time by setting short inactivity timeouts (ideally under 5 minutes), while being mindful of potential cost inflation from excessive small job start-stops.

Implement budget governance

Utilize Resource Monitors to track and control credit consumption, preventing unexpected overspending.

Optimize warehouse allocation

Rightsize warehouses to match workload demands and consider scheduled resizing (at no cost or downtime) for predictable variations.

Wisely consolidate/separate workloads

Balance isolation for performance with consolidation of light workloads to avoid excessive minimum charges, while also leveraging multi-

warehouses for governance and audit clarity.

Monitoring and profiling query performance

Monitoring query performance is not just about identifying which queries are slow. It's about understanding why they are slow, and what actions can be taken to improve them. Both BigQuery and Snowflake provide robust tools for profiling query execution, but using those tools effectively requires more than just knowing they exist. This section offers a guide to identifying performance bottlenecks and acting on them in a practical, iterative way.

BigQuery: Monitoring and profiling queries

BigQuery's execution graph, available in the query UI, is often the first place to look when investigating performance. If the graph shows a stage dominated by a large shuffle or an imbalanced join, it typically points to skewed keys or insufficient filtering. In such cases, adjusting join keys, improving filters, or even preaggregating data before the join can dramatically reduce execution time.

The `EXPLAIN` and `EXPLAIN ANALYZE` commands offer deeper visibility. The former gives a static view of the plan, while the latter actually runs the query and reports execution metrics by stage. Although `EXPLAIN ANALYZE` incurs query cost, it's often worth it for understanding where most resources are consumed. For critical workloads, this detailed feedback can justify schema or logic changes.

Beyond single-query tools, the `INFORMATION_SCHEMA.JOBS` view is extremely useful for spotting trends. Don't look just at long runtimes; instead, focus on high slot-milliseconds (slot-ms) with low output volume. These often indicate excessive resource usage without corresponding value, commonly caused by Cartesian joins, repeated scans, or poorly filtered aggregations over large datasets, leading to high slot-ms consumption for minimal output.

BigQuery's Query Insights feature offers automated suggestions (e.g., recommending filters or partitions), which are helpful for directional

guidance. However, they should be interpreted with caution. They often miss context-specific optimizations and shouldn't be followed blindly.

Most importantly, performance tuning in BigQuery is iterative. After any changes such as new cluster keys, partition adjustments, or query rewrites, reprofile the query using the execution graph or `EXPLAIN ANALYZE` to confirm improvement. Monitoring over time is key: performance degradation often happens gradually as data grows.

Snowflake: Monitoring and profiling queries

Snowflake's Snowsight UI provides the Query Profile view, which is structured as a tree of execution steps. Each step includes metrics like time, rows processed, and relative cost. The "Most Expensive" label points directly to where attention is needed. For example, if that node is a table scan, and it processed an unexpectedly large number of micropartitions, it's a strong signal that clustering is ineffective or filters aren't appropriately selective.

Exchange operators in Snowflake deserve special attention. They represent data reshuffling across compute nodes. When many appear, or if one dominates the plan, it usually points to suboptimal join strategies, such as missing predicates or data skew. Consider rewriting joins, reviewing sort orders, or rethinking data modeling to reduce these steps.

If a query spills to disk, it's likely that the current warehouse size is too small or the query is too complex for the current setup. You might experiment with temporarily scaling up, then measuring cost versus performance before making the change permanent. Remember, resizing warehouses in Snowflake incurs no downtime or additional reconfiguration.

Warehouse load monitoring can guide longer-term adjustments. Sustained CPU usage above 80% may require scaling up or enabling multi-cluster mode to improve throughput. Conversely, consistent usage under 20% over time might indicate that the warehouse can be safely scaled down without performance impact.

Caching and result reuse also offer meaningful gains. When queries are stable and data hasn't changed, Snowflake can serve results instantly at no cost. To encourage this, avoid introducing unnecessary variations in queries. For example, don't use `ORDER BY` unless the ordering is essential for downstream logic.

While Snowflake integrates well with external tools, it's essential to cultivate internal understanding of patterns. Dashboards and alerts are helpful, but there's no substitute for regularly and directly reviewing the profile of high-cost or slow-running queries. At the same time, in large-scale environments, automated tools can play a key role in identifying performance outliers or cost trends that may not be obvious through manual inspection alone. The ideal approach combines both automated detection with human-informed diagnosis.

TIP

Monitoring and profiling aren't one-and-done. Schema changes, evolving data volumes, and new use cases can all introduce performance regressions. The most effective teams incorporate regular review cycles: track query cost and slot usage trends, validate improvements with profiling tools, and keep a close eye on warehouse behavior or slot utilization metrics.

As with all advanced SQL practices, profiling is a process of continuous feedback and refinement. Whether you're using BigQuery or Snowflake, the real value comes not just from seeing where a query is slow but also knowing what to change and then validating that it worked.

Achieving robust performance in BigQuery and Snowflake requires a combination of concurrency management, resource isolation, cost optimization, and rigorous profiling, all leveraging the cloud's elasticity to scale resources dynamically. Unlike traditional databases, these platforms offer a flexible toolbox to balance speed and cost through continuous adjustments: allocating slots/clusters for SLAs and optimizing queries/schemas for efficiency.

BigQuery versus Snowflake tuning isn't about which is "better," but about using each effectively. BigQuery emphasizes automated workload management and shared capacity with reservation-based isolation. Snowflake emphasizes predictable performance through dedicated warehouses and explicit control. Regardless of platform, consistent monitoring of platform-specific metrics (slots/bytes versus credits/warehouse) is essential, and careful SQL and schema design support most performance gains.

Summary

The cloud has changed the way we think about SQL performance. Traditional tuning practices still apply, but they must now be adapted to dynamic, distributed execution engines. In BigQuery and Snowflake, performance tuning is no longer just about indexes and query plans, it's also about understanding the compute model, monitoring usage patterns, and aligning cost with performance expectations.

Throughout this chapter, we explored how both platforms offer different levers for tuning. BigQuery emphasizes automatic scaling with shared slots and reservation-based control. Snowflake focuses on isolating workloads through dedicated, rightsized virtual warehouses. In both, strategies like query profiling, workload isolation, cost governance, and concurrency tuning are essential for maintaining performance at scale.

Tuning is not a onetime task. It's a feedback loop: observe, test, adjust. Tools like `EXPLAIN ANALYZE`, query profiles, and resource monitors help identify inefficiencies, but improvement comes from understanding how each platform executes SQL.

Mastering these techniques empowers SQL practitioners not only to reduce latency and cost but also to build systems that are robust, predictable, and ready for growth, setting the stage for the more architectural thinking that comes next.

Writing efficient queries is only part of the picture. As data volumes, stakeholders, and use cases grow, SQL practitioners must also consider how those queries fit into larger, modular pipelines and scalable system architectures. In the next chapter, we move from query optimization to the broader challenge of architecting SQL-driven solutions, where we explore how to design reliable data pipelines, organize SQL models effectively, and align infrastructure choices with evolving business needs.

Chapter 3. Architecting SQL-Driven Solutions

So far, we've traveled the SQL journey from its evolution and expanding capabilities in Chapter 1, to advanced query techniques and performance optimization strategies in Chapter 2. Now it is time to turn our focus to better understand how those queries fit into scalable, SQL-centered architectures. For mid- to senior-level practitioners, the role extends beyond simply writing queries, since it often involves shaping the design of entire systems and orchestrating data pipelines that rely heavily on SQL as their foundation.

In this chapter, we'll explore the principles behind architecting SQL-driven solutions, namely, how to build modular and maintainable data pipelines, decide between batch and streaming approaches, structure the layers of a modern data platform, and avoid common architectural issues. We will introduce concepts such as data mesh and data products for scaling data ownership across teams and data contracts as the formal agreements that ensure reliability, quality, and consistency between producers and consumers.

We will also address operational considerations that make a system *production-ready*, such as testing, observability, and continuous integration/continuous delivery (CI/CD) practices for SQL pipelines, and leveraging infrastructure-as-code for managing cloud resources. Whether you're working in a traditional RDBMS or a cloud native warehouse, this chapter will also help you think holistically about data systems with SQL as the central foundation, integrating both technical and organizational best practices.

But first, before we dive into the details of design principles and architecture, it's important to define what we mean by a data pipeline.

A *data pipeline* is a structured sequence of processes that ingests (batch and/or streaming), transforms, and delivers data from raw sources to its final destination, often a curated dataset consumed by analysts, applications, or ML systems. These pipelines may include ingestion mechanisms (like file drops, APIs, or message queues), transformation engines (like SQL, Python, or Spark), and delivery or serving layers. SQL-driven pipelines follow the same shape but rely primarily on SQL for transformation and modeling, and often for serving outputs as tables and views.

Figure 3-1 provides a high-level overview of a data pipeline; later sections reference this diagram as we explore the concepts.

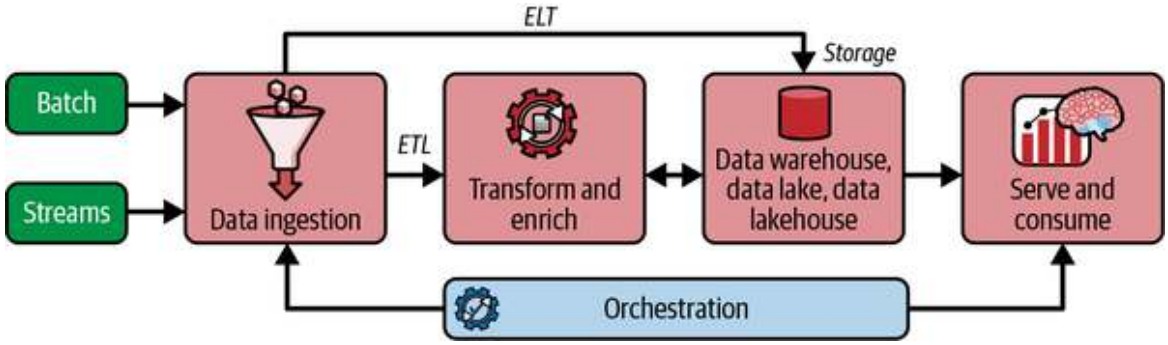


Figure 3-1. Conceptual overview of a modern data pipeline

Foundational Design Principles for SQL Data Pipelines

Foundational design principles are what separate ad-hoc SQL scripts from production-grade pipelines. These principles provide the discipline and structure needed to move from "getting it to work" to building pipelines that continue to deliver value over time. These principles include modularity, clear lineage, maintainability, idempotency, automation, and alignment with business SLAs. Just as important, they also include data contracts—formal agreements between producers and consumers of data that define what the data will look like, how often it will arrive, and what quality standards it must meet. Together, these principles shape SQL

pipelines that are not only technically correct but that are also resilient to change, transparent to stakeholders, and trustworthy across teams.

Designing Modular SQL Data Pipelines

A well-architected SQL pipeline is more than a list of queries executed in sequence: it's a set of composable steps with explicit inputs and outputs. Modularity keeps changes local, makes testing easier, and reduces fragility as pipelines grow.

Modularity and composability

Instead of writing one long SQL script that tries to handle every part of your data transformation process, it's better to break the work into smaller, reusable steps. Each part of the pipeline, such as loading raw data, cleaning and standardizing it, enriching with reference data, or joining and aggregating datasets, should be handled in its own SQL query, view, or table. Each of these steps should have clear inputs and outputs.

This approach has many advantages. Breaking your pipeline into smaller pieces makes it easier to read, test, and fix when something goes wrong. If a business rule changes or there's an error, you can go straight to the step that needs attention without changing the rest. These smaller pieces also work like building blocks: you can combine or reuse them to support new use cases or build additional datasets without starting from scratch. This saves time and makes your pipeline easier to maintain as things grow or change.

> **TIP**
>
> Think of building with LEGO bricks versus carving from a single block of wood. LEGO bricks are modular, reusable pieces with clear connections. You can easily swap a wheel or reuse it on a different model without affecting the rest. A single block of wood means any change requires recarving the whole, which is slow and hard to reuse. Your SQL pipeline should be like LEGO—small, independent, reusable components.

Modularity is not only about readability and maintenance; it also affects performance and cost. In distributed SQL engines, the way you structure transformations influences how much data is processed at each step. For example, when working with very large tables, filtering or aggregating datasets before joining them can significantly reduce the amount of data that needs to be handled later. Deciding where to break down the workflow, apply filters, or store intermediate results therefore impacts not just clarity but also efficiency and scalability.

Data lineage and dependency management

As your SQL data pipelines grow, understanding how different parts depend on each other becomes absolutely essential. In a large pipeline, it quickly becomes impossible to track all the connections between tables, views, and reports manually. For example, you need to know which raw tables feed into cleaned datasets and which are then used by reports or dashboards. Without that knowledge, changing one SQL query or table might accidentally break something further downstream, leading to broken reports, wrong results, or spending long hours trying to fix unexpected issues.

The best way to deal with all of these is to use tools that automatically show how data flows through your system, or at least to document those relationships clearly and consistently. This is called *data lineage*. Lineage gives you a clear view of where your data comes from and where it goes—which tables and columns are connected to others—so you can safely understand the impact of any change before making it. Lineage turns risky guesswork into confident decision-making and makes it much easier to improve, debug, or refactor your pipeline. Having strong data lineage and dependency tracking is key to keeping your pipelines reliable, flexible, and easy to manage.

> **TIP**
>
> Imagine you're a city planner managing a complex water system. Every pipe is connected. Without a detailed map (data lineage), changing one valve could accidentally cut water to a hospital. Similarly, without knowing how your SQL tables and reports depend on each other, even small changes can cause major issues.

Data lineage is also critical beyond day-to-day development. It plays a direct role in managing business SLAs, as the depth and structure of your pipeline influence how quickly data can move from source to consumption. Understanding how many upstream dependencies a dataset has helps teams assess delivery timelines, identify bottlenecks, and respond more effectively when incidents occur.

Lineage is equally important for governance and compliance. For example, if regulations require removing or anonymizing specific user data across the data platform, strong lineage tracking provides confidence that every dependent table, transformation, and downstream report is identified and updated. Without clear lineage, compliance becomes guesswork; with it, governance becomes systematic and controlled.

Maintainability and readability

Writing clean, well-organized SQL isn't just a nice-to-have. It's what keeps your data pipelines working well over time. As your project grows and more people work on it, messy or confusing code becomes a real problem. To make things easier for your future self and for your team, aim for consistency: use clear, predictable names for tables and columns, stick to the same formatting (like indentation and capitalization), and follow a shared SQL style throughout the project.

Comments are equally important. In data pipelines, good comments don't repeat what the SQL already makes obvious; instead, they explain *why* something is done. They capture business intent, clarify nonobvious design or compute decisions, document assumptions and constraints, and highlight performance trade-offs. As pipelines evolve, these contextual notes help

future contributors understand the reasoning behind the logic, reducing the risk of unintended side effects when changes are introduced.

One of the best ways to keep things organized is to layer your SQL models. We will detail this extensively in "Layered Architecture: Separating Storage, Transformation, and Orchestration", but generally speaking layering means splitting your pipeline into clear stages, like these:

Raw or staging tables

: These mirror the source systems, with little or no transformation.

Intermediate views or tables

: These apply most of the business logic: cleaning, standardizing, joining, and early aggregations.

Final tables or data marts

: These are polished, ready-to-use datasets built for reporting, dashboards, or machine learning.

This layered structure makes it much easier to follow what's happening at each step. If someone new joins the team, they can quickly understand how the data flows and where specific logic lives. It also helps avoid mistakes since you don't have to untangle a huge block of SQL just to fix a small issue.

Keep in mind that simpler pipelines that are easy to read and change are usually better than complex ones that try to do everything perfectly but are hard to maintain. Clean, readable SQL helps your pipeline stay flexible, scalable, and easy to build on as your needs evolve.

> **TIP**
>
> A clean SQL pipeline is like a well-organized kitchen. Labeled pipes, bundled wires, and accessible appliances make repairs easy and upgrades straightforward. A messy kitchen, with tangled wires and hidden pipes, makes even small tasks frustrating and risky. Well-structured, documented SQL code is simple to debug, change, and build upon.

Idempotency and rerunnability

In any data pipeline, things can go wrong: jobs fail, systems crash, or data doesn't show up as expected. So it's important to design each step of your pipeline to be *idempotent*. This means you can run the same operation multiple times, and it will always produce the same result. If something fails and you need to rerun part of your pipeline, you won't end up duplicating records, breaking logic, or corrupting data.

A simple, effective way to make a step idempotent is the delete-and-reload approach. Before you insert new data for a given time period or partition, you first remove whatever data is already there. Then you insert a fresh, complete version. Because you start clean each time, rerunning the step always gives the same outcome.

Another common technique is to use SQL `MERGE` statements (also known as `UPSERT` operations). These let you insert new records, update existing ones, or delete outdated rows, all in one statement. This means that even if the same `MERGE` runs twice, the final state of the data remains correct (new rows are only inserted once, updates are applied idempotently, and so on). In other scenarios, you might implement snapshotting of source data with timestamps or versioning, so that rerunning a load for a given date replaces the old snapshot with the new one consistently.

Designing with idempotency in mind makes pipelines much easier to recover when something goes wrong. If a step fails because of a network issue or bad data, you can just rerun it without worrying about side effects. This simplifies debugging, reduces risk, and builds trust in your data workflows.

> **TIP**
>
> Think of a thermostat controlling room temperature. Its job is to keep the room at 22°C. No matter how often it checks (runs), its goal is always the same single temperature; checking more often doesn't double the heat. An idempotent SQL step works the same way: run it once or ten times, the table result is always the same correct, consistent dataset, making retries safe.

Automation and orchestration

Running SQL scripts manually might work for quick tests or small projects, but production-grade pipelines need to run automatically and reliably. Manual processes are slow and error-prone, and they don't scale. To build trustworthy systems, you need to automate your pipelines and use an orchestrator to manage when and how things run.

Orchestration tools like Apache Airflow, Prefect, or even dbt's built-in scheduling features help you run complex pipelines by handling task dependencies, retries, and scheduling. Enterprise-grade relational databases like Microsoft SQL Server or Oracle Database are not missing these capabilities entirely: they typically provide job schedulers, stored procedure chaining, and dependency handling through native components (for example, SQL Server Agent). However, these mechanisms are database-scoped and primarily designed to orchestrate work inside the database engine.

Modern orchestration tools instead treat pipelines as first-class workflows that may span multiple systems, including databases, object storage, streaming platforms, and compute engines. Rather than embedding control flow inside individual jobs or stored procedures, they rely on *directed acyclic graphs* (*DAGs*) to describe when tasks are allowed to run and how they depend on one another. A task starts only after all of its upstream dependencies have completed successfully, enabling reliable retries, partial re-execution, and clear operational visibility across the entire pipeline. This topic is explored in more detail in "Orchestration and scheduling".

Automation ensures that your pipeline runs the same way every time, and it removes the need for someone to babysit the process. Automation also makes your pipeline easier to test and deploy across environments such as development, staging, and production. For any data platform to scale, orchestration and automation are essential.

> **TIP**
>
> Think of orchestration as the air traffic control of your data pipeline. Air traffic control doesn't fly the planes, but it does ensure every plane takes off, lands safely, and runs in the right order without collisions (errors). Your orchestrator is the control tower that keeps all your data tasks on track, consistently.

Service-level agreements and data freshness

Not every data pipeline needs real-time data, so trying to build for maximum speed can quickly lead to unnecessary cost and complexity. That's why it's important to work with your stakeholders to define clear service-level agreements (SLAs). These agreements help you understand exactly how fresh the data needs to be and how reliable it must be, so you can design the pipeline accordingly.

Start by asking the right questions:

- Does the data need to update every hour, or is once per day enough?
- Are there specific times when the data must be available?
- What level of delay or quality issue is acceptable before it becomes a problem?
- What is the data availability SLA from upstream systems?

Once you have those answers, you can make smart decisions about how the pipeline should be built:

Batch versus streaming

Choose batch processing when data needs to be updated every few hours or once per day. Use streaming when the data must be live or nearly real-time. We'll explore this trade-off in more detail in "Batch Versus Streaming: Choosing the Right SQL Pipeline Approach".

Scheduling

Choose the right job frequency to meet user expectations. No more, no less.

Monitoring

Set up alerts to catch problems before users do, based on the SLAs you've defined.

For example, a dashboard used in weekly business meetings might be fine with daily batch updates, which are simpler to manage and less expensive to run. Conversely, a fraud detection system or recommendation engine might need real-time streaming to be effective.

By setting SLAs up front and designing your pipeline to meet them, you avoid building something too complex or too expensive. You also ensure that the right users get the right data at the right time.

> **TIP**
>
> Delivering fresh data is like running a train schedule. Users don't just need the train to run; they need it to run on time, every time. If the train is late, trust is lost. Your data pipeline's SLAs are that timetable, ensuring data arrives on time at its destination (reports, dashboards) with the expected freshness, building user trust and enabling timely decisions.

By following these core principles—namely, building modular components, tracking dependencies with clear lineage, designing idempotent steps for safe retries, automating your workflows, and aligning freshness with real business needs—you create SQL data pipelines that are easier to manage,

scale, and trust. These aren't just nice ideas on paper. They're practical habits that help your data systems stay strong, even as data and other factors change.

In practice, this means keeping your SQL code organized in version control systems like Git and using tools like dbt that help you build modular, well-documented, and testable pipelines. It also means using lineage tools that show you how your data flows, so you can catch problems early and understand the impact of any change. And it means setting up orchestrators that not only schedule your jobs but also manage dependencies, handle failures, and send alerts, making sure everything runs reliably without needing constant attention.

When you apply these principles consistently, the result is a pipeline that's reliable, flexible, and easy to understand, not just for you but for everyone on your team. It becomes a solid foundation that can grow with your organization while still delivering clean, fresh, and dependable data.

Formalizing Data Agreements: Data Contracts

As data pipelines mature and multiple teams begin producing and consuming data, it becomes essential to establish clear agreements about the data itself. This is where data contracts come in. A data contract (see Example 3-1) is a formal agreement that explicitly defines the schema, format, and expectations of data exchanged between producers (the team or system providing the data) and consumers (the team or system using the data). It treats data as a product with a defined interface, much like an API contract in software engineering.

Example 3-1. A data contract

```
data_contract:
  name: customer_orders_contract_v1
  description: >
    Defines the schema, quality, and delivery expectations for customer
    orders data produced by the transactional system and consumed by the
    analytics platform.
  version: 1.0
  producer: "ecommerce_app.orders_service"
  consumer: "analytics_team.orders_pipeline"
```

```yaml
schema:
  table: orders
  primary_key: order_id
  fields:
    - name: order_id
      type: STRING
      constraints: [NOT NULL, UNIQUE]
    - name: customer_id
      type: STRING
      constraints: [NOT NULL]
    - name: order_date
      type: TIMESTAMP
      constraints: [NOT NULL]
    - name: status
      type: STRING
      allowed_values: ["pending", "paid", "shipped", "canceled"]
    - name: total_amount
      type: DECIMAL(12,2)
      constraints: [NOT NULL, ">=0"]

data_quality:
  expectations:
    - no_nulls: [order_id, customer_id, order_date, total_amount]
    - valid_enum: {status: ["pending", "paid", "shipped", "canceled"]}
    - currency: "total_amount must be >= 0"
  monitoring:
    freshness_sla: "Data available by 06:00 UTC daily"
    completeness: "At least 99.5% of transactions per day must be present"

delivery:
  cadence: "daily batch, delivered by 06:00 UTC"
  storage_format: "Parquet"
  location: "s3://analytics-raw/orders/"
  partitioning: "by order_date (daily)"

security_and_governance:
  pii_fields: [customer_id]
  handling:
    customer_id: "must be hashed in downstream gold models"
  access_control:
    roles:
      - name: analytics_read
        permissions: ["SELECT"]
      - name: finance_reporting
        permissions: ["SELECT", "EXPORT"]

lifecycle:
```

```yaml
contract_valid_from: "2025-07-25"
versioning_policy: >
  Minor changes (e.g. adding nullable columns) allowed with notification.
  Breaking changes require new contract version.
deprecation_policy: "Deprecated versions supported for 90 days."
```

Many of these elements map directly into SQL constructs. The schema definition aligns with table data definition language (DDL), where primary keys, `NOT NULL` constraints, and `CHECK` conditions are first-class citizens. Enumerated values resemble SQL domains, and partitioning strategies often translate into partitioned tables in warehouses like BigQuery or Snowflake. This direct connection means that contracts are not abstract governance artifacts; instead, they can be codified and validated through SQL itself, ensuring that producers and consumers share the same formal language for data integrity.

Data contracts shift the data quality paradigm from reactive to proactive. By specifying up front what the data will look like (schema, types, and so on), how often it will be updated, and what quality standards it must meet, you prevent breaking changes and unexpected surprises downstream. As a result, consumers can build with confidence that the data will conform to the contract. And if the producer needs to change something, they must version the contract or negotiate updates, rather than simply pushing a change that could break dashboards or models without warning.

Figure 3-2 shows the overall responsibility flow and negotiation loop around a data contract. Another way to look at this is that data contracts formalize the relationship between producers and consumers. Producers publish and adhere to the contract, while consumers build on its guarantees. As a result, a feedback loop ensures that contracts evolve with changing needs.

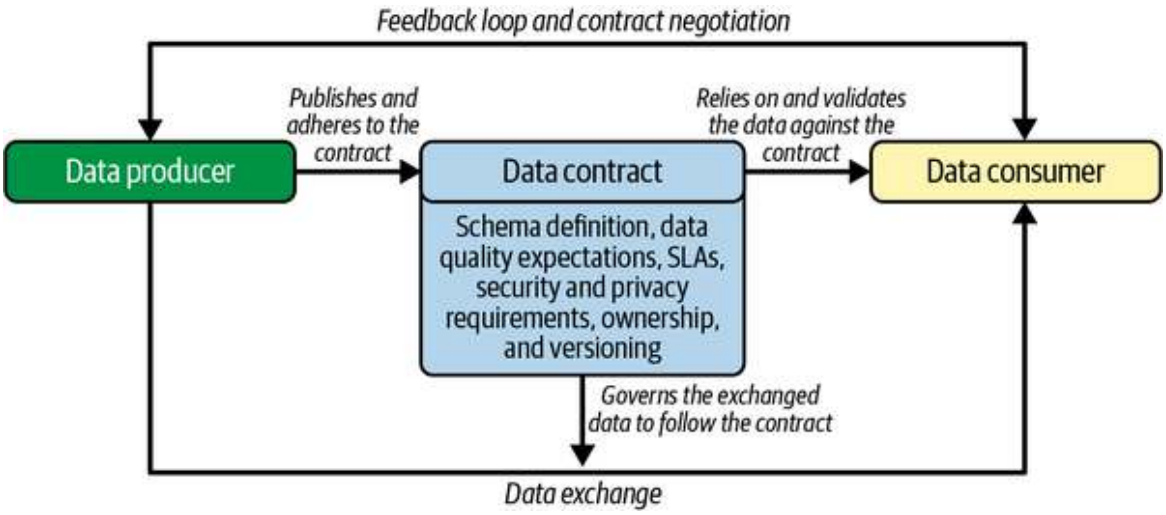


Figure 3-2. Data contract responsibility flow and negotiation loop

Some core elements typically covered in a data contract include the following:

Schema definition

The exact structure of the data, for example, field names, data types, acceptable values or ranges, primary keys, and relationships. This might be expressed in a formal schema format (JSON Schema, Avro, protocol buffers, and so forth) to allow automatic validation.

Data format and storage details

The data format of the data (JSON, CSV, Parquet, and others), character encodings (like UTF-8), file naming conventions or table naming, and partitioning strategy if applicable. This formatting ensures producers and consumers are literally speaking the same language about data and storage.

Data quality expectations

The expected levels of accuracy, completeness, and consistency, for example, which fields can/cannot be null, valid ranges or enumerations for values, uniqueness constraints, or other business rules the data is expected to satisfy. These expectations often tie into automated

validation checks (if a producer's output violates a rule, the contract can catch it).

Timeliness and SLAs

When and how often data will be delivered or updated (e.g., "by 8 a.m. daily" or "within 15 minutes of source capture"). This sets expectations on freshness similar to SLAs we discussed but as part of the formal contract.

Security and privacy requirements

How sensitive data is handled. For example, if certain fields contain Personally Identifiable Information (PII), the contract might require encryption at rest or mandate that data be masked or hashed for consumers. It can also define access controls (who is allowed to see this data) and compliance measures (GDPR, HIPAA, and others, if relevant).

Ownership and stewardship

Who is responsible for this data product on the producer side and perhaps a point of contact on the consumer side. Clear ownership means any issues with the data have a defined go-to person or team.

Lifecycle and evolution

How changes to the data contract are managed (versioning). For instance, if a new column is added or a column's meaning changes, the contract might stipulate that the new version will be published while the old version is maintained for X weeks to give consumers time to migrate. In production environments, updates to an active contract should also include formal communication and acknowledgment from downstream consumers. Tooling can validate structural changes, but it cannot automatically confirm that every dependent system has adapted safely. Requiring explicit notification and alignment helps prevent

unexpected SLA breaches or data quality issues during contract evolution.

By formalizing these details, data contracts provide a host of benefits. They prevent accidental breaking changes since a producer cannot just drop a column or change its type without updating the contract and informing consumers. They ensure a high level of data reliability and consistency, since both sides agree on what "good data" looks like. This simplifies data integration by reducing the need for constant ad-hoc communication between teams about schema and data meaning; much of that is codified in the contract.

Data contracts also improve compliance with regulations and internal policies, since requirements around privacy and governance are explicitly documented, promoting smoother collaboration across distributed teams, because the contract serves as a clear reference point for what is expected from each side.

Perhaps most importantly, data contracts fundamentally move data quality processes to *shift-left*, meaning issues are caught at the source or early in the pipeline rather than by downstream consumers. Instead of data engineering teams or analysts discovering that a source system changed a field and now dashboards are broken (a reactive scenario), the contract approach means the producer validates and communicates changes up front. This proactive behavior significantly reduces the fire-fighting mode that many teams find themselves in due to unforeseen data issues.

Managing data contracts with versioning allows controlled evolution. Just like an API, a data contract might need to change as business needs change; with version control, a new version of the contract can be introduced while the old version is still honored for a deprecation period. This way, downstream processes don't break overnight. Tooling is emerging around data contracts (for example, contracts defined in code or YAML that can be checked into Git and validated in CI/CD pipelines).

For SQL practitioners, these guarantees translate into stability across models and queries. Downstream transformations in dbt or ad-hoc SQL analysis rely on schemas that won't suddenly change, enumerations that remain consistent, and SLAs that align with reporting timelines. Versioned contracts further ensure that when evolution does occur, SQL models can migrate in a controlled way rather than breaking overnight. By embedding these expectations into the pipeline, data contracts allow SQL to remain a dependable interface for exploration, reporting, and analytics at scale.

Implementing data contracts is a key step toward treating *data as a product*. They formalize the interface between data producers and consumers, embedding quality, governance, and security expectations directly into the pipeline. For SQL practitioners, this means models and queries can be built on predictable, contract-backed tables, reducing defensive coding and ensuring transformations remain stable even as sources evolve. As a result, consumers know what to expect, producers know exactly what they are accountable for, and organizations gain both operational reliability and compliance confidence. Many modern data architectures, especially in decentralized or data mesh approaches, consider data contracts a foundation of reliable, scalable data sharing.

Architectural Patterns for Data Pipelines

How data moves, transforms, and becomes available for use is at the heart of any SQL-driven solution. The architecture you design isn't just about where data lives but also how it flows between systems and layers, and what integration patterns you adopt to balance scale, freshness, and complexity.

Within architectural patterns for data pipelines, we'll explore the major patterns that shape SQL pipelines: from batch versus streaming decisions, to transformation strategies like ETL and ELT, to emerging paradigms such as zero-ETL and secure data sharing. We'll also step back to consider broader architectural approaches like data mesh, where the organization of teams and ownership plays as important a role as the technical design.

Together, these patterns provide a toolkit of options that you can adapt to your specific context, workload, and organizational maturity.

Batch Versus Streaming: Choosing the Right SQL Pipeline Approach

One of the most important decisions you'll make when designing a SQL-based data pipeline is whether it should run in batch or streaming (often called real-time) mode. Both approaches can be built using SQL-centric tools, but they come with very different trade-offs. The choice affects your overall system design, the tools you'll need, and how you balance data freshness, cost, and complexity. Understanding these differences early on is key to building a pipeline that delivers the right data at the right time, without overcomplicating your stack.

In *batch processing*, data is collected over a set period, such as hourly, daily, or weekly, and then processed in large chunks. This is how most traditional SQL pipelines work. For example, a nightly job might extract all of yesterday's transactions and run several SQL steps to, ultimately, update tables that are used for machine learning or advanced analytics processes. Because you're working with a complete dataset, batch jobs are often simpler to understand and easier to troubleshoot.

Batch pipelines are usually easier to design and maintain as well. You can use familiar tools like relational databases or cloud data warehouses, write your logic in standard SQL, and run heavy transformations or aggregations efficiently. These systems are great at handling large amounts of data in a single go, and they make it easier to ensure that results are complete and consistent. If something goes wrong, it's often enough to fix the issue and just rerun the batch.

However, the main trade-off of batch processing is data latency. If your batch runs only once per day, users won't see new data until the next day. You can improve freshness by running more frequent batches, such as every hour, but that can put more load on your systems and make scheduling more

complex. That's why it's important to match your batch frequency to the actual needs of your users and use cases.

Streaming pipelines, on the other hand, process data continuously, often as soon as it arrives. Instead of waiting to collect data over a period, as in batch processing, streaming systems handle each new record or small group of records right away. This immediate processing capability makes it possible to gain near real-time insights, which is particularly ideal for use cases like live dashboards, fraud detection, or continuous system monitoring where timely information is critical.

In the SQL world, streaming usually relies on tools that support SQL queries on real-time data. This includes technologies like ksqlDB, Flink SQL, and Spark Structured Streaming. These systems allow you to write SQL queries that work on constantly changing streams of data. Some cloud data warehouses, like BigQuery and Snowflake, also offer streaming capabilities, such as micro-batch inserts and frequently refreshed materialized views. These features blur the line between traditional batch and real-time processing, creating a flexible range of options, depending on how fresh your data needs to be.

The biggest advantage of streaming is low latency, as data becomes available to users almost immediately. But this power comes with more complexity in the overall solution. Streaming systems need to handle late-arriving data, out-of-order events, and continuously running queries. Ensuring idempotency (no duplicates) and exactly-once processing is much harder in streaming than in batch jobs. Also, not all SQL logic works the same way with streaming data. For example, aggregations often require defining time windows to group events over a certain period. A common pattern is the *sliding window*, where metrics are continuously recalculated over a moving time range (for example, the last 10 minutes or last hour), updating as new events arrive. This allows streaming pipelines to provide continuously refreshed insights without waiting for a fixed batch boundary.

It's also helpful to distinguish between *real-time* and *near real-time*. Real-time systems typically produce outputs almost immediately after events

occur. For example, a fraud detection system might evaluate a transaction within seconds of its submission. Near real-time systems operate with short but defined delays, such as generating updated metrics every few minutes or recalculating the last hour of activity at regular intervals. Both rely on streaming principles, but the acceptable latency and transformation complexities differ depending on the business need.

As we've noted, when choosing between batch and streaming for your SQL pipeline, it's important to weigh the trade-offs across several key areas, from consistency and latency to tooling and cost. Table 3-1 summarizes the main differences between the two approaches to help you decide which best fits your needs.

Table 3-1. Comparison of batch and streaming SQL pipelines

Category	Batch pipelines	Streaming pipelines
Consistency and timeliness	Provides consistent, complete snapshots (for example, daily totals); simplifies SQL logic. Ideal for full-dataset transformations.	Delivers low latency (near real-time). May involve late-arriving or out-of-order data, requiring more complex correction logic.
Tooling and ecosystem	Uses familiar tools like traditional relational databases, cloud data warehouses, and ELT or ETL orchestration (for example, Python, dbt).	Requires specialized, stream-ready tools such as Apache Kafka (with ksqlDB), Apache Flink (with Flink SQL), or Spark Structured Streaming. Cloud native options like BigQuery (with Pub/Sub ingestion) or Snowflake (with Snowpipe) also offer streaming capabilities.

Category	Batch pipelines	Streaming pipelines
Cost and performance	Generally lower cost when scheduled smartly; highly efficient for processing large volumes of data in a single run.	Runs continuously, which can lead to higher infrastructure costs. Needs careful tuning and monitoring to avoid unnecessary compute expenses.
Middle ground	Micro-batching (for example, every 5–15 minutes) offers a balance of lower latency with manageable operational overhead.	True streaming provides second-level freshness but introduces significant operational complexity and higher infrastructure demand.

While batch and streaming are often presented as either-or decisions, modern data platforms increasingly combine both. This hybrid approach allows teams to take advantage of the strengths of each, using batch for accuracy and scale, and streaming for speed and responsiveness.

A well-known example is the *Lambda architecture*. In this model, represented in Figure 3-3, a streaming "speed layer" delivers real-time updates for urgent use cases, while a separate batch layer processes complete, accurate datasets in the background. The results from both layers are merged into a final output that users consume. This approach adds complexity, but it allows you to balance freshness with reliability, for

example, using real-time SQL pipelines to power a live dashboard while still running daily batch jobs to generate accurate historical aggregates.

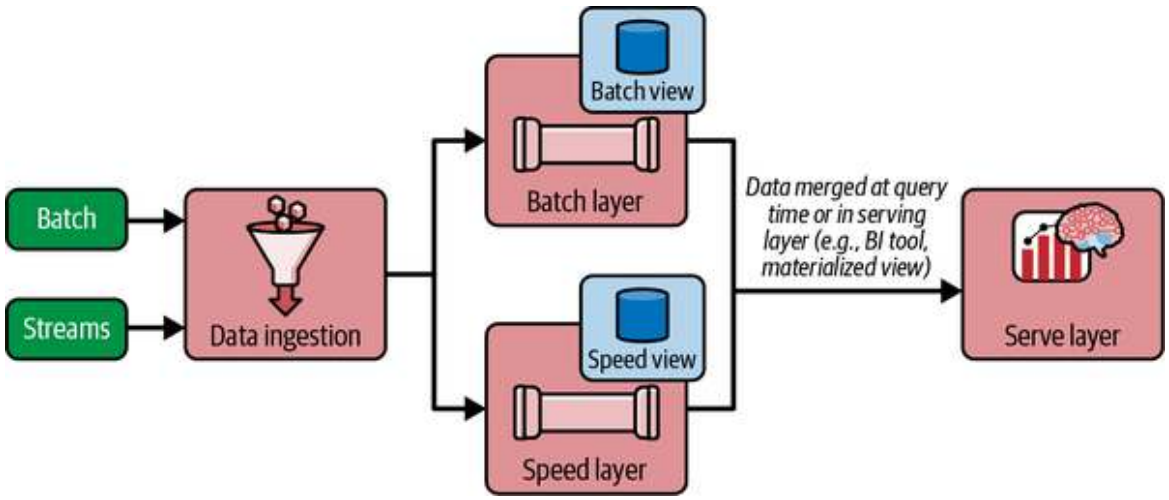


Figure 3-3. Lambda architecture data flow

An alternative is the *Kappa architecture*, which simplifies things by relying solely on streaming. Instead of maintaining two separate layers, all processing is done on a single, real-time data stream. If historical reprocessing is needed, such as replaying events due to a logic change, the system reads from a durable, append-only log or storage layer that retains the original stream. Kappa is often easier to operate and understand, especially when built on top of tools that support SQL on streaming data, such as Flink SQL (from Apache Flink), ksqlDB, or Spark Structured Streaming. Figure 3-4 highlights the Kappa architecture data flow.

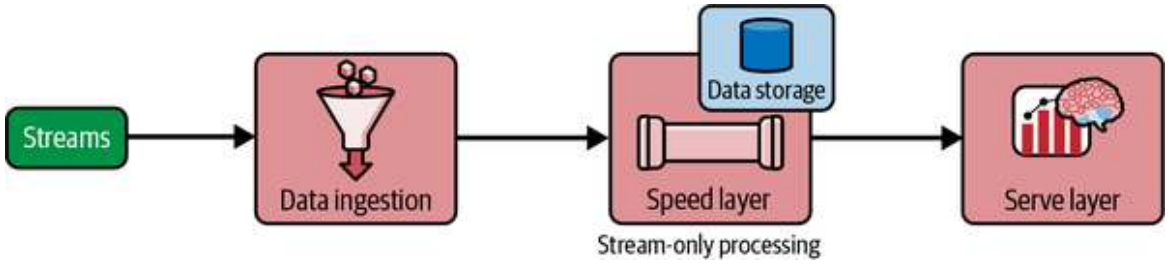


Figure 3-4. Kappa architecture data flow

Even if these models originate from distributed systems thinking, they are useful mental models for modern SQL pipelines. The Lambda architecture separates batch and real-time processing into two layers. The Kappa architecture simplifies this by relying solely on streaming, with historical

reprocessing handled by replaying data from an append-only log. In practice, most teams blend elements of both, depending on their requirements for freshness, scale, and complexity. No matter which approach you adopt, SQL is the common foundation. In batch pipelines, SQL expresses transformations over entire datasets, whereas in streaming pipelines SQL is extended with concepts like windows and time-based joins to handle data as it arrives. Choosing between batch and streaming is therefore more than just a system design decision. It also determines how your SQL code is structured, how queries are optimized, and how models evolve over time. In Chapter 5, we'll bring these concepts into practice by showing how SQL is used to ingest and transform data in both batch and streaming pipelines, highlighting the differences side by side.

Transformation Patterns: ETL, ELT, and Streaming

While architectural patterns like Lambda and Kappa configure the overall data flow, another important design decision happens within each layer, particularly in batch pipelines: should data be transformed before it's loaded into your system (ETL), or after it's already stored in the warehouse (ELT)? This distinction affects both pipeline complexity and how SQL is used in your transformation logic.

In traditional extract, transform, load (ETL), data is extracted from source systems, transformed using external tools or code, often in Java, Python, or even Spark, and then loaded in its final form into a database. This approach was common when databases were limited in compute power, and when transformation logic lived mostly outside the warehouse. ETL is still used today, particularly in batch-oriented pipelines and within the batch layer of a Lambda architecture, where transformations are applied before data enters the data platform.

For SQL practitioners, this often meant less direct involvement in the transformation layer, since most of the heavy lifting happened outside the database. SQL tended to be used only at the very end for loading or reporting, which limited visibility and made logic harder to maintain collaboratively.

Figure 3-5 shows a typical ETL pattern, where transformation happens before the data is stored.

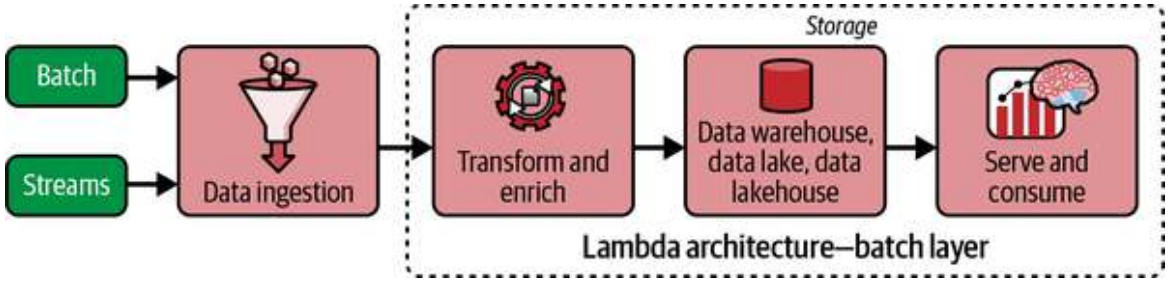


Figure 3-5. ETL pattern, common in Lambda batch layers

ETL can be efficient when source data volumes are smaller or when using an external engine optimized for transformation, but it has downsides. One major drawback is that the raw source data is never fully replicated to the warehouse—if you later need a piece of information that was dropped or changed during the ETL process, you might have to re-extract from the source. ETL processes can also be complex to maintain, as they often involve custom scripts or tools outside the database, and scaling that external transformation engine can be challenging.

However, with the rise of cloud native data warehouses and SQL-first transformation tools like dbt, the extract, load, transfer (ELT) pipeline has become the modern standard. A key enabler of this shift has been the dramatic reduction in storage costs over the last two decades, making it economically feasible to retain large volumes of raw data and transform them later inside the warehouse. In ELT, raw data is loaded into staging tables or schemas with minimal modification. All transformation logic, such as joins, data cleaning and enrichment, or even aggregations, is then expressed in SQL and executed inside the warehouse. This approach is not only more performant on modern platforms, but it also keeps transformation logic close to the data, making it easier to test, debug, version, and collaborate on.

Figure 3-6 illustrates the ELT approach, where data is loaded first into storage and then transformed inside the storage system.

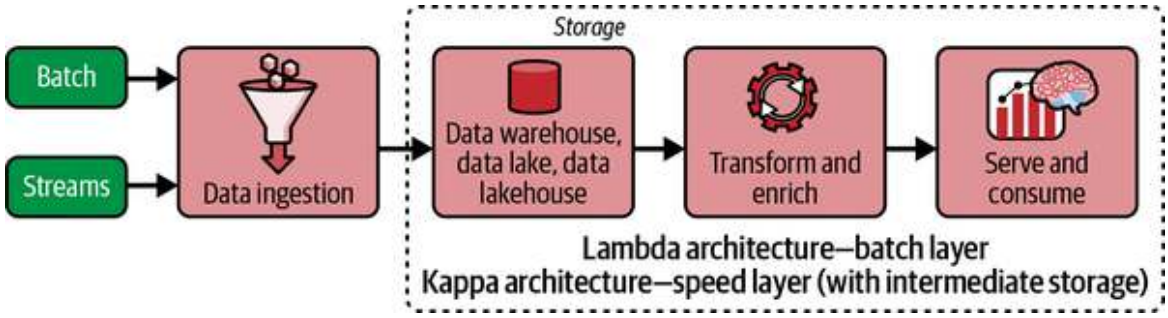


Figure 3-6. ELT pattern, common in modern SQL workflows

The ELT approach leverages the fact that modern warehouses (Snowflake, BigQuery, Redshift, and so on) can handle large-scale transformations very efficiently with their compute engines. By loading data first, you also retain the raw data (Bronze layer) and can always revisit or reprocess it with new logic. ELT simplifies pipelines by using the warehouse as the single transformation engine—no need for an external ETL server for most use cases. It also aligns well with modular SQL design: you can create staging tables (raw data), intermediate transformation views, and final tables all in SQL, orchestrated by a tool like dbt, a structure that maps well to the layered architecture patterns introduced earlier and explored in more detail in "Layered Architecture: Separating Storage, Transformation, and Orchestration".

Streaming transformations can be seen as a real-time extension of ELT/ETL. In streaming, data is continuously extracted from sources and immediately transformed on the fly as it's loaded into a system or as it flows through. There are two flavors: one is stream-first pipelines (such as Kafka → Flink jobs → database) where transformation happens in a streaming processor before final storage—akin to continuous ETL. The other is in-database streaming features (such as materialized views that update in near real-time or streaming ingestion with SQL triggers), which blur into ELT, just on a micro-batch scale. Figure 3-7 illustrates a streaming SQL pattern: data flows through continuous queries, and results are incrementally updated. No explicit batch or persistent storage is required.

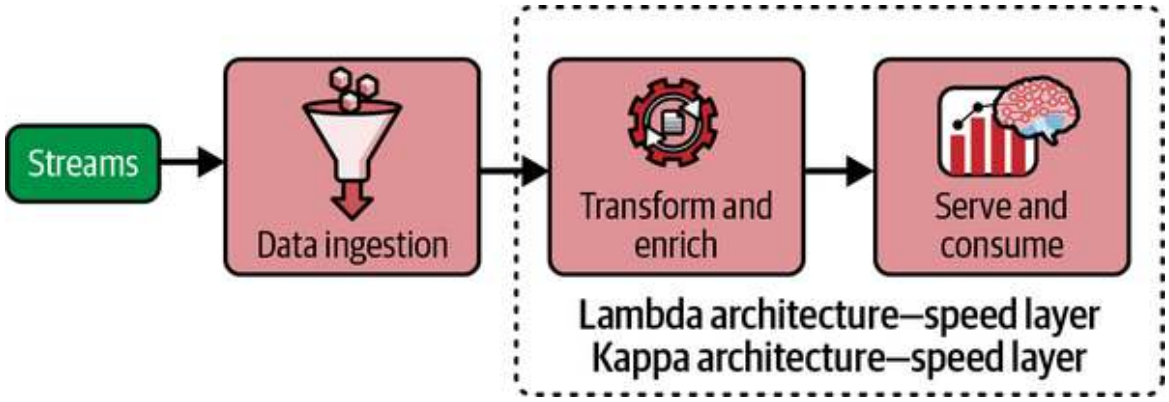


Figure 3-7. Streaming SQL pattern used in real-time pipelines (speed layers of Lambda or Kappa architectures)

In practice, the choice between ETL and ELT is largely about leveraging the strengths of your infrastructure. ELT is preferred in cloud native architectures due to scalability and simplicity. It keeps all transformation logic in SQL and benefits from the warehouse's power. ETL might still appear in cases with legacy systems or when using specialized transformation engines. Streaming is dictated by latency requirements. It's not a separate category as much as an additional consideration that can be applied (you could do streaming ELT, for example, by continuously loading raw data and then continuously transforming it).

Most organizations today use a mix: ELT for the bulk of pipelines, and some streaming for specific real-time needs. Understanding these patterns is key for SQL practitioners: ETL reduces SQL to a supporting role, while ELT places SQL at the center of all transformation logic. Streaming extends this SQL-driven approach into real time, where continuous queries and incremental updates become part of the toolkit. This framing not only guides platform and tool selection, but it also shapes how SQL code itself is written, shared, and maintained across the organization.

Beyond ETL/ELT and Streaming: Zero-ETL and Data Sharing

ETL, ELT, and streaming patterns describe how data is moved and transformed, but an emerging new set of integration patterns challenges the need for traditional pipelines altogether. These approaches, often referred to

as *zero-ETL* and *data sharing*, represent a shift in data architecture by minimizing or eliminating data movement between systems.

The zero-ETL approach is less a specific technique and more a design principle. It enables operational data to become available in an analytical platform, such as a data warehouse or lakehouse, with minimal manual pipeline code. This is typically achieved through built-in, automated syncs, often managed by the cloud provider. Instead of building and maintaining traditional extraction and load jobs, developers configure a connection (sometimes called an integrated sync or zero-ETL bridge) and focus on writing SQL transformations directly in the destination system.

For SQL practitioners, this shifts the center of gravity entirely toward the database or warehouse. Instead of stitching together pipelines in Python or Spark, analysts and engineers can work directly in SQL from the start, applying transformations on synced or federated tables as if they were native.

Despite the name, zero-ETL doesn't mean transformation disappears. Under the hood, extraction, loading, and even some transformation still occur, but they're abstracted away. This simplification reduces infrastructure complexity, lowers latency, and frees teams from maintaining pipeline logic.

A concrete example is the concept of lakehouse systems that combine features of data lakes and warehouses, or products like Azure Cosmos DB and Synapse Link, or Snowflake's external table integrations, where data from one system is queryable in another with minimal setup. Figure 3-8 illustrates how zero-ETL brings source data into the warehouse with minimal pipeline logic.

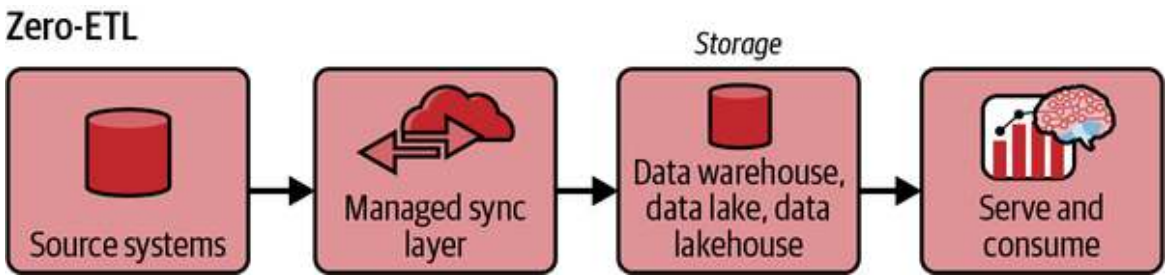


Figure 3-8. With zero-ETL, operational data is synced into the warehouse through a built-in integration, eliminating the need for manual pipelines

Similarly, *data sharing* is a pattern where data is directly and securely shared across accounts or organizations, often within the same cloud data warehouse ecosystem, without copying or duplicating the data. An example is Snowflake's data sharing feature or Databricks' Delta Sharing protocol. In these setups, consumers query the shared data directly from the provider's platform using SQL, as if it were native to their own environment.

This model allows for fine-grained access control and eliminates the need for pipeline development or file-based transfers. A single dataset can be shared with multiple external users, systems, or teams, each receiving access only to what they're authorized to see.

Figure 3-9 illustrates how a data provider can expose selected tables or views to various consumers through a secure sharing interface.

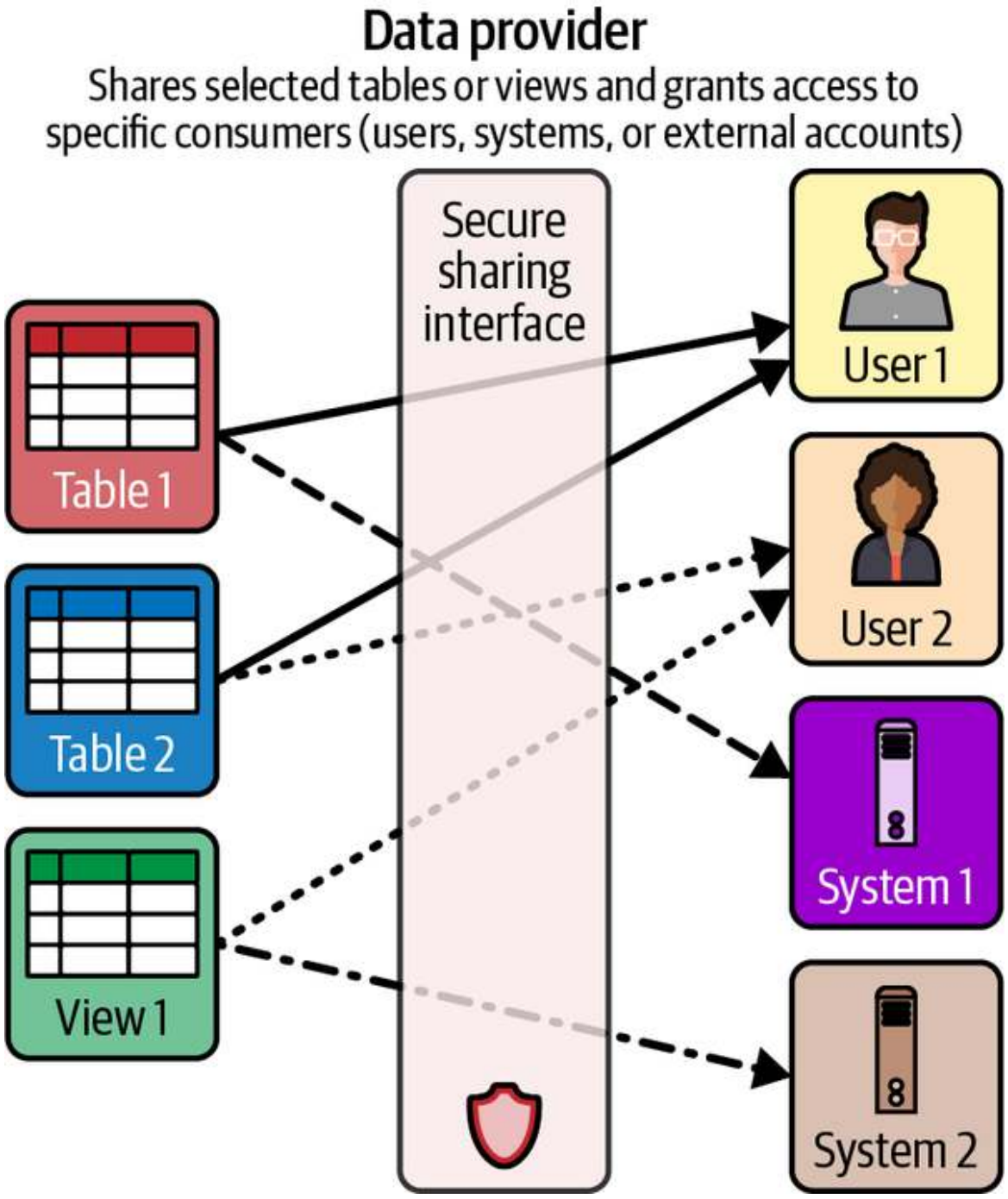


Figure 3-9. With data sharing, a data provider securely shares specific tables or views with multiple consumers, who query them directly from the source

Data sharing is relevant for ensuring consistency (everyone sees the same "single source of truth" data, rather than multiple copies that could diverge) and for efficiency (avoiding large data transfers and storage duplication). It's increasingly used for data marketplaces or intercompany analytics

where, for instance, a company can provide customers with direct read-only access to relevant data in near real-time.

Both zero-ETL and data sharing shift the emphasis from building pipelines and toward declarative, SQL-driven access. In practice, this means SQL becomes the primary interface: with zero-ETL, transformations happen directly in the warehouse; with data sharing, external datasets can be queried in SQL without ever moving or duplicating them. For SQL practitioners, these patterns reduce operational overhead while expanding the range of data they can analyze, making it easier to collaborate across domains and organizations in near real-time.

Data Mesh and Data Products

Both the classic and emerging patterns we've discussed can be seen through the lens of an "all-inclusive" architectural paradigm called *data mesh*. Data mesh is an approach, illustrated in Figure 3-10, to organizing data architecture (and data teams) that treats data as a product, emphasizes domain ownership of data, provides a self-service data infrastructure, and implements federated governance across the organization. In other words, it's not just a technical blueprint but also an organizational one.

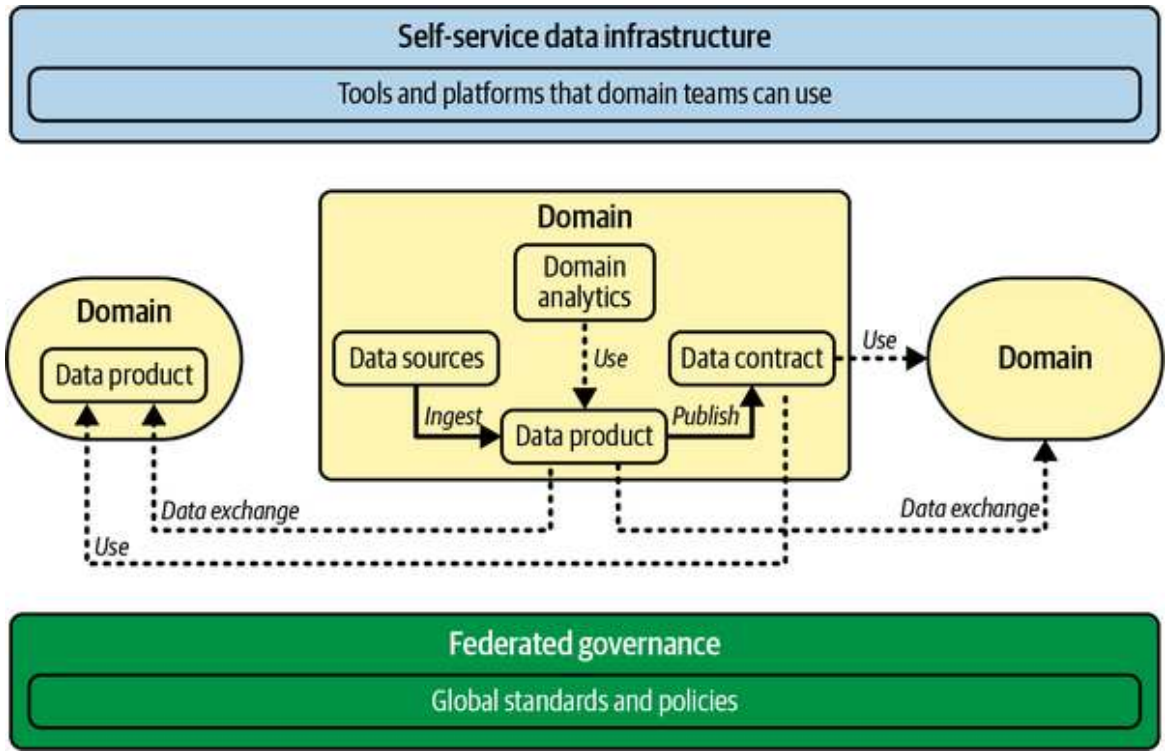


Figure 3-10. Data mesh architecture

In a *data mesh*, responsibility for data is decentralized: each domain or business unit (for example, Sales, Marketing, Finance) owns its data pipelines and datasets as products, delivering them to others in the company who need them. *Data as a product* means those teams treat their datasets with the same care and quality as a software team treats a customer-facing product, with clear documentation, quality assurances (tests, contracts), and an interface for users, most commonly through SQL queries but also via APIs or data shares when needed. Consumers of a data product are like customers: they should have a good experience using the data, and their needs drive the product features.

To enable this, data mesh relies on a self-service data platform where a central infrastructure team provides tools and platforms (for ingestion, transformation, storage, and so forth) that domain teams can use easily, without needing deep expertise in the underlying tech. This lowers the barrier for each domain to create and manage data products. For instance, a platform team might provide templated pipelines, standardized CI/CD

tooling for data (so that each team can deploy tests and contracts easily), a data catalog for discovery, and so forth.

The fourth principle, federated computational governance, ensures that while domains operate independently, they adhere to some global standards and policies. Instead of a central team manually enforcing rules, governance is embedded into the platform and automated. For example, common security controls, data cataloging requirements, and data contract enforcement can be built into the tooling. A federation of stakeholders (from each domain and central data governance) collaborate to define these standards, balancing autonomy with compliance. For SQL practitioners, this often means standardized testing frameworks, reusable dbt macros, or automated SQL validations that enforce governance rules consistently across domains.

In practice, implementing a data mesh means that SQL-driven solutions might be owned by different teams but made interoperable. One domain's SQL pipelines produce a data product (say, a cleaned Customer table), which is made discoverable by and shareable with others, possibly via a data sharing mechanism or an API. Another team can use it as a building block in their own SQL models, with a data contract ensuring the schema and quality. The organization ends up with a distributed, but coordinated, data architecture.

It's worth noting that data mesh is a relatively new concept (championed by Zhamak Dehghani in 2019) and not a fit for every organization since it adds complexity and assumes a certain scale of data and independence of teams. But its core ideas are influencing modern data architecture thinking. If your SQL platform is growing beyond a single team's scope, data mesh principles can guide how to scale out responsibly.

To tie this back to our chapter's focus: architecting SQL-driven solutions isn't only about technical patterns; it's also about people and processes. Data contracts, for example, are not just technical schemas; they represent agreements between producer and consumer teams (an idea very much in line with data mesh's distributed ownership). The need for strong

governance and standardized infrastructure becomes greater as you decentralize. The ultimate goal is to enable faster innovation (teams can move quickly with their own data) without sacrificing consistency and quality across the company.

Data mesh provides a vision of a large-scale SQL (and general data) architecture where data is treated as a first-class product, owned by domains, supported by common platforms, and governed in a federated way. It connects to modular pipeline design (each product pipeline is independent), emphasizes data contracts (clear interfaces), requires robust automation and CI/CD, and relies on layered architecture and anti-pattern avoidance to scale safely.

As your data platform evolves, these principles help keep SQL-driven solutions scalable organizationally as well as technically. By treating SQL models and pipelines as interoperable products, teams can innovate independently while still ensuring their SQL code integrates smoothly with others across the company.

Architectures for Scalability

Imagine it's 2014. Your company has just launched, powered by a simple, uncomplicated data stack: a single PostgreSQL instance running on a modest cloud virtual machine (VM). It handled everything, from transactions and ad-hoc queries to a few lightweight BI dashboards.

For a while, this was perfect. But as the company grew, so did the demands: daily sales summaries, marketing attribution models, multiyear reporting. Soon, query runtimes stretched from seconds to minutes, and Monday morning dashboards were blocked as analysts and ETL jobs fought over the same resources.

The fixes came in waves: adding indexes, upgrading the server hardware, scheduling heavy jobs at night. Then came read replicas for reporting, partitioned tables for large datasets, and more monitoring scripts than anyone wanted to maintain. But each fix brought complexity: replication

lag, maintenance windows, growing hardware costs. Eventually, you hit a wall: more CPU cores or RAM would help for a while, but not for long.

The turning point came with a move to a cloud data warehouse. The real difference wasn't just raw speed. It was the architecture: separating storage from compute, scaling resources up or out on demand, and isolating workloads so ETL and BI could run side-by-side without stepping on each other. Scalability shifted from being a constant firefight to something intentionally built into the system from day one.

Next, we compare these approaches in more detail, focusing on how they handle growth, performance, and operational complexity.

Traditional RDBMS Scaling

A *traditional relational database* is a monolithic system. Storage, compute, and query execution all live within the same engine, sharing the same physical resources, as illustrated in Figure 3-11. This design is simple to operate, but it brings natural limits on scalability. When workloads grow, the primary scaling strategy is *vertical scaling* by adding more CPU cores, more RAM, or faster storage to a single machine. This system can work well for moderate growth, but eventually it hits limits in both hardware and budget.

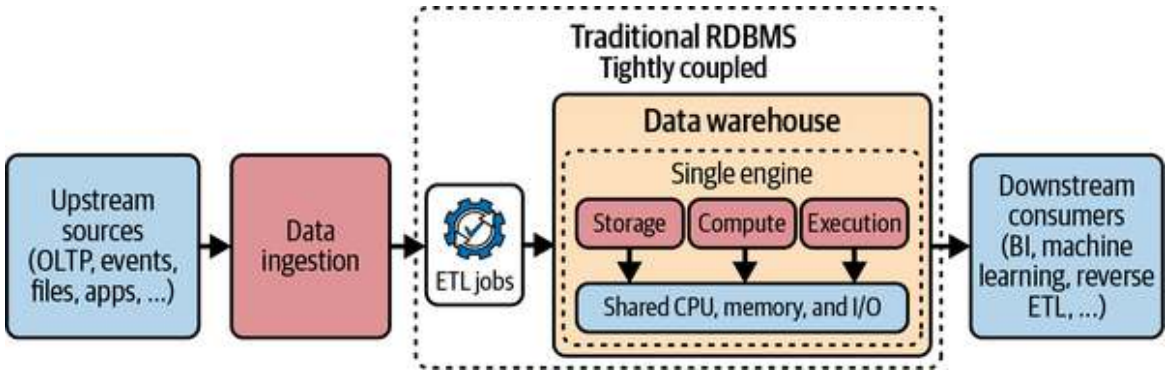


Figure 3-11. Traditional data architecture

Horizontal scaling exists in these systems, but it's far from effortless. You might deploy read replicas to offload analytical queries from the primary instance or implement sharding to split data across multiple servers. Both

options require significant engineering investment and trade-offs: replicas introduce lag and consistency considerations, while sharding adds complexity to application logic, making joins and aggregations across shards more difficult.

From a SQL practitioner's perspective, much of the optimization work focuses on squeezing maximum performance out of this single engine, which often involves indexing to speed up table lookups as in Example 3-2, partitioning and carefully managing query plans.

Example 3-2. Indexing a frequently filtered column to speed up table lookups

```
-- Indexing a frequently filtered column to speed up lookups
CREATE INDEX idx_orders_order_date ON orders(order_date);

-- Range partitioning a large table to improve pruning in queries
CREATE TABLE orders_2024 PARTITION OF orders
FOR VALUES FROM ('2024-01-01') TO ('2024-12-31');
```

Cloud Native Scaling

Cloud native data platforms break apart the monolithic design. Storage, compute, and query execution are decoupled so each can scale independently (Figure 3-12). This shifts the scaling model from constrained to elastic: keep all historical data in low-cost storage, spin up compute clusters only when needed, and run multiple workloads without them competing for the same resources.

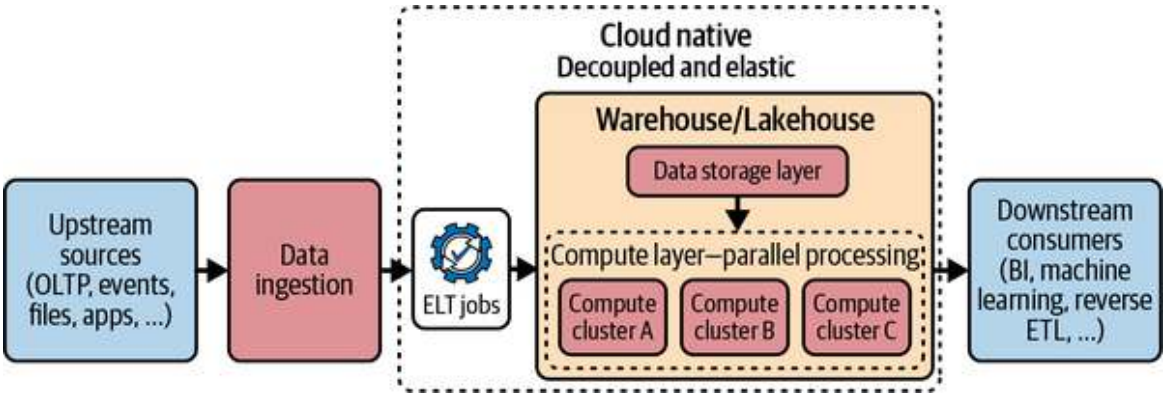


Figure 3-12. Cloud native data architecture

Scaling becomes a matter of provisioning for the moment:

- Running month-end close? Spin up extra compute, finish the job, and scale back down, paying only for what you use.
- Need parallel pipelines? Assign each to its own compute pool so ingestion, transformation, and BI queries can run without stepping on each other.

From a SQL user's perspective, this removes many operational bottlenecks. You can run complex joins and aggregations without worrying about a single server's RAM limit or carefully rebalancing shards. Performance tuning still matters (partitioning, clustering, statistics), but the ceiling is much higher.

Cloud native platforms also integrate capabilities that used to require additional tooling:

- Multi-cluster concurrency scaling
- Time travel and zero-copy cloning
- Cross-region replication
- Native support for semi-structured formats (JSON, Parquet, Avro)

The result is an environment built for scale, where architectural constraints are fewer, and you design pipelines for elasticity and parallelism from the start.

For SQL practitioners, this shift also means the ceiling on query complexity rises dramatically as multi-terabyte joins, advanced window functions, and semi-structured parsing can be expressed directly in SQL without hitting the hardware bottlenecks of a single server.

Concurrency and Isolation in Practice

In traditional systems, concurrency is often a zero-sum game. All queries, whether a simple lookup or a massive aggregation, compete for the same

CPU, memory, and I/O bandwidth. One badly optimized report can slow down critical transactions. To protect operational workloads, teams often impose strict limits on analytical queries or offload them to reporting replicas, adding cost and complexity.

Cloud native systems approach this differently, allowing you to isolate workloads by allocating dedicated compute resources:

- In Snowflake, spin up a separate warehouse for BI dashboards so reporting queries never slow ingestion jobs.
- In BigQuery, assign workloads to a dedicated slot reservation, guaranteeing throughput without contention.

This isn't just convenience; it enables parallel execution of layered architectures like the medallion model: Bronze ingestion, Silver enrichment, and Gold curation can run simultaneously in their own compute zones, without resource conflict. The result is predictable performance and the ability to scale both teams and workloads independently, shifting some focus from SQL practitioners from micro-optimizing queries to higher-level planning of compute allocation and cost management.

Hybrid and Multi-Cloud Patterns

While cloud native platforms have transformed the economics and flexibility of analytics, most organizations don't operate in a single, cleanly defined environment. Legacy systems, regulatory constraints, and cost optimization strategies often lead to *hybrid architectures* that mix on-premises and cloud resources, or *multi-cloud* setups that span more than one provider. These patterns are about leveraging the strengths of each system while still offering users a unified SQL experience.

Hybrid architecture involves combining a private or on-premises environment with a public cloud. A common driver is to keep sensitive operational data in a local PostgreSQL database to meet compliance needs and maintain low transaction latency, while replicating selected tables to a cloud data warehouse like Snowflake for large-scale analytical joins and

aggregations (Figure 3-13). Analysts get the benefit of modern cloud-scale querying, while sensitive data never leaves controlled infrastructure.



Figure 3-13. Hybrid architecture example

In contrast, a multi-cloud setup intentionally uses services from two or more public cloud providers (for example, GCP, AWS, Azure) within the same data strategy. The motivation can be risk mitigation, vendor neutrality, or simply picking the best tool from each provider. These patterns often take several forms:

Multiple warehouses in different clouds

An organization might use BigQuery on GCP for its large-scale historical analytics due to its serverless model, while using a dedicated Snowflake instance on AWS for financial reporting to meet specific regulatory or performance requirements.

One warehouse, mixed cloud services

You could use a central BigQuery warehouse on Google Cloud Platform (GCP) while ingesting raw files from an S3 bucket on Amazon Web Services (AWS), training machine learning models in Azure Machine Learning, and serving APIs from AWS Lambda. In this model, the data warehouse acts as the central hub, but the surrounding services are chosen based on their individual strengths across different providers.

Federated and virtualized queries across systems

This pattern allows you to query data without physically moving it. Using a federated query engine like BigQuery's `EXTERNAL_QUERY` (Example 3-3) you can join data from a MySQL database, a PostgreSQL instance, and a cloud storage bucket like S3 in a single SQL statement. This enables you to access real-time data from an

operational database and combine it with historical data in a warehouse without a slow, costly ingestion process.

Example 3-3. Gathering data from MySQL through BigQuery

```
-- BigQuery federated query from MySQL
SELECT *
FROM EXTERNAL_QUERY(
  'my-mysql-connection',
  'SELECT id, created_at FROM orders WHERE created_at >= NOW() - INTERVAL 1 DAY'
);
```

In each case, SQL remains the unifying interface, allowing practitioners to query across on-prem, cloud, and even cross-cloud systems without rewriting logic in other languages.

Actually, many mature data platforms combine more than one of these patterns. Figure 3-14 illustrates a complex multi-cloud scenario that blends them: multiple cloud warehouses across providers, cloud native services from other vendors, and a federated query layer to connect everything. This kind of composite approach allows organizations to choose the best tool for each workload while still offering a unified SQL experience.

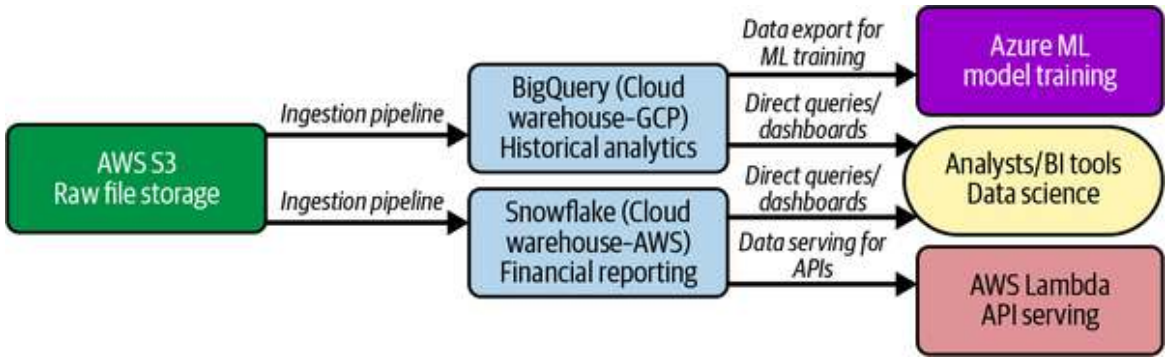


Figure 3-14. Complex multi-cloud architecture combining multiple patterns

Architectural decisions shape not only your platform's scalability but also its operational resilience and adaptability. Traditional RDBMS systems are reliable workhorses, ideal for small-scale, transactional workloads where simplicity and low latency matter most. But as data volumes, complexity, and concurrency needs grow, the inherent limits of monolithic scaling emerge.

Cloud native warehouses and lakehouses shift the equation entirely. By decoupling storage from compute, they allow elastic scaling, workload isolation, and feature-rich operations that make large-scale SQL-driven pipelines possible without the same operational pain.

Hybrid architectures bridge the gap between legacy and modern systems, enabling organizations to adopt cloud capabilities without abandoning existing investments. Multi-cloud patterns, on the other hand, are often driven by strategic choices, leveraging multiple vendors to optimize for cost, best-in-class services, or risk mitigation. In both cases, careful planning is required to avoid unnecessary data movement and governance gaps.

To contrast these architectural patterns, Table 3-2 compares traditional RDBMS, cloud native, hybrid, and multi-cloud approaches across key dimensions, summarizing not only their technical differences but also the trade-offs that influence scalability, governance, cost, and day-to-day operations in SQL-driven environments.

Table 3-2. Traditional RDBMS, cloud native, hybrid, and multi-cloud archite scalability, performance, and operational dimensions

Feature	Traditional RDBMS	Cloud native warehouse/lakehouse	Hybrid
Architecture	Monolithic: storage, compute, and query execution tightly coupled in a single engine	Decoupled: storage and compute scale independently, often multi-cluster	Mixed: on-prem or private clou paired with public clou analytics
Scaling model	Primarily vertical scaling; horizontal scaling via replicas or sharding (complex)	Elastic, on-demand scaling for both storage and compute; multi-cluster concurrency	Selective scaling in e environmer cloud portic elastic, on-prem limite
Concurrency	All workloads share CPU, memory, I/O; high risk of contention	Workload isolation via dedicated compute pools or slot reservations	Isolation within each environmer cross-syster queries may bottleneck

Feature	Traditional RDBMS	Cloud native warehouse/lakehouse	Hybrid
Complex queries	Limited by single-node resources; large joins slow	Distributed execution; handles multi-terabyte joins efficiently	On-prem limited; clo portion distributed
Semi-structured data	Requires ETL into relational schema	Native support for JSON, Avro, Parquet	Supported i cloud portic on-prem ma require ETI
Disaster recovery and resilience	Manual replication and failover	Built-in replication, failover, time travel, cloning	Separate DI strategies p environmer
Cost model	CapEx: fixed hardware costs, always-on	OpEx: pay-per-use compute, low-cost storage	Mixed CapEx/OpE optimizatio via workloa placement

Feature	Traditional RDBMS	Cloud native warehouse/lakehouse	Hybrid
Governance and security	Centralized permissions; schema-level control	Integrated cloud IAM; fine-grained access control	Separate governance per environmer must unify policies
Example use cases	OLTP + light analytics; smaller orgs	High-scale analytics, BI, ML pipelines	Compliance driven workloads; gradual clou adoption

Ultimately, the most successful architectures are not about chasing the newest platform, but about aligning system capabilities with workload needs, building a foundation where SQL can operate at scale, reliably, and cost-effectively, regardless of whether it runs on a single node, an elastic cloud service, or across multiple clouds.

Building and Operationalizing Scalable SQL Data Platforms

Designing scalable SQL pipelines is not just about choosing the right flow (batch versus streaming) or the right transformation pattern (ETL versus ELT). It's about turning those architectural choices into a production-ready platform. This means thinking about how data moves through layers, how queries are structured and maintained, and how the whole system can grow without breaking. In this section, we'll shift from conceptual decisions to

operational practices: how layered architectures bring order, how transformations become reliable models, how orchestration ensures predictability, and how supporting practices like testing, CI/CD, and infrastructure as code turn SQL pipelines into scalable, maintainable data platforms.

Layered Architecture: Separating Storage, Transformation, and Orchestration

By this point, we've explored the foundations of how SQL is used for data transformation, from optimizing queries to the broader architectural decisions between batch and streaming pipelines. We've seen how patterns like ELT have largely replaced ETL in modern platforms, and how SQL continues to evolve to meet both historical and real-time needs. But building a robust platform is about more than choosing between paradigms; instead, it's also about how we structure and manage those decisions in practice. While batch versus streaming and ETL versus ELT define how data flows, layered architectures determine how SQL practitioners structure, manage, and scale those flows in practice.

In modern data platforms, SQL doesn't operate in a vacuum. Queries live inside pipelines, pipelines operate on top of data platforms, and all of this has to run in a predictable, maintainable, and scalable way. That's where the concept of *layered architecture* becomes essential.

A layered architecture separates the concerns of data ingestion, storage, transformation, orchestration, and consumption, serving as a clear example of how to design a modular and maintainable data pipeline. A layered architecture allows teams to develop, maintain, and troubleshoot each part of the pipeline independently, while still contributing to a cohesive, end-to-end system. This structure also reflects how most modern data stacks are actually built, with clear handoffs between systems, tools, and responsibilities.

> **NOTE**
>
> Lambda and Kappa architectures describe how data flows, whether through batch, streaming, or both. A layered architecture, on the other hand, focuses on where responsibilities lie across your SQL pipeline. The two are not competing ideas. In fact, Lambda or Kappa architectures can (and should) still adopt a layered structure to improve clarity, modularity, and maintainability. Think of this as zooming in: once you've chosen a flow model, you still need to structure your pipeline components cleanly within it.

At the beginning of this chapter, with Figure 3-1, we illustrated a high-level view of how a data pipeline works. That diagram served as a conceptual starting point. Figure 3-15 builds on it, following the modern ELT standard, revisiting and expanding the visual to reflect a more complete and structured view that explicitly aligns with the layered architecture principles we will explore.

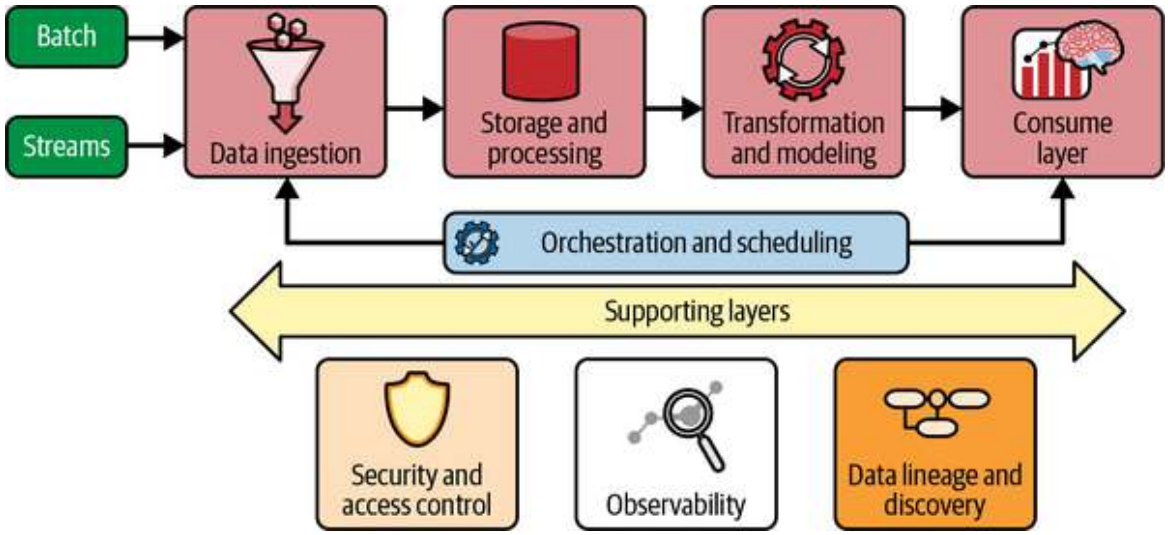


Figure 3-15. Layered architecture

The following sections describe each layer in this architecture and how they contribute to building scalable, SQL-centric pipelines.

Data ingestion

Whether your pipeline follows a batch-oriented or streaming-first architecture, the ingestion layer is the starting point of any SQL-driven data

platform. It's responsible for collecting raw data from various source systems and delivering it reliably to your analytical environment, often a data warehouse or lakehouse. This process needs to be accurate and consistent, since every downstream step, including transformation, modeling, and consumption, depends on trusting this incoming data.

Source systems can vary significantly. Ingestion might pull structured data from relational databases, fetch JSON records from APIs, load files from cloud storage, or collect fast-moving event data from messaging systems like Kafka. Each source might deliver data in a different format or at a different speed, so the ingestion layer plays a key role in handling these differences and turning them into something that can be reliably stored and processed.

In practical terms, ingestion happens in one of two ways: batch or streaming.

In *batch ingestion*, data is collected on a defined schedule, such as every hour or once a day, and loaded in larger chunks. This is common when working with traditional systems, or when real-time freshness isn't required. In SQL-based platforms, batch ingestion typically lands data in raw staging tables. While external tools like Fivetran or Airbyte are often used to move data into the warehouse, they are primarily data delivery tools, designed to land data in a format from which SQL can take over, rather than being SQL-based tools themselves.

In *streaming ingestion*, data is ingested continuously as it's generated. When new events occur, such as a customer making a purchase or a sensor update, that data is immediately pushed into the platform. This real-time delivery is typically handled by technologies like Kafka, Pub/Sub, or Kinesis, which act as buffers and routers for high-throughput event streams. In a SQL-compatible environment, streamed data may land in append-only tables, be captured in micro-batches, or become queryable via incrementally refreshed materialized views.

No matter how the data is ingested, the goals are always the same:

- All data should be delivered completely, with no loss or duplication.
- Important metadata like timestamps and primary keys should be kept intact.
- The process should handle errors, schema changes, or temporary source outages.
- The result should be cleanly structured and ready for the next stage of processing, supporting, for example, downstream SQL transformations.

The ingestion layer is not responsible for applying business rules or cleaning the data. It should avoid implementing complex transformation logic, such as filtering, enrichment, joins, or aggregations. The goal is to make raw data accessible and queryable (structured tables or partitioned files) so SQL-driven logic can take over cleanly in downstream layers.

Storage and processing

Once data is ingested, it must be stored in a system that supports efficient querying, transformation, and long-term access. This is the role of the storage and processing layer. In modern SQL-first architectures, this typically means a cloud native data warehouse (such as Snowflake, BigQuery, or Redshift) or a lakehouse (like Databricks, Apache Iceberg, or Delta Lake). These systems are designed to support large-scale analytical workloads efficiently.

This layer is where data is persisted and your SQL queries are executed. It supports everything from lightweight filtering to complex joins, aggregations, and modeling steps, but with a focus on infrastructure-level execution. This is where performance matters: materialization strategies, partitioning, and clustering directly impact how efficiently queries run and scale.

Storage systems typically evolve through different levels of maturity—raw, refined, and curated—based on how the data is processed and organized.

While terminology varies, a common and intuitive model to describe this evolution is the *medallion architecture*. Figure 3-16 illustrates how data flows through this layered structure, from raw ingestion to curated outputs ready for consumption.

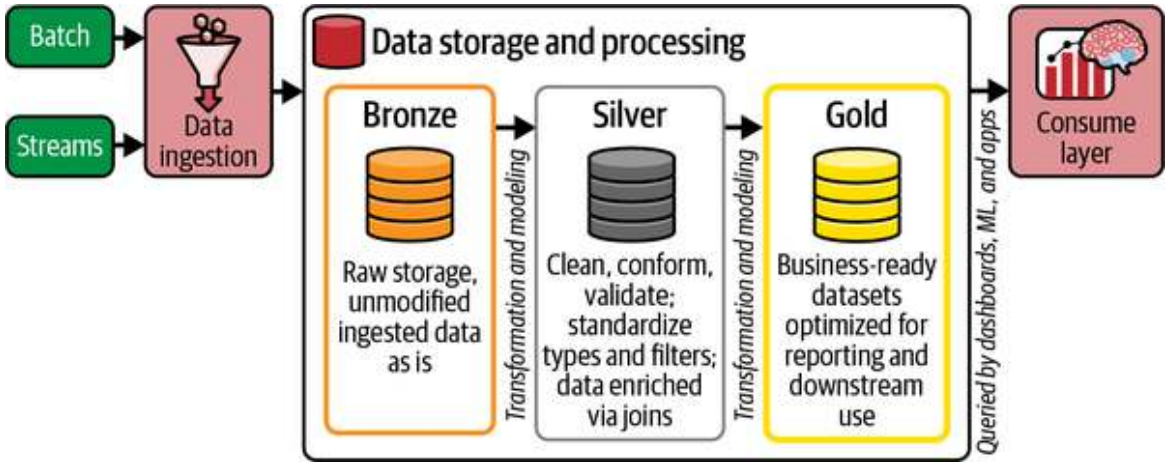


Figure 3-16. Medallion-style storage and processing flow within a SQL pipeline; data progresses from Bronze (raw) to Silver (refined) to Gold (curated), to power downstream consumption

Bronze (raw)

Unchanged data as it was ingested, for example, replicated tables from an operational system or events streamed in from Kafka. These datasets are typically schema-aligned but unfiltered, and they are often kept in an immutable state to serve as a reliable source of truth. The Bronze layer is essential for validation, historical traceability, reprocessing, and backfills, allowing teams to replay transformations or correct downstream logic without losing the original data.

Silver (refined)

Data that has been cleaned, conformed, and made consistent. This stage may handle nulls, correct types, and introduce joins across systems. Silver datasets serve as trusted, standardized foundations for analytical work, preserving enough detail for exploratory and ad hoc analysis before business-facing aggregation occurs in the Gold layer.

Gold (curated)

> Highly refined, aggregated, and business-facing datasets, such as KPI tables or metrics ready to feed dashboards and machine learning models. Gold data reflects key business definitions and is often governed more tightly. It represents curated, business-defined narratives designed for consumption across reporting tools, executive dashboards, and operational decision-making.

In practice, your SQL queries, either manual or managed through tools like dbt, flow through these stages. The storage engine is responsible not only for keeping your data but also for processing these queries efficiently. That means platform-level decisions matter: how data is partitioned, clustered, or sorted can dramatically affect performance. Some warehouses support columnar storage, automatic clustering, or materialized views that refresh incrementally. Others provide data sharing and streaming capabilities directly within the storage layer.

Treat this layer as both storage and compute, especially in cloud platforms where these concerns are separated internally. Even when the platform abstracts away operational details, SQL users still need to understand the trade-offs of table types, refresh strategies, and materialization approaches.

Lastly, even if the transformation logic lives in the next layer, it's worth noting that the storage and processing layer is what makes that logic possible. This is where your models are materialized, your indexes (if applicable) are leveraged, and your queries are optimized for performance. A poorly organized storage layer can quickly turn a clean pipeline into a bottleneck. Conversely, a well-managed warehouse or lakehouse enables clean modular transformations and consistent delivery of data products downstream.

Transformation and modeling

Once data is ingested and stored in a queryable format, the next step is to transform it into structures that are usable, interpretable, and aligned with business needs. This is where the core logic of your data pipeline lives. It's not just where SQL is executed, but where logic is written, versioned, and

maintained. Here, SQL becomes expressive: cleaning data, joining entities, applying business rules, and building reusable models that support everything from BI dashboards to ML pipelines.

Although SQL transformations execute in the same environment as the storage layer, the goal here isn't performance optimization or persistence configuration but applying business logic and shape data into relevant data structures. The transformation layer is about logic and structure, not infrastructure.

In SQL-driven platforms, this layer is typically implemented as a series of versioned SQL queries or models, often managed by transformation tools like dbt. These tools orchestrate transformations declaratively, materializing results as views or tables while maintaining a clear lineage of dependencies between models.

Staged modeling layers

Transformations don't usually happen in a single step. Instead, they are broken down into stages, which makes pipelines easier to test, debug, and extend. These modeling layers often follow the same sequence introduced in the medallion architecture:

Staging (Bronze-aligned)

> This is the initial transformation layer, focused on making raw data usable. It typically includes light operations such as renaming columns, applying data type casts, and filtering corrupt or irrelevant records. Models in this layer tend to mirror the structure of the source systems closely and are meant to provide a clean and stable baseline for subsequent transformations.

Intermediate or enriched (Silver-aligned)

> This layer focuses on integration and business logic. Here, you join data from multiple domains, enrich records with reference tables, apply filters, or normalize schemas. A good example is combining order events with customer dimensions to create a unified behavioral view.

This stage often carries the most transformation logic and is where your models begin to reflect how the business actually operates.

Final or curated (Gold-aligned)

This is the output that consumers will query directly. These models are often fully aggregated or denormalized for performance, audited for correctness, and optimized for usability. Outputs can include metrics tables, KPI snapshots, denormalized reporting views, and features for machine learning models. Gold-layer models are designed to be used directly by less technical users and BI tools, and they are typically subject to stricter validation, documentation, and access controls. Still, they may internally reference the intermediate layer, which is why testing and governance should span the entire pipeline, not just the final outputs.

Aligning transformations to this structure provides clarity and stability. Figure 3-17 illustrates how this layered modeling approach is implemented in practice. Each box represents a SQL model, typically defined by a single query, and arrows represent data dependencies. The example is based on a simplified ecommerce dataset, where raw tables include customers, orders, payments, and products.

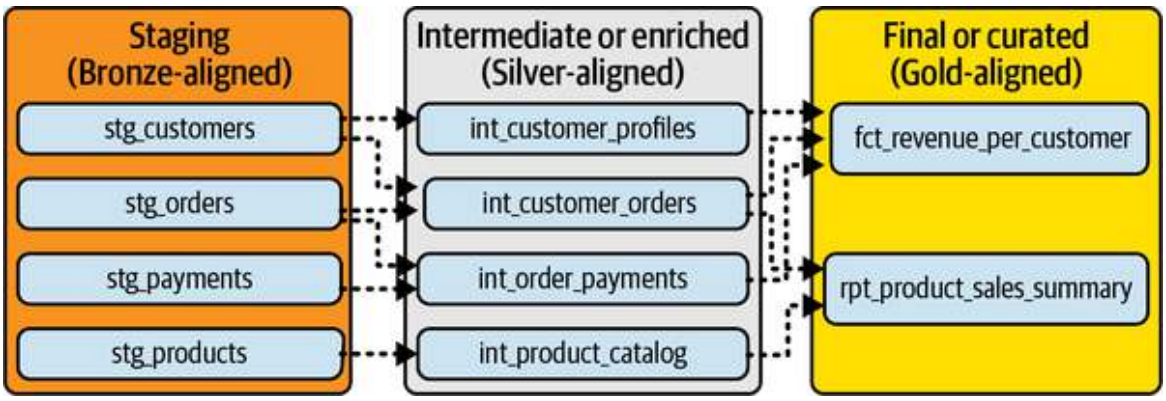


Figure 3-17. Example of layered SQL transformations using a common naming convention (`stg_`, `int_`, `fct_`, `rpt_`)

In the staging layer (`stg_`), raw inputs are cleaned and standardized. Intermediate models (`int_`) apply business logic, for example, combining

orders with customer or payment data, and prepare enriched records for downstream use. The final layer includes curated outputs, such as `fct_revenue_per_customer`, which aggregates revenue by customer, or `rpt_product_sales_summary`, which provides product-level sales reporting.

> **NOTE**
>
> The `stg_`, `int_`, `fct_`, and `rpt_` prefixes shown here are illustrative. Even if these are widely adopted naming patterns, especially in dbt and modular SQL development, teams can adopt their own conventions as long as they are consistent and well-documented.

Transformation patterns

As discussed earlier, most modern data pipelines follow an ELT pattern: data is extracted and loaded first and then transformed directly within the data warehouse or lakehouse. This shift, from traditional ETL, is one of the key enablers of SQL-centric engineering. Platforms like Snowflake, BigQuery, and Databricks are optimized for running transformations in place, allowing SQL users to take full advantage of the warehouse's scalability and performance.

ELT doesn't mean transformations are flat or unstructured. Robust pipelines apply the same engineering principles as application development: break logic into reusable parts and write tests. When designing transformations, keep the following principles in mind:

Clear separation of concerns

Isolate business rules into dedicated models.

Modularity

Build small, composable steps rather than one large SQL file.

Reusability

Abstract common logic (like currency conversions or user segmentation) into shared models.

Testability

Write tests for assumptions like uniqueness, null constraints, or value ranges.

Traceability

Use comments, documentation, and naming conventions to make logic easier to audit.

Following these principles leads to pipelines that are easier to understand and change, especially in teams with shared ownership.

SQL as a modeling language

As pipelines grow more complex, readability and declarative intent become key to managing change. Because SQL describes what to produce, and not how, it's easier to understand and reason about, even across large teams. This makes SQL particularly well-suited to modeling: building views or tables that represent the structure of the business in a transparent and consistent way.

In this context, *modeling* refers not just to transformation logic but to the design of reusable data structures that reflect shared definitions. A model might define what a *customer* is, what counts as *revenue*, or how to segment users into cohorts. These definitions become shared assets across the organization.

That said, modeling in SQL does require some engineering rigor. Best practices include the following:

- Using CTEs to structure logic
- Avoiding overly nested subqueries when unnecessary
- Applying consistent naming conventions

- Annotating logic with comments for clarity
- Breaking long queries into logical components
- Materializing intermediate datasets when appropriate to optimize performance, manage resource usage, or reuse results across multiple downstream steps

These practices were introduced earlier as ways to write clean and performant SQL. In the context of data modeling, they become even more important, not just to improve query readability but also to define stable, testable building blocks that can scale with your team and business.

Chapter 4 covers these modeling decisions, from schema design to dimensional modeling, and from slowly changing dimensions to time-based metrics.

By combining the structure of transformation layers with strong SQL modeling practices, this stage becomes more than a pass-through for logic. It becomes the core of your data platform: where logic lives, where meaning is applied, and where trust in the data is earned.

Orchestration and scheduling

After data is ingested, stored, and transformed, it must be executed in a reliable, repeatable way. Tasks need to run in the right sequence, at the right time, and with full awareness of dependencies across the pipeline. That's the role of orchestration: coordinating the moving parts of a SQL-based system so that everything flows in a controlled and predictable manner.

Without orchestration, even a small data pipeline can quickly become unmanageable. Imagine having to manually trigger dozens of SQL models every day, ensuring they run in the correct order, validating that upstream tables are ready, checking for errors, and tracking execution timing by hand. It's not just inefficient; it's fragile. One missed step, like refreshing a report before its source data is updated, can cascade into broken dashboards, outdated metrics, and a loss of trust in the data platform.

Orchestration transforms a pipeline from a collection of queries into a functioning system. It defines how each piece fits together: which models depend on which, when they should run, how they're triggered, and what to do if something goes wrong. It turns static SQL into an operational flow that the business can rely on.

In SQL-driven platforms, orchestration doesn't execute the transformation logic itself. Instead, it acts as the coordinator, handing off execution to the underlying data warehouse or lakehouse, and managing how, when, and in what order that logic should run. This separation of concerns is key allowing the SQL models to remain clean, declarative, and focused purely on business logic, while orchestration ensures that the entire pipeline behaves like a reliable system.

To make this possible, orchestration layers take on several responsibilities:

Ordering

> Ensuring the correct execution sequence between tasks, which guarantee that dependent models are never built before their inputs. This helps keep your data reliable by making sure each transformation uses up-to-date, correct inputs.

Scheduling

> Defining when tasks should run, whether on a fixed cadence like every hour or every morning at 4 a.m., or in response to an event, such as a new file arriving or an upstream job completing successfully. This is what turns your logic into a process that runs automatically and on schedule.

State tracking

> Monitoring and persisting the status of every task in the pipeline, such as succeeded, failed, skipped, retried, or stale; this helps teams troubleshoot failures, spot delayed jobs, and understand the health of their data pipeline at any point in time.

Triggering workflows

> Triggering follow-up processes after a model has run successfully, for instance, building a curated reporting table only after all required staging and enrichment models have completed.

One of the most important aspects of orchestration is its ability to resolve dependencies between tasks. SQL models rarely operate in isolation. A model that calculates revenue might depend on clean transaction data, product details, and user information. Running these models out of order can lead to incomplete outputs, out-of-date dashboards, or data issues that go undetected.

To solve this, most modern orchestration tools, including dbt, Airflow, Dagster, or Prefect, represent pipelines as DAGs. In a DAG, each node is a task or SQL model, and each edge defines a dependency. The orchestrator walks this graph to determine the correct execution order, ensuring that no model runs until all its prerequisites are complete. Figure 3-18 illustrates this concept with a DAG, optimized for readability in documentation.

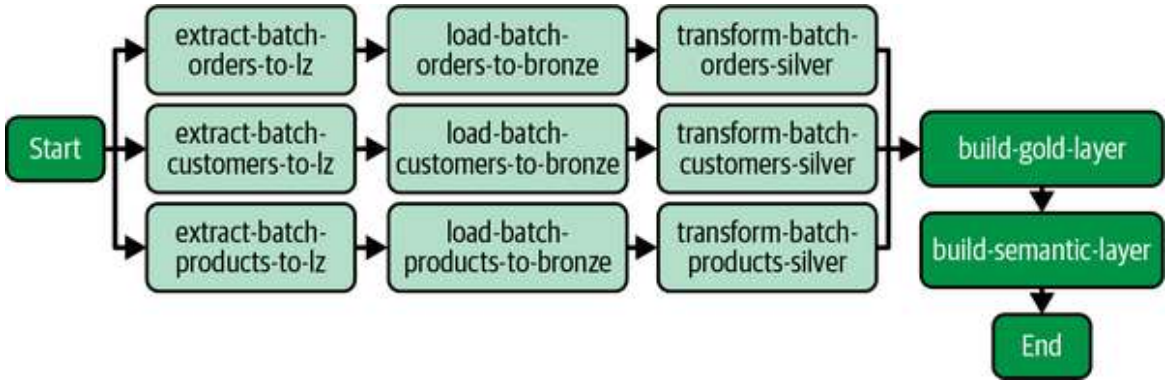


Figure 3-18. DAG example

This dependency resolution is not just a technical requirement; it's a safeguard against silent failure. A model can execute without error yet still rely on outdated or incomplete upstream data. These inconsistencies are among the hardest to detect and the most damaging to data trust. A well-defined DAG ensures that models run only when their inputs are fresh, reducing the risk of broken logic reaching end users.

In production systems, correctness depends not just on what is computed but also when. Orchestration ensures both, not by running SQL but by managing the lifecycle of SQL. And in doing so, it elevates your data models from isolated logic to coordinated, reliable workflows.

As we have discussed, scheduling is one of the defining responsibilities of the orchestration layer. It determines when each part of your pipeline should run. Depending on the use case, that timing can take very different forms:

Time-based schedules

> Time-based schedules trigger jobs at fixed intervals, such as every hour, every night at midnight, or every Monday at 8 a.m. This approach works especially well for batch pipelines or business reporting cycles that follow a predictable cadence. For example, if stakeholders expect a daily sales dashboard to be refreshed by 7 a.m., a time-based schedule ensures that the relevant models are run at the right time to meet that expectation.

Event-driven schedules

> Event-driven schedules are more dynamic. Instead of relying on the clock, they respond to specific signals, such as a new file appearing in cloud storage, a webhook from an upstream system, or the successful completion of a previous step. These schedules are particularly useful in streaming or near-real-time scenarios, where fresh data may arrive at irregular intervals and speed matters more than predictability.

Modern orchestrators typically support both types of scheduling, and many data teams adopt a hybrid approach. You might run core reporting tables on a daily schedule, while also triggering quick updates to lookup tables as soon as new data is ingested. This mix of reliability and responsiveness helps balance latency, performance, and cost.

In addition to controlling execution timing, orchestration also contributes to pipeline visibility. This is an essential part of operating data systems in production. Orchestrators track the outcome of each task, maintain logs of

what ran and when, and provide context when things go wrong. If a job fails or runs longer than expected, the orchestrator gives you the tools to diagnose and respond. While full observability is covered later in "Supporting layers", it's worth noting here that orchestration is one of the first layers that surfaces operational metadata for debugging and system health monitoring.

When designing the orchestration logic, especially for SQL-first systems, a few principles help keep things clean, maintainable, and scalable:

Model-based orchestration

> Let the structure of your SQL models (for example, staging → intermediate → curated) guide the order of operations. This aligns the orchestration graph with your mental model of the pipeline.

Minimize hardcoding

> Avoid rigid schedules when data availability is variable. Let data dependencies drive execution when possible, especially when using tools like dbt that can infer relationships automatically.

Fail fast, fail clearly

> Design your tasks to identify errors early, and log them in a way that helps developers troubleshoot without guesswork.

Separate orchestration logic from transformation logic

> Don't mix orchestration concerns into your SQL code. Scheduling should live in external orchestration tools or configurations, not inside the transformations themselves.

Ultimately, the orchestration layer is what makes the pipeline run. It doesn't define your data logic, but it does ensure that logic is executed in the right sequence, at the right time, and with the right dependencies. Without it, even the cleanest transformation logic can fail to deliver trustworthy results.

If the transformation layer is where meaning is created, then orchestration is what brings that meaning to life, turning your SQL code into a dependable, automated system that your organization can trust every day.

Consumption layer

The consumption layer is the last mile of your data pipeline, the point where all previous layers, from ingestion to transformation, are finally put to work. This is where data moves from systems to people, from structured models to strategic decisions. If the upstream pipeline is the factory, the consumption layer is the storefront: the place where value is realized and trust is either reinforced or lost.

This layer depends entirely on what comes before it, and it is also the most visible. It's where business users open dashboards, analysts run ad hoc queries, data scientists extract features for training models, and applications retrieve metrics via APIs. Any break, delay, or inconsistency here ripples quickly through the organization, often with real consequences.

Imagine a sales leader refreshing a dashboard in Power BI or Looker and seeing the wrong number. Or two teams arriving in a meeting with two conflicting reports on customer churn. These moments don't just signal broken data; they damage confidence in the entire platform. That's why the consumption layer must be stable, curated, and well-governed. Trust is earned upstream, but it's tested here.

In modern SQL-driven platforms, this layer connects directly to the outputs of your transformation layer, most often the curated Gold models described in the medallion architecture. BI tools like Looker, Metabase, and Power BI typically read from these production-grade tables using semantic layers, direct SQL queries, or prebuilt views. Reverse ETL platforms like Hightouch or Census sync curated attributes back into operational systems, closing the loop between analytics and action. Even when these tools abstract away complexity, almost everything they do is ultimately expressed and executed as SQL under the hood, reinforcing SQL as the primary interface for consuming data.

And because this layer is so close to decision-making, it must be treated with discipline. While business-facing dashboards and reports should rely only on curated models, other users, like data scientists and analytics engineers, may work directly with intermediate models, especially when those models contain enriched or denormalized data useful for exploration or machine learning. What matters most is that access is intentional, documented, and aligned with how stable each model is expected to be.

Regardless of which layer is being queried, testing and documentation should span the full transformation pipeline, not just the final outputs. Validating raw ingested data, enforcing schema constraints in staging, and applying business logic checks in intermediate layers is essential for trust. After all, a well-tested pipeline isn't just about ensuring Gold-layer tables are accurate; it's about catching issues as early as possible in the flow.

Best practices for the consumption layer

A well-designed consumption layer certainly contributes to performance, but it also is essential for clarity, safety, and trust. The following principles help ensure that your data is not only accessible but reliable:

Expose only curated models

Limit access to Gold-layer outputs. This reduces the risk of stakeholders building reports on incomplete or inconsistent data.

Enforce semantic alignment

Use semantic layers or standardized naming conventions to help nontechnical users query data consistently.

Treat reporting as a product

Every dashboard is an interface to your models. Version it, test it, and monitor usage.

Avoid tight coupling

Decouple dashboards and apps from volatile upstream logic. If a transformation changes, you shouldn't have to fix 10 dashboards.

Document with intent

Make it easy to understand what each model or table is for, and what it isn't. Clear ownership and documentation are key.

Just like the rest of the pipeline, this layer should be designed for evolution. New metrics, new use cases, and new questions will arise. By maintaining a clean separation between business-facing outputs and the upstream transformations that feed them, you can adapt confidently without breaking trust.

Connecting back to the architecture

This layer sits directly on top of the Gold models from your medallion-based transformation flow. And in the broader layered architecture of your pipeline, it aligns with the final segment, the Output or Delivery phase, which turns raw data into actionable insight.

The strength of this layer derives from clarity, not complexity. Consumers shouldn't need to understand how the data was ingested, partitioned, or transformed. But they do need to trust that what they're seeing reflects the truth of the business. That trust starts with the SQL practitioner who defines and protects these final outputs.

From Design to Production: Scaling SQL Data Solutions

Designing a SQL-driven architecture is only a part of the picture. The real challenge begins when those pipelines need to run in production, reliably and at scale. It's not enough for queries to work in isolation: they must be tested, deployed, monitored, and secured as part of a living system that adapts to change. This is the layer where concepts like modularity and layered design meet the realities of daily operation.

In practice, operationalizing SQL pipelines means embedding engineering rigor into data workflows. Robust testing strategies ensure correctness long before data reaches stakeholders. CI/CD pipelines bring automation and speed, making deployments repeatable and safe. Infrastructure as Code gives teams reproducibility and control as platforms expand across projects, environments, or even clouds. Supporting layers, such as observability, lineage and discovery, and security, provide visibility, accountability, and governance, ensuring trust and scalability as usage grows.

Together, these practices turn SQL pipelines into production-grade systems. They allow teams to move quickly without sacrificing stability, to evolve architectures without losing control, and to scale solutions that remain transparent and trustworthy as business demands increase.

Supporting layers

As your data platform evolves, ingesting more sources, serving more teams, and supporting increasingly critical decisions, it is no longer sufficient to simply run SQL transformations in the correct order and produce curated outputs. Production-grade systems need more than correctness. They require visibility, accountability, and control. That's where supporting layers come in.

These aren't the parts of your architecture that move or transform data directly. Instead, they work behind the scenes to ensure the entire system operates with resilience, clarity, and trust. They allow teams to monitor for problems before they become outages, trace how data flows through complex pipelines, and enforce responsible access to sensitive or business-critical datasets.

In modern SQL-first environments, these supporting capabilities typically fall into three essential categories: observability, data lineage and discovery, and security and access control. Each solves a distinct operational challenge, but together they form the core of a sustainable data platform, one that can scale confidently while remaining trustworthy and transparent.

Observability: Knowing what's working (and what's not)

Imagine this scenario: a stakeholder opens a dashboard first thing on a Monday morning and notices that the revenue numbers are suspiciously low. Panic sets in. Is the business actually underperforming? Did something go wrong with the data logic? Was the source data delayed, or did the transformation fail silently? Without clear visibility into the health of your data pipelines, your team is left in a state of chaos with no answers—or worse, they are unaware there's even a problem.

This is exactly what *observability* is designed to solve. At its core, observability provides a window into how your pipelines are running, not just whether they completed, but whether they behaved as expected. It helps teams monitor both the operational and functional aspects of data workflows, navigating the information they need to catch issues before they impact downstream consumers.

With strong observability in place, you and your team are capable of answering these key questions:

- Did this model complete successfully, or did it fail midway?
- Was the number of rows processed within a normal range?
- Has the schema changed since the last successful run?
- Is this dataset fresh enough to meet its SLA, or is it stale?

Modern observability stacks go beyond basic logging. They continuously monitor pipeline execution status, flag unusual behavior, and trigger alerts when thresholds are crossed. These systems often integrate with orchestration and transformation layers, giving engineers a single place to inspect logs, analyze error messages, understand query performance, or trace lineage to root causes.

But observability isn't just about reacting to failure. Instead it acts as a feedback loop, helping you detect anomalies early, measure pipeline health over time, and proactively improve your system's reliability. When done well, observability becomes an integral part of development and maintenance, not just a last-resort tool for when things go wrong.

Observability should never be an afterthought. Effective systems are designed with observability in mind from the start, across ingestion, transformation, orchestration, modeling, and consumption layers. Decisions about logging, metadata capture, validation checks, and metric tracking should be embedded into the pipeline's architecture rather than added later in response to incidents. Designing for observability ensures that when issues arise, visibility is already built into the system.

In a world where data products are consumed in real-time and decisions are made at the speed of dashboards, observability becomes your early warning system, helping your team stay ahead of problems, not behind them.

Lineage and discovery: Seeing the forest and the trees

As data platforms evolve, so do their pipelines. New tables are introduced, models are refactored, business logic gets updated, and before long, complexity quietly creeps in. Familiar and slightly anxious questions echo across teams:

- What is this table used for?
- Can I delete this model, or is it powering a report?
- Where does this column actually come from?

These are not trivial questions. In a system with hundreds or thousands of models, the inability to answer them reliably is more than an inconvenience: it's a risk. This is where data lineage and discovery come in, turning uncertainty into visibility and opacity into structure:

Lineage

Lineage is about tracing the flow of data through your system, from the moment it's ingested, through each transformation, all the way to final consumption. It reveals how tables depend on one another, what feeds into what, and what will break if a particular model changes. It answers the question "If I change this logic, what else do I affect?" Lineage diagrams visualize these relationships, often building on the same DAG

structure used by orchestrators, but with a richer semantic layer for both engineering and business audiences.

Discovery

Discovery focuses on the meaning and metadata of your datasets, helping you understand what a dataset is: its schema, owner, purpose, update frequency, data quality, and use. Discovery tools act like internal data catalogs: searchable, browsable, and integrated into your development workflows. They change your warehouse from a jungle of table names into a navigable, documented resource.

Together, lineage and discovery form the GPS of your data platform, enabling analysts to self-serve confidently, engineers to make safe changes, and data stewards to trace the origins of every metric. They also play a critical role in compliance and auditability, helping teams demonstrate control and accountability over how data is sourced, transformed, and used.

Without these tools, even the most well-structured SQL pipelines can degrade into fragile, misunderstood, and change-resistant black boxes. With them, your platform becomes more than functional: it becomes navigable, transparent, and resilient.

Security and access control: Guardrails, not roadblocks

Structure and clarity lose their meaning if the wrong people have access to sensitive data, or if the right people can't access the data they need. As your platform matures, security can't be an afterthought. It must be embedded into the architecture as a core design principle, balancing control with usability.

Security and access control define who can see what, and under what conditions. This applies across all layers of your pipeline, from raw ingestion zones to curated business outputs, and includes decisions such as:

- Enforcing different permissions on raw (Bronze) versus curated (Gold) data

- Masking sensitive columns like personally identifiable information (PII) or payment data
- Logging data access and query behavior for audit and compliance tracking
- Defining role-based access that aligns with team structures and responsibilities

It's easy to view governance as a bureaucratic obstacle. But in reality, it's the foundation of trust, compliance, and operational integrity. Whether you're bound by external regulations like GDPR, HIPAA, or SOC 2, or simply aiming for internal best practices, a consistent principle applies: access should be intentional. Users should only have visibility into the data they actually need to perform their job: no more, no less. This is the essence of the principle of least privilege.

And, when implemented well, access control is not a blocker. It's a design enabler. It lets you open up curated outputs to a broad audience without compromising sensitive zones like raw ingestion layers or intermediate modeling logic. In this way, security becomes a tool for safe scalability, not a constraint on innovation.

As your data platform grows, so too does the importance of control. A five-model DAG might survive with loose ownership and open access. A fifty-model DAG won't. A small team might remember table ownership and column meaning. A cross-functional organization cannot, and it shouldn't have to.

That's why observability, lineage, and access control form the foundation of sustainable data operations. They don't move data themselves, but they make sure everything else does, securely, transparently, and with confidence. These are not optional add-ons. They are essential architectural layers that allow SQL-centric systems to scale without chaos.

Implementing testing strategies

Testing is the safety net of a production-grade SQL pipeline. It's the practice that catches issues before they affect end users, ensuring that changes to logic, schema, or data sources don't silently break downstream models or reports. In a modern SQL-first architecture, testing is not a phase at the end; it's part of every stage of development, deployment, and monitoring. These tests are SQL queries themselves, often automated through dbt or custom assertions.

Observability tells you what is happening in production; testing aims to prevent issues from ever reaching production in the first place. The most resilient pipelines combine both: proactive checks before changes go live, and continuous monitoring once they do.

Just as architectures are layered, tests should be too:

- At the transformation layer, validate that joins, filters, and aggregations produce expected results.
- At the orchestration layer, confirm that dependencies, schedules, and triggers produce fresh, correct outputs.
- At the consumption layer, ensure that final datasets and dashboards reflect accurate, consistent metrics.

Table 3-3 summarizes the multiple testing strategies that can be added into a SQL-driven data platform.

Table 3-3. Comprehensive testing strategies for a SQL-driven data platform

Test type	Description	Key focus	When to apply
Unit testing	Verifies a small, isolated piece of transformation logic	Correctness of calculations, filters, joins in isolation	During development of new models or logic changes
Integration testing	Validates that multiple models or components work together	Data flow between models, dependency resolution	Before promoting changes to staging or production
End-to-end testing	Checks the entire pipeline from ingestion to consumption	Data completeness, schema alignment, correctness of final outputs	Before production releases or after major changes
Performance testing	Measures query runtime and resource usage under realistic load	Latency, cost efficiency, scalability	Before launching high-volume pipelines or after optimizations

Test type	Description	Key focus	When to apply
Regression testing	Detects unintended changes in results after modifications	Stability of key metrics and outputs	Any time existing logic is altered
Data quality testing	Checks for missing values, duplicates, outliers, invalid formats, and distribution anomalies	Accuracy and reliability of critical datasets, often using defined thresholds rather than strict equality (e.g., acceptable row count variance)	Continuously, especially in production
Security testing	Validates that sensitive data is protected and access rules are enforced	Row/column-level security, masking policies	Whenever permissions or access patterns change
Load testing	Simulates high-volume workloads	Resource scaling and failure behavior under stress	Before seasonal spikes or major data increases

Test type	Description	Key focus	When to apply
Compliance testing	Confirms adherence to legal/regulatory requirements	Data retention, anonymization, deletion policies	Regularly, especially before audits
User acceptance testing (UAT)	Ensures outputs meet stakeholder expectations	Metric definitions, usability, accessibility	Before final production sign-off

A good testing strategy not only protects data quality; it also builds trust between engineers, analysts, and business stakeholders. It's the difference between reacting to failures and confidently deploying new features, knowing they've been vetted from every angle.

Continuous integration and continuous delivery for SQL pipelines

If testing is your safety net, continuous integration/continuous delivery (CI/CD) is the automation that makes your data pipeline usable every day without slowing the team down. In modern data teams, SQL pipelines evolve constantly with new models, schema changes, optimizations, and bug fixes. Without automation, deploying those changes safely is manual, slow, and error-prone.

CI/CD brings the rigor of software engineering into SQL development; however, unlike software applications, what moves through CI/CD pipelines are SQL models and queries:

Continuous integration (CI)

Automatically validates changes when they're committed, running tests to catch issues early

Continuous delivery (CD)

> Ensures that, once changes pass validation, they're deployed quickly and consistently to staging and production environments

In SQL-first workflows shown in Figure 3-19 and further described in Table 3-4, CI/CD integrates seamlessly with tools like dbt for transformations, orchestrators like Airflow or Dagster, and version control platforms like GitHub or GitLab.

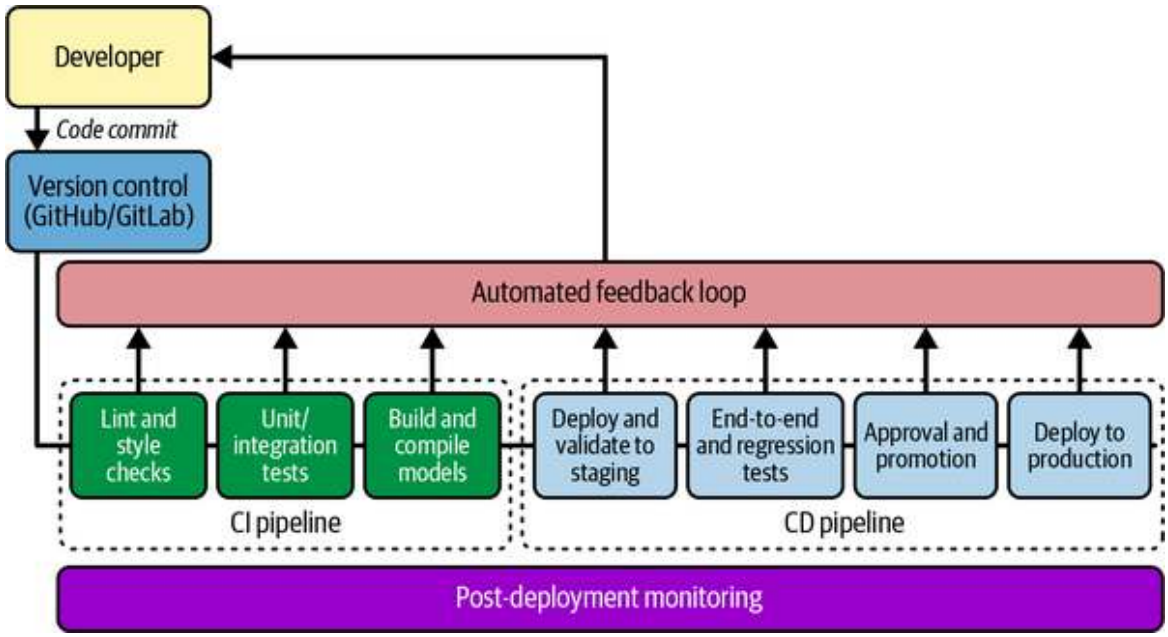


Figure 3-19. CI/CD workflow

Table 3-4. CI/CD workflow steps for SQL/dbt pipelines

Step	Purpose	SQL/dbt-specific actions
Version control commit	Track all changes in code and logic	Commit SQL models, macros, and tests with clear, descriptive messages
Automated linting and style checks	Enforce consistency and readability	Run SQLFluff or dbt-style checks
Automated unit and integration tests	Catch issues early	Use dbt test or custom SQL queries
Build and compile models	Ensure models compile and dependencies resolve	Run dbt compile to detect syntax or reference errors
Deploy to staging	Validate in a safe environment	Trigger orchestrated runs against staging datasets
End-to-end and regression tests	Confirm stability before production	Compare staging outputs to production baselines
Approval and promotion	Controlled release process	Require peer review and stakeholder sign-off

Step	Purpose	SQL/dbt-specific actions
Deploy to production	Apply changes to live datasets	Orchestrator triggers production jobs after approval
Post-deployment monitoring	Detect issues quickly	Use observability tools for anomalies and performance checks

Even a minimal CI/CD setup—linting, unit tests, and automated staging deploys—can dramatically improve reliability and speed. Over time, teams can expand to include performance tests, automated rollback plans, and blue/green deployment strategies for SQL models.

Infrastructure as Code (IaC) in scaling and for data platforms

One enabling technology for managing scalable architectures is *Infrastructure as Code* (IaC). IaC is the practice of defining your infrastructure (servers, databases, network configs, and so on) in machine-readable configuration files or scripts, rather than via manual setup or clicking through consoles. In a data platform context, this means database instances, warehouses, pipelines, and related resources are defined in code and can be deployed or updated automatically.

IaC can be used in hybrid setups, but it is especially important in cloud contexts where using IaC tools (like Terraform and CloudFormation) might frequently create and tear down resources (like ephemeral test environments or scaled-out clusters). IaC ensures that if you need to scale out your platform (for example, add a new environment, expand to a new region or cloud), you can do so reliably with the same configurations. Additionally, IaC documents your architecture and reduces manual errors in configuration. In other words, leveraging IaC brings several benefits to SQL-driven solutions, since the database schemas, roles, and warehouses that queries run on are reproducible and version-controlled like code:

Reproducibility

You can spin up an identical environment (for development, testing, or new teams) using the same IaC scripts. For example, if your data warehouse and all its schema and permissions are defined in Terraform scripts, onboarding a new environment is much faster and less error-prone.

Version control

The configuration lives in Git (or another VCS), so changes are tracked. If someone modifies the data infrastructure (e.g., opens a new port or creates a new schema), that change is reviewed and logged just like a code change, thus reducing accidental misconfigurations.

Automation

Combined with CI/CD pipelines, IaC allows automated provisioning and teardown. If you need to temporarily scale out, you can run a script to deploy more nodes, and later tear them down—all as part of automated jobs.

Consistency across clouds

IaC tools like Terraform are cloud-agnostic to an extent. You can manage multi-cloud or hybrid setups in one place, ensuring consistent settings (like naming conventions, tags, security rules) across all.

For example, consider managing Snowflake with IaC: you could have Terraform code that defines Snowflake warehouses, databases, roles, and grants, as in Example 3-4. If you want to create a new warehouse for a new project, you update the config and apply it, and Terraform ensures the Snowflake account matches the desired state. Similarly, with BigQuery, you could define datasets, access roles, and even scheduled queries in code.

Example 3-4. Terraform configuration for Snowflake defining a warehouse, database, schema, role, and grants

```hcl
# Provider omitted for brevity

# Warehouse
resource "snowflake_warehouse" "analytics_wh" {
  name           = "ANALYTICS_WH"
  warehouse_size = "XSMALL"
  auto_suspend   = 60
  auto_resume    = true
}

# Role
resource "snowflake_role" "analytics_role" {
  name = "ANALYST_ROLE"
}

# Database + Schema
resource "snowflake_database" "analytics_db" {
  name = "ANALYTICS_DB"
}

resource "snowflake_schema" "core" {
  database = snowflake_database.analytics_db.name
  name     = "CORE"
}

# Grants: give the role usage on warehouse/db/schema and read on schema
resource "snowflake_warehouse_grant" "wh_usage" {
  warehouse_name = snowflake_warehouse.analytics_wh.name
  privilege      = "USAGE"
  roles          = [snowflake_role.analytics_role.name]
}

resource "snowflake_database_grant" "db_usage" {
  database_name = snowflake_database.analytics_db.name
  privilege     = "USAGE"
  roles         = [snowflake_role.analytics_role.name]
}

resource "snowflake_schema_grant" "schema_read" {
  database_name = snowflake_database.analytics_db.name
  schema_name   = snowflake_schema.core.name
  privileges    = ["USAGE", "SELECT"]
  roles         = [snowflake_role.analytics_role.name]
}

# Attach role to users (example)
resource "snowflake_role_grants" "analytics_role_grants" {
  role_name = snowflake_role.analytics_role.name
```

```
  users     = ["analyst_user1", "analyst_user2"]
}
```

Similarly, Example 3-5 shows how with BigQuery, you could use IaC to define datasets and access roles, while leaving table schemas and scheduled queries to SQL-first tools like dbt or orchestration frameworks.

Example 3-5. Terraform configuration for BigQuery defining a dataset and access roles in a single block

```
# Provider omitted for brevity

resource "google_bigquery_dataset" "sales_ds" {
  dataset_id                  = "sales"
  friendly_name               = "Sales Analytics Dataset"
  description                 = "Dataset for sales reporting and analytics"
  location                    = var.gcp_region
  delete_contents_on_destroy = true

  access {
    role          = "READER"
    user_by_email = "analyst-team@mycompany.com"
  }

  access {
    role          = "WRITER"
    user_by_email = "analytics-ci@my-gcp-project.iam.gserviceaccount.com"
  }
}
```

Another scenario is presented in Example 3-6: using IaC to manage development versus production environments. You might define a template data pipeline architecture in code. When a developer wants their own sandbox, you run the IaC to provision all necessary components (their own schema, perhaps a smaller warehouse, etc.) that mirror production. This promotes better testing and experimentation without risking prod.

Example 3-6. Terraform configuration for Snowflake parameterized by environment (dev versus prod); warehouses, databases, and roles are provisioned consistently, with differences in size and privileges driven by variables

```
variable "env" { type = string } # "dev" or "prod"

locals {
```

```
  name_suffix = upper(var.env)          # DEV / PROD
  wh_size     = var.env == "prod" ? "MEDIUM" : "XSMALL"
}

resource "snowflake_warehouse" "wh" {
  name           = "ANALYTICS_${local.name_suffix}"
  warehouse_size = local.wh_size
  auto_suspend   = 60
  auto_resume    = true
}

resource "snowflake_database" "db" {
  name = "ANALYTICS_${local.name_suffix}"
}

resource "snowflake_schema" "core" {
  database = snowflake_database.db.name
  name     = "CORE"
}

resource "snowflake_role" "role" {
  name = "ANALYTICS_${local.name_suffix}_ROLE"
}

resource "snowflake_warehouse_grant" "wh_usage" {
  warehouse_name = snowflake_warehouse.wh.name
  privilege      = "USAGE"
  roles          = [snowflake_role.role.name]
}

resource "snowflake_database_grant" "db_usage" {
  database_name = snowflake_database.db.name
  privilege     = "USAGE"
  roles         = [snowflake_role.role.name]
}

resource "snowflake_schema_grant" "schema_readwrite" {
  database_name = snowflake_database.db.name
  schema_name   = snowflake_schema.core.name
  privileges    = var.env == "prod" ? ["USAGE","SELECT"] :
                  ["USAGE","SELECT","INSERT","UPDATE","DELETE"]
  roles         = [snowflake_role.role.name]
}
```

IaC is becoming a standard part of data architecture. It treats infrastructure as a part of the development process rather than as a static backdrop. As data teams embrace DevOps/DataOps, knowing how to use IaC (and tools

like Terraform, AWS CloudFormation, Azure ARM templates, among others) is increasingly important. It ensures that as your SQL-driven platform scales, it does so in a controlled, documented manner.

> **NOTE**
>
> From an organizational view, IaC also shifts some responsibilities to data engineers. They are not just writing SQL but also provisioning the environment for it. This is a powerful role that may require upskilling. However, it ultimately makes the team more self-sufficient and speeds up iteration by removing bottlenecks in infrastructure provisioning.

Together, these layers and operational practices transform SQL pipelines into production-grade platforms. They ensure that queries aren't just written but tested, orchestrated, monitored, and deployed as part of a scalable system. For SQL practitioners, this is where writing queries evolves into building data products with solutions that are reliable, maintainable, and able to grow with the business. In Chapter 5, we'll build on this foundation by focusing more directly on advanced SQL modeling practices, showing how concepts such as layered transformations, modular queries, and shared definitions are applied in practice.

Avoiding Common Architectural Anti-Patterns

At this point, we've walked through the layered architecture of SQL-driven pipelines, from ingestion and storage to transformation, orchestration, and consumption, along with the supporting layers that make them production-ready. But even the best-designed architectures are vulnerable to mistakes that quietly impact scalability, maintainability, and trust over time.

These anti-patterns often don't appear as single catastrophic failures. Instead, they sneak in gradually as pipelines grow, requirements change, and "just this once" shortcuts accumulate. Spotting and addressing them

early can mean the difference between a pipeline that scales with the business and one that eventually grinds to a halt under its own weight.

The next sections discuss some of the most common anti-patterns in SQL environments, why they matter, and how to avoid them.

Tight Coupling Between Pipeline Components

Tight coupling happens when downstream processes rely on upstream tables that were never intended for direct consumption, such as dashboards querying staging or intermediate models. When internal structures change, these dependencies break silently, slowing development and reducing trust.

For instance, a dashboard directly queries a staging model that was never meant for consumption (Example 3-7). A small internal change like renaming a column, thus changing deduplication logic, silently breaks downstream users.

Example 3-7. Tight coupling query: bad practice

```
-- Dashboard query
SELECT
  customer_id,
  SUM(amount) AS total_sales
FROM staging.orders
GROUP BY customer_id;
```

A better approach: establish stable, documented output contracts like final tables, semantic layers, or published views for downstream use, while keeping upstream objects private, as represented in Example 3-8. Treat these published datasets like data products: tested, versioned, and owned, so internal changes don't cascade into production failures.

Example 3-8. Reduce tight coupling query

```
-- consume.customer_sales (published model)
SELECT
  customer_id,
  total_sales
FROM consume.customer_sales;
```

Monolithic "All-in-One" SQL Scripts

Putting the entire pipeline—ingestion, transformation, and output—into a single SQL script (Example 3-9) may seem simpler at first, but it quickly becomes fragile. Debugging is harder, intermediate results are invisible, and one failure forces a full rerun. Parallel execution becomes impossible.

Example 3-9. Monolithic "all-in-one" SQL script: bad practice

```
WITH raw AS (...),
cleaned AS (...),
deduplicated AS (...),
aggregated AS (...)
SELECT * FROM aggregated;
```

A better approach: break transformations into modular SQL models with explicit dependencies and materialized checkpoints. Tools like dbt enable partial reruns, parallelism, and targeted optimizations, making the pipeline more maintainable and faster to evolve.

Missing Lineage and Monitoring

Without lineage, you don't know who depends on what. Without monitoring, you may not even know something broke until a user complains. These gaps lead to hesitation when making changes, or worse, to unintentional outages.

A better approach: include observability into the architecture from inception. Track data lineage down to the column level, monitor row counts and freshness, and alert on anomalies. This makes silent failures visible and actionable before they affect decision-making.

Over-Coupling to Specific Technologies

Relying heavily on vendor-specific SQL extensions or proprietary features can lock you into a single platform, making migration or integration costly. One example is the use of the `QUALIFY` clause, as in Example 3-10, which is perfectly valid in some platforms, but not portable.

Example 3-10. Over-coupling to specific technologies: bad practice

```
SELECT
  *,
  QUALIFY ROW_NUMBER() OVER (PARTITION BY id
                             ORDER BY updated_at DESC) = 1
FROM orders;
```

A better approach: use standard SQL for core transformations where possible, and isolate platform-specific logic so it can be replaced if needed. Document these dependencies so there are no surprises during a platform change.

Not Designing for Evolution

Architectures that assume fixed inputs, logic, or scale will eventually break as requirements evolve. Pipelines that can't reprocess historical data or easily add new sources are the first examples that something wasn't correctly designed.

A better approach: parameterize pipelines for backfills, keep sources configurable, and design transformations to accommodate change. Flexibility today is resilience tomorrow.

Skipping Tests for SQL Models

Without automated tests, errors in joins, aggregations, or data assumptions often slip into production unnoticed, eroding trust in the platform.

A better approach: implement tests directly in SQL or via frameworks like dbt. Validate uniqueness, null constraints, and business rules at every stage, not just the final outputs. Testing protects the integrity of your SQL pipelines and catches issues before they reach consumers.

Ignoring Governance and Security

Granting broad, uncontrolled access may be convenient in the short term, but it risks compliance breaches and inconsistent data handling as the platform grows.

A better approach: apply role-based access controls, column-level security, and masking for sensitive fields. Separate sensitive datasets into dedicated schemas and automate retention policies. A secure architecture protects the business and builds trust.

Ultimately, every anti-pattern noted in this section ties directly back to the layered architecture we've been building throughout this chapter. A coupled transformation layer depends on stable consumption contracts. Orchestration works best when pipelines are modular. Observability, lineage, and governance are critical to keeping each layer transparent and safe.

Avoiding these risks isn't about blindly following rules but about preserving the qualities that make SQL-driven pipelines scalable, models reliable, and architectures trustworthy in the first place.

Summary

In this chapter, we explored how SQL-driven architectures evolve from small, monolithic systems to cloud native, hybrid, and multi-cloud environments designed for scale, flexibility, and resilience. We examined each architectural layer, from ingestion to storage and processing, transformation and modeling, orchestration and scheduling, consumption, and the supporting capabilities that keep the whole system reliable and secure.

The main takeaway is that scalable SQL-driven architectures are built, not improvised. They depend on deliberate choices about how layers interact, how workloads are isolated, and how the platform is governed. SQL remains central, not just for querying but as the connective tissue across ingestion, transformation and modeling, and consumption. And while the examples here focus on SQL-first solutions, these ideas aren't unique to SQL. The same principles, such as keeping components simple, reusing what works, continuous testing, and designing with change in mind, apply to almost any kind of data solution, no matter the language or tools you use. Learning them in a SQL-first context gives you a solid base you can carry

into other platforms and technologies, helping your systems adapt as your needs grow.

With the architectural foundation in place, the next step is to go deeper into how data models are actually designed and implemented within this framework. In the next chapter, we will move beyond the staged Bronze/Silver/Gold structure into normalization, dimensional models, denormalization patterns, and newer semantic/graph approaches that affect how SQL is written and governed.

Part II. SQL for Data Engineering and Data Science

Chapter 4. Advanced Data Modeling with SQL

Modern data platforms depend on strong data models to organize information efficiently and meet business requirements. This chapter explores both classic SQL modeling methods and new developments in semantic modeling. We begin with established techniques such as Third Normal Form normalization, star and snowflake dimensional schemas, and the one big table approach, then move on to knowledge graphs and semantic models.

Throughout, we use practical examples in MySQL and other open source tools, set in a Human Resources context, to make each concept clear and actionable. By the end, you'll have a practical understanding of the strengths of each modeling style and see how SQL remains fundamental even as knowledge graphs and other new paradigms become more common.

Traditional SQL Data Modeling Techniques

Early database design focused on organizing data in relational tables, guided by rigorous principles. The goal was typically to minimize redundancy, ensure data integrity, and optimize transactional operations. Let's review three cornerstone techniques that have dominated the industry in the past few decades: 3NF normalization, the star schema (and its variant, the snowflake schema), and the recent, popular approach of one big table (OBT) denormalization. We'll use a simple HR example—employees and departments—to illustrate these.

Third Normal Form—Structured, Relational Modeling

Relational databases often start with a normalized model, commonly achieving Third Normal Form (3NF). A model in 3NF is structured to eliminate data redundancy and update anomalies. In practical terms, each fact is stored once in one place, rather than duplicated across multiple records.

This makes the data model efficient, organized, and free from unnecessary redundancies. By removing transitive dependencies (where a column depends on another non-key column), 3NF ensures that each non-key attribute depends only on the primary key of its table.

This is a classic example of a *transitive dependency*. The attribute `dept_name` depends on `dept_id`, which in turn depends on the primary key of the employee record. In 3NF design, attributes that depend on non-key attributes are moved to separate tables so that each piece of information has a single authoritative location.

Suppose we start with a single table of employees that includes department info as in Example 4-1.

Example 4-1. An inefficient single-table model

```
-- Before normalization: a single table with redundant dept info
CREATE TABLE EmployeeFlat (
    emp_id    INT PRIMARY KEY,
    emp_name  VARCHAR(100),
    dept_id   INT,
    dept_name VARCHAR(100)  -- `Department` name repeats for
                            -- each employee in the dept
);
```

In this design, department names are repeated for each employee within a department, violating Third Normal Form. This is because the department name depends on the department ID, not directly on the employee ID.

To resolve this, we move department details into a separate table and reference them using keys (see Example 4-2).

Example 4-2. A more efficient model

```
-- After normalization to 3NF: separate `Employee` and `Department` tables
CREATE TABLE `Department` (
    dept_id   INT PRIMARY KEY,
```

```
    dept_name VARCHAR(100)
);

CREATE TABLE `Employee` (
    emp_id   INT PRIMARY KEY,
    emp_name VARCHAR(100),
    dept_id  INT,
    FOREIGN KEY (dept_id) REFERENCES `Department`(dept_id)
);
```

As shown in Example 4-2, each department name is stored only once in the `Department` table, while the `Employee` table references departments by their ID.

This follows the core 3NF principle: all data about a given subject, such as departments, is kept in a single location. The result is a model with minimal duplication, reduced data redundancy, and higher information density, ensuring the same information is never stored in multiple places.

A schema designed in 3NF provides a very strong foundation for data integrity. Each piece of information is stored in exactly one place, which dramatically reduces redundancy and ambiguity. If a department name changes, for example, the update happens in a single row in the `Department` table, and that change is immediately reflected everywhere the department is referenced through joins. This design eliminates update anomalies, prevents contradictory values from creeping into the system, and keeps the data model internally consistent.

From an operational perspective, 3NF schemas tend to be clean, predictable, and robust. The structure closely mirrors how transactional systems behave in the real world: employees belong to departments, salaries belong to employees, job titles belong to roles. Because each concept is modeled independently, insert, update, and delete operations behave reliably. This makes 3NF an excellent fit for systems of record such as HR platforms, payroll systems, or identity management systems, where correctness and consistency matter more than query simplicity. In that sense, normalization “cleans up” the data model, enforcing discipline and clarity at the cost of convenience.

That cost becomes visible as soon as you start querying across entities. The primary drawback of 3NF is that related data is spread across multiple tables, which means queries frequently require joins. Even simple questions such as “list employees and their department names” require at least one join between `Employee` and `Department`. As the model grows, queries often involve several joins across multiple tables. These joins are not just harder to write and reason about; they also introduce performance overhead, particularly when tables are large or poorly indexed.

Highly normalized schemas can also become cognitively expensive. An analyst or data scientist must understand the full relational structure to answer even basic analytical questions. In reporting or exploratory analysis, this friction adds up quickly. Queries become longer, harder to maintain, and more prone to subtle errors. While modern databases are quite good at optimizing joins, the fundamental reality remains: normalization optimizes for write correctness, not read convenience. As a result, normalized schemas often struggle to meet the performance and usability expectations of analytics workloads.

This leads to a clear guiding rule when 3NF is the right choice. Normalized models are ideal for transactional systems (OLTP), where data is frequently inserted, updated, and deleted, and where consistency is not negotiable. In an HR system, core tables such as `Employee`, `Department`, `JobRole`, and `Manager` are naturally modeled in 3NF. This ensures that organizational structures remain coherent and that changes propagate correctly throughout the system.

However, when the goal shifts from operating on data to analyzing it—for example, summarizing head count by department, tracking salary evolution over time, or comparing attrition across teams—the limitations of 3NF become apparent. At that point, the normalized model is typically transformed into a more analysis-friendly shape, such as a star schema or a wide denormalized table. These derived models trade some redundancy for faster queries and simpler mental models, while the original 3NF schema continues to serve as the authoritative source of truth.

In practice, successful data platforms rarely choose between normalization and denormalization. Instead, they use 3NF as a stable core and build analytical models on top of it. Understanding this distinction is essential before moving on to dimensional modeling, which embraces many of the trade-offs that 3NF deliberately avoids.

Dimensional Modeling: Star Schema for Analytics

Imagine the frustration during the end-of-quarter review when the dashboard takes ages to load, delaying decision-making. This sluggish performance can often be traced back to traditional relational databases designed for transaction processing rather than complex analytical queries. When the objective shifts from simply operating on data to extracting insights from it, relational normalization quickly reveals its limitations. Analytical workloads require models that are easy to query, intuitive to understand, and perform well at scale. As a result, one of the most widely adopted strategies in analytics and data warehousing is the star schema, a particular form of dimensional modeling designed for these needs.

A star schema organizes data around two core concepts: facts and dimensions. At the center lies a fact table that stores measurable, quantitative values, such as amounts, counts, or durations. In an HR context, these facts might include head count, hiring volume, attrition events, or salary measures. Surrounding the fact table are dimension tables that provide descriptive context for the measures, for example, `Employee`, `Department`, `Date`, `Location`, or `JobRole`. Visually, the structure resembles a star: a central fact table connected to several dimension tables via foreign keys, each representing a different analytical perspective.

Unlike a fully normalized 3NF design, a star schema is intentionally optimized for data retrieval instead of data maintenance. Each dimension is typically stored in a single, denormalized table. Consequently, analytical queries usually require only one join per dimension, regardless of how many descriptive attributes are needed. This dramatically simplifies query patterns. In practice, a star schema requires only one level of joins between

the fact table and each dimension, significantly reducing query complexity and improving performance for BI workloads.

A helpful mental model for dimensional design is to distinguish between what you measure and how you slice it. If someone refers to measuring something, that item is a fact. If they wish to measure it by something, that "by" attribute is a dimension. For example, "salary amount" or "employee count" are facts, while phrases like "by department," "by month," or "by location" indicate dimensions. This rule applies across domains and is especially effective in HR analytics, where questions often focus on measuring people-related metrics across organizational structures and time.

To make this concept more tangible, envision a spreadsheet where each row represents a unique department and month combination. For an organization aiming to analyze employee counts per department over time, a standard star schema would feature a fact table called `FactHeadcount`. This table captures one row per department and time period, storing a metric such as `employee_count`. Surrounding it are dimension tables such as `DimDepartment`, containing department descriptors like `name` and `location`, and `DimDate`, which holds time-related attributes like `month`, `year`, and `quarter`.

In this design, the fact table contains only keys and measures, while descriptive attributes reside solely in the dimensions. The fact table links to each dimension via foreign keys, for example, `FactHeadcount.dept_id` referencing `DimDepartment.dept_id`. To answer a question like "employee count by department in 2026," the query simply joins the fact table to the `Department` and `Date` dimensions and filters by `year`. Because the dimensions are not further normalized, all relevant attributes are immediately accessible after the join.

This simplicity is the star schema's greatest strength. Analytical queries become shorter, more readable, and easier to reason about. Analysts do not need deep knowledge of the underlying operational schema. Fewer joins typically result in faster execution, especially in column-oriented analytical databases. Star schemas are also highly intuitive; often, even nontechnical

users can understand the model simply by looking at the table names and relationships.

Another major advantage is consistency through shared dimensions. The same-dimension table can be reused across multiple fact tables. For example, in an HR warehouse, a single `DimDepartment` might be shared by `FactHeadcount`, `FactHiring`, and `FactSalary`. This concept, known as conformed dimensions, ensures that all analyses use the same definitions. "`Department`" means the same thing everywhere, which is critical for trustworthy reporting and valid cross-metric comparisons.

Because of these advantages, star schemas are widely considered a cornerstone of data warehousing. They strike a practical balance between rigor and usability. To understand the full data journey, consider this linear flow: data moves from the source to a normalized 3NF model for correctness, then to star schema tables for analytics, and finally to dashboards for end-user insights. A common architectural pattern is to ingest and integrate data into a normalized 3NF model for correctness, and then populate star schema tables as a downstream step specifically for analytics. This approach preserves data integrity while delivering fast, easy-to-consume models.

Snowflake Schemas: Adding Normalization to Dimensions

A snowflake schema is a variation of the star schema that introduces additional normalization within the dimensions. Rather than storing all dimension attributes in a single table, some attributes are extracted into related subdimension tables, creating chains of joins that branch outward from the fact table. Visually, this structure resembles a snowflake rather than a star.

For instance, in the HR scenario, if the `Department` dimension includes a `Location` attribute that is itself a complex entity, the `Location` data might be normalized into a separate `DimLocation` table. The `FactHeadcount` table would join to `DimDepartment`, which then joins to `DimLocation`. This

change transforms the query path by adding an extra join. To answer a query like "employee count by location in 2026," the query would now need to navigate from `FactHeadcount` to `DimDepartment` and then to `DimLocation` to filter by the desired `Location` attribute. While this structure maintains the core fact-dimension framework, it introduces additional complexity by splitting dimensions into multiple related tables.

In practice, the storage savings from snowflaking are often modest. Dimension tables are typically small relative to fact tables, and modern databases compress repeated values efficiently. As a result, many teams prefer the simplicity of star schemas and accept some redundancy in exchange for easier queries and better performance. Snowflake schemas tend to make sense only when dimensions are especially large, highly hierarchical, or difficult to maintain in a single table.

In summary, a snowflake schema is best understood as a more normalized version of the star schema. It trades query simplicity and speed for reduced redundancy and stricter structure. While this approach is appropriate in certain situations, most analytical workloads favor classic star schemas, especially when performance, usability, and clarity are priorities.

To illustrate the practical differences between star and snowflake schemas, let's use the `FactHeadcount` example to see how the queries compare.

Example 4-3 shows a schema for a dimensional head count model.

Example 4-3. A star schema DDL: dimensional head count model

```
CREATE TABLE `DimDepartment` (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(100),
    location_id INT
);

CREATE TABLE `DimLocation` (
    location_id INT PRIMARY KEY,
    location_name VARCHAR(100)
);

CREATE TABLE `FactHeadcount` (
    dept_id INT,
    date_id INT,
```

```
    employee_count INT,
    PRIMARY KEY (dept_id, date_id)
);
```

In a star schema, where the department name is stored in the `DimDepartment` table, retrieving head count by department is straightforward, as shown in Example 4-4.

Example 4-4. A star schema query

```
SELECT d.dept_name, SUM(f.employee_count) AS total_employees
FROM `FactHeadcount` f
JOIN `DimDepartment` d ON f.dept_id = d.dept_id
WHERE f.year = 2026
GROUP BY d.dept_name;
```

In the star schema query, retrieving data such as head count by department is straightforward—only a single join between the fact table and the department dimension is required. This simplicity is one of the main reasons star schemas are favored for analytical workloads.

By contrast, in a snowflake schema as shown in Example 4-5, the model is more normalized—department and location are split into separate tables. For example, the `DimDepartment` table might store a `location_id`, and a separate `DimLocation` table holds the location details. As a result, to group results by location name, you need to join the fact table to the department dimension and then to the location dimension as well.

Example 4-5. A snowflake query

```
SELECT loc.location_name, SUM(f.employee_count) AS total_employees
FROM `FactHeadcount` f
JOIN `DimDepartment` d ON f.dept_id = d.dept_id
JOIN `DimLocation` loc ON d.location_id = loc.location_id
WHERE f.year = 2026
GROUP BY loc.location_name;
```

In a snowflake schema, retrieving information such as head count by location requires two joins: first from the fact table to the department dimension, and then from department to location.

If the schema were a star, with location details already included in the `DimDepartment` table, only a single join would be necessary. This

illustrates how the additional normalization in a snowflake schema can add complexity to queries compared to the star approach.

In both star and snowflake designs, SQL queries are far simpler than those required for a fully normalized OLTP schema. Instead of joining multiple tables such as `Employees`, `Departments`, and various assignment tables, analysts work with a small, predictable set of joins between a fact table and its dimensions. For example, imagine a monthly report that needs to calculate the head count by department. In a dimensional model, this task requires only a straightforward join of the `Employee` fact table with the `Department` dimension. This straightforwardness allows analysts to quickly generate insights, focusing their energies on answering business questions rather than untangling complex operational relationships. This simplicity is why dimensional models are so popular for user-friendly analytics.

Ultimately, choosing between a star schema and a snowflake schema depends on intent and context. Star schemas are typically the default for data warehousing and BI, especially when the goal is fast, straightforward queries over large datasets. These schemas excel in scenarios where query patterns are well-defined and repeated frequently, such as in dashboards and standard reports. For instance, a metric-driven dashboard showcasing daily sales margins can greatly benefit from the speed and efficiency of star schemas, enabling executives to make quick, informed decisions that can impact revenue. The denormalized dimensions keep queries concise and predictable, which is critical for supporting many concurrent users, and ensures that business-critical insights are delivered promptly and accurately.

In contrast, snowflake schemas are best viewed as a refinement, not a replacement, for star schemas. They are helpful when dimensions have deep hierarchies or when a large dimension table becomes unwieldy. Normalizing parts of a dimension can improve maintainability and structure, but it introduces more joins and complexity. Snowflaking should therefore be used selectively and only with clear justification. Before deciding to snowflake a dimension, teams should consider asking: *Will this separation significantly reduce data load times?* and *Could it simplify the understanding of data relationships for our analysts?* These questions can

help ensure that the decision to snowflake adds tangible value without unnecessarily complicating the model.

In modern analytical databases, raw performance is just one factor among many. Clarity and maintainability of the model have become equally important, as many systems now deliver strong performance even with highly denormalized schemas. As a result, the decision between star and snowflake schemas is often guided more by ease of use, long-term maintenance, and team familiarity than by execution speed alone. Most organizations benefit from starting with a star schema and introducing snowflaking when a specific dimension clearly justifies it.

The most effective dimensional models are pragmatic, not dogmatic. Start with a star schema to maximize accessibility and performance, and use snowflake dimensions only when complexity or maintenance truly demands them. By staying flexible and adapting to the data's real-world shape, you keep analytical models both efficient and approachable, laying a strong foundation for whatever future innovations data modeling may bring.

One Big Table: The Wide Denormalized Table Approach

An increasingly common modeling approach in the era of large-scale analytics and cloud data warehouses is the one big table (OBT) design. As the name suggests, this approach places all relevant analytical data into a single, wide table, denormalized to the extreme. OBT enables rapid insights by having all necessary data in one place, instantly accessible: for example, quickly determining margin by channel during a promotion period. Conceptually, an OBT fully flattens a star schema, joining the fact table with all of its dimensions in advance so that measures and descriptive attributes coexist as columns.

You may encounter this approach under several names, including wide tables, flat tables, or single-table models. Regardless of terminology, the core idea is to shift complexity from query time to data preparation time. This radical simplicity creates a table containing everything a typical analytical query needs but also introduces significant maintenance burdens.

The core motivation behind OBT is radical simplicity. Analytical queries require only scanning and filtering a single table, with no joins required. All necessary attributes, such as department name or location, are present as columns. This makes queries shorter, easier to write, and simpler to understand.

For example, instead of storing department identifiers in a fact table and resolving them via joins, an OBT includes `dept_name`, `location`, and other department attributes directly with each fact row. This allows questions like "employee count by department in 2026" to be answered with a simple aggregation and no joins required. At this point, teams must decide whether to store attributes as they were when the fact occurred (as-was) or as they are now (as-is). This choice affects historical analysis and consistency.

This design aligns extremely well with the execution model of modern columnar analytical databases. Systems such as BigQuery, Snowflake, and Redshift are optimized to efficiently scan large tables, reading only the columns required by a query. In this environment, avoiding joins can significantly reduce query latency. In practice, performance benchmarks have shown that a single denormalized table can outperform an equivalent star schema by a substantial margin, precisely because it avoids runtime join costs. This reality has led some platforms to offer pragmatic advice: simply denormalize aggressively when performance matters most.

OBT can also simplify data pipelines. Instead of building and maintaining multiple-dimensional tables and carefully managing foreign keys, a team can focus on producing a single analytical dataset tailored to a specific use case. For small teams, early-stage projects, or narrowly scoped analyses, this can dramatically reduce time to value. This scenario could be labeled as a *prototype phase* or *minimum viable mart,* helping teams map the advice to their specific project maturity. These labels signal when it might be appropriate to sunset or further develop the OBT into a more robust solution. In HR analytics, for example, a single `Employee360` table containing personal attributes, organizational context, and derived metrics can be far easier for consumers to work with than a constellation of related tables.

That said, the benefits of OBT come with significant trade-offs. Chief among them is redundancy. The same descriptive values are repeated across many rows, complicating updates and change management. For example, if a department name changes, that update must be propagated everywhere it appears. Managing historical correctness is especially challenging. Storing point-in-time (as-was) versus current-state (as-is) values is a key decision. This redundancy can also result in a storage multiplier effect, with requirements sometimes tripling due to dimension bloat. This is an important consideration for budgeting and stakeholder discussions.

OBTs also tend to grow very wide, often containing dozens or even hundreds of columns. This can make them difficult to understand, document, and evolve. Adding a new attribute sourced from a different system may require recomputing or backfilling large portions of the table. While incremental processing and partitioned backfills can mitigate this cost, the underlying reality remains: OBTs are less flexible in the face of schema change.

There are also governance and compliance considerations. Regulations such as GDPR require that personal data be correctable and, in some cases, removable. In a heavily denormalized table, identifying and updating all instances of an individual's data can be cumbersome. While OBT can satisfy regulatory requirements, it demands careful planning, strong metadata, and robust update mechanisms. Assigning clear responsibilities, such as a privacy officer for compliance and a data engineer for update workflows, clarifies accountability and supports effective compliance.

From a modeling perspective, OBT represents a philosophical shift. Structure is flattened away, so concepts previously defined through dimensions and relationships become implicit and duplicated. While this may be acceptable for isolated or short-term use cases, it limits reuse and semantic consistency across an organization. For example, a conformed `Customer` dimension ensures "VIP" has a consistent meaning. In contrast, independently built OBTs may encode business rules differently, leading to fragmentation. Dimensional models with conformed dimensions provide a shared foundation that OBTs alone cannot guarantee.

In practice, OBTs are best treated as delivery artifacts rather than foundational models. They are often used as downstream marts tailored for specific tools, dashboards, or performance-critical workloads. Some BI tools, like Power BI, are better suited to flat inputs due to technical constraints. This makes OBTs especially useful. They also serve as quick wins, helping teams meet immediate analytical needs while more structured models are developed.

Modern ELT workflows often embrace this hybrid approach. A normalized or dimensional warehouse serves as the system of record, while one or more OBTs are materialized on top for convenience and speed. Tools like dbt make this pattern easy to implement, allowing a single SQL model to join fact and dimension tables into a wide representation. Analysts benefit from querying the OBT for simplicity (see Example 4-6), while engineers maintain the underlying models for flexibility and governance.

In summary, OBT modeling trades structural elegance for brute-force performance and usability. It works exceptionally well with modern analytical engines and can dramatically simplify consumption. However, it increases maintenance costs, reduces flexibility, and requires greater discipline around governance. For these reasons, OBT is most valuable when used deliberately and in conjunction with well-modeled underlying data—not as the sole representation of analytical truth.

Example 4-6. A one big table (OBT) query

```
SELECT
    year,
    month,
    location_name,
    department_name,
    SUM(employee_count)        AS total_employees,
    AVG(avg_salary)            AS average_salary,
    COUNT(DISTINCT employee_id) AS unique_employees
FROM `HR_Analytics_OBT`
WHERE year = 2025
  AND location_region = 'EMEA'
GROUP BY
    year,
    month,
    location_name,
```

```
    department_name
ORDER BY
    year,
    month,
    location_name,
    department_name;
```

Lakehouse Modeling and ELT Workflows

In modern cloud data platforms, dimensional models and wide analytical tables are often built using ELT workflows inside a lakehouse architecture. Rather than transforming data before loading it into the warehouse, ELT pipelines first load raw data into a central storage layer and then apply transformations using SQL.

In this architecture, the data lake stores raw and semi-structured data, while the warehouse or lakehouse engine performs transformations and analytics directly on top of that data. Tools such as dbt have become a common way to manage these transformations. dbt allows teams to define analytical models as SQL files, organize them into dependency graphs, and materialize them as tables or views.

Within a dbt-driven workflow, a typical modeling pattern might look like this:

- Raw ingestion tables capture source system data with minimal transformation.
- Staging models standardize naming, types, and basic cleaning.
- Intermediate models apply business logic and join datasets.
- Final models expose star schemas, wide analytical tables, or domain-specific marts.

This layered approach aligns well with many of the modeling strategies discussed in this chapter. Normalized schemas often exist in staging layers to ensure correctness, dimensional models appear in curated analytical layers, and OBTs may be materialized as downstream marts optimized for dashboards or specific tools.

Lakehouse architectures therefore do not replace traditional data modeling techniques; instead, they provide a modern operational framework for building and maintaining them using SQL, version control, and automated pipelines.

Summary of Traditional Models

We've examined several traditional data modeling techniques, each with its unique strengths and trade-offs. Consider a retail sales transaction scenario to illustrate these models. 3NF offers robust data integrity and consistency. It is ideal for ensuring accurate and reliable records in transactional systems, but it can increase query complexity when retrieving sales data across different stores. A star schema is more suited for analytical contexts. It simplifies data retrieval and allows for faster generation of reports on sales trends, but it introduces some redundancy due to its denormalized nature. An OBT approach optimizes for query simplicity and speed by consolidating all sales-related data into a single wide table. This leads to quick access to sales summaries, though it may incur higher storage costs and redundancy. Choosing the appropriate model depends on your specific goals and use cases.

Table 4-1 summarizes the data modeling approaches used in SQL-based systems, from fully normalized relational models to highly denormalized analytical layouts. Rather than presenting these models as competing alternatives, the comparison highlights the trade-offs each approach makes in data integrity, query performance, flexibility, governance, and usability. In practice, modern data platforms often use several of these models simultaneously, applying each where it fits best. Understanding these differences is essential before moving beyond tabular models toward semantic and graph-based representations of data.

Table 4-1. Common data modeling approaches in SQL analytics, highlightin offs in integrity, performance, flexibility, and usability

Category	3NF (normalized)	Star schema	Snowflake schema	One table
Primary goal	Maximize data integrity and correctness; minimize redundancy	Optimize analytical querying and usability	Balance normalization with analytical access	Maxi query simp perfo
Typical use case	OLTP systems and systems of record (for example, HR core systems)	Data warehouses and BI reporting	Warehouses with complex or hierarchical dimensions	Perfo critic analy dashb and t speci
Structure	Highly normalized tables with many relationships	One fact table surrounded by denormalized dimensions	Fact table with partially normalized dimensions	Singl table conta facts dime
Query complexity	High: many joins required, even for simple questions	Low: one join per dimension	Medium: additional joins inside dimensions	Very joins

Category	3NF (normalized)	Star schema	Snowflake schema	One table
Read performance	Poor to moderate for analytics workloads	High and predictable	Moderate to high, depending on join depth	Very mode colur datab
Write and update efficiency	Excellent: changes applied in one place	Good, but dimension updates may affect multiple rows	Better than star for some dimension updates	Poor: may touch rows
Schema flexibility	Rigid but stable	Moderately flexible	More rigid as dimensions deepen	Low: chang requi backf
Data redundancy	Minimal	Moderate and intentional	Lower than star, higher than 3NF	High
Governance and compliance	Strong: easy to correct and audit data	Strong with proper dimension management	Strong but more complex	Requ caref planr GDP data

Category	3NF (normalized)	Star schema	Snowflake schema	One table
Cognitive load for users	High: requires deep schema knowledge	Low: intuitive and business-friendly	Medium: additional joins add mental overhead	Very table like a sprea
When it breaks down	Analytics become slow and hard to maintain	Very wide or rapidly changing dimensions	Query performance and complexity grow	Chan mana consi and r suffe

Next, let's shift gears to the innovations in data modeling that have emerged in recent years. Specifically, we'll discuss semantic models and knowledge graphs. These approaches extend beyond traditional tables and rows to capture the meaning of data and its complex relationships. This shift addresses a significant limitation of conventional models. Traditional models struggle with metadata sprawl, schema drift, and complex cross-domain joins. By resolving these pain points, semantic models and knowledge graphs offer more flexibility and clarity, making them increasingly compelling for modern databases. We'll also explore how these new paradigms integrate with the SQL world.

Knowledge Graphs and Semantic Modeling: The New Frontier for SQL

In the last decade, knowledge graphs have gained popularity as a way to model and integrate data, especially in complex domains. This renewed interest is fueled by the pressing needs of today's business landscape. For

instance, when an organization undergoes a merger and must reconcile disparate HR systems overnight, a knowledge graph can efficiently map and connect diverse data sources.

Unlike a relational model, which organizes information in tables, or a star schema, which still relies on predefined tables and relationships, a knowledge graph represents data as a network of entities and relationships. In this network, entities are the nodes and relationships are the edges. Alongside knowledge graphs, semantic modeling has also emerged, using formal ontologies and metadata to give data deeper meaning.

Although these graph models may appear quite different from traditional relational databases, they build on similar foundational ideas. Just as 3NF seeks to minimize redundancy and ensure data integrity through carefully designed relationships, knowledge graphs focus on making relationships and entities explicit to provide a comprehensive view of the data landscape. In the following sections, we will explore knowledge graphs, show how they differ from traditional models, and examine how SQL fits into this new paradigm.

What Is a Knowledge Graph?

At its core, a knowledge graph is a collection of interlinked facts. It organizes information as a graph, consisting of nodes and edges. The nodes represent entities, such as real-world objects or abstract concepts. The edges show the relationships between these entities. In this context, a *fact* is an atomic assertion about entities or their relationships. This differs from the *fact table* in data warehousing, which typically contains quantitative data. Instead, a knowledge graph is a semantic data model that connects entities and relationships within a unified network.

There are several definitions of knowledge graphs, but most agree that they visualize the connections between interlinked entities. For example, one might define a knowledge graph as a structure that shows how real-world objects and concepts are related. Another common definition describes a

knowledge graph as a collection of interlinked facts on a specific topic or domain.

Knowledge graphs are often categorized by their scope. An enterprise ontology covers a wide range of data within an organization and usually requires significant governance. In contrast, a domain ontology focuses on a specific subject area and often demands less oversight. Understanding the type of ontology you are building is important because it influences the knowledge graph's scope, complexity, and maintenance requirements.

To support real-world knowledge graphs, a growing ecosystem of specialized technologies has emerged. Neo4j is one of the most prominent and widely adopted graph databases, offering robust support for storing and querying complex networks of connected data. All of these databases are designed to efficiently manage nodes and edges while enabling rapidly traversing intricate relationships.

A major innovation in this field is the development of intuitive, pattern-based graph query languages. Cypher, created by Neo4j, has become the most popular and accessible of these languages. It allows users to express queries in a way that closely mirrors natural graph thinking. For instance, to find all departments connected to Alice through a `memberOf` relationship, a Cypher query would look like Example 4-7.

Example 4-7. A knowledge graph query (Cypher)

```
MATCH (alice:Person {name: 'Alice'})-[:memberOf]->(dept:`Department`)
RETURN dept;
```

Because relationships are stored directly in the graph, the database engine can quickly retrieve Alice's departments by following the existing edges, without needing costly join operations. This pre-materializing of relationships not only improves query performance but also provides a more intuitive way to explore connected data.

With Neo4j, Cypher, and similar technologies, you can see practical examples of how graph databases empower powerful traversal and pattern-matching queries that go beyond what is possible with traditional SQL.

Graph Technologies: Neo4j and Cypher

In the world of finance, identifying and preventing fraud is a mission-critical task. Imagine how a bank might use graph database technology to detect fraud rings.

By mapping intricate connections between transactions, account holders, and external agents, the bank can uncover hidden patterns of fraudulent activity that would be difficult to detect with traditional database queries. To support these innovative data querying approaches, a new generation of graph database technologies has emerged. Neo4j stands out as one of the most widely used and recognized platforms for managing and exploring connected data. In addition to Neo4j, leading cloud providers offer their own graph database solutions, including Amazon Neptune, Azure Cosmos DB (with Gremlin API), and TigerGraph.

These platforms are designed to efficiently store, traverse, and analyze networks of relationships at scale. A key innovation of graph databases is their use of specialized query languages. Cypher, created by Neo4j, is the most widely adopted. It allows users to write expressive, pattern-based queries that mirror the way people naturally think about traversing networks. For example, a simple Cypher query like `MATCH (a)-[r]->(b) RETURN a, r, b` illustrates how relationships are pre-materialized, showing direct connections without the need for inference. These pre-materialized relationships (connections between entities) are stored as direct pointers instead of being inferred from keys. This makes traversing relationships efficient and intuitive, eliminating the need for the costly join operations common in traditional relational systems.

This design has several important implications:

Relationships are treated as first-class data, not as secondary artifacts.

> In traditional relational databases, relationships are often managed through foreign keys, which can complicate data architecture and make models more challenging to comprehend. This side-by-side comparison highlights the simplicity and clarity brought by treating connections as

first-class entities, fundamentally shifting how we understand and organize data.

Query structures naturally reflect real-world connections.

For many data modelers, a common frustration is dealing with endless join rewrites in traditional relational databases, since trying to accurately reflect complex relationships often leads to cumbersome, convoluted SQL queries. Recognizing these headaches, graph databases offer a welcome relief. By enabling more intuitive data modeling, they save time and reduce complexity in query development, making the design process less stressful and more efficient.

Complex, join-heavy questions become simpler and more intuitive, which enables more efficient data retrieval and faster decision-making.

For instance, what traditionally required a cumbersome 12-table join operation in a relational database can often be reduced to a simple two-hop traversal in a graph database. This dramatic simplification not only saves time but also makes complex queries more accessible and easier to manage.

Performance depends on the depth of relationships traversed, not on table size.

Think of it like a subway ride: the more stations you stop at, the longer your trip takes. Similarly, in a graph database, the deeper the relationship, the more computational effort required. This approach results in optimized operations and can reduce computational costs, making it efficient for queries focusing on deeply interconnected data.

However, it is important to remember that graph queries do not always outperform SQL in every scenario. For tasks like aggregations and set-based analysis—such as calculating monthly revenue—relational databases often deliver better performance and simpler solutions. SQL excels at handling large datasets for numerical calculations and aggregate results. In

contrast, for relationship-centric questions, Cypher and other graph query languages offer a level of expressiveness and clarity that is difficult to achieve with SQL alone.

This distinction grows even more significant as organizations move beyond traditional tables and adopt semantic models or ontologies. In these newer paradigms, relationships carry explicit meaning, and inference becomes a central aspect of querying and analysis. To transition smoothly from current graph models to future Web Ontology Language (OWL) work, it is essential to focus on data mapping and vocabulary alignment, bridging today's practices with tomorrow's ontology. This initial step ensures that the shift from tables to graphs is manageable, not mysterious. The change is not merely technical; it represents a fundamental transformation in how we understand and organize complex information. In the coming chapters, we will explore how to design ontologies and map relationships using frameworks like OWL, equipping you to harness the full capabilities of semantic technology.

Querying with Cypher Versus SQL

To understand how graph queries differ from SQL, let's continue with a concrete example from the Human Resources domain.

Imagine an HR knowledge graph where employees, departments, divisions, skills, and projects are all modeled as nodes. Relationships between these entities are explicit and typed. If Alice works in the Sales department, the graph contains a relationship between the `Person` node representing Alice and the `Department` node representing Sales, labeled `memberOf`. If Sales belongs to a larger division, another relationship connects Sales to a `Division` node using `partOf`. If Alice has skills or certifications, additional relationships, such as `hasSkill`, connect her node to `Skill` nodes.

Unlike relational models, which infer relationships through joins, a knowledge graph stores these connections directly. This enables queries to traverse the data model by following explicit relationships, making the

connections between entities both visible and actionable. The result is a more natural way to represent and query real-world scenarios.

Consider the question: “Which departments does Alice belong to?”

In a knowledge graph, this question is answered using a direct traversal from the person node to the connected department nodes. This approach highlights the intuitive nature of graph queries: rather than specifying join logic, you simply describe the path you want to follow, as shown in Example 4-8.

Example 4-8. A knowledge graph query (Cypher)

```
MATCH (alice:Person {name: 'Alice'})-[:memberOf]->(dept:`Department`)
RETURN dept;
```

This example follows the existing `memberOf` relationship from the `Person` node representing Alice to the related `Department` nodes. There is no need for a join table, no foreign key matching, and no inference at query time; all connections are already established in the graph. The structure of the query closely matches the way we think about organizational relationships.

Here’s a more complex scenario. Suppose you want to identify all employees who work in the Sales Division.

In a knowledge graph, answering this question requires following a chain of relationships through the data. Each step along the way—employee to department, department to division—represents a meaningful connection. The power of graph databases and their query languages lies in their ability to express this logic concisely. Even as relationships grow in number and complexity, the queries remain readable and closely mirror the structure of the real-world organization, as shown in Example 4-9.

Example 4-9. A multi-hop graph traversal (Cypher)

```
MATCH (p:Person)-[:memberOf]->(:`Department`)-[:partOf]->(:Division {name:
'Sales'})
RETURN p.name AS employee_name;
```

In this example, the graph query outlines a path through the network of data, rather than piecing together results from a series of table joins as in

SQL. Each relationship is traversed directly, allowing the query's structure to closely mirror the original business question. This approach makes graph queries especially powerful for uncovering connections and dependencies that might be obscured by a tabular model.

One of the core strengths of Cypher and graph databases is their readability, even as you introduce more conditions or relationship types. For instance, you can combine multiple relationship constraints in a single, straightforward query.

Suppose you want to find all employees in the Sales division who possess the skill "Data Analysis."

Step 1 is to locate the Sales division. The query starts by identifying the Sales division within the graph, using the direct relationships among department, division, and employee nodes.

Step 2 is to filter `Employees` by `Skill`. Next, the query filters the employees based on their skills, selecting only those who have `Data Analysis` in their skill set. This stepwise approach brings clarity to query formulation and reflects the natural sequence of reasoning that analysts use.

Each `MATCH` clause introduces a new constraint but retains the overall query clarity. Rather than specifying how tables should be joined, you simply describe the connections between entities. This approach makes the logic easier to follow, maintain, and adapt as business questions evolve. Example 4-10 illustrates how this can be achieved with Cypher.

Example 4-10. A multi-constraint knowledge graph query (Cypher)

```
MATCH (p:Person)-[:memberOf]->(:`Department`)-[:partOf]->(:Division {name:
'Sales'})
MATCH (p)-[:hasSkill]->(:Skill {name: 'Data Analysis'})
RETURN p.name AS employee_name;
```

Semantic Models and Ontologies

A semantic model goes beyond capturing just the structure of data; it encodes the meaning behind it. Instead of focusing solely on storing values,

a semantic model explicitly defines what entities represent, how they are connected, and the rules that govern their relationships.

In knowledge graphs, semantic modeling is most often realized through an ontology. An *ontology* serves as a schema for the graph, defining the types of entities within a domain, the relationships among them, and, at times, logical constraints or hierarchical structures.

A semantic knowledge model captures and represents the meaning of data by explicitly defining domain-relevant objects, concepts, and the relationships between them. This approach produces a model that humans can understand and machines can interpret.

For example, in an HR domain, a semantic model might:

- Define `Employee` and `Department` as entity types. An employee works in a department, and a department has a name attribute. These entities are often represented using URIs, enabling data to be linked across different systems. For instance, a semantic identifier like “ex: `Employee/123` `worksIn` ex: `Department/Sales`" illustrates how such connections can be established and understood across organizational boundaries.
- Capture that an employee reports to another employee, further enriching the organization’s web of relationships.

Once these concepts and relationships are formally defined, any system or application accessing the data can interpret its meaning, not just its storage. This shift enables more intelligent data integration, sharing, and automated reasoning.

RDF and OWL: The Foundations of Semantic Modeling

The foundational standards for semantic modeling on the web are Resource Description Framework (RDF) and Web Ontology Language (OWL). Both are defined by the World Wide Web Consortium (W3C) as part of the Semantic Web stack and are widely used for building knowledge graphs.

RDF represents data as triples: subject, predicate, and object. For example, imagine Alice starting her first day in the Sales department. This fact can be modeled as a triple: *Alice works in Sales,* with `worksIn` as the predicate connecting her to her department. Each element is identified by a URI, ensuring global uniqueness and interoperability. As more triples are added, a rich and interconnected graph of information emerges.

OWL builds on top of RDF, allowing for much richer domain definitions. With OWL, you can define classes, properties, hierarchies, and even logical rules that support inference. For example, you can express rules that allow new facts to be inferred from existing relationships. Consider the following scenario:

- An employee works in the Sales department.
- The Sales department is part of the North Division.
- Therefore, by logical inference, the employee works in the North Division as well.

Reasoners use these rules to automatically derive new facts, showcasing OWL's powerful reasoning capabilities. This automated inference transforms static data into a dynamic web of knowledge, enabling systems to answer questions that go beyond explicitly stored facts.

Together, RDF and OWL provide the foundation for semantic knowledge graphs, structures that enrich data with explicit meaning and enhance it with powerful reasoning. This approach elevates the value and utility of data, enabling organizations to unlock deeper insights and support advanced, intelligent applications.

Example 4-11 shows how simple HR facts and schema constraints can be expressed using RDF triples.

Example 4-11. RDF triples for an HR domain

```
:Alice       :worksIn     :SalesDept .
:SalesDept   :hasName     "Sales" .
:worksIn     rdfs:domain  :`Employee` .
:worksIn     rdfs:range   :`Department` .
```

The first two triples represent data. The latter two define ontology rules, stating that `worksIn` links an `Employee` to a `Department`. These schema-level statements allow systems to infer types and validate relationships.

RDF data is typically queried using SPARQL query language, and this is where SQL comes on the scene again. SPARQL plays a role similar to SQL but operates over graph patterns instead of tables, as in Example 4-12.

Example 4-12. A SPARQL query

```
PREFIX ex: <http://example.com/hr#>

SELECT ?empName ?deptName
WHERE {
  ?emp a ex:`Employee` ;
       ex:worksIn ?dept ;
       ex:hasName ?empName .
  ?dept ex:hasName ?deptName .
}
```

This query matches patterns in the graph, navigating relationships rather than joining tables. While this approach seems unfamiliar at first, the underlying idea mirrors SQL joins conceptually, operating directly on graph structures.

Property Graphs and Cypher

While RDF centers on semantic web standards, many organizations turn to property graphs for more flexible data modeling. In property graphs, both nodes and relationships can have labels and properties, making it easier to represent rich, real-world contexts. Neo4j is one of the leading property graph databases and features the Cypher query language, designed to make graph pattern matching intuitive and expressive.

In a property graph model:

- Nodes represent entities, such as people or departments. Each node can have one or more labels that describe its type, as well as properties that store relevant information.

- Relationships connect nodes, are always directed (from one node to another), and are typed to indicate their meaning, for instance, `WORKS_IN` or `MANAGES`. Relationships can also store properties, such as the date when the relationship was established.

Cypher queries use a clear, visual syntax inspired by ASCII art to describe patterns in the graph. This syntax allows straightforward expression of even complex graph traversals in a way that is both readable and closely aligned with how people visualize networks (see Example 4-13).

Example 4-13. A simple Cypher query

```
MATCH (e:`Employee`)-[:WORKS_IN]->(d:`Department` {name: "Sales"})
RETURN e.name;
```

This query directly mirrors the mental model of the graph: find employees connected to the Sales department via a `WORKS_IN` relationship. For comparison, the equivalent SQL query requires a join as shown in Example 4-14.

Example 4-14. An equivalent SQL query

```
SELECT e.emp_name
FROM `Employee` e
JOIN `Department` d ON e.dept_id = d.dept_id
WHERE d.dept_name = 'Sales';
```

PostgreSQL and GraphDB

An interesting insight into how SQL and Cypher can work together to deliver a hybrid approach comes from PostgreSQL.

PostgreSQL's approach to graph querying is purposefully incremental and rooted in its robust SQL heritage. Rather than reinventing the wheel, PostgreSQL enables organizations to layer graph capabilities on top of their existing relational infrastructure. This is achieved through extensions like Apache AGE, which allow developers to define property graphs natively within PostgreSQL. With Apache AGE, you can create nodes and relationships as first-class citizens and then populate the graph directly from your established relational tables. These relationships and entities are

accessible with Cypher-style query patterns, making it possible to harness the expressiveness of graph traversal while retaining the familiarity and reliability of SQL.

In practical deployments, SQL continues to serve as the foundation for core data operations—handling ingestion, cleansing, transformation, and traditional analytics. Graph queries, on the other hand, are introduced purposefully to address scenarios where relationships and network patterns are the focus. For example, Example 4-15 demonstrates a traversal that navigates from a person to their associated department, all without the complexity of join operations. Importantly, this query is executed entirely within PostgreSQL, eliminating the need to export data or maintain a separate graph database.

Example 4-15. Querying a property graph inside PostgreSQL

```
SELECT *
FROM cypher('hr_graph', $$
  MATCH (p:Person {name: 'Alice'})-[:memberOf]->(d:`Department`)
  RETURN d.name
$$) AS (department_name text);
```

This hybrid workflow empowers teams to leverage graph modeling precisely where it delivers additional insight, such as in organizational hierarchies, fraud detection, social networks, and recommendation systems. At the same time, they can maintain their existing relational models, SQL-based ETL pipelines, and familiar operational practices. PostgreSQL's extensible architecture thus enables a gradual, low-risk adoption of graph analytics, allowing organizations to experiment, iterate, and expand graph use cases without disrupting their data ecosystem.

In practice, organizations must decide whether to run graph workloads in a dedicated graph database or within a relational system extended with graph capabilities. Dedicated graph databases such as Neo4j or TigerGraph typically provide stronger performance for deep relationship traversal and large graph analytics. Relational extensions such as PostgreSQL AGE or SQL Server Graph offer a pragmatic alternative, allowing teams to reuse existing infrastructure and SQL pipelines while selectively introducing

graph queries. The trade-off is therefore not purely technical but architectural: operational simplicity versus specialized graph performance.

What's important is not the syntax but the workflow: SQL prepares the data, and the graph navigates the relationships. You stay in the PostgreSQL ecosystem, reuse existing tables and pipelines, and introduce graph queries only when they add clarity or performance. For SQL users, this makes graph databases feel less like a radical departure and more like a natural extension of relational modeling when relationships become the primary question.

SQL and Graphs: Complementary, Not Competing

Imagine a company's nightly ETL job responsible for aggregating sales data from various sources. As the clock strikes midnight, SQL demonstrates its reliability and precision; it ingests, cleanses, and transforms batches of transactional data, ensuring consistency and accuracy for the morning reports. However, as the data increasingly emphasizes connections, such as customer interactions and product dependencies, SQL starts to show its limitations in handling these relationship-dense queries. This is where hybrid architectures leverage the power of graph models to efficiently traverse and analyze these complex networks.

SQL remains the backbone of most data platforms. It is the primary language for ingesting, cleansing, and transforming data; performing aggregations and generating reports; and building analytical models such as star schemas and OBTs. Even in modern, cloud native architectures, SQL remains the common ground for data processing and analysis.

Graphs are increasingly pivotal in achieving key organizational outcomes, such as enabling faster fraud detection and delivering richer recommendations. By centering relationships and connectivity, graph models excel at traversing complex networks, supporting relationship-centric analysis, accommodating flexible schemas, and enriching datasets

with semantic context. As data grows more interconnected, graphs are being used for organizational modeling, recommendation engines, and contextual reasoning.

Modern data platforms are increasingly adopting hybrid architectures that combine both paradigms. The SQL:2023 standard, for example, introduces property graph querying natively within SQL, enabling users to express graph patterns as virtual tables. In practice, many teams follow a pragmatic approach: they prepare and model data with SQL and tools like dbt and then project selected entities and relationships into a graph database for advanced analysis and exploration.

A modern data platform typically consists of several complementary layers. At its core is a relational warehouse, modeled using a blend of normalized schemas, star schemas, and denormalized tables, all built and managed with SQL. On top of this foundation, a knowledge graph captures key entities and relationships, structured by an explicit ontology that ensures semantic consistency. Analysts and data scientists select their querying approach—whether from the warehouse or the knowledge graph—based on the task at hand, often combining results from both to gain deeper insights. However, integrating these systems is not without challenges. Data lineage between the warehouse and the graph can become convoluted, and ontology drift poses a threat to maintaining semantic accuracy. Mitigation strategies, such as establishing robust data governance frameworks and continuously updating ontologies, can help manage these risks and build trust with seasoned BI leaders considering pilot implementations.

In the bustling operations room of DigiCom Analytics, a team of data engineers sits around a table, grappling with a pressing challenge. Their current 3NF system, while perfect for maintaining data integrity and consistency, struggles with executing complex analytical queries promptly. As they delve deeper into resolving slow joins and cumbersome queries, they realize the need for a more flexible approach. The breakthrough comes when they experiment with graph traversal, which not only accelerates query performance but also reveals unexpected connections in customer

data. This pivotal shift uncovers a hidden fraud ring, affirming the power of hybrid modeling architectures.

Summary

This chapter explores how data modeling with SQL has evolved from classic relational principles to modern, hybrid architectures that incorporate semantic models and knowledge graphs. Rather than presenting these approaches as competing alternatives, we framed them as complementary techniques, each optimized for different types of problems.

We began with traditional SQL modeling techniques. 3NF remains the foundation for transactional systems, prioritizing data integrity, consistency, and correctness. While highly normalized schemas excel at write-heavy workloads, they introduce complexity for analytical queries. Dimensional modeling emerged as a pragmatic compromise: star schemas organize data into facts and dimensions, enabling fast, intuitive queries for business intelligence. Snowflake schemas partially normalize dimensions, trading simplicity for maintainability in more complex hierarchies. At the extreme, the OBT approach consolidates analytical data into a single wide table, optimizing for simple queries and performance at the expense of redundancy and flexibility.

The chapter then explored semantic modeling and knowledge graphs. Knowledge graphs shift the focus from rows and joins to entities and relationships, treating connections as first-class data. Graph databases and query languages like Cypher enable expressive, traversal-based queries that are difficult in SQL, especially for relationship-heavy or multi-hop questions. Semantic models, supported by standards such as RDF and OWL, formally define concepts, relationships, and inference rules, adding an extra layer of meaning.

Crucially, the chapter emphasized that SQL is not displaced by these innovations. SQL remains the backbone of modern data platforms, powering ingestion, cleansing, transformation, aggregation, and analytical modeling. Graphs and semantic layers extend SQL's reach, enabling new

questions when relationships and context are central. Hybrid approaches, such as PostgreSQL's graph extensions and the SQL:2023 property graph standard, show how relational and graph paradigms are converging.

The key lesson is not to search for a single "best" data model, but to choose the right abstraction for each problem. Normalized models ensure correctness, dimensional models optimize analytics, wide tables accelerate consumption, and graphs unlock relational insight. Mature data platforms combine these techniques, guided by clear semantics and strong governance.

Key takeaways for this chapter include:

- 3NF models are ideal for systems of record and transactional workloads where consistency and correctness are critical.
- Star schemas simplify analytical querying and remain the dominant pattern for data warehouses and BI.
- Snowflake schemas introduce additional normalization in dimensions and should be used selectively.
- OBTs maximize query simplicity and performance but require careful governance and change management.
- Knowledge graphs excel at relationship-centric and multi-hop queries that are difficult to express in SQL.
- Semantic models make data meaning explicit, enabling better integration, reasoning, and reuse.
- SQL remains foundational, even in graph- and AI-driven architectures, serving as both the preparation and execution layer.
- Hybrid architectures enable SQL and graph data to coexist, allowing teams to adopt new paradigms incrementally and pragmatically.

Understanding these models and their trade-offs equips data engineers and scientists to design robust, adaptable systems. As data ecosystems grow in complexity, the ability to combine SQL-based, semantic, and graph-based approaches will become increasingly valuable. To apply these concepts, sketch a mix of models for an upcoming project. Reflect on the problems you encounter and consider how integrating these approaches could enhance your solutions. By doing so, you'll solidify your understanding and prepare for innovative designs that meet future challenges.

Chapter 5. Building Data Engineering Solutions with SQL

Building a data engineering solution entirely with SQL as the primary language may sound ambitious, but modern platforms and tools make it close to achievable. In this chapter, we design and implement an end-to-end data pipeline where SQL is the primary language for defining data structures, transformations, and analytical semantics, while the surrounding platform components remain SQL-friendly. The goal is to show that SQL, coupled with the right ecosystem, can anchor everything from real-time data ingestion to building a data lakehouse, while following best practices for scalability and maintainability.

Some parts of the system, specifically streaming ingestion engines (such as Spark Structured Streaming), orchestration services, and storage maintenance processes like compaction and snapshot management, involve system-level concerns beyond pure SQL syntax. Some of these components are used in practical form throughout the chapter, but the focus remains on how they integrate with SQL-based modeling layers rather than on the low-level operational engineering required to run them at scale. The goal is not to turn the reader into a platform engineer but to show how SQL-driven layers integrate into a broader data platform where surrounding systems are designed to remain SQL-compatible rather than SQL-exclusive.

Our scenario is a simple retail example with three core tables: `Customers`, `Products`, and `Orders`. Customers purchase products, generating orders with quantities of those products. This small schema represents a typical online store or point-of-sale system. The pipeline follows a Lambda-style architecture: streaming ingestion for near-real-time analytics and a batch ingestion component for comprehensive historical data analysis. The entire solution will be built using SQL-centric tools, highlighting how SQL can be used at every step of a data engineering project.

> **NOTE**
>
> All code developed in this chapter is available in the book's GitHub repository. Please refer to the *chapter_05* folder, which contains dedicated subdirectories for producers, ingestion, and transformation jobs. Each one includes a comprehensive README with detailed setup and execution instructions. Following these guides alongside this chapter will give you a hands-on experience in building a data engineering pipeline from scratch.

Pipeline Overview and Design Principles

Our pipeline will simulate an ecommerce data platform. The source operational system is normalized (with separate tables for customers, products, orders: a typical OLTP schema). We need to ingest data from this source in two ways: a continuous *stream of orders* for real-time analysis, and periodic *batch extracts* of all data for full accuracy and backfills. Downstream, we'll store the data in a *data lakehouse* using an open table format (Apache Iceberg) on cloud object storage, and then perform transformations to create analytics-friendly models (such as aggregated sales, customer lifetime value, etc.). Finally, we'll apply best practices to ensure the pipeline is scalable, testable, and maintainable.

Figure 5-1 illustrates the overall architecture of the pipeline, combining a real-time stream and a batch process into a unified SQL-based data platform (Lambda architecture).

In the speed layer, order events flow from the source system into a Pub/Sub topic in real time. In the batch layer, the source periodically exposes full extracts of each table via REST APIs. Both feeds land data into the Bronze layer (raw storage) backed by Iceberg. Downstream, a series of SQL transformations in dbt produce cleaned Silver layer views and aggregated Gold layer tables, ready to be consumed, whether directly or through the consume layer where the *semantic layer* resides. As introduced in Chapter 4, the semantic layer centralizes business definitions, metrics, and governance rules, ensuring consistent interpretation across analytical consumers.

Why Lambda architecture? While unified streaming architectures (often called Kappa) are increasingly adopted, many organizations still operate hybrid environments where batch ingestion remains practical, cost-efficient, and operationally simpler. This chapter presents both to reflect real-world diversity rather than prescribe a single model.

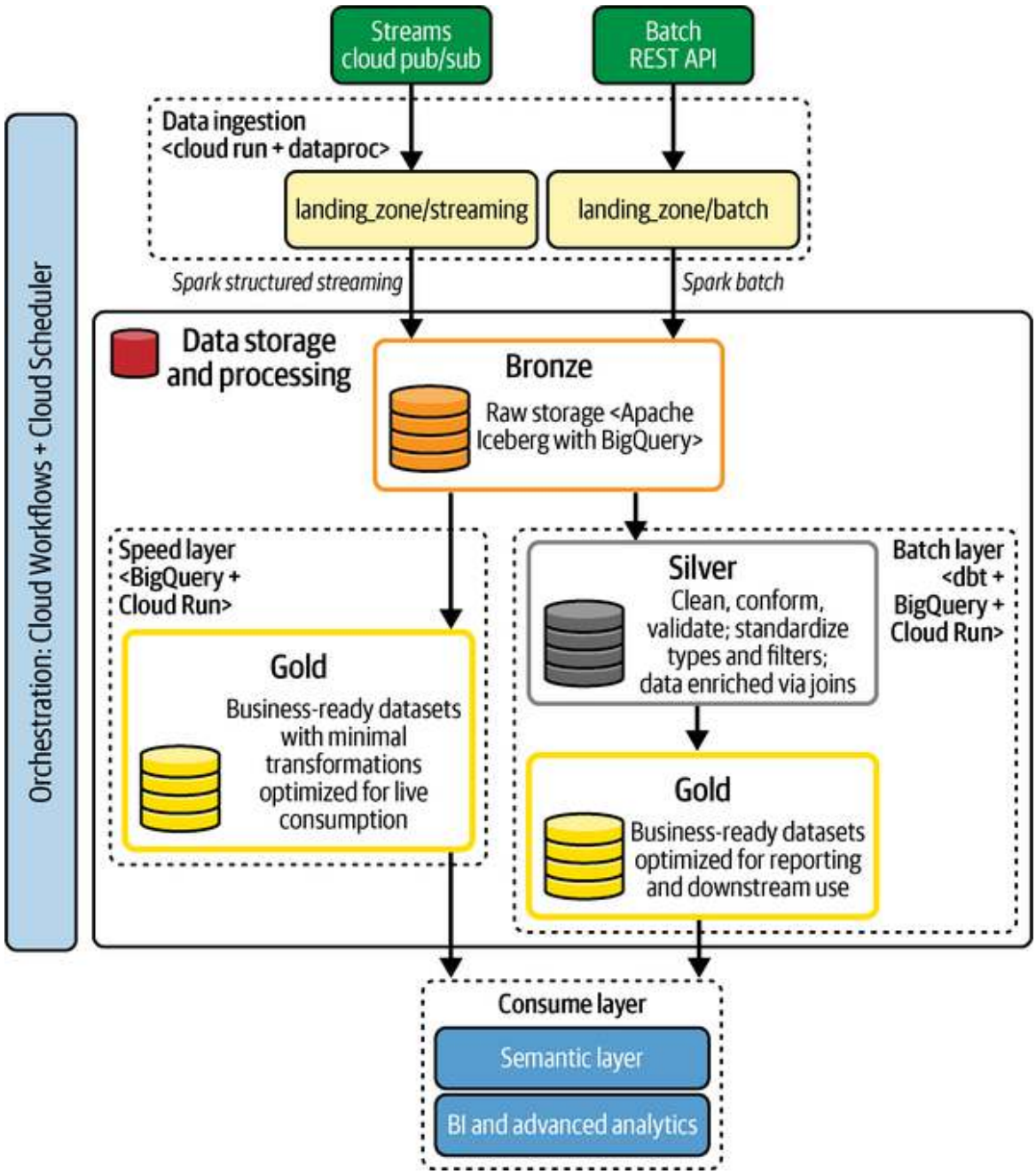


Figure 5-1. Pipeline architecture

To design this architecture, several design principles were considered:

SQL first

Use SQL or SQL-like for several layers of our data platform, whenever possible. All business logic (cleaning, joins, aggregations) is done in SQL (via dbt models or SQL-based streaming), rather than in application code. This ensures clarity, declarative logic, and the ability to leverage query optimizations. When we cannot use pure SQL (for example, reading from Pub/Sub), we still aim to use SQL-friendly engines (Spark) and keep any non-SQL code minimal.

Lambda architecture

We combine streaming and batch pipelines to get the best of both. The streaming pipeline provides low-latency updates for fresh data, while the batch pipeline ensures completeness and accuracy by reconciling any missing or late data. This was introduced in "Batch Versus Streaming: Choosing the Right SQL Pipeline Approach", and here we implement it in practice. The design ensures that both pipelines ultimately feed into the same storage and schema, so they can be unified in downstream analysis.

Unified lakehouse with Apache Iceberg

We follow a modern lakehouse paradigm where data is stored in a data lake (cloud object storage) but managed with data warehouse-like features (schemas, SQL access, ACID transactions) using Apache Iceberg. The raw data lands as files in a *landing zone* and then is structured into the Bronze layer as Iceberg tables on cloud storage. This gives us the best of both worlds: low-cost storage and openness of a data lake, with the reliability and schema management of a warehouse. Iceberg also enables *time travel* (querying older snapshots of the data) and *schema evolution*, which are invaluable for data governance.

Modularity and loose coupling

Each component of the pipeline is decoupled and modular. The data producers (APIs and Pub/Sub) are independent of the ingestion logic.

Ingestion (landing data to Bronze) is separated from transformation (Silver/Gold models), and orchestration ties them together. This makes the system easier to maintain and test. For example, we can run the batch ingestion job independently or refresh a particular dbt model in isolation. Each layer has clear contracts: Bronze is raw immutable data, Silver is standardized clean data, Gold is business aggregates.

Low maintenance and managed services

We favor managed services to reduce operational burden. GCP's Pub/Sub, Cloud Run, Dataproc Serverless, Cloud Workflows, BigQuery, and BigLake are used so that we don't manage servers or complex infrastructure. Infrastructure-as-Code (`gcloud` commands) and configuration are used to set up these resources, aligning with best practices (as discussed in "Building and Operationalizing Scalable SQL Data Platforms").

Cost efficiency

The entire pipeline is designed to run within free tier limits or minimal cost. For example, we use Pub/Sub (which has a free tier) instead of running a Kafka cluster, and we use serverless Spark, which can auto-terminate to avoid idle costs. We also use BigQuery's external table capability (BigLake) to query data in Cloud Storage without needing to pay for storage twice or ingest data into BigQuery. Wherever possible, we use small-scale data and short-lived jobs to control costs. In a production setting, we would be able to scale up resources and ensure proper reservations, but the architecture would remain the same.

Next, we describe each part of the pipeline, from data ingestion (streaming and batch) to the storage layer, transformations, orchestration, and finally consumption and governance. Before diving into the implementation, a few initial setup steps are needed on Google Cloud Platform (GCP). You can follow along in a cloud shell or the console.

Setting Up the GCP Environment

Setting up Google Cloud is relatively simple, yet you need a number of configuration considerations to ensure a correct setup. All of the following steps can be done within GCP's free tier (though you may need to enable billing on the project for certain operations, as noted below). We assume you have a GCP account; if not, you can sign up for a free trial on Google Cloud. The following setup uses the *Google Cloud Shell* (a browser-based terminal in the GCP console) for convenience, but you can also use the gcloud CLI locally if you prefer.

Create a GCP project

We create a new GCP project named Advanced SQL DE Demo and with the Project ID `advance-sql-de-demo` to isolate our resources. Using the `gcloud` command line interface (CLI), you can create a project and set it as the default for subsequent commands, as shown in Example 5-1.

Example 5-1. Creating and selecting a new Google Cloud project for our demo environment

```
gcloud projects create advance-sql-de-demo --name="Advanced SQL DE Demo"
gcloud config set project advance-sql-de-demo
```

Enable required APIs

Our pipeline will use several GCP services (Cloud Storage, BigQuery, Pub/Sub, Dataproc, etc.), which require their APIs to be enabled in the project. Example 5-2 shows the `gcloud` command to enable all necessary services in one go.

Example 5-2. Enabling GCP APIs for Storage, BigQuery, BigLake, Dataproc, Pub/Sub, and Cloud Run

```
# Enable necessary GCP services
gcloud services enable \
    storage.googleapis.com \
    bigquery.googleapis.com \
    bigqueryconnection.googleapis.com \
    biglake.googleapis.com \
    dataproc.googleapis.com \
    pubsub.googleapis.com \
```

```
run.googleapis.com \
workflows.googleapis.com \
cloudscheduler.googleapis.com
```

These APIs cover Cloud Storage, BigQuery, and BigQuery Connection/BigLake (for the Iceberg integration), Dataproc (to run Spark jobs), Pub/Sub (for streaming ingest), Cloud Run (to host data producer services), and Cloud Workflows and Scheduler (for orchestration). Enabling them ensures that we have access to all the managed services needed for the pipeline.

Create a cloud storage bucket

Next, create a GCS bucket to hold the lakehouse files (e.g., Parquet files and Iceberg metadata). Use a regional bucket (for example, `europe-west1`). Example 5-3 shows how to create this bucket.

Example 5-3. Creating a GCS bucket for the data lakehouse files

```
gsutil mb -p advance-sql-de-demo
          -c STANDARD
          -l europe-west1
          gs://advance-sql-de-bucket/
```

This bucket will serve as the data lakehouse storage. If this command fails with a 403 error about billing (*billing account for the owning project is disabled*), it means you need to attach a billing account to your project, even if we plan to stay within the free tier limits or minimal cost. Go to the Google billing page and link a billing account to the project, then retry the bucket creation command (you must be signed into your Google account to access the billing page).

Set up BigLake (Iceberg) metastore

Google Cloud's BigLake service will serve as the metastore for our Iceberg tables. This will allow BigQuery and other engines to discover and query the data in the bucket. Note that, at the time of writing, the BigLake Iceberg integration is new and the `gcloud` CLI support for it is in alpha, which might be unstable. Set up the catalog via the BigLake Metastore page for

the project `advance-sql-de-demo` by clicking Create Catalogue and filling in the following details:

Cloud Storage Bucket

`advance-sql-de-bucket`

Authentication

End-user credentials (which is sufficient for a demo environment)

After creation, you have a BigLake Iceberg catalog associated with the bucket (serverless and managed by GCP). This catalog is essentially our single source of truth for table metadata, accessible by BigQuery and also by Spark or Flink through the Iceberg API (via a REST endpoint).

Create Pub/Sub topic and subscription

Prepare the streaming input by creating a Pub/Sub topic called `orders` and a subscription for the streaming pipeline. Example 5-4 shows these commands.

Example 5-4. Creating a Pub/Sub topic for orders and a subscription for the pipeline

```
gcloud pubsub topics create orders

# Create a subscription that our data pipeline will use to pull messages
gcloud pubsub subscriptions create orders-to-gcs-sub \
    --topic=orders
```

The subscription ensures that our ingestion (streaming job) can receive messages, even if it's not running at the exact moment messages are published (Pub/Sub will retain them until acknowledged). The name `orders-to-gcs-sub` hints that initially we might land these messages to Cloud Storage as an intermediary step.

> **NOTE**
>
> While this section configures most of the GCP environment, including Pub/Sub topics and subscriptions, this step is optional at this stage. It is documented here for completeness, but it is also covered in *chapter_05/producers/README.md*, where it appears as a required step for publishing data to Pub/Sub and, consequently, for correctly setting up the environment.

Create a BigQuery dataset and BigLake connection

In BigQuery, we need to establish a BigQuery connection to access the BigLake (Iceberg) catalog. This connection allows BigQuery to query the data in our Cloud Storage bucket. In the BigQuery page, go to your project, Connections > Create connection and create a new connection with type `Vertex AI remote models, remote functions, BigLake and Spanner (Cloud Resource)`. Give it a name (e.g., `adv_de_sql_biglake_conn`).

The BigQuery ←→ BigLake connection uses a service account under the hood, found inside the Connection info > Service account id. We must grant that service account access to the Cloud Storage bucket; otherwise BigQuery cannot read the files. In your Cloud Storage Bucket, go to the bucket's permissions and add the connection's service account with Storage Object Admin or similar read/write role (for simplicity in a demo, we give broad access). This step ensures that BigQuery queries (on Iceberg tables) won't be blocked by permissions.

At this point, our cloud infrastructure is ready: we have a bucket for the data lakehouse, a BigLake catalog for Iceberg, a Pub/Sub stream, and BigQuery set up to interface with the data. The last thing we need is to ensure our local or cluster environment (where we'll run the ingestion jobs) can authenticate to GCP. If you are using Cloud Shell, you're already authenticated as your user. If you run code from elsewhere (like a local machine or a Dataproc cluster), you should ensure credentials are available. One simple approach is to use Application Default Credentials via the Cloud SDK: for example, running `gcloud auth application-default`

`login` will get you a local credential that Spark or other SDKs can pick up. Alternatively, you could create a dedicated service account key and run `gcloud auth activate-service-account --key-file=REPLACE-BY-YOUR-SA-KEY.json`. Either way, make sure the environment where your SQL jobs run can access GCP resources.

With the Google Cloud environment configured, we are ready to start building the pipeline itself.

> **TIP**
>
> Hands-On Checkpoint: If you would like a more hands-on technical experience, this is a good point to implement the producers from the book's GitHub repository in the *chapter_05/producers* directory and follow the *README.md*, which provides the setup instructions. This service simulates our source APIs and publishes events, required for our ingestion part.

Data Ingestion in a SQL-Centric Architecture

Data ingestion is the first stage of our pipeline. In this section, we demonstrate two ingestion patterns: *streaming ingestion* for real-time orders and *batch ingestion* for periodic snapshots of core entities (`customers`, `products`, and `orders`). Traditionally, streaming ingestion might be implemented with low-level Java or Scala consumers, while batch ingestion often relies on custom scripts or ETL tools.

In our approach, ingestion is implemented using technologies that operate on relational abstractions and integrate naturally with SQL-based systems. Modern stream processing frameworks such as Apache Spark Structured Streaming (and Apache Flink SQL) allow data to be processed as tables or DataFrames, which map directly to SQL concepts. This keeps ingestion aligned with the broader SQL-centric architecture, even when the runtime itself is not a pure SQL engine.

For batch ingestion, if our source was a database, we could even use SQL directly to pull data (using connectors). In our case the source is a REST

API (simulated), so a small script is used to retrieve the data. Once the data is landed in the platform, however, it enters the same SQL-compatible environment as the streaming data, ultimately being stored in Iceberg tables and consumed downstream through SQL engines.

Let's break down the ingestion implementation.

Streaming Ingestion with Spark Structured Streaming and a Flavor of Apache Flink SQL

In our scenario, the streaming ingestion (the pipeline's speed layer) begins as orders are placed in the ecommerce system, and we want them to appear in our analytics pipeline within minutes. A producers application simulates this by publishing each new order as an event to a Pub/Sub topic. Our streaming ingestion layer captures those events, lands them durably in cloud storage, and converts them into structured Iceberg tables in the Bronze layer.

We implement streaming ingestion as a two-step pipeline for modularity, reliability, and replayability:

1. Pub/Sub to landing zone (Cloud Run Job)

A small stateless service will subscribe to the Pub/Sub topic and write incoming messages as JSON files into a GCS bucket (the landing zone). This ensures we have a durable, file-based log of all events exactly as they were produced. It also decouples the rest of the pipeline from Pub/Sub since downstream jobs just read files from storage, which is often simpler and more reliable for replay or audit (file: *ingestion/streaming_ingestion/main.py*, Cloud Run Job: `ch05-streaming-ingestion-job`).

2. Landing zone to Iceberg (Spark Structured Streaming)

A Spark streaming job will continuously monitor the landing zone folder for new files, parse them, and append them to an Iceberg table into the Bronze layer. This job runs on Dataproc Serverless, meaning we

don’t manage any cluster. It is Google’s responsibility to spin up resources as needed for the job (file: *ingestion/spark/jobs/streaming/01_lz_orders_to_bronze_layer.py*).

This design effectively follows a *Lambda* style ingestion pattern: raw events are captured immediately and durably, while Spark handles near-real-time micro-batch processing into a structured analytical store. Decoupling Pub/Sub from downstream processing via cloud storage simplifies recovery, replay, and auditing.

The next sections explain the streaming ingestion flow in detail.

Pub/Sub to GCS (landing zone)

The Cloud Run job is a Python program that uses the Pub/Sub subscriber client library. It polls the subscription for messages in small batches and writes any received records to Google Cloud Storage. We store these records using newline-delimited JSON (NDJSON), so each Pub/Sub message becomes a single JSON object per line. This format is easy for downstream systems such as Spark to process and remains readable for manual inspection when troubleshooting is required.

Example 5-5 shows an excerpt of the code responsible for writing a batch of Pub/Sub messages to GCS as an NDJSON file. Messages are accumulated in memory and uploaded in a single write operation, rather than one file per message, which helps avoid small files and reduces storage overhead.

Example 5-5. Excerpt of the code that writes a batch of Pub/Sub messages to GCS as an NDJSON file

```
# JSON records as newline-delimited JSON (one JSON object per line) and upload to GCS
payload = "\n".join(
    json.dumps(r, separators=(",", ":"), ensure_ascii=False)
    for r in records
) + "\n"
blob.upload_from_string(payload, content_type="application/x-ndjson")
```

Each file is written to a path that encodes the current date and hour in UTC, which later allows us to partition tables efficiently by time. A sample path

structure is shown in Example 5-6.

Example 5-6. Sample path scheme of a streaming file written to landing zone

```
gs://<bucket>/landing_zone/streaming/orders/
  ingest_dt=2026-04-25/
  ingest_hr=11/
  part-<uuid>.jsonl
```

The Cloud Run job determines the folder path dynamically, based on the current timestamp each time it writes a batch of messages. Before persisting the data, the job enriches every record with operational metadata that supports debugging and traceability, including the Pub/Sub message ID, the publish timestamp, and the ingestion timestamp.

If a message payload is not valid JSON, the raw payload is preserved in a `_corrupt_record` field together with an `_ingest_error` flag. This ensures that no data is silently dropped. Messages are acknowledged only after the file has been successfully written to GCS, providing an *at-least-once* delivery guarantee, which becomes effectively *exactly-once* when combined with downstream checkpointing.

Although Pub/Sub offers a native subscription-to-Cloud-Storage feature that could replace this step in a production environment, we implement the landing logic manually here to make enrichment, validation, and error handling explicit. Persisting raw events in object storage also creates a durable buffer that decouples ingestion from downstream processing. If processing lags or needs to restart, the data remains safely on disk and can be replayed deterministically.

GCS to Iceberg and Spark structured streaming (Bronze layer)

The next step of the ingestion part of our pipeline is to leverage Spark Structured Streaming deployed via Dataproc Serverless to capture the data inside the landing zone and append it to the `bronze.live_orders` Iceberg table. This stage is entirely managed by Spark's SQL engine and operates on a micro-batch model. Every few seconds, Spark checks for new files,

processes them in a batch, performs validation and basic cleanup, and writes valid data to the Iceberg table.

The Spark job is submitted on-demand using Dataproc Serverless, so we don't need to maintain any cluster or infrastructure. The core of this ingestion logic can be found in the job at *ingestion/spark/jobs/streaming/01_lz_orders_to_bronze_layer.py*. The code excerpt in Example 5-7 reflects this core process.

Example 5-7. Spark Structured Streaming job (landing zone → Bronze) for live orders ingestion

```
df = (
    spark.readStream
    .schema(landing_zone_schema())
    .option("maxFilesPerTrigger", max_files)
    .json(input_path)
)

query = (
    df.writeStream
    .queryName("ch05_orders_to_iceberg")
    .foreachBatch(write_batch_factory(
        quarantine_path=quarantine_path,
        catalog=args.CATALOG_NAME,
        db=args.DB_NAME,
        table=args.TABLE_NAME,
    ))
    .option("checkpointLocation", checkpoint_path)
    .trigger(processingTime=f"{trigger_seconds} seconds")
    .start()
)

    #TTL Logic: Wait for a specific amount of time or indefinitely
    if args.ttl:

        ...

        finished_cleanly = query.awaitTermination(timeout_seconds)

        ...

    else:
        ...

        query.awaitTermination()
```

In the code above, `spark.readStream` is used to continuously read JSON files from the landing zone (`gs://<bucket>/landing_zone/streaming/orders/`). We provide an explicit schema for the orders (defined in `landing_zone_schema()` to match the JSON fields) and set `maxFilesPerTrigger=50` to control how many files are processed in each micro-batch.

Instead of using a simple file sink, we use `df.writeStream.foreachBatch(...)` with a custom batch writer. This pattern gives us fine-grained control over each batch: we validate records, enrich them with metadata, and separate invalid data into a quarantine path. Only validated rows are appended to Iceberg. In Structured Streaming, the `foreachBatch` function receives each micro-batch as a static DataFrame, which allows us to use Spark's Data Source V2 batch write API (`writeTo(...).append()`) safely within a streaming job. This bridges streaming ingestion with Iceberg's transactional batch write semantics.

The `query.awaitTermination()` call keeps the Spark job running indefinitely, processing new files as they arrive. In production, this job (or an equivalent long-running stream) typically runs continuously. In the demo, we run it for a bounded period (for example, a Dataproc Serverless batch with a time-to-live like `--ttl="10m"`) to simulate continuous ingestion without needing to manually cancel the job.

The Spark job also performs the following steps behind the scenes:

Schema and parsing

> We provide Spark with a schema (`landing_zone_schema()`) for the JSON (this schema matches the fields we expect in the order JSON, including those metadata fields added earlier). Spark then reads the JSON files into a DataFrame using this schema. By specifying the schema, we avoid expensive schema inference on each batch and ensure consistency.

Table creation and partitioning

We ensure the target Iceberg table exists with the desired schema. The job issues a Spark SQL command to create the table if it doesn't exist, specifying all columns and data types. We partition the Iceberg table by `ingest_date` (date of ingestion). Partitioning by date is a simple strategy that works well for time-series data and preserves query performance (queries can skip partitions easily). The table creation looks roughly like Example 5-8.

Example 5-8. Iceberg table creation example

```
CREATE TABLE IF NOT EXISTS local.bronze.live_orders (

    id BIGINT,

    customer_id BIGINT,

    product_id BIGINT,

    quantity INT,

    total_amount DOUBLE,

    unit_price DOUBLE,

    currency STRING,

    status STRING,

    source STRING,

    created_at TIMESTAMP,

    pubsub_message_id STRING,

    pubsub_publish_time TIMESTAMP,

    ingest_ts TIMESTAMP,

    ingest_date DATE,

    ingest_dt STRING,

    ingest_hr STRING,

    input_file STRING

)
```

```
USING iceberg

PARTITIONED BY (ingest_date);
```

All columns correspond to either the original order fields or metadata we added. Some fields like `unit_price` (which is `total_amount/quantity`) and `status`/`source` are set later in transformation, but we include them for completeness in the Bronze table structure, even if their values might be null or placeholder at ingestion.

Minimal data transformations (bronze cleanup)

We perform minimal transformations in this stage—just enough to prepare the data for reliable storage. Specifically, our Spark job will:

- Add `input_file` column (source filepath).
- Parse and normalize timestamps (`created_at`, `ingest_ts`, `pubsub_publish_time`).
- Derive partitioning columns (`ingest_date`, `ingest_dt`, `ingest_hr`).
- Fill missing fields: if `status` or `source` are missing, set to *unknown* and *pubsub* respectively.
- Calculate `unit_price = total_amount / quantity`.
- Rename Pub/Sub specific fields for clarity (e.g., `_pubsub_message_id` to `pubsub_message_id`).

  All of this is done with Spark DataFrame operations (which are essentially SQL transformations under the hood).

Validation and quarantine

We implement a basic data validation to separate obviously bad records. For example, a valid order should have non-null IDs, positive quantity, a currency code, etc. Records failing these conditions are considered invalid. In the Spark streaming job, we split the incoming batch into valid and invalid DataFrames based on these rules. If there are any invalid rows (perhaps due to upstream bugs or unexpected data), we do not write them to the Iceberg table. Instead, we write them out as JSON to a quarantine path (`gs://<bucket>/quarantine/streaming/live_orders/...`) for later inspection. This quarantine is an append-only log of problematic data, stored in GCS (partitioned by date as well). It's good practice to quarantine since doing so prevents bad data from polluting your main tables while ensuring data is not lost.

Append to Iceberg

The valid records are appended to the `bronze.live_orders` table using Spark's Iceberg integration (Example 5-9).

Example 5-9. Append data to Iceberg table: Python code

```
df_valid.writeTo("local.bronze.live_orders").append()
```

Each micro-batch becomes a new Iceberg snapshot, offering atomicity and enabling rollback or time travel if needed.

We configure the streaming job with a trigger interval (e.g., 15 seconds) and a checkpoint location in GCS for state (so Spark can remember what files it has processed and ensure exactly-once processing of files). In practice, this job could be left running for long periods. In our case, we might run it only during demonstrations and then cancel it—but in a live system, you'd keep it running or use multiple structured streaming jobs if needed for scalability.

After this, our streaming orders end up in the `bronze.live_orders` Iceberg table (accessible also via BigQuery external table as

`bronze.live_orders`). New orders flow in continuously. We have achieved a near-real-time ingestion using mostly SQL engines: the only custom code was a small Python job to interface Pub/Sub with GCS, and the rest was handled by Spark's SQL processing.

When new Iceberg snapshots are written, refresh the BigQuery external tables so they pick up the latest Iceberg metadata. In our repo, we include a helper Cloud Run Job (`ch05-refresh-bq-iceberg-tables`) that uses the BigQuery API to refresh or re-create the external tables for each Iceberg table. See additional information in "Iceberg schema evolution". Ensure that the BigQuery `bronze` dataset has external tables for `customers`, `products`, `orders`, and `live_orders` and that they point to the Iceberg catalog.

> **WARNING**
>
> In this streaming setup, ensure the Producers API and Pub/Sub topic are up and running before starting the ingestion. If the producers aren't running, the streaming job will have no data to consume. In our example, the Cloud Run job and Spark streaming job will simply wait if no messages arrive. Be cautious about long-running jobs and cost: a continuously running Spark stream may consume resources even if data is trickling. Design triggers and timeouts appropriately (for example, our demo limits the streaming job to run a for a certain time defined on the `ttl` arg).

Why Spark in a SQL-centric architecture?

Although Spark Structured Streaming is not a pure SQL engine, it plays a pragmatic and important role in a SQL-centric data architecture. Spark is one of the most widely adopted distributed processing engines in the industry, and its Structured APIs are fundamentally built around relational abstractions. DataFrames and Datasets represent logical tables, transformations compile into query plans, and Spark SQL provides a fully capable SQL interface on top of the same execution engine. In practice, many DataFrame operations are simply a programmatic way of expressing relational transformations that could also be written in SQL.

In our pipeline, Spark is not used as a business logic engine. Its role is deliberately constrained to data movement and ingestion orchestration, not semantic transformation. Spark handles concerns such as streaming reads, file layout, partitioning, and reliable writes into Iceberg tables. The transformation and modeling logic, where business meaning is defined, remains fully SQL-driven in BigQuery and dbt.

Another practical reason for selecting Spark in this architecture is its mature and native integration with Apache Iceberg. Spark was one of the earliest engines to support Iceberg's table format and catalog abstractions, including *snapshot isolation*, *schema evolution*, *partition evolution*, and transactional writes. This makes Spark a reliable engine for producing data into Iceberg tables without requiring proprietary connectors or format-specific workarounds.

Apache Flink SQL

As noted previously, we chose Spark Structured Streaming for our implementation due to its easy integration with Iceberg and ability to reuse Spark SQL skills. Yet, it's worth noting an alternative approach: using Apache Flink with SQL for the streaming ingestion. Flink is another distributed stream processing engine that has excellent SQL support. With Flink SQL, you can declare *source tables* (e.g., a Pub/Sub subscription or Kafka topic) and *sink tables* (e.g., an Iceberg table), and then issue SQL `INSERT` statements that continuously apply. For example, in Flink we could handle reading from the stream and writing to Iceberg continuously, as shown in Example 5-10.

Example 5-10. Flink SQL statement

```
CREATE TABLE OrdersStream (
  id BIGINT,
  customer_id BIGINT,
  product_id BIGINT,
  quantity INT,
  total_amount DOUBLE,
  currency STRING,
  created_at TIMESTAMP(3),
  WATERMARK FOR created_at AS created_at
) WITH (... Pub/Sub connector properties ...);
```

```
CREATE TABLE LiveOrdersIceberg (
  id BIGINT, customer_id BIGINT, ..., ingest_date DATE
) WITH (... Iceberg connector properties ...);

INSERT INTO LiveOrdersIceberg
SELECT *, CAST(CURRENT_TIMESTAMP AS DATE) as ingest_date
FROM OrdersStream;
```

We won't implement Flink in our pipeline, but it's a powerful concept: streaming ETL can be done with pure SQL in engines like Flink. GCP's equivalent would be using Dataflow with SQL or Beam SQL, though at the time of writing, Beam's SQL is not as straightforward as Flink's. Our Spark approach is Python code-heavy, but you could imagine doing it with Spark SQL queries as well. The key is that we stay within the realm of declarative processing as much as possible, and you have choices of stream processors.

Batch Ingestion for Bulk Data

While streaming ingestion covers the near-real-time flow of new orders, many systems also require batch ingestion to guarantee completeness and consistency. In our pipeline, we ingest *daily full snapshots* of `customers`, `products`, and `orders` from the source system. This batch layer acts as a safety net: if any events are missed in streaming, arrive late, or if a source does not emit fine-grained change events, the batch pipeline ensures the Bronze layer reflects the correct state of the world.

The source system exposes REST API endpoints that return the full dataset for each entity. From this, the batch ingestion process follows a simple, robust sequence:

1. API to landing zone (Cloud Run job)

> We have three separate Cloud Run jobs, one for each entity: `customers`, `products`, `orders`. Each job, when executed, calls the producers API (e.g., the `<API>/customers` endpoint) to fetch the entire dataset, and then writes the results as JSON to the landing zone on GCS. We include the current date and a timestamp in the filepath (so each *snapshot* is

isolated) (file: *ingestion/batch_ingestion/main.py*, Cloud Run Job: `ch05-batch-ingestion-<entity>`).

2. Landing zone to Iceberg (Spark batch jobs writing to Iceberg)

After the JSON snapshots are in GCS, we run a Spark job, executed inside Dataproc Serverless, to load each into the corresponding Iceberg Bronze table (`customers`, `products`, `orders`). This involves parsing the JSON and doing an upsert or partitioned overwrite in Iceberg (file: *ingestion/spark/jobs/batch/01_lz_batch_to_bronze_layer.py*).

This separation maintains consistency with what we did on streaming ingestion and ensures the extraction and the initial light transformations are loosely coupled, aligning with a Lambda-style architecture, also allowing the batch pipeline to periodically reconcile the system's full state.

The next sections describe the batch ingestion flow in detail.

API to GCS (landing zone)

As introduced earlier, to pull data from the API we create one Cloud Run Job per entity, namely `customers`, `products`, and `orders`. Each job is a small Python script that executes a well-defined sequence of steps to retrieve data and persist it into the landing zone.

At runtime, the job reads environment variables such as `API_BASE_URL`, which identifies the producers service, `ENTITY`, which specifies the dataset to fetch, and `BUCKET`/`GCS_PREFIX`, which defines where the output artifacts are written. It then issues an HTTP `GET` request to the corresponding API endpoint. For example, a request to `https://<API>/customers` returns a JSON array containing the full set of customer records.

The response payload is processed using a helper function such as `fetch_list(...)`, after which each record is enriched with ingestion metadata. This includes the ingestion timestamp (`_ingest_time_utc`), an ingestion type flag (`_ingest_type = "batch"`) to distinguish batch data

from streaming), and the entity name (`_entity`). These fields allow downstream processes to identify the origin and context of each record unambiguously.

The enriched records are then written to GCS as newline-delimited JSON using `write_ndjson(...)`. Each batch is stored in a snapshot-isolated folder structure, as illustrated in Example 5-11.

Example 5-11. Sample path scheme of a batch file written to landing zone

```
gs://<bucket>/landing_zone/batch/<entity>/
  snapshot_dt=2026-01-07/
  snapshot_ts=20260107T000000Z/
  <entity>.ndjson
```

The use of both `snapshot_dt` and `snapshot_ts` in the folder hierarchy ensures that every batch ingestion is isolated in its own directory. This makes each snapshot explicit and prevents data from different runs from being accidentally mixed. The job writes NDJSON files in the same way as the streaming ingestion, typically producing a single file per execution, although larger payloads could be split if needed.

This design assumes that the producers API exposes *consistent snapshots* for each entity. For example, the orders batch endpoint returns all orders up to the current point in time, effectively behaving like a full dump of the operational table. Under this assumption, each batch ingestion reflects the latest known state of the source system. In more advanced scenarios, the API could expose incremental flags or support change data capture, but for the purposes of this architecture, a full snapshot provides a simple, reliable solution.

Using Cloud Run jobs for batch ingestion keeps the process on-demand and serverless. Jobs can be triggered manually or orchestrated through schedulers such as Workflows or Airflow. Because each entity is handled by an independent job, batch ingestions can run in parallel or in a controlled sequence, depending on operational requirements.

Once these jobs complete, snapshot data is available in the landing zone under paths such as:

- *landing_zone/batch/customers/...*
- *landing_zone/batch/products/...*
- *landing_zone/batch/orders/...*

GCS to Iceberg with Spark batch processing (Bronze layer)

After the snapshot files land in GCS, we run a Spark batch job (separate from the streaming job) to load them into Iceberg Bronze tables. This job is submitted to Dataproc Serverless and is parameterized by entity type. We run it once per entity, typically after the corresponding Cloud Run job completes.

For each entity, the Spark batch job performs a set of steps. The first ensures the Spark reads the JSON files from the landing zone path for the given entity. Spark's reader can handle the NDJSON. We provide a schema matching the expected fields for that entity (for example, customers have id, name, email, etc., plus the `_ingest_time_utc`, `_ingest_type`, `_entity` fields). All snapshots should conform to that schema.

Continuing in the same step the function also extracts the snapshot partition info from the filepath. Our filepath included `snapshot_dt` and `snapshot_ts` directories. The Spark job captures those by parsing the input filepath (using `input_file_name()` and regex) to get columns `snapshot_dt` and `snapshot_ts` for each record. This means each record knows which snapshot it came from (when it was pulled).

Example 5-12 shows a part of the code used for this first step.

Example 5-12. Spark batch processing

```
df = (
    spark.read
    .schema(schema)
    .json(input_path)  # NDJSON is supported by spark.read.json
    .withColumn("_input_file", input_file_name())
)

# Extract snapshot_dt / snapshot_ts from path like:
# .../snapshot_dt=YYYY-MM-DD/snapshot_ts=YYYYMMDDTHHMMSSZ/...
```

```
df = (
    df.withColumn("snapshot_dt",
                  regexp_extract(col("_input_file"),
                  r"snapshot_dt=([0-9]{4}-[0-9]{2}-[0-9]{2})", 1))
      .withColumn("snapshot_ts",
                  regexp_extract(col("_input_file"),
                  r"snapshot_ts=([0-9]{8}T[0-9]{6}Z)", 1))
)

return df
```

Next, there's some basic normalization (function `normalize_<entity>(..)`). We cast data types appropriately and select the desired columns. For instance, in orders data, the `created_at` string is parsed to a timestamp, and any fields not present in older snapshots can be given defaults. The code also sets a `status` default and a `source` default for orders (if not provided)—in the batch case, we might mark `source="batch_snapshot"` to differentiate from streaming ones. The customers and products are more straightforward (just casting types and passing through the fields with perhaps minor renaming).

The next step is one of the most important parts: define the *upsert versus append* logic. Traditionally, we use the append of the data, partitioning the Iceberg table by `snapshot_dt` and treat each snapshot as separate. We keep historical snapshots so we can time travel or compare changes over time. You could also choose to keep only the latest by overwriting, but here we'll accumulate them in Bronze. We also keep an additional option that allows you to choose to replace the `<entity>` Bronze table on each batch load (essentially keeping it synced to the latest state). We do this by using a parameter `--RESET_TABLE=true` when running the <entity> batch spark job, which triggers the job to drop and recreate the table before inserting the new snapshot. This means `bronze.<entity>` will always reflect the entity's latest full snapshot. This decision is a trade-off: we lose historical snapshot ability for the specific entity in Bronze, but we avoid duplicates.

The final step is moving data from the landing zone into the Iceberg Bronze layer. For each entity, the Spark job ensures the table exists and then performs a *dynamic partition overwrite*.

Unlike traditional Hive-style tables, Iceberg is *partition-aware*. We don't need to manually point Spark to specific directories; instead, we use the Spark DataFrameWriter API, which allows Iceberg to handle the metadata swap atomically. To do that, first, we ensure the Iceberg table is defined with the correct partitioning scheme (Example 5-13). Iceberg stores this partitioning logic in its own metadata.

Example 5-13. Iceberg SQL instruction: create table if it doesn't exist

```
spark.sql("CREATE TABLE IF NOT EXISTS bronze.<entity> (
          ...
          snapshot_dt STRING,
          snapshot_ts STRING,
        )
        USING iceberg
        PARTITIONED BY (snapshot_dt, snapshot_ts)")
```

Then, instead of using a standard `INSERT` or a manual SQL `PARTITION` declaration, we use the writeTo API: `.writeTo(table).overwritePartitions()`. This tells Iceberg to identify which partitions are present in the new DataFrame and replace only those specific partitions in the table.

> **TIP**
>
> Hands-On Checkpoint: After running the Spark batch job for all three entities—`customers`, `products`, `orders`—you should see the corresponding Iceberg tables (`bronze.customers`, `bronze.products`, `bronze.orders`) populated alongside the previously ingested `bronze.live_orders` from streaming.

At this point, the landing zone has done its job (it's a raw data repository), and our Bronze layer (the structured lakehouse) is up to date with both streaming and batch data. This completes the ingestion pipeline implemented under *chapter_05/ingestion*.

Storage Layer

The *storage layer* of our data platform is built as a modern data lakehouse, combining the scalability of a data lake with the reliability of a data warehouse. In practice, this means raw data lands in GCS and is managed with Apache Iceberg's open table format, while still being queryable directly from BigQuery. This approach provides ACID guarantees, schema flexibility, and even time travel capabilities on the data lake, all accessible through familiar SQL in BigQuery. It forms the foundation (the Bronze layer) on which our Silver and Gold layers are built.

The Data Lakehouse

A *data lakehouse* is an architectural pattern that merges the best of data lakes and data warehouses. In our case, we store data in open formats on cheap, scalable cloud storage, but with a warehouse-like management layer on top (Iceberg) that enables transactions, schema enforcement, and indexing. This hybrid design lets us decouple storage and compute: data is persisted in GCS, and multiple engines (e.g., Spark for ingestion, BigQuery for analytics) can operate on the same data without duplication. The lakehouse ensures that raw data is preserved and auditable, while providing the governance, ACID consistency, and performance features of a warehouse.

Implementing a lakehouse architecture introduces several key advantages to our storage layer:

Single source of truth

> Data is ingested once into the Bronze layer on GCS and never manually copied; both batch and streaming sources end up in the same unified tables. BigQuery references this data in place (via BigLake), ensuring analysts query the *latest truth* directly from storage without redundant ETL processes.

ACID transactions and reliability

> Apache Iceberg manages our dataset's writes so they are atomic, consistent, isolated, and durable. An ingestion job either succeeds

completely or fails entirely, preventing partial or corrupt data from ever being visible to downstream users.

Schema management and evolution

As source schemas change over time, the lakehouse can adapt without breaking downstream queries. Iceberg allows adding or removing columns and evolving partition strategies without expensive reprocessing. This means our Bronze tables can gracefully absorb changes (such as a new field in `customers` data) in a controlled way.

Performance via pruning and indexing

Unlike raw cloud storage, the table format maintains metadata (e.g., min/max statistics and partition indexes) that query engines use to skip irrelevant data. BigQuery, when querying these Iceberg tables, reads only the necessary Parquet files thanks to Iceberg's manifest-based pruning, which accelerates scans on large partitions.

Seamless integration of batch and stream data

The lakehouse Bronze layer captures both streaming events and batch snapshots into the same tables, naturally supporting Lambda architecture. For example, new incremental `orders` from Pub/Sub and full daily snapshots of orders can coexist in the `bronze.orders` table, with each commit adding a snapshot. The Iceberg metadata keeps these writes isolated yet consistent, so analysts don't have to worry about merge conflicts. They just see a complete table update after each load. Note that, to better demonstrate the differences between real-time and bulk ingestion, this book maintains two separate tracks for `order` data: incremental events from Pub/Sub into `bronze.live_orders` and full snapshots from the REST API into `bronze.orders`.

Open table formats

One foundational technology enables all the advantages of a lakehouse: *open table formats*. These types of tables are an emerging class of data storage technology (e.g., Apache Iceberg, Delta Lake, Apache Hudi) designed for transactional data lakes. Open table formats introduce a *metadata layer* on top of flat files, turning a collection of Parquet/Optimized Row Columnar (ORC) files in cloud storage into a queryable table with ACID properties and snapshots. They are *open* because they are vendor-neutral and widely supported, preventing lock-in while ensuring interoperability across engines and clouds.

Our project chose Apache Iceberg as the open table format for the Bronze layer, due to its robust design and compatibility with BigQuery. Iceberg and its peers solve classic data lake problems by providing:

- Atomic transactions for reliable updates
- Flexible schema evolution and partition management
- Time travel querying on historical data
- Efficient query planning through table metadata and indexes

In essence, formats like Iceberg bring database-like features to cloud storage, enabling a true lakehouse. Using Iceberg ensures that raw data in GCS is not just files in a bucket but a well-managed table that multiple tools can read consistently. BigQuery's support for Iceberg tables is a testament to the industry trend: even cloud warehouses are embracing these open formats so that data can live in the lake without sacrificing management features.

Data Storage and Management with Apache Iceberg

Now that we've established some basic concepts, this section shows how Apache Iceberg is implemented as our storage layer. Raw JSON data from various sources (streaming and batch) is first landed in a GCS landing zone. From there, structured Bronze tables are materialized in Iceberg format on GCS via Apache Spark jobs.

Each table (e.g., `bronze.customers`, `bronze.orders`) corresponds to a managed directory in our cloud storage warehouse directory (*gs://<bucket>/iceberg/warehouse/bronze/<entity>*). The Spark ingestion jobs write to these tables using Iceberg's Hadoop catalog, which manages metadata in the GCS filesystem itself. As a result, after ingestion, our bucket contains an *iceberg/warehouse/bronze/<entity>* folder with subdirectories for each table and a metadata folder per table tracking all commits (e.g., *.../bronze/orders/metadata/ contains v1.metadata.json*, *v2.metadata.json*, etc.). This multilevel metadata approach allows engines like BigQuery to plan queries by reading small JSON files rather than scanning every Parquet file in the bucket.

To optimize query performance and organization, the Bronze tables are partitioned by ingestion date. Each record includes partitioning fields: `ingest_date` (the daily date) for `bronze.live_orders`, and `snapshot_dt` and `snapshot_ts` (the daily timestamps) for the batch entities. The Iceberg tables are partitioned by these fields. Physically, this means data files are stored in paths such as *.../bronze/live_orders/data/ingest_date=2026-01-15/part-...parquet* or *.../bronze/orders/data/snapshot_dt=2026-01-15/snapshot_ts=20260115T183139Z/part-...parquet*, respectively. This simple date-based partitioning allows BigQuery and other engines to easily prune data when filtering by date, while keeping the partition strategy stable even as the schema evolves. Iceberg supports *partition evolution*, allowing us to change from daily to hourly partitions in the future without rewriting historical data.

Another benefit is *data skipping*: since each snapshot's metadata knows which partitions and ranges of values each file covers, a query for, say, `WHERE id = 123` can quickly eliminate most files if Iceberg's statistics indicate they don't contain that customer. This kind of pushdown happens behind the scenes and is one reason we can store large volumes of raw data without choking BigQuery: the overhead of the external table is low because BigQuery reads only what Iceberg tells it is necessary. In effect this creates a *separation of concerns*: Iceberg handles data layout, metadata, and

consistency; BigQuery handles heavy SQL lifting and leaves storage management to the Iceberg layer.

Finally, in terms of *maintenance*, Iceberg provides capabilities such as snapshot expiration and data file compaction. Over time, we may accumulate many snapshots (e.g., daily loads for a year). We can configure a retention period or run a procedure to expire old snapshots once they're no longer needed, freeing storage but still preserving an audit window (e.g., as the last 30 days of history).

In streaming ingestion scenarios, compaction should be treated as an architectural requirement rather than an optional optimization. Frequent micro-batch commits naturally produce many small data files, which over time increase metadata size and degrade query planning efficiency. In the case of engines such as BigQuery reading Iceberg tables externally, excessive small files also increase object listing overhead. Scheduling periodic data file rewrite operations (for example, Iceberg's `rewrite_data_files`) is therefore a core component of maintaining a healthy lakehouse.

These table management tasks can be performed via scripts or SQL procedures (e.g., `CALL system.expire_snapshots(...)`) without downtime. What's important is that the storage layer is self-managing to a large extent. We don't manually delete or overwrite files; instead, we let Iceberg orchestrate all changes through its transactions. This reduces the risk of human error in raw data storage and means the Bronze layer is always in a well-defined state.

Beyond the benefits already discussed, Iceberg offers two indispensable tools for data management: the ability to evolve schemas seamlessly and the capacity to leverage historical snapshots for time travel.

Iceberg schema evolution

Iceberg relies on schema management and evolution. We predefined the schema for each Bronze table to match the expected source data (e.g., `customers` has `id`, `name`, etc.). Iceberg requires a table schema on creation,

which it uses to validate and store data. Over time, if upstream systems send new fields or change types, Iceberg's schema evolution capability lets us modify the table schema safely. For instance, we can add a new column to `bronze.customers` or rename an existing one without reloading all data. Old snapshots of the table retain the previous schema, but queries against the latest snapshot see the updated schema. In our implementation, we've leveraged this by keeping the Bronze schema flexible and using Iceberg's support for optional fields. The pipeline's code also includes a safeguard to reset or drop tables (`--RESET_TABLE=true`) if a serious mismatch is detected during development, but in production you would typically do an Iceberg schema update instead of dropping data. Thanks to Iceberg, schema changes become a metadata operation rather than a full rewrite—ensuring that analysts always see a consistent, up-to-date schema in BigQuery even as the raw data structures evolve.

Another powerful aspect of this design is that BigQuery can query the Iceberg tables directly as external tables. We created a BigQuery dataset called `bronze` (*/sql/ddl/01_bq_bronze_schema.sql*) and, for each Iceberg table on GCS, defined a corresponding external table in BigQuery (using BigLake connectors). Example 5-14 shows the example for the `products` entity.

Example 5-14. Creating a BigQuery external table

```
CREATE OR REPLACE EXTERNAL TABLE `<project_id>.bronze.products`
WITH CONNECTION
   `<project_id>.<region>.<bigquery_connection>`
OPTIONS (
  format = 'ICEBERG',
  uris = [
    'gs://<bucket>/iceberg/warehouse
         /bronze/products/metadata/v2.metadata.json'
  ]
);
```

These external tables use Iceberg's metadata for table schema and file locations. In practice, each BigQuery external table points to the latest Iceberg snapshot of the underlying data. For example, the `bronze.products` table in BigQuery is defined with the Iceberg format and

a URI that references the current Iceberg metadata JSON (`vN.metadata.json`) for the products table on GCS. BigQuery uses a federated connection to GCS to read the data via this metadata pointer.

This setup means that analysts can run standard SQL queries in BigQuery against `bronze.customers` or `bronze.orders` and get the most up-to-date data from the data lake, with Iceberg ensuring that the data is consistent.

Finally, there's one small caveat when using BigQuery external tables on Iceberg: BigQuery does not automatically track Iceberg's latest snapshot. Whenever new data is ingested and a new Iceberg snapshot is created, the BigQuery table's pointer (the metadata URI) needs to be updated to that new snapshot. We implemented a lightweight mechanism to handle this: a refresh job that updates BigQuery table definitions to the newest snapshot in place. The job (`ch05-refresh-bq-iceberg-tables`) reads each table's Iceberg *version-hint.text* (which tells us the latest snapshot number) and constructs the path to the new `vN.metadata.json` file. Under the hood, this job runs `CREATE OR REPLACE EXTERNAL TABLE ... OPTIONS (format = 'ICEBERG', uris=[...vN.metadata.json])` in BigQuery for each table, effectively repointing the external table to the newest data commit, from the *version-hint.text* file. Typically we invoke this refresh job after each batch ingestion or periodically during streaming.

Note that we intentionally configure BigQuery Iceberg external tables by directly referencing Iceberg metadata files in object storage rather than using BigLake's managed catalog. This preserves Iceberg's engine-agnostic architecture, ensures that metadata and table state remain portable across compute engines and cloud platforms, avoids vendor lock-in, and keeps storage as the source of truth. This also aligns with Iceberg's design principle that metadata should be independent of any single query engine.

This choice reflects a deliberate trade-off. Using a managed catalog such as BigLake Metastore can simplify synchronization by automatically tracking new snapshots. However, it introduces tighter coupling between table state and the query engine's catalog layer. By managing refresh through a lightweight orchestration job instead, we maintain portability and open-

format governance, accepting the need for explicit synchronization as part of the platform's operational responsibility.

Auditing and recovery: Time travel with SQL

A standout advantage of our Iceberg-based storage layer is time travel, the ability to query historical data easily. Iceberg retains past snapshots, which we can leverage for auditing and recovery scenarios. Imagine a case where a bug in a transformation overwrote some values, or an analyst asks, "What did the table look like before yesterday's load?" In a traditional pipeline, you might be out of luck (unless you kept backups). In our lakehouse, we simply query the Bronze table as of a previous point in time. Each Iceberg snapshot is essentially a versioned dataset, so we have an immutable record of the data at that snapshot. This is invaluable for debugging and trust: Iceberg maintains a history of table snapshots, allowing users to query data as it existed at a specific point in time. For example, we could retrieve last week's state of `bronze.products` to investigate an anomaly in the product catalog, or we could compare two snapshots to see what changed (which products were added or modified).

Practically, there are a couple of ways to do time travel queries. In Spark or another engine with full Iceberg support, you can use SQL syntax like `SELECT * FROM bronze.products TIMESTAMP AS OF '2026-01-10 00:00:00'` or `... VERSION AS OF 42` (where 42 is a snapshot ID). That's the point where snapshot history stops being an implementation detail and becomes a capability the business can rely on. In our lakehouse, analysts can time travel in BigQuery against the Bronze Iceberg tables via BigLake, as shown in Example 5-15.

Example 5-15. Time travel with BigQuery and Iceberg

```
SELECT *
FROM bronze.customers
FOR SYSTEM_TIME AS OF
     TIMESTAMP_SUB(CURRENT_TIMESTAMP(), INTERVAL 1 DAY);
```

This is the *audit glance* use case: *What did we have yesterday? Did this customer exist? Was the email different?* It's also a clean way to validate

batch loads, sanity check a release, or confirm whether an anomaly was introduced recently. But there's another, more complex, real-word case, where we may need to compare two or more points in time for the same table. This is where BigQuery provides some nuance, due to its use of `FOR SYSTEM_TIME AS OF`, since all references to a certain table in the query must use the same timestamp. So if you write something like:

- `bronze.customers FOR SYSTEM_TIME AS OF ...` (yesterday)
- joined to `bronze.customers` (today / live)

BigQuery throws: `Query error: Table 'bronze.customers' is referenced with and without 'FOR SYSTEM_TIME AS OF' expression. If a 'FOR SYSTEM_TIME AS OF' expression is used, all references of a table should use the same TIMESTAMP value.`

This is not a limitation of Iceberg; it's a query-planning constraint in BigQuery. The fix is simple and operationally clean: materialize one snapshot as a separate object so BigQuery sees it as a distinct table. The most efficient pattern is a temporary table, as seen in Example 5-16.

Example 5-16. Time travel with BigQuery and Iceberg with two points in time

```
 -- Step 1: Materialize the historical snapshot into a temp table
CREATE OR REPLACE TEMP TABLE yesterday_customers AS
SELECT *
FROM bronze.customers
FOR SYSTEM_TIME AS OF TIMESTAMP_SUB(CURRENT_TIMESTAMP(), INTERVAL 1 DAY);

-- Step 2: Compare the temp table to the live table
SELECT
  COALESCE(cust.id, yestcust.id) AS customer_id,
  yestcust.name AS name_yesterday,
  cust.name AS name_today,
  yestcust.email AS email_yesterday,
  cust.email AS email_today
FROM yesterday_customers as yestcust
FULL OUTER JOIN bronze.customers as cust
 ON yestcust.id = cust.id
WHERE
```

```
  yestcust.name != cust.name
  OR yestcust.email != cust.email
order by 1;
```

With this solution, we can compare the same table at two or more points in time, and:

- Avoid manual cleanup. Temporary tables disappear automatically at the end of the session, so analysts can run comparisons without leaving behind a trail of snapshot tables.
- Make time travel usable for governance. Once you can diff two points in time with plain SQL, you can operationalize audits and answer questions such as:
  - What changed since yesterday?
  - Which customers had PII updates?
  - Did an ingestion run drop records?
  - Which dimension attributes drifted?

The storage layer built on Apache Iceberg provides the strengths of a data warehouse (consistency, structured querying, time travel) on top of the flexibility of a data lake (cheap storage, semi-structured data, multiple producers/ingestion). By designing the Bronze tier around open-table formats, data is ingested once into GCS and then queried as managed tables in BigQuery.

Transformation and Data Modeling with SQL

After raw data lands in the Bronze layer, the next step is to refine it into analytical tables. This project uses dbt on BigQuery to build a dedicated SQL transformation layer. The dbt project defines a clear two-tier model structure, Silver and Gold, that standardizes data and prepares it for analysis. In the Silver layer, each model corresponds to a single source entity (e.g., `orders`, `products`, `customers`) and produces a cleaned,

conformed view of the latest data. In the Gold layer, the models serve business needs directly: dimension tables and fact tables are materialized for analysts, and additional marts aggregate facts into common metrics, creating the foundations for building a Semantic layer. This approach aligns perfectly with our SQL-first philosophy and ensures that by the time data reaches business consumers, it is well-structured, tested, and aligned with defined business logic.

To achieve our goal, we followed some dbt best practices in building these models:

- Used *sources* to represent the Bronze external tables. In dbt, we declare the bronze tables as sources (with their schema and expected fields). This way, we can use `{{ source('bronze', 'customers') }}` in our models. Sources allow testing (e.g., freshness or count of source data if needed) and documentation of upstream data.
- Organized models into folders by layer (a *models/silver/* folder and *models/gold/* folder). We used dbt's configuration to automatically assign the BigQuery dataset (schema) based on folder: all models in *silver/* go to dataset `silver`, and in *gold/* to dataset `gold`. This avoids hardcoding dataset names in every model and ensures a clean separation of layers.
- Wrote *tests* for things like primary key uniqueness in dimensions, referential integrity, and basic data quality.
- Enabled *documentation* by writing descriptions for models and columns in YAML, so that the dbt docs site is informative. For instance, describing `fact_orders.total_amount` as *The total monetary amount of the order, in the original currency*.

Next, let's look at the physical implementation. As introduced, our dbt project cleanly separates concerns into Silver and Gold layers, complemented by integrated testing and documentation. The next sections describe how they function in our pipeline.

Silver and Gold Model Design in dbt: Building a Medallion Architecture

The dbt project's folder structure cleanly separates Silver and Gold transformations. Under the hood, dbt is configured to treat any model in the `silver/` directory as a BigQuery *view* in the `silver` dataset, and models in `gold/` as tables in the `gold` dataset. This means Silver models are essentially *live views* over the latest Bronze data (always up to date, without storing duplicates), while Gold models persist transformed results for fast access. The choice of views for Silver keeps storage costs low and ensures freshness on each run, whereas tables for Gold allow heavy aggregations or joins to be pre-computed for performance.

> **NOTE**
>
> The *view-based Silver* strategy is a great starting point for development and small-to-medium datasets, enough for our current setup, but as you scale, it often becomes a bottleneck. For production environments with massive data volumes, transitioning to *incremental materializations*, or an hybrid approach, is recommended to improve performance and reduce query costs.

This Silver/Gold design implements a medallion architecture within SQL. Silver models focus on standardizing and cleaning the raw data, and Gold models create the business-friendly schema (often a star schema of fact and dimension tables), normally also part, or source, of a Semantic layer. In practice, this means each Silver model draws directly from the Bronze tables, and each Gold model builds on one or more Silver models to form final outputs. The `dbt ref()` function establishes these dependencies between models, letting dbt automatically build and update them in the correct order as a pipeline (no manual job scheduling needed). The configured schema naming also ensures each environment is isolated: for example, the custom schema generation macro (`generate_schema_name(..)`) maps models to the appropriate dataset

(`silver` or `gold`, see Table 5-1) based on their folder, so developers can run the same project in dev or prod without code changes.

Table 5-1. Map of layers to distinct BigQuery datasets

Layer	BigQuery Dataset
Bronze	`bronze`
Silver	`silver`
Gold	`gold`

Staging with Silver models

Silver models prepare raw data for deeper transformations. Each Silver model is roughly scoped to a single source table and contains only the logic needed to clean, typecast, and filter that data, nothing more. This keeps the staging layer simple and focused. In our project, for example, the model `stg_orders` is a lightweight SQL view that selects all columns from the raw Bronze orders table and filters it to the latest snapshot of data. It uses a common macro to ensure only the newest records (by ingestion timestamp) are included (`latest_snapshot_where(..)`), preventing duplication of outdated records. The SQL in Example 5-17 shows how `stg_orders` pulls from the source and applies the snapshot filter and type casting.

Example 5-17. Staging orders code

```
with src as (
  select *
  from {{ source('bronze', 'orders') }}
  {{ latest_snapshot_where(source('bronze', 'orders')) }}
)

select
  cast(id as int64) as order_id,
```

```
  cast(customer_id as int64) as customer_id,
  cast(product_id as int64) as product_id,
  ...
  cast(ingest_ts as timestamp) as ingest_time
from src
```

Here, the `{{ source('bronze', 'orders') }}` reference tells dbt to read from the raw Bronze orders table, and the custom `latest_snapshot_where` macro appends a `WHERE` clause to pick the maximum ingest timestamp from that table. This effectively implements a deduplication by snapshot: if the Bronze table contains multiple snapshots or versions of the same entity, the Silver view will show only the most recent version. Following the source query, the `SELECT` statement casts each field to a strict type (e.g., ensuring IDs are 64-bit integers, timestamps are proper `timestamp` types, etc.). This enforces a consistent schema regardless of how data was ingested (for instance, JSON sources might have treated some numbers as strings, but in Silver they become typed columns). The Silver models for customers and products follow the staging pattern introduced in "Storage and processing": they select from `bronze.customers` or `bronze.products`, filter to the latest data, and cast columns to well-defined types. While the Silver layer conceptually allows for cross-system joins, we have opted here for a narrowly focused, one-to-one transformation. This adheres to the best practice of using staging models to rename columns and cast data types while avoiding joins or aggregations to improve modularity and debuggability. By keeping these models simple, we ensure a strong, isolated foundation before moving into the complex business logic of the Gold layer.

Building facts and dimensions in the Gold layer

If Silver models are the clean ingredients, the Gold models are the finished dishes ready for consumption. The Gold layer of this project contains fact tables, dimension tables, and marts that together form a star schema and a Semantic layer for analytics. Many Gold models are straightforward selections from the Silver layer, but with a change in materialization. For

instance, `fct_orders` (a fact table of orders) simply selects all fields from the `stg_orders` view, as in Example 5-18.

Example 5-18. Fact orders code

```
select
  order_id,
  customer_id,
  product_id,
  quantity,
  unit_price,
  total_amount,
  created_at
from {{ ref('stg_orders') }}
```

This might seem like an identity operation, but it serves two purposes: it materializes the data as a BigQuery table (in the Gold dataset) and it gives the table a business-oriented name (`fct_orders`) and structure. By materializing `fct_orders` as a table, we create a stable snapshot of orders that won't change until the next `dbt run`, and heavy queries (like large aggregations or joins) can hit this table without repeatedly scanning the raw data. The fact table contains measures such as `quantity` and `total_amount` that analysts care about, and keys that link to dimensions.

Alongside the facts, dimension tables in Gold provide the descriptive context. In our case, `dim_customers` and `dim_products` are built from their Silver counterparts to serve as reference data for analyses. For example, `dim_products` selects product IDs, names, categories, and prices from `stg_products`, again materializing them as a table in BigQuery. Splitting out dimensions like this has a practical benefit: business users or BI tools can join facts to dimensions to get human-friendly attributes (e.g., product category names) without repeatedly hitting the source-of-truth tables. It also decouples the core entity information from the fact events. For instance, if product attributes are updated (new category or price change in a new snapshot), the dimension table can be refreshed independently. The Gold layer's dimension models are materialized as full tables (not just views) to provide a stable reference that can be indexed or cached as needed for fast joins.

Fact tables exist to define what happened, at the most atomic level the business can agree on. However, facts are not where the business decides how to talk about performance. That conversation happens at a different level. Questions like "What is daily revenue by category?" or "How many customers placed more than five orders this month?" are not row-level concerns. They depend on aggregation, time grain, and analytical intent. Trying to force these definitions into the fact table would either change the table's grain or lead to an explosion of pre-computed metrics that are hard to reason about and even harder to maintain.

This is where marts enter the picture. In this project, *marts* are not arbitrary slices of data, nor are they places where logic is casually duplicated. Marts are where metric-level truth is defined, once and explicitly. A model such as `mart_revenue_by_category_daily`, Example 5-19, does not redefine what revenue means per order. Instead, it builds on the canonical definition already present in `fct_orders` and expresses how the business wants to view that revenue at a daily, categorical level.

Example 5-19. Mart model

```
select
  date(o.created_at) as order_date,
  p.category,
  count(*) as orders,
  sum(o.total_amount) as revenue
from {{ ref('fct_orders') }} o
join {{ ref('dim_products') }} p
  on o.product_id = p.product_id
group by 1,2
```

The main point is that a single source of truth is not achieved by collapsing all logic into one table. It is achieved by clearly separating responsibilities. Facts own atomic truth. Marts own analytical interpretation. When that boundary is respected, consistency naturally follows.

Seen this way, marts also represent the first concrete step toward a Semantic layer. They don't yet provide metric APIs or BI-level abstractions, but they do something equally important: they define metrics in code, give them names, document them, and make them discoverable. Instead of letting

every dashboard reinvent “revenue,” marts establish shared, governed outputs that downstream tools can trust. We’ll return to the idea of a full Semantic layer in “Consume Layer”. For now, marts serve as its practical foundation.

Tests and documentation

As the transformation layer grows, correctness becomes less about writing clever SQL and more about enforcing expectations. This is where dbt fundamentally changes the nature of SQL transformations.

In this project, tests are not an afterthought. They are written alongside models and encode assumptions that would otherwise remain implicit. A dimension assumes that critical attributes are never null. A revenue metric assumes prices are positive. Rather than relying on tribal knowledge, these assumptions are formalized and checked every time the pipeline runs.

dbt provides two primary methods, or types of tests, for validating data:

Generic tests (YAML-based)

> These are defined directly in our *schema.yml* files and are highly efficient for standard validations. dbt provides four built-in generic tests:
>
> `unique`
>
> > Ensures a column (like `order_id`) has no duplicate values
>
> `not_null`
>
> > Ensures a column (like `customer_id`) never contains empty values
>
> `accepted_values`
>
> > Validates that a column contains only specific strings (e.g., `status` must be *ordered*, *shipped*, or *delivered*)
>
> `relationships`

Checks referential integrity (e.g., every `order.customer_id` exists in the `customers` table)

Singular tests (SQL-based)

For complex business logic that generic tests can't cover, we write custom SQL queries. A *singular test* is simply a `SELECT` statement that returns rows that should not exist. For example, a revenue metric assumes prices are positive (`tests/orders_amount_positive.sql`). If the query returns any rows where `total_amount <= 0`, the test fails, as shown in Example 5-20.

Example 5-20. Test example, validating `total_amount` metric

```
select *
from {{ ref('fct_orders') }}
where total_amount is null or total_amount <= 0
```

When the `dbt build` executes, models are built and tests are evaluated immediately. If a test fails, the run stops. No Gold tables are updated. From an operational perspective, this is a powerful shift. Data quality issues are surfaced at the transformation boundary, before analysts ever see inconsistent numbers. The pipeline doesn't quietly degrade, because it refuses to publish results that violate its own contracts.

Documentation follows the same philosophy. Each model and column is described in *schema.yml* files that live next to the SQL (Example 5-21). dbt then turns these descriptions into a browsable documentation site that includes model lineage, field definitions, and dependencies. The transformation layer becomes self-describing. Anyone can trace a metric from a dashboard back through marts, facts, and staging models to its original source.

Example 5-21. Documentation sample within schema.yml in the models/gold folder; dbt uses this to generate the documentation page

```
version: 2
```

```
- name: dim_products
  description: >
    Products dimension table containing product details.
  columns:
    - name: product_id
      description: Primary product identifier.
    - name: product_name
      description: Name of the product.
    - name: category
      description: Category to which the product belongs.
    - name: price
      description: Price of the product.

  ...

- name: mart_sales
  description: >
    Sales mart aggregating revenue and total sales by order date.
```

All of this logic would be far less valuable if it ran only on a developer's laptop. In this project, dbt is treated as production infrastructure: As with the other ingestion jobs, transformations are also executed using a Cloud Run job called `ch05-dbt-job`. When triggered, the job runs `dbt build` to materialize Silver and Gold models and execute all tests, followed by `dbt docs generate` to produce up-to-date documentation. The job runs under a dedicated service account with just enough permissions to create BigQuery jobs and write to the appropriate datasets.

This setup deliberately avoids long-lived virtual machines or heavyweight CI pipelines. Cloud Run provides a clean execution model in which the job starts, runs dbt, and exits. It can be triggered manually, scheduled, or orchestrated alongside ingestion jobs. The transformation layer thus becomes just another managed step in the platform.

Because this project uses dbt Core rather than dbt Cloud, the documentation generated by `dbt docs generate` exists as local artifacts instead of a hosted service. To make these insights accessible, we publish the generated documentation to a dedicated, public-access GCS bucket as part of the job execution. This bucket is intentionally isolated from the primary data storage to preserve security while still providing an easy way for readers

and stakeholders to inspect lineage, model definitions, and dependencies. Detailed setup instructions and the deployment script are available in *chapter_05/transformations/README.md*.

When the job completes, the results are immediately visible. The Silver dataset contains views representing the current, conformed state of each entity. The Gold dataset contains materialized fact tables and marts ready for analysis. At the same time, the generated documentation is uploaded to a GCS bucket, making lineage and definitions accessible through a browser.

Ultimately, what emerges is more durable than a mere collection of SQL files. The transformation layer evolves into a shared language, a single source of truth where logic, assumptions, and business definitions are versioned and visible to engineers and analysts alike. By centralizing this knowledge, the data platform becomes a transparent, governed asset accessible to the entire organization.

> **TIP**
>
> Hands-On Checkpoint: We have now established our Silver and Gold models, as well as their supporting tests and documentation. If you are following along and want to continue to gain practical experience with these transformations, now is the ideal time to implement the code. You can find the complete dbt project in the book's GitHub repository in the *chapter_05/transformations* directory.

Orchestration and Workflow Management

Designing the pipeline and writing the code are only part of the story. We also need to *orchestrate* these components reliably. Orchestration involves scheduling jobs, managing dependencies between them, and handling failures or retries. In our solution, we have multiple moving pieces: Cloud Run jobs (for ingestion into the landing zone), Dataproc Spark jobs (to build the Iceberg table within the Bronze layer), and dbt transformation runs. Manually running them in order, as we did during development, is not viable for production. We need an automated workflow.

For our pipeline, we decided to use Google Cloud Workflows, a lightweight, serverless orchestration service within Google Cloud, to coordinate the pieces. Cloud Workflows allows us to define a directed acyclic graph (DAG) of tasks in YAML, calling various GCP services and waiting for their completion. It's similar in spirit to tools like Apache Airflow (Cloud Composer on GCP) or Prefect, but it requires no infrastructure setup and is tightly integrated with GCP APIs, aligning with our principles of low maintenance and managed services to reduce operational burden.

To modularize our pipeline orchestration we defined several workflow YAML files (in *chapter_05/orchestration/*):

workflow_full_batch_orchestration.yml

> A master workflow, deployed as `ch05-wf-full-batch-orchestration`, that executes all batch ingestion and processing steps sequentially. This workflow first invokes each entity-specific batch ingestion workflow (`customers`, `products`, `orders`), then triggers the downstream steps to refresh BigQuery tables pointers and run dbt. It ensures the batch pipeline runs in order from raw data pull to final transformations.

workflow_ingest_customers.yml, ...ingest-products.yml, ...ingest-orders.yml

> Individual ingestion workflows, deployed as `ch05-wf-ingest-<entity>`, for each entity. Each of these orchestrates a single batch data load: calling a Cloud Run Job to ingest raw JSON from the source API into the landing zone and then submitting a Dataproc Spark batch to convert that JSON into the Bronze table for that entity. These workflows include loops to poll the Dataproc job status and retry if needed.

workflow_ingest_customers.yml

> A streaming ingestion workflow, deployed as `ch05-wf-ingest-live_orders`, for live orders. This workflow handles near-real-time

data: it triggers a Cloud Run job that reads Pub/Sub events and writes JSON to the landing zone (streaming path) and then starts a Dataproc Structured Streaming job to continually append new orders to the Bronze table. This workflow is separate from the full batch flow, since it runs continuously (or on a frequent schedule) rather than as a one-off sequence.

Cloud Workflows Orchestration Pipeline

The full batch orchestration workflow, `ch05-wf-full-batch-orchestration`, serves as the entry point when we want to refresh all data. It uses Cloud Workflow's ability to call other workflows and services. For example, the master workflow calls each ingest workflow via an HTTP `POST` to the Workflows REST API. It waits for one to finish before moving to the next, ensuring customers, products, and orders are ingested in sequence. Once all three batch ingestions succeed, the master workflow calls two final Cloud Run jobs: one to refresh BigQuery's external table pointers to the new Iceberg data, and another to run the dbt transformation job that builds the Silver/Gold layers. The master workflow's steps are as follows:

1. Invoke `ch05-wf-ingest-customers` workflow; wait for completion
2. Invoke `ch05-wf-ingest-products` workflow; wait for completion
3. Invoke `ch05-wf-ingest-orders` workflow; wait for completion
4. Call a Cloud Run job to refresh BigQuery external tables (making new Bronze data visible to BigQuery)
5. Call a Cloud Run job to run dbt (executing all pending SQL models for Silver/Gold updates)

Each entity-specific ingestion workflow encapsulates the logic for a single source. Taking the customers workflow as an example, Example 5-22

shows its two main steps: first calling a Cloud Run job for batch ingestion, then submitting a Dataproc batch for the Bronze step. Please note that to maintain readability, some configuration lines in this snippet have been broken or truncated. For the complete, executable version of this logic, please refer to *orchestration/workflow_ingest_customers.yml*. The workflow reads configuration like `PROJECT_ID`, `BUCKET`, `REGION` and `SERVICE_ACCOUNT` from environment variables (set at deploy time) so it knows where to run jobs and where data lives. It then uses the `googleapis.run.v1.namespaces.jobs.run` connector to start the Cloud Run job `ch05-batch-ingestion-customers`, and an HTTP call to the Dataproc batches API to run the PySpark job for landing zone → Bronze. The Cloud Run job writes a full snapshot of customers to the GCS landing zone, and the Spark job (running *01_lz_batch_to_bronze_layer.py*) reads that JSON and writes the `bronze.customers` Iceberg table.

Example 5-22. Cloud Workflows excerpt for batch ingest (customers)

```
- init:
    assign:
        ...
        - job_name: ch05-batch-ingestion-customers
        ...

 - batch_ingestion_customers:
     call: googleapis.run.v1.namespaces.jobs.run
     args:
         name: ${"namespaces/" +
               project_id + "/jobs/" +
               job_name}
         location: ${job_location}
     result: run_execution

 - spark_batch_customers_to_bronze:
     call: http.post
         url: ${"https://dataproc.googleapis.com/v1/projects/" +
               project_id +
               "/locations/"+
               job_location +
               "/batches"}
         auth:
             type: OAuth2
             scopes: "https://www.googleapis.com/auth/cloud-platform"
```

```
        headers:
            Content-Type: "application/json"
        query:
            batchId: ${batch_id}
        body:
            environmentConfig:
                executionConfig:
                    serviceAccount: ${service_account}
            pysparkBatch:
                mainPythonFileUri: ${"gs://" + bucket +
                                "/chapter_05/ingestion/spark
                                /jobs/batch
                                /01_lz_batch_to_bronze_layer.py"}
                args:
                - "--ENTITY=customers"
                - ${"--INPUT_PATH=gs://" + bucket +
                                 "/landing_zone/batch/customers/"}
                - "--CATALOG_NAME=local"
                - "--DB_NAME=bronze"
                - "--TABLE_NAME=customers"
        ...             # (other Spark configs omitted)
```

Here, the workflow references the PySpark job file in GCS (*mainPythonFileUri* pointing to a *gs://...* path). The workflow passes in parameters like `--ENTITY=customers` and input and output paths, so the Spark code knows which dataset to process and where to read/write. It also specifies the service account to run the Dataproc job (for permissions) and configures resources (CPU, memory) and packages (Iceberg runtime JAR) via the `runtimeConfig` and `properties` in the YAML. After submitting the batch, the workflow enters a loop to poll the Dataproc job status until it succeeds, handling retries or failures appropriately. This pattern is the same for products and orders workflows. Only the job names and arguments (e.g., `--ENTITY=products`) differ.

> **NOTE**
>
> Cloud Workflows cannot access local files, so any code needed by the Dataproc step must be uploaded to Cloud Storage beforehand. In this case we copy our Spark job scripts into the bucket under the same folder structure used locally. For example, the file *chapter_05/ingestion/spark/jobs/batch/01_lz_batch_to_bronze_layer.py* is copied to *gs://<BUCKET>/chapter_05/ingestion/spark/jobs/batch/* so that the workflow's Dataproc call can retrieve it.

The live orders (streaming) workflow operates on a similar principle but with a streaming twist. The `ch05-wf-ingest-live_orders` workflow kicks off a Cloud Run job (`ch05-streaming-ingestion-job`) that subscribes to Pub/Sub order events and writes them as they arrive to the landing zone's streaming path. Next, it submits a Dataproc Serverless Spark job running *01_lz_orders_to_bronze_layer.py* in Structured Streaming mode. This Spark job will continuously process new files from *gs://<BUCKET>/landing_zone/streaming/orders/* and upsert them into the `bronze.live_orders` table.

In the workflow YAML, we see parameters tuned for streaming, such as `--MAX_FILES_PER_TRIGGER=50` and `--TRIGGER_SECONDS=15` to control micro-batch size and frequency. We also set a `--CHECKPOINT_PATH` on GCS for Spark to maintain state between triggers. To avoid an indefinite run in this demo context, the Dataproc batch is given a time-to-live (TTL) of 10 minutes, after which it will terminate. The `live_orders` workflow doesn't loop on the Spark job in the same way (since streaming jobs run until canceled or TTL), but it does call the refresh external tables job at the end once the streaming ingestion job finishes or times out. This refresh step updates BigQuery's pointer to the latest Iceberg snapshot, ensuring that any new streaming data in Bronze becomes queryable via the external table.

Scheduling the Workflows

You can run these workflows on demand (for example, via the Google Cloud CLI or in the GCP Console as shown in the README), but in

production you would likely schedule the full batch pipeline to run at regular intervals. Cloud Workflows integrates with Cloud Scheduler to support cron-based triggers. For instance, running the `ch05-wf-full-batch-orchestration` every night at 8 p.m. This is configured either through the GCP Console (by adding a Scheduler trigger to the workflow) or via the `gcloud` scheduler command, as shown in Example 5-23.

Example 5-23. Creating a Cloud Scheduler job

```
gcloud scheduler jobs create http run-daily-workflow \
  --schedule="0 20 * * *" \
  --uri="https://workflowexecutions.googleapis.com/v1/projects
            /$PROJECT_ID/locations/$REGION/workflows
            /ch05-wf-full-batch-orchestration/executions" \
  --http-method=POST \
  --oauth-service-account-email=$SERVICE_ACCOUNT \
  --message-body="{}" \
  --location=$REGION
```

Cloud Scheduler will invoke the workflow's execution URL at the set time, so that the pipeline can run unattended. The streaming workflow, on the other hand, might be scheduled to start up periodically (if using TTL, as in our case), or you could design it to run continuously by removing the time limit. In a real deployment, streaming jobs often run 24/7, and you may not need Cloud Scheduler for them unless you want to auto-restart the job on a schedule (for instance, to rotate logs or pick up code changes).

Alternative Orchestration Options

While Cloud Workflows provides a simple, fully managed way to orchestrate GCP services (and it fits our use case by minimizing infrastructure), it's not the only orchestration tool. Another option is Apache Airflow, a popular open source orchestrator that expresses workflows as Python DAGs, designed for complex data workflows and ETL pipelines. Airflow offers a rich UI, built-in scheduling, and a large library of operators to integrate with various systems, which can be advantageous for very complex pipelines or multi-cloud scenarios. However, it requires running (or paying for) an Airflow environment, and

tasks execute with a bit more overhead (it's best for batch jobs that can tolerate a few seconds of scheduler latency).

Another modern option is Prefect, a workflow orchestration framework that also uses Python but aims to be lightweight and flexible. Prefect can orchestrate workloads in a hybrid fashion (cloud-managed orchestration with on-prem or cloud execution) and emphasizes a good developer experience.

Both Airflow and Prefect can achieve the same end-to-end coordination we built here. For example, you can write DAGs or flows that trigger Cloud Run jobs and Dataproc jobs, but they require adding components to the stack. For learning purposes, Cloud Workflows keeps the orchestration simple and within GCP; as the pipeline grows in complexity, Airflow or Prefect become practical alternatives.

Now that we have an orchestration layer in place, our data pipeline is fully automated. We can ingest raw data (batch and streaming), land it in the data lake (GCS), transform it into structured tables (Iceberg Bronze, then dbt Silver/Gold in BigQuery), all orchestrated by Cloud Workflows. With scheduling, this pipeline can run end-to-end on a schedule or on-demand, reliably managing dependencies between jobs.

> **TIP**
>
> Hands-On Checkpoint: We have now set up our orchestration that will automatically manage our data pipelines. If you are following along and want to continue gaining practical experience, now is the ideal time to implement the orchestration. You can find the complete orchestration and scheduling configurations in the book's GitHub repository in the *chapter_05/orchestration* directory.

Next, we move to the consumption layer to unlock the full potential of the data pipeline and configurations we've established.

Consume Layer

Modern data pipelines culminate not in a database but in how data is consumed; that is, by people and systems. The Consume layer serves as the interface and contract through which end users and downstream systems access data. Unlike Bronze, Silver, and Gold layers (which physically store and refine data), the Consume layer is not a new storage tier but a semantic and access layer built on top of Gold. Its purpose is to expose data in a stable, meaningful form for analysts, data scientists, and applications, while shielding consumers from breaking changes and enforcing consistent business logic.

A Stable Interface for Data Consumption

In a medallion architecture, Gold tables (facts, dimensions, marts) are the curated, clean data. However, simply handing users these tables is often insufficient. The Consume layer provides a stable interface (via SQL views, published schemas, or semantic endpoints) that acts as a contract between data engineering and data consumers. This means:

No raw or transient exposure

> Analysts or applications never query a raw Bronze table directly. For example, a streaming Bronze table like `bronze.live_orders` would never be exposed. Instead, the Gold/Consume layer should expose a refined view (or other governed object) that selects and shapes the live orders data for safe consumption.

Stable schemas and logic

> The schemas and business logic exposed in the Consume layer change rarely, and if they must, changes are controlled and versioned. Under the hood, Gold models can evolve (new columns, schema changes, new pipelines), but the public face, the consumption interface, remains consistent. Dashboards, reports, and data apps continue to see the same fields and definitions, even if we swap out how we compute them.

Unified access for all consumers

> Whether it's a BI analyst writing SQL, a data scientist featurizing data, or an application retrieving metrics via an API, all are served from the same single source of truth. Each consumer might use different tools, but they all hit the same governed interface (the Consume layer) rather than disparate tables or one-off logic.

By treating consumption as a first-class layer (much like an API for our warehouse), we formalize how data is used. In practice, the Consume layer often corresponds to a dedicated dataset or schema (for example, `consume` in our scenario) containing views and stable queries. These views encapsulate business logic and hide underlying complexity. As long as the view definitions remain consistent, consumers get the same results and fields, even if the underlying implementation changes.

When are raw Gold tables safe to use directly? In some cases, a Gold model is simple and stable enough to be used as is. For example, a well-defined dimension table (say `dim_products` with product attributes) might be exposed directly if its structure is rock-solid and unlikely to change. Similarly, a fact table with a straightforward grain and clear documentation (`fct_orders` with one row per order) could be queried directly by power users. These are cases where the physical Gold model already aligns perfectly with a long-term semantic contract, so no further abstraction is needed.

When is an abstraction required? Whenever the physical schema is subject to change or the usage logic is complex. For instance, if computing a metric correctly requires joining multiple tables or applying filters (for instance, *revenue* might need to join orders to products for category information, and exclude canceled orders), it's better to expose a prepared metric view rather than expect each user to get the join and filters right. Likewise, when delivering real-time data or experimental features, an interface layer ensures that even if the underlying storage or format changes (say, switching the streaming technology or table format), the consuming queries don't have to change. In short, the Consume layer ensures decoupling where consumers

are protected from how data is produced and stored, relying only on the contract we provide.

Bridging Semantics to Execution

In Chapter 4, we designed a semantic model for our domain by defining entities, relationships, and even ontologies (RDF/OWL) to capture business meaning. The Consume layer brings those semantic definitions to life in SQL. The goal is to connect the conceptual model to executable patterns that users can actually query.

Bridging this gap means embedding semantics explicitly in the data layer. Relying on well-named columns or documentation is not enough to guarantee consistent usage. We need the semantics (the definitions of metrics, the relationships between entities, the permitted aggregations) to be part of the queryable interface. In the Consume layer, we distinguish between physical and semantic models:

Physical models are the actual tables in our warehouse (the outputs of the Gold layer: facts, dimensions, data marts). They store the data with certain structure and performance considerations.

Semantic models consist of the defined entities, metrics, and relationships that give that data meaning. The semantic model lives on top of the physical tables. It encodes things like *Order has a relationship to Customer and Product* or *Total Revenue is defined as the sum of* `total_amount` *after excluding certain statuses* and so on.

In Chapter 3, we approached design from a conceptual perspective (thinking in terms of entities like Orders and Customers). Now we carry that forward: every important business concept should correspond to either a stable column, a documented metric, or a governed view in the Consume layer. This is how modeling choices lead to tangible usage patterns.

Concretely, the Consume layer includes semantic models and governance rules on top of our tables:

- We define explicit metrics and entities. For example, a metric "Total Revenue = sum of `total_amount` on the Orders fact table", or an entity Order that has relationships to Customer and Product. These formal definitions mirror our ontology. By declaring them, we tell the platform what our data means and how it can be used.
- We encode those definitions in our data platform. Rather than just saying *order_id is the primary key* in documentation, we can declare it in a semantic model. Rather than assuming everyone knows to count distinct `order_id` for number of orders, we formally define a measure for `order_count`.

The Consume layer operationalizes the semantic decisions from modeling. It ensures that the meanings we defined (in ontology or in design discussions) translate into actual SQL patterns and objects.

Physical models versus semantic models in practice

To make this concrete, consider our Orders data within the `models/gold` directory. Physically, by the end of the Gold layer we have `fct_orders` (a fact table with one row per order, containing fields like `order_id`, `customer_id`, `product_id`, `total_amount`, `created_at`, etc.). We also have dimension tables like `dim_customers`, `dim_products` for descriptive attributes. These are the physical models.

Now, on top of these, we build a semantic model for Orders, within `models/consume` and call it *orders_semantic.yml*. This semantic model formally declares things that were previously implicit:

Entities

For Orders, we mark `order_id` as the primary key (entity) of this model. We mark `customer_id` and `product_id` as foreign key entities, indicating that an Order references a Customer and a Product. (In a full implementation, we would have separate semantic models for Customers and Products that these link to.)

Dimensions

We declare `created_at` as a time dimension of the Orders model, to the granularity of the day for analysis. We could also expose other dimensions like `order_status` or any attribute that doesn't change the grain of the fact.

Measures

We define fundamental measures on the Orders model, for example an `order_count` measure (count of distinct orders) and a `total_amount_sum` measure (sum of the order's total amount). Each measure is tied to an aggregation type and optionally to a time dimension (e.g., we aggregate `order_count` by the day it was created).

Metrics

We then define higher-level metrics that consumers care about. A metric might map one-to-one with a measure (for example, the Total Orders metric might use the `order_count` measure) or combine measures (for example, Average Order Value is `total_revenue / total_orders`). Metrics can also alias measures with more business-friendly naming or apply filters (e.g., a metric for "Total Revenue (Excluding Cancellations)" might apply a `where` clause to exclude canceled orders).

We encode all of this in a declarative config as part of our dbt project. dbt's new Semantic layer (powered by MetricFlow) provides a YAML syntax to declare semantic models. For our Orders example, the semantic model looks like Example 5-24.

Example 5-24. Semantic model for Orders data

```yaml
semantic_models:
  - name: orders_semantic
    model: ref('fct_orders')
    description: "Semantic model for orders (one row per order)."
    entities:
      - name: order_id
```

```yaml
      type: primary
    - name: customer_id
      type: foreign
    - name: product_id
      type: foreign
  dimensions:
    - name: created_at
      type: time
      type_params:
        time_granularity: day
  measures:
    - name: order_count
      description: "Number of orders"
      agg: count_distinct
      expr: order_id
      agg_time_dimension: created_at
    - name: total_amount_sum
      description: "Total order amount (gross)"
      agg: sum
      expr: total_amount
      agg_time_dimension: created_at

metrics:
  - name: total_revenue
    description: "Total revenue of orders (sum of order amounts)"
    type: simple
    label: total_revenue
    type_params:
      measure: total_amount_sum
  - name: total_orders
    description: "Total number of orders placed"
    type: simple
    label: total_orders
    type_params:
      measure: order_count
  - name: avg_order_value
    description: "Average order value (revenue per order)"
    type: ratio
    label: avg_order_value
    type_params:
      numerator: total_revenue
      denominator: total_orders
```

This YAML formally captures the semantics of our Orders data. Notice how it mirrors the concepts we discussed:

- We marked `order_id` as the unique identifier and linked orders to customers and products. This means the Semantic layer *knows* that an Order can be related to a Customer or Product.
- We defined time granularity on `created_at` so metrics can be aggregated by day.
- We defined measures for the core quantitative fields (count of orders, sum of amount).
- We defined metrics: `total_orders`, `total_revenue`, and `avg_order_value`, which directly or indirectly use those measures.

These definitions give us a semantic graph: `Orders -> Customer`, `Orders -> Product` relationships, along with metrics that can be computed from Orders. By declaring foreign key entities, we have essentially encoded the ontology path *an Order has a Product, which has attributes like category*. The Semantic layer can use that. For example, because we declared `product_id` as a foreign key to a Product model, the system can infer that an Order can be aggregated by any attribute of Product (like `product.category`) via that relationship. This unlocks a powerful capability: inferred joins. The Semantic layer understands the join path, so analysts don't have to manually join tables to group an order metric by a product attribute.

Also note that in our semantic model, *measures* live inside the Orders semantic model (they are tied to the fact table and are raw aggregations like sum or count). *Metrics* are exposed to end users and often are just aliases or simple transformations of measures (as in the preceding example where `total_revenue` is just the sum measure). In some cases, metrics can add business rules on top of measures (like filtering out certain orders or computing ratios or percentages). Here our metrics are straightforward, but we could imagine more complex ones. The key is that by defining them here, everyone uses the same definition. If tomorrow we change what counts as a *completed order* (say, exclude test orders), we would adjust the

metric definition in one place, and all downstream consumers automatically get the updated logic.

SQL-First Semantic Execution

Defining a semantic model is half the battle. We also need to execute queries in line with these definitions. Our approach is SQL-first: consumers will primarily use SQL to get their answers, but with the Semantic layer ensuring those SQL queries are consistent and easy to write.

There are a few ways to enforce semantic intent using SQL:

1. Canonical SQL views

We can create dbt models or views that reflect the semantic definitions. For instance, using the Orders semantic model above, we can create a metrics view inspired by it, that provides preaggregated metrics by day (and possibly by other common dimensions). For example, a view for daily orders metrics could be as shown in Example 5-25.

Example 5-25. Daily order metrics view

```
SELECT
  DATE(o.created_at) AS order_date,
  COUNT(DISTINCT o.order_id) AS total_orders,
  SUM(o.total_amount) AS total_revenue,
  SAFE_DIVIDE(SUM(o.total_amount),
  COUNT(DISTINCT o.order_id)) AS avg_order_value
FROM {{ ref('fct_orders') }} o
GROUP BY DATE(o.created_at)
```

This SQL view directly implements our metrics with the correct formulas (count distinct for orders, sum for revenue, and a ratio for average). By querying `daily_order_metrics`, consumers get consistent results since the logic for these metrics is centralized in one

place. If the definition of a metric needs to change, we can change the view query (or underlying model) instead of chasing down dozens of dashboard queries.

2. Query templates or macros

In a dbt context, we can also use SQL macros to enforce consistent joins or filters. For example, if many metrics require excluding canceled orders, we might provide a macro or a base model that already does `WHERE status <> 'canceled'` so that every metric inherits that rule. This way, we ensure no one forgets the filter. We can also have a *time spine* join macro to ensure proper date coverage (more on this in the following sections).

3. Stable consumption schemas

We dedicate a schema (or dataset) to consumption-facing objects. This schema contains only curated, safe objects, mostly views or sometimes narrow tables that are meant for end user queries. By separating it, we can control access (e.g., give analysts access to `analytics_consume` schema but not to raw Bronze or intermediate Silver schemas). It's also a clear signal: anything in consume schema is ready for use and adheres to business definitions.

Using these SQL-based approaches, we can prevent the dreaded *metric drift*. All analysts are either querying the same pre-built metric views or using the same semantic definitions. They are not writing ad-hoc SQL that might calculate a metric in a slightly different way. For example, without a Semantic layer, one analyst might calculate average order value as `SUM(total_amount)/COUNT(order_id)`, while another (mistakenly) does `AVG(total_amount)`, which could give a different result if there are cancellations or zero amounts. The Semantic layer (or our governed views) codifies exactly how to compute it so everyone does it the same way.

> **WARNING**
>
> While we define our semantic model *orders_semantic.yml* in YAML, the SQL patterns above do not automatically inherit or enforce those definitions. In a SQL-first execution model, it's still up to us to reflect the semantic intent in how we build views, joins, and macros. This is a key difference: the YAML defines intent, but without a tool like MetricFlow actually parsing and using that semantic model to generate SQL, the link between semantic definition and SQL execution is manual.

Example: Semantic joins and inferred logic

To see the Semantic layer in action, consider an analyst asking, "What is the total revenue by product category, by day?" In a traditional SQL scenario, the analyst would have to know that *product category* lives in the `dim_products` table, and they'd write a join between `fct_orders` and `dim_products` on `product_id`, then group by `dim_products.category` and date.

With our Semantic layer, the analyst could simply request Total Revenue by `Product.Category` by Day, and the system can infer the rest. Conceptually, a semantic query engine would use our definitions to generate a SQL query like the one in Example 5-26.

Example 5-26. Query generated by the Semantic layer for Total Revenue by Product Category by Day

```
SELECT
  DATE(o.created_at) AS order_date,
  p.category AS product_category,
  SUM(o.total_amount) AS total_revenue
FROM analytics.fct_orders o
JOIN analytics.dim_products p
  ON o.product_id = p.product_id
GROUP BY 1, 2;
```

Here, the engine knew that `total_revenue` = sum of `fct_orders.total_amount`, and it knew that to get product category it must join to `dim_products` (because `product_id` was declared as a foreign key to the `Product` entity). It also automatically grouped by the day of

`created_at` because our metric is time bound. The analyst didn't have to remember table names, join keys, or aggregation details—the Semantic layer handled it using the definitions we provided.

What does this buy us beyond less typing? Consistency and abstraction. In pure SQL, every analyst or downstream consumer must remember to join the right tables and apply the right filters for each metric. That's prone to error as well as inconsistent. The Semantic layer, by contrast, centralizes these definitions so that everyone uses the same logic. As a result, having a centralized metric layer leads to more trust and accuracy in data, because there is one version of each truth (one definition of Total Revenue, etc.) rather than many slightly different ones.

The dbt Semantic layer (MetricFlow)

To implement this semantic approach in our SQL workflows, we leverage dbt's Semantic layer, which is currently powered by an open source library called MetricFlow. The semantic YAML model we showed for Orders is actually part of dbt's semantic configurations. When enabled, this layer allows tools (or dbt itself) to answer metric queries like the one from Example 5-26.

But why MetricFlow? MetricFlow (and dbt's Semantic layer) exists to provide a centralized metric definition framework and automated query generation. Instead of scattering metric logic across dozens of BI reports or SQL scripts, we define metrics once (in code) and then query them on the fly. MetricFlow will parse a request for, say, `total_revenue by product_category by day`, and know which tables to join and how to aggregate, as we have seen. It effectively builds a graph of your warehouse: nodes are semantic models (Orders, Products, Customers, etc.), and it knows the connections (foreign keys) between them. Given a metric query, it finds the correct path through this graph and produces an optimized SQL query to answer it.

This approach decouples meaning from physical tables. We no longer expose *just tables* to analysts with a hope and a prayer that they understand

them. We expose metrics and entities with formal definitions. The physical tables remain important (they store the data efficiently), but the Semantic layer is an abstraction that ensures any query hitting those tables does so in an approved, correct way. It's analogous to how a REST API abstracts a database: here the Semantic layer abstracts the data warehouse.

How do we use it? In practice, as of this writing, interacting with the dbt Semantic layer can be done through dbt itself or via API integrations. For example, dbt Cloud can act as a metric server so that BI tools like Tableau or Mode can fetch metrics by name. Or an analyst could use a command-line or Python interface to query metrics. In a development environment, you might run a command or script that says, for example, compute `total_revenue` by `product.category` using Example 5-27, and MetricFlow would generate a SQL query such as "What is the total revenue by product category, by day?" and execute it on BigQuery, returning the results.

Example 5-27. Generated query by the Semantic layer for Total Revenue by Product Category by Day

```
mf query --metrics total_revenue
         --group-by product_category
```

Note that the semantic definitions shown here are real and production grade, but the execution model is presented conceptually to avoid unnecessary friction in a SQL-focused book. At the time of writing, using dbt's MetricFlow on BigQuery with dbt Core involves additional setup overhead and version compatibility constraints that may not align with a typical development environment.

Rather than digressing into resolving those environment issues, this chapter focuses on demonstrating the structure and intent of the Semantic layer without requiring the reader to run a MetricFlow server or API. The definitions are in place and could be consumed by a semantic engine, but in this chapter they are intentionally used in ways that keep us firmly in SQL.

> **NOTE**
>
> If you want to experiment with MetricFlow, you need to install the `dbt-metricflow` adapter and ensure compatible versions of dbt and Python. At the time of writing, MetricFlow was compatible with dbt Core 1.9 but not 1.10+, and required Python 3.11. As environments evolve, always consult the latest documentation before attempting integration.

The takeaway is that our project's semantic YAML is not just ornamentation: it's the single source of truth for metric logic. Whether we use a fancy engine like MetricFlow or simply use it as documentation for writing consistent SQL, the semantics are now centralized.

Semantic Contracts are Testable Contracts

Earlier, we introduced testing as a core part of the transformation flow, by covering schema validity, freshness, and null constraints. But in the Consume layer, testing takes on a deeper role. Here, we're not just validating structure—we're validating meaning. The Consume layer defines *semantic contracts*: assertions about what the data represents, not just what it looks like. These must be enforced explicitly, because semantic guarantees don't emerge automatically from SQL logic or table schemas.

This is where semantic contract tests come in. They verify that key business logic holds true, even as upstream models evolve. These tests aren't about catching nulls; they're about protecting meaning, for example:

- `total_revenue` must equal the sum of `revenue_by_channel`.
- `churn_rate` must always fall between 0 and 1.
- The *top 5 product categories by revenue* must all exist in `dim_products`.

These checks can be implemented as `dbt tests` or lightweight, scheduled validation queries. Either way, the goal is the same: make meaning auditable. As with functional tests in software engineering, these tests

reduce the risk of silent *semantic drift*, when a metric still runs but no longer means what it used to.

In a SQL-first system, testing becomes the final guardrail for semantics. Combined with clear definitions, stable views, and curated consumption schemas, semantic contract tests ensure that our consumption layer remains trustworthy and not just performant.

Operationalizing the Semantic layer

With the Semantic layer in place, there are two parallel ways to operationalize it for day-to-day use:

Leverage the semantic model in modern tools

> As tooling matures, we can connect BI tools or notebooks directly to the Semantic layer. For example, dbt's Semantic layer can act as a service that BI tools query. In that scenario, an analyst might never write `SELECT` statements for metrics at all. Instead, they drag *Total Revenue* and *Product Category* into a dashboard, and behind the scenes the semantic service generates the SQL. This is the future of consumption: a metrics-as-a-service approach. It promises users always get the correct calculation, and if we update a definition (say, "exclude refunded orders from revenue"), all reports update together.

Use semantic definitions as the source of truth for SQL

> Even without a dedicated metrics server, we can use the semantic model to drive our own SQL development. The YAML definitions are authoritative: they should inform any new data mart or report we create. For example, if a data scientist wants to create a feature for customer lifetime value, they should refer to the semantic model for how *Total Orders* or *Total Revenue* is defined, rather than writing a fresh SQL from scratch. We could even programmatically generate some views from the YAML (with a bit of scripting) to ensure they never diverge. The key point is that the Semantic layer's definitions are treated as

code: we review them, version them, and everyone on the team can rely on them, knowing they represent the latest business logic.

In practice, our Gold layer marts can be seen as materializations of semantic intent. For instance, our `fct_orders` table is a physical object that aligns with the *Orders* semantic model. If we create a new mart like `fct_customer_lifetime_value`, we would design it using existing semantic metrics (so that, say, total revenue per customer in that table uses the same sum definition). This flips the script: rather than semantics being an afterthought on top of tables, the tables are built to satisfy the semantic model's needs. Marts thus become optional performance optimizations (you might preaggregate some metrics by certain dimensions for speed) while the semantic model remains the logic source. This approach ensures that whether a metric is queried on the fly or pre-computed in a table, the definition is identical.

Live Data in Consume Layer

Apart from the Semantic layer and its whole relevance, another special challenge in the Consume layer is incorporating live or real-time data. In our architecture, new orders are continuously arriving via a streaming pipeline into a Bronze Iceberg table `bronze.live_orders`. These represent orders that have occurred since the last batch load of the warehouse. We want consumers to be able to see (near) real-time order data, but we cannot expose the raw `bronze.live_orders` directly, since:

- It's an external Iceberg table, which might contain duplicate entries (for example, if an order record was updated or if the streaming job retries, there could be multiple versions of the same order).
- The schema might be raw or not fully typed (depending on how the data lands).
- It likely lacks some of the cleansing that our batch data went through (the Silver layer casting, etc.).

The solution is to create a thin stabilization model in BigQuery as part of the Consume layer. This model reads from `bronze.live_orders`, applies minimal transformations (deduplication, type enforcement, and basic validity checks), and makes the data queryable in BigQuery's native format. We implement this as a dbt model materialized as a view, named `live_orders_stabilized` within the `gold` layer (Example 5-28).

Example 5-28. Live orders with minimal transformations and ready for consumption

```
WITH filtered AS (
  SELECT *
  FROM {{ source('bronze', 'live_orders') }}
  WHERE DATE(created_at) = CURRENT_DATE()
    AND ingest_ts >= TIMESTAMP_SUB(TIMESTAMP(CURRENT_DATE()), INTERVAL 48 HOUR)
),
ranked AS (
  SELECT
    *,
    ROW_NUMBER() OVER (PARTITION BY id ORDER BY ingest_ts DESC) AS row_num
  FROM filtered
)
SELECT
  CAST(id AS INT64) AS order_id,
  CAST(customer_id AS INT64) AS customer_id,
  CAST(product_id AS INT64) AS product_id,
  CAST(quantity AS INT64) AS quantity,
  CAST(unit_price AS NUMERIC) AS unit_price,
  CAST(total_amount AS NUMERIC) AS total_amount,
  CAST(created_at AS TIMESTAMP) AS created_at,
  CAST(ingest_ts AS TIMESTAMP) AS ingest_time
FROM ranked
WHERE row_num = 1;
```

In this model, we select from the raw Iceberg source `bronze.live_orders` (registered as a source in dbt) and use a window function to remove duplicates. The assumption is that each order has a unique ID, and if multiple entries share the same ID, we keep only the latest one (`ROW_NUMBER() = 1` ordered by ingest timestamp). We filter by `created_at` to satisfy the business definition of *today's orders*, while the `ingest_ts` guardrail bounds the scan over the external Iceberg table and tolerates late-arriving events without scanning historical partitions. We also

cast each column to the appropriate type, as BigQuery may otherwise interpret data from Iceberg using generic types.

This model is materialized as a view, not as a physical table. When dbt materializes a model as a view, it defines the SQL logic, documentation, lineage, and tests for the model, while the warehouse evaluates the query at read time. This allows the model to reflect newly ingested streaming data as it becomes queryable through the underlying Iceberg table, without depending on a dbt execution cycle.

Also, this approach provides low-latency access to streaming-ingested data, but it is not intended to deliver subsecond event-time stream processing. Freshness is bounded by the ingestion engine's commit cadence and the warehouse's visibility of new Iceberg snapshots.

`live_orders_stabilized` contains near-real-time orders in a form consistent with `fct_orders` (same columns and types). The next step is to expose it for consumption by creating a view in the `consume` layer schema (Example 5-29).

Example 5-29. Live orders BigQuery view for consumption

```
SELECT *
FROM {{ ref('live_orders_stabilized') }}
```

This view is what end users will query to get the latest orders. It simply selects all from the stabilized table. But why add a view on top of the stabilized table? Here are two reasons:

- It allows us to place the view in a different dataset (`consume`) that might have different access permissions. For example, analysts might have access to `consume.live_orders_latest` but not to the raw `bronze.live_orders` or even the intermediate stabilized table if it resides in an internal schema.
- It acts as a clear contract endpoint. We can later change how we produce `live_orders_stabilized` (or even swap out its source from Iceberg to another source) without anyone needing to point to a new table, since the view name remains the same.

With `live_orders_latest` in place, real-time consumers (like a dashboard showing today's orders, or a feature that reacts to new orders) can simply query this view. They are guaranteed a schema identical to `fct_orders` (since we selected matching columns). They don't have to worry about duplicates (we removed them) or about partial writes (Iceberg snapshots are handled by selecting only latest records, etc.). The contract is: *this view always shows a list of all new orders generated for that day, without duplicates*.

At this point, we have transformed our data journey: raw data landed in Bronze, refined through Silver, organized in Gold, and now exposed via a semantic-rich Consume layer. The key outcome is that consumers can query using business terms and stable schemas rather than raw tables and ad-hoc logic. They might interact with this layer in different ways: writing SQL against views, using a BI tool's metric interface, or hitting an API. But in all cases, the answer is governed by the definitions we've put in place.

By implementing a SQL-first Semantic layer, we ensure that as our organization grows, we don't devolve into chaos with each team defining metrics their own way. New analyses can build on existing concepts easily. If someone needs a new metric, we add it to the semantic model (and potentially materialize it if needed), and it becomes available to all.

This applies not only to batch-mode consumption but also to real-time data use cases. As shown with `live_orders_latest`, the same semantic and contract principles apply, even when the data is fresh. The stabilized view acts as a trusted interface, not a raw stream. Consumers can build real-time dashboards or features knowing that duplicates have been removed, types are enforced, and the shape of the data matches our core fact tables. By treating live data as a consumption interface problem, not a transformation pipeline, we preserve consistency across batch and streaming without fragmenting logic.

Security and Governance Considerations

Security and governance are tied into each layer of the architecture; they are not thought of or implemented separately. Throughout our current pipeline, developed in this chapter, we designed the system so that data is accessed and transformed in controlled ways, with clear boundaries and accountability. This section demonstrates how identity, schema design, and dbt features enforce security and governance as continuous, platform-embedded concerns rather than bolted-on processes.

Identity and Execution Boundaries

A key principle is separation of identities by function. Each major component of the pipeline runs under a dedicated service account with the minimum necessary permissions. For example, the dbt transformation layer uses a service account named `ch05-dbt-runner-sa`, which is granted BigQuery job execution and data editor roles. In practice, this means the dbt job can submit queries and create tables/views but nothing more. The setup from the repository explicitly binds only the needed IAM roles (Example 5-30).

Example 5-30. Setting up dbt service account

```
gcloud projects add-iam-policy-binding "$PROJECT_ID" \
  --member="serviceAccount:ch05-dbt-runner-
sa@$PROJECT_ID.iam.gserviceaccount.com" \
  --role="roles/bigquery.jobUser"

gcloud projects add-iam-policy-binding "$PROJECT_ID" \
  --member="serviceAccount:ch05-dbt-runner-
sa@$PROJECT_ID.iam.gserviceaccount.com" \
  --role="roles/bigquery.dataEditor"
```

In contrast, the ingestion jobs use different identities with more restricted scopes. The streaming ingestion Cloud Run job operates as `ch05-streaming-ingestion-sa`, which has Pub/Sub subscribe rights and permission to write objects to the landing zone bucket, but notably it has no direct BigQuery access. Similarly, the batch ingestion jobs all share a `ch05-batch-ingestion-sa` identity that can write to Cloud Storage and register new data in BigQuery's Bronze layer, but they cannot touch higher layers or

run arbitrary queries. By isolating these execution identities, we ensure that a compromise or error in one stage (for example, the ingestion code) cannot directly affect the data models or consumption queries in another stage.

The data consumption side runs with separate, read-only credentials. Analysts and BI tools do not use the dbt runner account; instead, they receive identities with permission to only read curated Gold/Consume datasets (and specific views). This identity separation enforces least privilege: the dbt runner can build tables in Silver/Gold, but it isn't used for ad-hoc dashboard queries, and analyst accounts can query curated outputs but cannot edit or deploy dbt models. This division of labor through IAM is a foundational security measure baked into how the platform operates.

Dataset and Schema Boundaries as Governance

We also enforce governance through the way data is organized into separate datasets (schemas). Each layer of the pipeline corresponds to a distinct BigQuery dataset: Bronze, Silver, Gold, and Consume are physically isolated in their own schemas. This design is not just for naming clarity; it allows us to apply tiered access control. For instance, the raw Bronze tables (containing sensitive or unrefined data) can be locked down so only data engineers or the ingestion service accounts can read them, whereas the refined Gold data can be made accessible to a wider audience (e.g., analysts or data scientists) with read-only privileges.

In our dbt project configuration, this layering is codified. All Silver models are built in the `silver` dataset, Gold models in `gold`, and so on, via schema configuration in *dbt_project.yml*, as in Example 5-31.

Example 5-31. Model configurations for dbt_project.yml

```
models:
  ch05_dbt_transformations:
    silver:
      +schema: silver      # maps to BigQuery dataset "silver"
      +materialized: view
    gold:
      +schema: gold        # maps to BigQuery dataset "gold"
      +materialized: table
```

```
  consume:
    +schema: consume     # maps to BigQuery dataset "consume"
    +materialized: table
```

This project config ensures that models in each folder are created in the intended dataset. By programmatically mapping layer → dataset, we prevent a model from accidentally landing in the wrong place (governance by design). The boundaries also enable layered permissions. BigQuery allows setting IAM policies at the dataset level, so we can grant or revoke access per layer. For example, we might give the analytics team read-only access to the `gold` dataset, while restricting any access to `bronze`. In SQL terms, you could express this as shown in Example 5-32.

Example 5-32. Grant analysts read-only access to Gold data (conceptual example)

```
GRANT SELECT ON SCHEMA `<PROJECT_ID>.gold` TO "group:analyst-
team@yourcompany.com";
```

In practice, the repository uses IAM bindings (through `gcloud` commands) rather than SQL `GRANT` statements to manage these permissions. The effect, however, is the same: Bronze, Silver, Gold, and Consume form natural governance boundaries. Each subsequent layer contains more refined data and fewer people/service accounts with access. This minimizes risk. For instance, analysts query the polished data in Gold or Consume and cannot accidentally read raw PII in Bronze. These practices also create clear ownership: the team or service responsible for a layer can be given the appropriate role on that dataset (and no more). Governance is thus baked into the physical design of our SQL platform via schema separation and permission layering.

Governance via dbt (Documentation, Lineage, and Exposures)

Beyond access control, governance is also about knowledge and accountability: knowing where data comes from, what it means, and who is using it. Our SQL transformation layer (built with dbt) embeds these governance elements as first-class artifacts.

Lineage and documentation

Every dbt model has a defined source or upstream models, and dbt automatically captures this lineage. When we run `dbt docs generate` as part of the pipeline, it produces an HTML documentation site that includes a DAG of model dependencies, as illustrated in Figure 5-2. For example, the documentation lets us trace a Gold table back to the Silver view and ultimately to the Bronze source table that feeds it. This transparency is crucial for governance. Anyone can inspect how a KPI is computed or see the chain of transformations. Moreover, we write descriptions for models and fields (via YAML or in-model docstrings). These descriptions show up in the docs site, providing context like *Orders fact table used as the primary source for revenue and order-volume analytics*. In short, the platform doesn't rely on tribal knowledge: the definitions and lineage are part of the code. As an example, the repository's Cloud Run job that runs dbt also publishes the documentation to a GCS bucket for easy access, treating documentation as a built-in deliverable of the ETL process, not a separate effort.

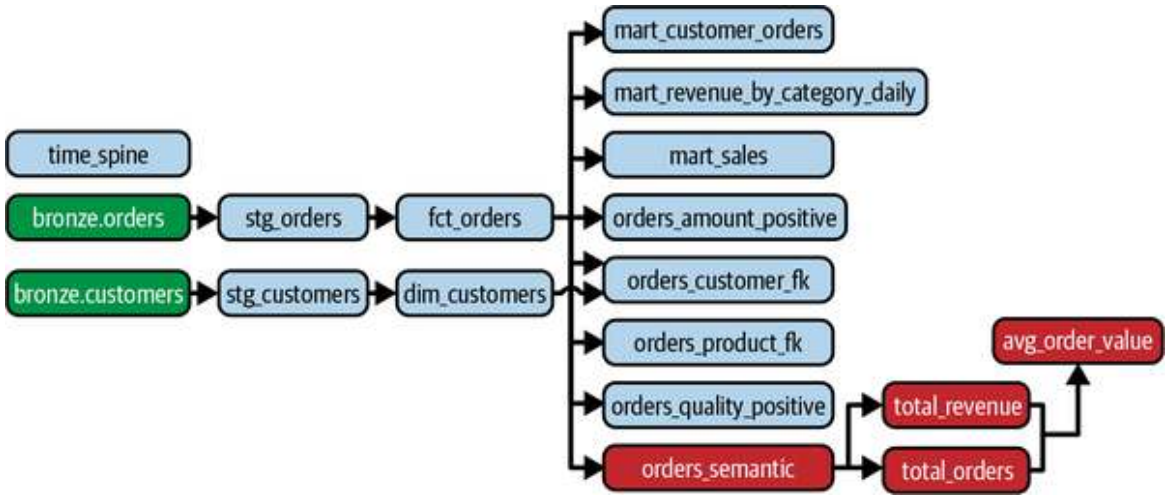


Figure 5-2. Partial lineage graph of our data pipeline

Exposures and ownership

dbt's exposures feature allows us to declare what external tools or reports depend on our data models. This is an operational governance step. By codifying, say, that a Power BI dashboard or weekly report relies on the

`gold.mart_sales` table, we create a clear link between upstream data changes and downstream stakeholders. While our repository within this chapter focuses on the data pipeline itself, you could imagine adding an exposure in the dbt project for the final dashboard. For example, we might define an exposure YAML like the one in Example 5-33.

Example 5-33. Exposure configuration in dbt (conceptual example)

```
exposures:
  - name: weekly_orders_report
    label: Executive Orders and Revenue Dashboard
    type: dashboard
    url: https://app.powerbi.com/groups/.../reports/...
    depends_on:
      - ref('mart_sales')
    owner:
      name: Data Analysts Team
      email: analyst-team@yourcompany.com
    description: >
      Executive dashboard showing weekly order volumes and revenue.
```

This declaration would appear in the docs, showing that the `mart_sales` model is used by `weekly_orders_report`. It means any change to that model can be traced to a business outcome (governance through impact analysis), and owners can be notified of changes. Even if this exact YAML isn't present in the repository, the practice is clear: we treat reports and data products as code, just like data models.

Testing and quality rules

Our dbt transformations also include tests (for example, ensuring every order has a valid customer ID, or that certain values are non-null). These tests are part of governance as well. They enforce data integrity rules and alert us to issues. In our pipeline, referential integrity tests (e.g., that `order.customer_id` exists in the customer dimension) act as automated governance checks in the pipeline. The fact that these tests run with each `dbt build` means governance is continuous: the moment a bad record appears, it's caught in the transformation step, not months later in an audit.

Overall, by using dbt we gain an audit trail of how data is produced and a built-in catalog of definitions. Governance is not a separate tool or a PDF document; it's the manifested metadata of our SQL code. Anyone with access to the project (or the published docs) can answer essential questions —What is this data? Where did it come from? Who is responsible for it?— without chasing down individuals. This level of transparency and documentation is critical in regulated industries and in general for maintaining trust in the data.

Security Versus Semantics: Consistent Definitions as Control

An often overlooked aspect of data governance is ensuring *semantic consistency*, ensuring everyone uses the same calculations and definitions for key business metrics. Our platform addresses this via the Consume layer's semantic models. Rather than having each analyst write their own SQL for metrics (which can lead to one team's *revenue* definition diverging from another's), we curate canonical SQL views and metric definitions that everyone can rely on. This acts as a form of *soft security*: it doesn't restrict access to data, but it restricts the *degrees of freedom* in interpretation, which protects the business from errors and inconsistency.

Concretely, in this chapter we implemented a semantic model for orders data (using dbt's Semantic layer features). We defined, for example, an Orders semantic model with entities, dimensions, measures, and metrics, such as an `order_count` measure and a `total_revenue` metric derived from it. By encoding these in YAML and generating a consistent view, we ensure that *Total Revenue* always means the sum of order amounts (excluding any canceled orders, if defined as such) and *Total Orders* always means count of distinct orders. The benefit is that if we ever need to change how a metric is calculated, we do it in one place. All dashboards and queries that use the semantic model will automatically reflect the update. The key is that by defining key concepts in the semantic model, everyone uses the same definition. If tomorrow we change what counts as a *completed order*, we adjust the metric definition in one place, and all

downstream consumers automatically get the updated logic. This approach greatly reduces the risk of inconsistent ad-hoc SQL creeping in, because analysts are empowered to query curated views or use predefined metrics rather than writing raw aggregations from scratch each time.

From a security perspective, providing a Semantic layer adds a safety net: even though users have access to the data, they are guided to the correct joins, filters, and calculations. It's easier to do the right thing (use the approved metrics) than to inadvertently do the wrong thing. This is complementary to access control. Even with open access to Gold data, governance is maintained because the Semantic layer ensures a single source of truth for business logic. In our SQL platform, this takes the form of dbt models in the Consume schema that join facts and dimensions or that expose metrics. For example, we could have a `consume.daily_orders_metrics` view that pulls the metrics from the Orders semantic model for each day, ready for any dashboard to use. By writing such a view once, we prevent dozens of slightly different SQL queries floating around. The semantic consistency is a form of governance that guards against the *multiple versions of the truth* problem.

Security in this context isn't only about locking down data; it's also about locking down definitions. The platform's design makes it hard to use data in unauthorized or unintended ways—not just through permissions but through a well-defined semantic framework. This ensures effective governance of both data access and data use.

Summary

This chapter brought together all the pieces to build an end-to-end data engineering solution with SQL. This journey started at data ingestion and ended with a governed, analytics-ready dataset, demonstrating how a modern pipeline can be constructed almost entirely with SQL-centric tools. The main takeaways from this chapter include:

Streaming and batch ingestion in a lakehouse

We implemented a pipeline that ingests streaming event data (orders) as well as batch data dumps (customers, products, historical orders). Raw JSON events land in a cloud storage landing zone and then get structured into Bronze tables backed by Apache Iceberg on GCS. This unified lakehouse approach allows us to handle real-time and batch sources in parallel. We used SQL engines everywhere: Google Cloud Pub/Sub and Spark wrote data to Iceberg, and BigQuery (with BigLake connectors) let us query that data using standard SQL. The important concept is that our raw data is immediately queryable in its structured form (Bronze) without complex ETL, a testament to the versatility of SQL in handling varied ingestion patterns.

SQL transformation with dbt (Silver/Gold layers)

Building on the raw base, we employed dbt to turn Bronze data into refined Silver views and Gold tables. The Silver layer applied cleansing and normalization (for example, standardizing schemas and ensuring we keep only the latest snapshot of each record), while the Gold layer organized the data into star-schema-like tables (facts and dimensions) ready for analytics. All transformation logic was expressed in SQL select statements within dbt models, proving that set-based SQL is up to the task of complex business logic. We also benefited from dbt's testing and modularity: as we created models, we added tests (ensuring data quality) and documentation. This practice reinforced that building data pipelines with SQL isn't just about writing queries, but also about engineering reproducible, well-tested data models.

Orchestration and automation

To operationalize the pipeline, we introduced cloud orchestration (Google Cloud Workflows and Cloud Scheduler) to schedule and execute each stage in order. The streaming ingestion job ran continuously, whereas batch jobs and the dbt transformation job were orchestrated to run in sequence (e.g., after new data lands, trigger the Spark job, then the dbt job). Orchestration ensured that our SQL pipeline could run reliably end to end without manual intervention.

More importantly, because each stage was containerized or defined as code, the entire workflow is reproducible. We demonstrated how to tie tasks together using GCP's workflow YAML, effectively creating a SQL-centric DAG of tasks. This highlighted that SQL pipelines benefit from the same automation and DevOps practices as code: scheduling, monitoring, and infrastructure-as-code were all part of the solution.

Consumption and the Semantic layer

At the end of the pipeline, we didn't just dump raw tables for analysts. We added a Consume layer, a semantic model and curated views that present the data in business-friendly terms. By defining metrics like `total_orders` and `total_revenue` in one place (and in code), we ensured all consumers are on the same page when they query the data. This layer acts as the interface between our engineering pipeline and the analytics consumers. It shows that SQL-based solutions can deliver a self-service analytics layer without requiring separate BI semantic servers: we used the power of SQL (through dbt and BigQuery) to implement what is essentially a semantic governance rulebook in queries. The result is that an analyst can simply `SELECT from` a `daily_orders_metrics` view or use a metric defined in the Semantic layer, confident that it's consistent with official definitions.

Embedded security and governance

Perhaps most importantly, we learned that security and governance must be embedded in every layer of the SQL pipeline. We saw that each function (ingestion, transformation, consumption) ran with principle of least privilege using dedicated service identities. We saw how dividing data into Bronze/Silver/Gold datasets creates natural checkpoints where access can be restricted or audited. And we leveraged dbt to document lineage and enforce tests, making the pipeline's workings transparent. By the time data reaches the consumer, it's not only clean and well-modeled, but it also comes with an explanation of where it came from and how it's defined. This end-to-end mindset—thinking about

governance at ingest time, at transform time, and at usage time—is what distinguishes a robust data platform from a collection of ad-hoc scripts. In short, governance is not *an afterthought*; it goes side by side with development.

Building a data engineering solution with SQL means embracing both the language and the ecosystem of tools around it. In this chapter, we showed that SQL can be the unifying thread, from data ingestion with SQL-based formats, to transformation in dbt, to analytical consumption, all while following software engineering best practices. We combined open formats (like Iceberg), cloud services, and SQL-centric frameworks to create a pipeline that is scalable, maintainable, and transparent. By walking through streaming versus batch, multi-layer modeling, orchestration, and governance, we've developed an end-to-end perspective on how modern data platforms can be built almost entirely with SQL. This sets the stage for deeper specialization: having proven the concept, teams can confidently leverage SQL for even more use cases, knowing that the principles of modular design, automation, and governance will scale along with their data.

In the next chapter, we extend this foundation into the domain of data science, exploring how modern SQL engines support statistical analysis, feature engineering, and even machine learning workflows. Having structured and governed the data platform, we now examine how SQL itself becomes an analytical execution layer.

Chapter 6. SQL in Data Science

For decades, SQL has been the basis of data management. Nearly every data professional, regardless of their specific domain, is likely to have encountered SQL as their entry point into data. Its presence is so fundamental that it often serves as the connective tissue between vast data warehouses and the analytic tools that drive business decisions, as seen in previous chapters.

However, SQL's role in data science is often perceived as limited—a mere vehicle for data extraction. Once the data is retrieved, the prevailing wisdom is to shift to more specialized tools like Python, R, or Julia for analysis, statistics, and machine learning applications. This view, while widespread, underestimates both the capabilities and the evolving potential of SQL itself.

Is this separation always necessary—or even optimal? While every tool has its strengths and intended use cases, this separation may prematurely sideline SQL's capabilities. Modern SQL engines have expanded far beyond simple `SELECT` statements. With the advent of advanced analytical functions, window operations, and even native machine learning capabilities (as seen in platforms like Google BigQuery and Snowflake), SQL is steadily encroaching on territory traditionally reserved for general-purpose programming languages, with newer capabilities being added frequently (e.g., the inclusion of LLMs as built-in functions in several data platforms).

But before relegating SQL to a supporting role in the data science workflow, it is essential to evaluate both its unique advantages and its current limitations—especially when compared to the popular data science programming languages. As the landscape of data analytics continues to evolve, so too does SQL, challenging the assumption that it should remain merely a preliminary step in larger analytical pipelines.

On the one hand, SQL offers several distinct strengths that are difficult to match:

Performance at scale

SQL engines are highly optimized for operations on massive datasets, often executing complex transformations faster than in-memory tools.

Declarative simplicity

The declarative nature of SQL allows analysts to specify what they want, not how to compute it, leading to concise and readable code.

Integration

SQL is natively integrated with most enterprise data platforms, reducing friction between data storage and computation. This also brings the compounded benefit of managing datasets for different purposes in the same place.

Visualization tooling

SQL integrates well with BI platforms, notebooks, and visualization libraries, enabling data scientists to turn query results into clear, presentation-ready visuals.

Advanced built-ins

Modern SQL dialects now offer window functions, statistical aggregates, and native support for complex analytics.

On the other hand, SQL's suitability depends on the needs and complexity of the project. For end-to-end workflows, relying solely on SQL can introduce friction, especially in procedural, iterative, or specialized contexts. Some notable limitations include:

Procedural complexity

SQL is not well-suited for procedural or iterative workflows, which can make implementing iterative algorithms—like machine learning or statistical models—very difficult.

Statistical and machine learning breadth

While SQL engines are catching up, their libraries and extensibility for statistical modeling and machine learning remain narrower than those of Python or R.

Visualization capabilities

SQL engines like BigQuery and Snowflake are progressively extending their visualization charts, but so far they are useful only for basic overview and analysis.

Package ecosystem

The vibrant open source communities and package ecosystems of Python and R enable quick prototyping and access to methods with more complex logic.

SQL isn't only an extraction step. In some workflows, it can cover exploration, feature engineering, and even modeling and scoring inside the warehouse.

Data Processing and Analysis

A key task on all data science projects is the analysis and investigation of the dataset at hand to fully grasp the subtleties of the subject under study. For this goal, several techniques can be executed to extract information and provide insights. Before jumping into more advanced analysis methods, several quick insights are easily gathered from the dataset that can serve as guidelines on posterior analysis.

Dataset Statistical Description

Covering a range of basic statistics, a dataset can be quickly analyzed through the combination of common native functions like `AVG` or `COUNT` to understand the distribution of data. This query executes calculations similar to other methods just like the `pandas.DataFrame.describe` method in Python.

For this example, we use a sample weather dataset (`oreilly.weather`) containing temperature, humidity, pressure, and rainfall measurements. The setup script to create this table is available in the chapter's companion code.

The first step is to transform the dataset from a wide format—where each variable has its own column—to a long format where each value appears alongside its variable name. This `UNPIVOT` operation (Example 6-1) makes subsequent calculations more manageable and scalable. The `stats` CTE then computes basic descriptive statistics for each variable: count of non-null values, count of nulls, mean, standard deviation, variance, minimum, maximum, and median using an approximate quantile method suitable for large datasets.

Example 6-1. Unpivot and compute basic statistics

```
WITH unpivoted AS (
  SELECT *
  FROM `oreilly.weather`
  UNPIVOT (
    value FOR variable IN (
      Temp9am, Temp3pm, Humidity9am, Pressure9am, Pressure3pm, Rainfall
    )
  )
),
stats AS (
  SELECT
    variable,
    COUNT(value) AS n,
    COUNT(*) - COUNT(value) AS NumNulls,
    AVG(value) AS avg,
    STDDEV(value) AS stddev,
    VARIANCE(value) AS variance,
    MIN(value) AS min,
    MAX(value) AS max,
    APPROX_QUANTILES(value, 2)[OFFSET(1)] AS median
  FROM unpivoted
```

```
  GROUP BY variable
)
SELECT * FROM stats;
```

Building on this foundation, we add higher-order statistics—skewness and kurtosis—to understand the shape and symmetry of each distribution (Example 6-2). *Skewness* measures how much the distribution deviates from symmetry, while *kurtosis* reflects the "tailedness" or concentration of values. Both are calculated using central moments: differences between each value and the mean, normalized by the standard deviation and variance. Some newer engines expose these directly as aggregate functions; for example, Presto provides `skewness(x)` and `kurtosis(x)`, which can simplify this type of statistical analysis.

Example 6-2. Add skewness and kurtosis calculations

```
-- Continuing from the previous CTEs (unpivoted, stats)
moment_calcs AS (
  SELECT
    u.variable,
    POWER(SUM(POWER(value - s.avg, 3)) / s.n, 1) / POWER(s.stddev, 3) AS
skewness,
    POWER(SUM(POWER(value - s.avg, 4)) / s.n, 1) / POWER(s.variance, 2) AS
kurtosis
  FROM unpivoted u
  JOIN stats s USING (variable)
  WHERE value IS NOT NULL
  GROUP BY u.variable, s.avg, s.stddev, s.variance, s.n
)
SELECT
  s.variable,
  s.n AS NumRecords,
  s.NumNulls,
  s.min AS Min,
  s.median AS Median,
  ROUND(s.avg, 2) AS Average,
  s.max AS Max,
  ROUND(s.stddev, 2) AS StdDev,
  ROUND(s.variance, 3) AS Variance,
  ROUND(m.skewness, 3) AS Skewness,
  ROUND(m.kurtosis, 3) AS Kurtosis
FROM stats s
JOIN moment_calcs m USING (variable)
ORDER BY variable;
```

The final `SELECT` combines basic statistics with moment calculations into a single summary table. The output (Table 6-1) presents each variable with its record count, null count, minimum, median, average, maximum, standard deviation, variance, skewness, and kurtosis—all rounded for readability. The results have been transposed here for easier reading.

Table 6-1. Example output from dataset statistics query (transposed)

Statistic	Humidity9am	Pressure3pm	Pressure9am	Ra
NumRecords	12	11	11	12
NumNulls	0	1	1	0
Min	48	1005.8	1009.5	0.0
Median	68	1014.1	1016.9	1.6
Average	69.33	1013.65	1016.32	3.2
Max	88	1019.8	1022.1	12
StdDev	12.14	4.21	3.78	4.0
Variance	147.33	17.76	14.32	16
Skewness	-0.156	0.124	-0.089	1.2
Kurtosis	2.089	2.145	2.312	3.5

From this output, we can extract several key insights about each weather variable:

Humidity9am

Has no missing values, with values well spread between 48% and 88%. The slight negative skewness (-0.156) indicates a nearly symmetric distribution with a minor tail toward lower humidity values. The kurtosis of 2.089 (below 3) suggests a slightly platykurtic distribution—flatter than a normal distribution.

Pressure variables

Both pressure readings show minimal skewness and kurtosis values near 2, indicating fairly symmetric, slightly flat distributions. The 3 p.m. pressure readings have one missing value and show slightly more variability.

Rainfall

This variable shows the most pronounced skewness (1.245), indicating a right-skewed distribution—typical for rainfall data where most days have little or no rain, but some days have heavy precipitation. The kurtosis of 3.584 (above 3) indicates heavier tails, meaning occasional extreme values.

Temperature variables

Both morning and afternoon temperatures show mild skewness and low kurtosis, suggesting relatively normal distributions. The 3 p.m. temperatures are consistently higher (average 24.46°C compared to 17.49°C at 9 a.m.) as expected.

By leveraging the `UNPIVOT` operation and calculated moments, this approach remains highly maintainable and can be easily extended to additional variables or statistics, avoiding repetitive and unwieldy SQL code.

Similarly, this approach can be extended and employ other methods such as `MODE` and `COUNT(DISTINCT)`, among others, to expand the analysis to account for categorical variables.

> **TIP**
>
> It's also possible to use platform-specific functions for summary statistics. For example, BigQuery provides `ML.FEATURE_INFO` on BigQuery, among other tools.

Correlation and Association

Pearson correlation quantifies linear association between two numeric variables and is available as `CORR(x, y)`, as seen in Example 6-3. *Spearman correlation* targets monotonic association: it measures whether one variable tends to increase (or decrease) as the other increases, without assuming a linear relationship.

Example 6-3. Pearson correlation

```
SELECT
  CORR(flipper_length_mm, body_mass_g) AS r
FROM `bigquery-public-data.ml_datasets.penguins`
WHERE flipper_length_mm IS NOT NULL
  AND body_mass_g IS NOT NULL;
```

This returns a single r value between −1 and +1. Values near 0 indicate weak linear association; values near ±1 indicate strong association, with direction given by the sign. Spearman correlation is computed as the Pearson correlation of the ranks and is less sensitive to outliers and nonlinear but monotonic relationships, as demonstrated in Example 6-4. As a general (but not absolute) rule, if Spearman is high while Pearson is modest, the relationship is likely monotonic but nonlinear; if both are high with the same sign, the relationship is strongly linear.

Example 6-4. Spearman correlation

```
WITH ranked AS (
  SELECT
    RANK() OVER (ORDER BY flipper_length_mm) AS rx,
```

```
    RANK() OVER (ORDER BY body_mass_g) AS ry
  FROM `bigquery-public-data.ml_datasets.penguins`
  WHERE flipper_length_mm IS NOT NULL
  AND body_mass_g IS NOT NULL
)
SELECT CORR(rx, ry) AS spearman_r
FROM ranked;
```

This approach is portable and robust. Use it when distributions are skewed, have heavy tails, or contain outliers that would distort Pearson. If this correlation is not available, the next section shows how to derive the same quantities from standard aggregates.

Regression from Aggregates

Sometimes only summary statistics are available rather than row-level detailed data. In a simple linear analysis, we can still recover a quick line of best fit—slope, intercept, and R^2 from aggregate statistics: the means of x and y, the (population) variance of x, the (population) covariance of x and y, and their correlation. These match the ordinary least-squares solution for simple linear regression when the aggregates are computed on the same records. Using population (`<method>_POP`) versus sample versions changes only small finite-sample constants. A quick linear fit can be recovered from aggregates by simply leveraging correlation `CORR`, covariance `COVAR_POP`, and variance `VAR_POP` of the population from the CTE *s* to derive *slope*, *intercept*, and r^2 as seen in Example 6-5.

Example 6-5. Regression-lite from aggregates

```
WITH s AS (
  SELECT
    AVG(weight_pounds)                        AS avg_y,
    AVG(gestation_weeks)                      AS avg_x,
    COVAR_POP(weight_pounds, gestation_weeks) AS cov,
    VAR_POP(gestation_weeks)                  AS varx,
    CORR(weight_pounds, gestation_weeks)      AS r
  FROM `bigquery-public-data.samples.natality`
  WHERE weight_pounds IS NOT NULL
    AND gestation_weeks IS NOT NULL
    -- Trim implausible values to limit the effect of extremes
    AND weight_pounds BETWEEN 3 AND 14
    AND gestation_weeks BETWEEN 30 AND 45
```

```
)
SELECT
  cov / varx                          AS slope,
  avg_y - (cov / varx) * avg_x        AS intercept,
  POW(r, 2)                           AS r2
FROM s;
```

This query estimates the relationship between birth weight (`weight_pounds`) and gestational age (`gestation_weeks`) in BigQuery. We trim extreme values to reduce the influence of extreme observations, which can distort aggregate measures.

> **TIP**
>
> `CORR` and ranking functions exist across major warehouses. Systems like SnowFlake directly expose regression aggregates such as `REGR_SLOPE` and `REGR_R2` for quick linear summaries.

Outlier Detection

Relevant to both the descriptive/diagnostic component of a data science project and the dataset curation for machine learning tasks, outlier detection and correction have a strong emphasis on avoiding data distortions, particularly in training and evaluation on ML.

One way to detect these extreme values is to compute the z-score and establish a threshold such as 3 standard deviations to flag outliers as seen in Example 6-6. In this case, the dataset is being filtered to values within the interval of 3 standard deviations, which accounts for 99.7% of observations on normal distributions.

Example 6-6. Outlier detection using z-score

```
WITH stats AS (
  SELECT
    AVG(tripduration) AS mean_duration,
    STDDEV(tripduration) AS stddev_duration
  FROM
    `bigquery-public-data.new_york_citibike.citibike_trips`
)
```

```
SELECT
  tripduration,
  (tripduration - mean_duration) / stddev_duration AS z_score
FROM
  `bigquery-public-data.new_york_citibike.citibike_trips`, stats
WHERE
  ABS((tripduration - mean_duration) / stddev_duration) > 3
LIMIT 20;
```

Another way to perform this is to select intervals of data within specific percentiles (Example 6-7). This tends to be more robust than z-score for cases of skewed data due to the distortion of the mean and standard deviation.

Example 6-7. Outlier detection using percentiles

```
WITH percentiles AS (
  SELECT
    APPROX_QUANTILES(tripduration, 100)[OFFSET(1)] AS p1,
    APPROX_QUANTILES(tripduration, 100)[OFFSET(99)] AS p99
  FROM
    `bigquery-public-data.new_york_citibike.citibike_trips`
)
SELECT
  *
FROM
  `bigquery-public-data.new_york_citibike.citibike_trips`, percentiles
WHERE
  tripduration < p1 OR tripduration > p99
LIMIT 20;
```

> **WARNING**
>
> `APPROX_QUANTILES` is an approximate BigQuery function. Though scalable and memory efficient, as is appropriate for large datasets, it does not compute the exact results. Consider using `PERCENTILE_DISC` or `PERCENTILE_CONT` for exact calculations, even though this function is more compute demanding.

Advanced Statistical Analysis

In modern data-driven organizations, statistical analysis is no longer confined to specialized tools or programming languages—it is increasingly

performed directly where the data lives: in SQL databases. As cloud data warehouses such as Google BigQuery, Snowflake, and others continue to power business analytics at scale, the ability to conduct advanced statistical tests, evaluate distribution assumptions, and calculate confidence intervals within SQL unlocks new agility and reproducibility for data science and analytics teams.

Traditionally, validating statistical assumptions or running hypothesis tests required exporting datasets into Python, R, or other statistical software. Today, however, mature SQL dialects and extensible user-defined functions (UDFs) bring much of this capability into the database layer itself. This not only reduces friction and error-prone data movement, but it also makes robust statistical reasoning accessible to a wider audience of analysts and decision-makers. On BigQuery, these can be specific SQL instructions aiming to address specific statistical needs, or small JavaScript/Python code to execute specific attributes, such as calculating a test statistic. On Snowflake, a user can also apply Python/Java/Scala code via a UDF through Snowflake Snowpark.

A practical challenge when using SQL for statistical analysis is that SQL engines do not come with preloaded probability distributions. Most continuous distributions (t, F, chi-squared, etc.) require numerical integration to compute p-values or critical values—iterative algorithms that are difficult to implement in set-based SQL. This is why helper functions like BigQuery's `bqutil.fn.t_test()` are valuable: they encapsulate these complex calculations. For distributions without such helper functions, analysts often resort to lookup tables, external tools, or approximate methods.

The same caveat applies to simulation-heavy methods such as Monte Carlo analysis. While modern warehouses can support this style of work through random-number generation and set expansion, it is not a natural fit for SQL because repeated simulation and procedural control flow are more naturally expressed in Python or R.

Normality Testing

Many ML algorithms and statistical procedures assume that data follows a normal distribution. Before applying techniques like linear regression, t-tests, or certain clustering algorithms, data scientists need to verify this assumption. The Kolmogorov-Smirnov (KS) test provides a rigorous way to determine this directly within SQL, eliminating the need to export data to specialized statistical software.

The KS test (Example 6-8) compares the empirical distribution of the data to a theoretical normal distribution fitted to the same data. Other methods exist, but as a rule:

Jarque–Bera

: Quick check for shape problems in large samples

Kolmogorov–Smirnov

: General-purpose tool to compare full distributions

Shapiro–Wilk

: Gold standard for normality in small/medium datasets—very sensitive and powerful

Notice that the tests are not whether the data matches some predetermined normal distribution, but rather whether it follows the normal distribution that best fits the observed data. This approach makes the test both fair to the normality assumption and more sensitive to detecting genuine departures from normal behavior.

Example 6-8. Kolmogorov–Smirnov test

```
WITH base AS (
  SELECT mother_age AS x
  FROM `bigquery-public-data.samples.natality`
  WHERE mother_age IS NOT NULL
  LIMIT 10000  -- Optional: limit for performance
),
fit AS (
  SELECT AVG(x) AS mu, STDDEV_SAMP(x) AS sigma, COUNT(*) AS n FROM base
),
```

```
ecdf AS (
  SELECT
    x,
    ROW_NUMBER() OVER (ORDER BY x) * 1.0 / COUNT(*) OVER () AS F_emp
  FROM base
),
theory AS (
  SELECT
    e.x,
    e.F_emp,
    bqutil.fn.normal_cdf(e.x, f.mu, f.sigma) AS F_norm
  FROM ecdf e CROSS JOIN fit f
)
SELECT
  MAX(ABS(F_emp - F_norm)) AS D_stat
FROM theory;
```

The process begins by selecting a sample of mother ages from the dataset, removing any missing values and limiting the size of the sample for performance reasons. This selected data forms the foundation for the analysis.

Next, the query calculates two key summary statistics from the data: the mean and the standard deviation. These serve as the parameters for the normal distribution we're going to compare against. This fitted normal distribution reflects what the data would look like if it followed a perfectly normal bell curve centered around the observed average.

Then the query constructs what's called the *empirical cumulative distribution function*, or ECDF. For each age in the dataset, the ECDF tells us what proportion of the sample is less than or equal to that age. This function gives a stepwise representation of the actual data distribution.

In parallel, the query calculates the theoretical cumulative distribution function, or CDF, for the normal distribution using the same ages. For each value, it determines what proportion of a perfect normal distribution (based on the earlier mean and standard deviation) would fall below that point. This represents what we'd expect if the data were perfectly normal.

The core of the KS test lies in comparing these two distributions—the empirical and the theoretical—point by point. For each value in the sample,

it measures the absolute difference between the empirical and theoretical CDFs. Finally, it identifies the maximum of these differences. This maximum gap is known as the KS D-statistic.

The result—the *D-statistic*—quantifies the largest deviation between what the data actually looks like and what we'd expect under a normal distribution. Data scientists typically compare this D-statistic to critical values from statistical tables to make formal decisions about normality, but even without formal hypothesis testing, the magnitude of the D-statistic provides valuable intuition about how well the normal assumption fits the data at hand.

Confidence Intervals: Estimating the Mean

Confidence intervals express the uncertainty around a sample estimate. When the mean of a numeric variable is computed, the sample average is a point estimate, and the confidence interval adds a margin of error around it. In frequentist terms, a 95% confidence interval means that if we repeatedly drew similar samples and constructed the interval in the same way, about 95% of those intervals would contain the true population mean. For a single computed interval, the practical takeaway is that it provides a range of plausible values for the population mean given the data and method used. Wider intervals indicate more uncertainty; narrower ones suggest a more precise estimate.

For example, if a sample mean is `10.0` and the 95% confidence interval is `[9.5, 10.5]`, then `10.0` is the best point estimate and the interval reflects the uncertainty around that estimate. It should be read as a margin of uncertainty around the sampling process, not as a literal 95% probability statement about this one fixed interval.

The following SQL pattern (Example 6-9) illustrates this calculation. For each year in the dataset, it computes the mean, sample standard deviation, and count and then derives the confidence bounds accordingly.

Example 6-9. Confidence intervals for normal approximation, 95% confidence level

```sql
SELECT
  year AS grp,
  AVG(mother_age) AS mean_age,
  STDDEV_SAMP(mother_age) AS s,
  COUNT(*) AS n,
  -- Replace [z] with desired z/t critical value, e.g., 1.96 for ~95% CI
  AVG(mother_age) - 1.96 * STDDEV_SAMP(mother_age) / SQRT(COUNT(*)) AS ci_lower,
  AVG(mother_age) + 1.96 * STDDEV_SAMP(mother_age) / SQRT(COUNT(*)) AS ci_upper
FROM `bigquery-public-data.samples.natality`
WHERE mother_age IS NOT NULL
GROUP BY year
ORDER BY year;
```

In this example, *z* represents a critical value chosen according to the desired confidence level. For instance, different confidence levels—such as 90%, 95%, or 99%—correspond to different values; in this case for 95% there is `z=1.96`. These can be inserted directly or passed in as parameters if the confidence level is configurable.

The output (Table 6-2) shows the confidence intervals for mother's age across different years.

Table 6-2. Example output from confidence interval query

grp	mean_age	s	n	ci_lo
1969	24.737	5.641	1800103	24.72
1970	24.615	5.541	1868900	24.60
1971	24.560	5.482	1781774	24.55
1972	24.441	5.498	1749402	24.43
1973	24.414	5.433	1839736	24.40
1974	24.411	5.333	2029150	24.40

Looking at the results, we observe that the mean mother's age decreased slightly from 1969 to 1974 (from ~24.7 to ~24.4 years). The confidence intervals are extremely narrow (typically ±0.008 years) due to the large sample sizes (1.7–2 million records per year). This precision means we can be highly confident that the observed year-over-year differences are statistically significant, and not due to sampling variability.

Two-Sample Tests

Two-sample tests are fundamental statistical tools for comparing a numeric metric between two distinct groups. Whether we are analyzing conversion rates between A/B test variants, comparing user engagement across different customer segments, or evaluating performance metrics between regions, these tests help determine if observed differences are statistically significant or could reasonably be attributed to random variation.

The core question these tests address is: “Are the distributions of our metric meaningfully different between the two groups?” Two essential approaches are the parametric *t-test* for normally distributed data with similar variances, and the non-parametric *Mann–Whitney U* test for data that doesn’t meet these assumptions.

Two-sample t-test

The two-sample t-test (Example 6-10) compares the means of two independent groups to determine if they differ significantly. It assumes both groups follow approximately normal distributions with similar variances, making it most suitable when: our data is approximately normally distributed, the two groups have similar variances, we want to compare means specifically, and sample sizes are reasonably large (typically $n > 30$ per group to invoke the Central Limit Theorem).

The test calculates a *t-statistic* that measures how many standard errors the difference in means is away from zero. A larger magnitude *t-value* indicates a larger standardized difference between group means, providing a standardized way to assess whether the observed difference is meaningful or could easily be due to random sampling variation.

This query demonstrates an application of the two-sample t-test using Chicago crime data. The analysis compares daily crime counts between January and July 2022 to determine if there is a statistically significant difference between these two months.

Example 6-10. Two-sample t-test

```
WITH daily_counts AS (
  SELECT
    DATE(date) AS crime_day,
    CAST(COUNT(*) AS FLOAT64) AS crimes,
    EXTRACT(MONTH FROM date) AS crime_month
  FROM `bigquery-public-data.chicago_crime.crime`
  WHERE
    DATE(date) BETWEEN '2022-01-01' AND '2022-01-31'
    OR DATE(date) BETWEEN '2022-07-01' AND '2022-07-31'
  GROUP BY crime_day, crime_month
),
samples AS (
```

```
  SELECT
    ARRAY_AGG(CASE WHEN crime_month = 1 THEN crimes END) AS jan_crimes,
    ARRAY_AGG(CASE WHEN crime_month = 7 THEN crimes END) AS jul_crimes
  FROM daily_counts
)
SELECT
  (SELECT AVG(x) FROM UNNEST(jan_crimes) x) AS jan_mean_daily_crimes,
  (SELECT AVG(x) FROM UNNEST(jul_crimes) x) AS jul_mean_daily_crimes,
  bqutil.fn.t_test(jan_crimes, jul_crimes).t_value AS t_statistic,
  bqutil.fn.t_test(jan_crimes, jul_crimes).dof AS degrees_of_freedom
FROM samples;
```

The query structure follows a logical three-step approach. The `daily_counts` CTE first aggregates the raw crime data by calculating the total number of crimes for each day, while also extracting the month for later grouping. The date filtering restricts the analysis to January 2022 (month 1) and July 2022 (month 7), creating two distinct groups for comparison. Converting the count to `FLOAT64` ensures compatibility with the statistical functions.

The `samples` CTE then transforms this daily data into the array format required by the *t-test* function, through the usage of `ARRAY_AGG`. Using conditional aggregation with `CASE WHEN`, it creates two separate arrays: `jan_crimes` containing daily crime counts for January, and `jul_crimes` containing daily crime counts for July.

The final `SELECT` statement computes the mean daily crime count for each month using scalar subqueries over the arrays, then applies the *t-test* function to compare them. The `bqutil.fn.t_test()` function returns a struct containing statistical results, from which the query extracts the t-statistic and degrees of freedom. Including the mean values alongside the test statistics provides essential context: the t-statistic indicates whether the difference is statistically significant, while the means reveal the direction and magnitude of that difference.

The output (Table 6-3) shows the comparison between January and July 2022.

Table 6-3. Example output from t-test query

jan_mean_daily_crimes	jul_mean_daily_crimes	t_statistic
508.194	721.065	10.740

The results reveal that July averages approximately 213 more crimes per day than January (721 versus 508). If the absolute value of the calculated t-statistic is greater than the critical value at a given significance level (e.g., 5%), we reject the null hypothesis and conclude that the difference is statistically significant. In this case, the t-statistic of 10.74 with 60 degrees of freedom far exceeds the critical value of ~2.0 for a two-sided test at the 95% confidence level, confirming that the difference in Chicago crime rates between these two months is statistically significant and not due to random variation.

> **NOTE**
>
> Count data is inherently nonnormal—counts are discrete and nonnegative, typically following a Poisson distribution. However, for a large city like Chicago with high daily crime counts, the normal approximation is reasonable due to the Central Limit Theorem. Be cautious when applying this approach to smaller datasets (e.g., crime counts in a small town), where the nonnormality of counts becomes more pronounced and a non-parametric test like Mann–Whitney U would be more appropriate.

Mann–Whitney U

The Mann–Whitney U test (Example 6-11)—also known as the Wilcoxon rank-sum test—is a non-parametric alternative for comparing two independent groups. *Non-parametric* means the test does not assume any specific distributional form for the data—not just normality, but any parametric distribution (normal, exponential, Poisson, etc.). Instead of comparing means, it evaluates whether one group tends to have

systematically higher or lower values than the other by ranking all observations and comparing rank sums.

This test becomes the preferred choice when:

- Working with data that is skewed, heavy-tailed, or contains outliers
- Dealing with ordinal data
- Sample sizes are small
- The underlying distribution is unknown or cannot be assumed
- Comparing entire distributions rather than just means

The robustness to distributional assumptions makes it particularly valuable for real-world business data, which rarely follows textbook distributions.

Example 6-11. Mann–Whitney U

```
WITH samples AS (
  SELECT
    ARRAY_AGG(CASE WHEN sex = 'MALE' THEN body_mass_g END IGNORE NULLS)
      AS male_mass,
    ARRAY_AGG(CASE WHEN sex = 'FEMALE' THEN body_mass_g END IGNORE NULLS)
      AS female_mass
  FROM `bigquery-public-data.ml_datasets.penguins`
  WHERE
    sex IN ('MALE', 'FEMALE')
    AND body_mass_g IS NOT NULL
)
SELECT
  bqutil.fn.mannwhitneyu(male_mass, female_mass, 'two-sided').U AS u_statistic,
  bqutil.fn.mannwhitneyu(male_mass, female_mass, 'two-sided').p AS p_value
FROM samples;
```

This query applies the Mann–Whitney U test to compare body mass distributions between male and female penguins in the Palmer Penguins dataset. The analysis determines whether there are statistically significant differences in body mass between the sexes without assuming normal distributions.

The `samples` CTE transforms the penguin data into the array format required by the `mannwhitneyu` function using conditional aggregation. The

`CASE WHEN` statements separate the body mass data by `sex`: `male_mass` contains body mass values for male penguins, while `female_mass` contains values for female penguins. Converting the body mass to `FLOAT64` ensures compatibility with the statistical functions.

The `WHERE` clause filters the data to include only records with explicitly defined `sex` values (`MALE` or `FEMALE`) and non-null body mass measurements, ensuring clean input data for the statistical test. This filtering prevents any ambiguous or missing data from affecting the analysis.

Lastly, the Mann–Whitney U test is applied with a two-sided alternative hypothesis, which tests whether the distributions differ in either direction rather than testing for a specific directional difference. The function returns a `STRUCT` containing the test results—a `STRUCT` in SQL is a composite data type that groups multiple named fields, similar to a record or object in other languages. From this struct, the query extracts both the `U statistic` and the `p-value`. By comparing the `p-value` with the significance level α (which is 1—Confidence level), there are two possible decisions:

`p-value` $> \alpha$

> The null hypothesis is not rejected and therefore there is not enough evidence to conclude that the distributions are different.

`p-value` $< \alpha$

> The null hypothesis is rejected as it is suggested that the distributions of the two groups are statistically different.

Multiple Groups Testing

When analysis extends beyond comparing two groups, different statistical approaches become necessary. Multiple group comparisons require tests that can handle the increased complexity while controlling for the higher probability of false discoveries that comes with multiple comparisons.

Similarly, categorical data relationships demand specialized tests designed for contingency table analysis.

Two techniques matter here: one-way *analysis of variance* (*ANOVA*) for comparing means across multiple groups when normality assumptions are met, and the Kruskal—Wallis test as the non-parametric alternative for comparing distributions across multiple groups.

One-way ANOVA: A handmade analysis

One-way ANOVA tests whether the means of three or more independent groups are statistically equal. It assumes that data within each group follows a normal distribution with equal variances across groups. ANOVA partitions the total variance in the data into variance between groups and variance within groups, then compares these components using an F-statistic.

This test becomes the standard choice when comparing means across multiple treatment conditions with normally distributed data, analyzing performance metrics across several regions or time periods with adequate sample sizes, evaluating standardized test scores across different schools or programs, or when the assumptions of normality and equal variances are reasonably met. The test determines whether significant differences exist among group means, though like Kruskal—Wallis, it does not identify which specific groups differ.

ANOVA represents one of the most fundamental statistical procedures for comparing means across multiple groups, yet implementing it directly in SQL requires careful construction of the underlying mathematical components. Unlike many statistical functions that are provided either via built-in functions or commonly accepted UDFs, ANOVA demands a manual approach that, while more complex, offers complete transparency and control over each calculation step.

This implementation demonstrates a complete one-way ANOVA analysis using birth weight data from public health records, examining whether significant differences exist across racial groups. The next examples build

the ANOVA query in stages—data preparation, descriptive statistics, overall statistics, and the final F-test—ending in a single query that performs the full analysis.

Before starting with ANOVA, be aware that personalized UDFs on BigQuery can only be created on datasets belonging to the user. As such, the recommendation is to make a copy of a public dataset to a local destination to fully take advantage of these capabilities (Example 6-12).

Example 6-12. One-way ANOVA: migration of public dataset to local environment

```
CREATE OR REPLACE TABLE oreilly.natality_sample AS
SELECT
  mother_race,
  weight_pounds,
  year
FROM `bigquery-public-data.samples.natality`
WHERE weight_pounds IS NOT NULL
    AND mother_race IS NOT NULL;
```

The foundation of any ANOVA analysis begins with proper data preparation and filtering. The initial data extraction (Example 6-13) establishes the analytical dataset by selecting relevant variables and applying quality controls that ensure statistical validity.

Example 6-13. One-way ANOVA: data preparation

```
-- Step 1: Prepare sample data from public natality dataset
WITH birth_data AS (
  SELECT
    mother_race,
    weight_pounds
  FROM `oreilly.natality_sample`
  WHERE weight_pounds IS NOT NULL
    AND weight_pounds > 0
    AND mother_race IN (1, 2, 3)  -- White, Black, American Indian
    AND year = 2008
  LIMIT 1000  -- Keep sample size manageable
),
```

This preparation phase extracts only the essential variables required for the analysis: the grouping variable `mother_race` and the dependent variable `weight_pounds`. The filtering conditions serve multiple statistical purposes.

Removing null values ensures complete cases for analysis, while the constraint `weight_pounds > 0` eliminates biologically impossible values that could skew results. Restricting the analysis to three specific racial groups creates a focused comparison, while limiting to 2008 data maintains temporal consistency. The sample size limitation serves practical purposes for demonstration, though production analyses would typically use the full dataset.

Data interpretation requires converting numeric codes into meaningful labels, transforming abstract database values into comprehensible analytical categories. This recoding step (Example 6-14) enhances both analysis clarity and result presentation.

Example 6-14. One-way ANOVA: data interpretation

```
-- Step 3: Map race codes to readable names
coded_data AS (
  SELECT
    CASE mother_race
      WHEN 1 THEN 'White'
      WHEN 2 THEN 'Black'
      WHEN 3 THEN 'American_Indian'
    END as race_group,
    weight_pounds
  FROM birth_data
),
```

The `CASE` statement transforms numeric race codes into descriptive labels, eliminating potential confusion about what each code represents. This transformation becomes particularly important when presenting results to stakeholders who may not be familiar with the database's coding scheme. The resulting `race_group` variable provides immediate interpretability while maintaining the analytical structure necessary for subsequent calculations.

Next, the group-level statistics (Example 6-15) capture the essential characteristics needed to quantify both within-group and between-group variation.

Example 6-15. One-way ANOVA: descriptive statistics

```
-- Step 3: Calculate group statistics
group_stats AS (
  SELECT
    race_group,
    COUNT(weight_pounds) as n,
    AVG(weight_pounds) as group_mean,
    VAR_POP(weight_pounds) as group_variance
  FROM coded_data
  GROUP BY race_group
),
```

Each group's descriptive statistics serve specific roles in ANOVA calculations. The sample size `n` determines the weight each group contributes to overall calculations and affects degrees of freedom. The `group_mean` provides the central values that will be compared statistically, while `group_variance` quantifies the spread within each group. Using `VAR_POP` rather than `VAR_SAMP` maintains consistency with the mathematical formulation of ANOVA, treating each group as a complete population for variance calculation purposes.

Overall dataset characteristics complement group-specific statistics by providing reference points for measuring between-group differences. These overall statistics (Example 6-16) establish the analytical framework within which group comparisons occur.

Example 6-16. One-way ANOVA: overall statistics

```
-- Step 4: Calculate overall statistics
overall_stats AS (
  SELECT
    COUNT(weight_pounds) as total_n,
    AVG(weight_pounds) as grand_mean,
    COUNT(DISTINCT race_group) as num_groups
  FROM coded_data
),
```

The total sample size `total_n` determines overall degrees of freedom, while the `grand_mean` serves as the reference point for measuring how much each group mean deviates from the overall average. The number of groups (`num_groups`) directly affects the degrees of freedom calculations

and determines the structure of the F-distribution used for significance testing.

The last auxiliary step is actually the mathematical core of ANOVA, involving partitioning total variation into components attributable to differences between groups and variation within groups. This split (Example 6-17) implements the classic ANOVA formulation through careful calculation of *sums of squares*, *degrees of freedom*, and *mean squares*.

Example 6-17. One-way ANOVA: core components

```
-- Step 5: Calculate ANOVA components
anova_calc AS (
  SELECT
    -- Degrees of freedom
    os.num_groups - 1 as df_between,
    os.total_n - os.num_groups as df_within,

    -- Sum of Squares Between Groups
    SUM(gs.n * POWER(gs.group_mean - os.grand_mean,2)) as ssb,

    -- Sum of Squares Within Groups
    SUM((gs.n - 1) * gs.group_variance) as ssw,

    -- Mean Squares
    SUM(gs.n * POWER(gs.group_mean - os.grand_mean,2)) / (os.num_groups - 1) as
msb,
    SUM((gs.n - 1) * gs.group_variance) / (os.total_n - os.num_groups) as msw

  FROM group_stats gs
  CROSS JOIN overall_stats os
  GROUP BY os.num_groups, os.total_n, os.grand_mean
)
```

This calculation implements the fundamental ANOVA decomposition. *Degrees of freedom* follow standard formulations: between-group degrees of freedom equal the number of groups minus one, while within-group degrees of freedom equal the total sample size minus the number of groups. The *sum of squares between groups* (`ssb`) measures how much group means deviate from the grand mean, weighted by each group's sample size. The *sum of squares within groups* (`ssw`) aggregates the variation within each

group, representing unexplained or error variation. *Mean squares* divide each sum of squares by its corresponding degrees of freedom, creating standardized measures of variation that enable meaningful comparison through the F-statistic.

To achieve the final ANOVA results and present them in a format similar to a traditional reference table, the data is ultimately transformed into the last SQL snippet (Example 6-18).

Example 6-18. One-way ANOVA: final results

```
-- Final Results: Group summaries and ANOVA table
SELECT 'GROUP_SUMMARY' as result_type, race_group as category,
  CAST(n as STRING) as value_1,
  CAST(ROUND(group_mean, 3) as STRING) as value_2,
  CAST(ROUND(SQRT(group_variance), 3) as STRING) as value_3
FROM group_stats

UNION ALL

SELECT 'ANOVA_TABLE' as result_type, 'Between_Groups' as category,
  CAST(df_between as STRING) as value_1,
  CAST(ROUND(ssb, 2) as STRING) as value_2,
  CAST(ROUND(msb, 4) as STRING) as value_3
FROM anova_calc

UNION ALL

SELECT 'ANOVA_TABLE', 'Within_Groups',
  CAST(df_within as STRING),
  CAST(ROUND(ssw, 2) as STRING),
  CAST(ROUND(msw, 4) as STRING)
FROM anova_calc

UNION ALL

SELECT 'ANOVA_TABLE', 'F_Statistic',
  NULL as value_1,
  NULL as value_2,
  CAST(ROUND(msb/msw, 4) as STRING) as value_3
FROM anova_calc

ORDER BY result_type DESC, value_1 NULLS LAST
```

The `UNION ALL` structure combines multiple result sets into a single output table, with the `result_type` field distinguishing between group summaries and ANOVA table components. Group summaries present sample sizes (`value_1`), means (`value_2`), and standard deviations (calculated as the square root of variance in `value_3`) for each racial group. The ANOVA table follows traditional format, showing degrees of freedom, sums of squares, and mean squares for both between-group and within-group sources of variation. The F-statistic appears as the ratio of mean square between to mean square within, providing the test statistic for significance evaluation.

An exemplary case can be observed in Table 6-4, which illustrates the expected output format of this test.

Table 6-4. Output from handmade one-way ANOVA

result_type	category	value_1	value_2
GROUP_SUMMARY	American_Indian	10	7.838
GROUP_SUMMARY	Black	37	6.547
GROUP_SUMMARY	White	953	7.297
ANOVA_TABLE	Between_Groups	2	23.22
ANOVA_TABLE	Within_Groups	997	1587.91
ANOVA_TABLE	F_Statistic		

These results are missing the `p_value` for easy comparison with a defined significance level, exposing the need of creating a UDF to determine if the calculated F-statistic leads to the rejection of the null hypothesis. To do so, we create a UDF in JavaScript that will calculate the critical value for a significance level of 5%, as seen in Example 6-19.

Example 6-19. One-way ANOVA: create UDF in JavaScript

```
CREATE OR REPLACE FUNCTION oreilly.f_critical_05(
  df1 INT64,
  df2 INT64
)
RETURNS FLOAT64
LANGUAGE js AS """
  // Simplified lookup table for common F critical values at α = 0.05
  const f_table = {
    '1,10': 4.96, '1,15': 4.54, '1,20': 4.35, '1,30': 4.17,
    '2,10': 4.10, '2,15': 3.68, '2,20': 3.49, '2,30': 3.32,
    '3,10': 3.71, '3,15': 3.29, '3,20': 3.10, '3,30': 2.92,
    '4,10': 3.48, '4,15': 3.06, '4,20': 2.87, '4,30': 2.69
  };

  const key = df1 + ',' + df2;

  // Return exact match if available
  if (f_table[key]) return f_table[key];

  // Simple approximation for other values
  return 3.0 + (10 / df2) - (0.1 * df1);
""";
```

Finally, with this function created within the original dataset, the results can be encapsulated to generate the conclusions of this test (Example 6-20).

Example 6-20. One-way ANOVA: conclusion of hypothesis test

```
SELECT
  ROUND(msb/msw, 4) AS F_Statistic,
  df_between,
  df_within,
  ROUND(oreilly.f_critical_05(df_between, df_within), 3) AS F_Critical,
  CASE
    WHEN msb/msw > oreilly.f_critical_05(df_between, df_within)
      THEN 'SIGNIFICANT - Reject H0'
    ELSE 'NOT SIGNIFICANT - Fail to reject H0'
```

```
  END AS Conclusion
FROM anova_calc;
```

Given that the interest is only to determine the rejection of the null hypothesis, the results in Example 6-18 can be encapsulated into another CTE or completely removed because only the `anova_calc` and previous CTEs are required to compute the `F_Critical`. If `F_Statistic > F_Critical` then the null hypothesis is rejected, indicating significant differences exist among group means.

Kruskal—Wallis test

The Kruskal—Wallis test (Example 6-21) serves as the non-parametric extension of one-way ANOVA, comparing distributions across three or more independent groups without assuming normality. Like the Mann–Whitney U test, it relies on ranking all observations across groups and comparing the rank sums, making it robust to outliers and non-normal distributions.

The Kruskal—Wallis test becomes essential when comparing multiple treatment conditions, analyzing performance across several regions or time periods, evaluating metrics across different customer segments, or when data violates ANOVA assumptions of normality and equal variances. This test determines whether at least one group differs significantly from the others, though it does not specify which particular groups differ.

Example 6-21. Kruskal—Wallis test

```
WITH samples AS (
  SELECT
    STRUCT(
      species AS factor,
      CAST(body_mass_g AS FLOAT64) AS val
    ) AS sample_struct
  FROM `bigquery-public-data.ml_datasets.penguins`
  WHERE body_mass_g IS NOT NULL
)
SELECT
  bqutil.fn.kruskal_wallis(ARRAY_AGG(sample_struct)).H AS h_statistic,
  bqutil.fn.kruskal_wallis(ARRAY_AGG(sample_struct)).p AS p_value,
  bqutil.fn.kruskal_wallis(ARRAY_AGG(sample_struct)).dof AS degrees_of_freedom
FROM samples;
```

This query demonstrates the Kruskal—Wallis test using a struct-based approach to compare body mass distributions across penguin species. The `samples` CTE creates a structured data format where each row becomes a `STRUCT` containing both the grouping factor (`species`) and the measurement value (`body_mass`). This approach eliminates the need for conditional aggregation by preserving the group-value relationship within each struct.

The `STRUCT` format with `factor` and `val` fields is mandatory for using this Kruskal—Wallis function directly, hence the standardized way of grouping the data. Converting `body_mass_g` to `FLOAT64` ensures numerical compatibility, while filtering for non-null values maintains data quality. This structure captures all penguin species present in the dataset automatically, without requiring explicit enumeration of each species.

The `SELECT` statement aggregates all structs into a single array through `ARRAY_AGG` and applies the Kruskal—Wallis test. This approach scales naturally to any number of groups without modifying the query structure. The function returns multiple components: the H-statistic (equivalent to the chi-squared test statistic), the `p-value` indicating statistical significance, and the degrees of freedom (typically the number of groups minus one). A significantly low `p-value` (e.g., 0.05) indicates that at least one species differs in body mass distribution from the others, though post-hoc testing would be required to identify specific pairwise differences. A high `p-value` suggests that the observed differences could plausibly be due to random chance.

Machine Learning

Modern cloud data platforms have transformed from simple query engines into comprehensive machine learning platforms. This evolution addresses a fundamental challenge: the friction between where data lives (data warehouses) and where ML models are traditionally built (separate ML platforms). By bringing ML directly to the data, organizations can eliminate

costly data movement, reduce latency, and leverage the scale and security of their existing data infrastructure.

Drilling deeper into the analysis of when to choose and employ SQL-based ML platforms, here are some scenarios to consider:

Analyst-driven analytics

When business analysts need to build predictive models without learning new programming languages

Data governance requirements

When models must operate within existing security and compliance frameworks

Real-time scoring

When predictions need to be generated as part of regular SQL queries

Exploratory analysis

When rapidly prototyping models on large datasets stored in the warehouse

Consider alternatives to SQL-based ML strategies for these scenarios:

Complex feature engineering

Advanced preprocessing may be easier in Python/R

Cutting-edge algorithms

Latest research implementations are typically in specialized ML frameworks first

Multi-platform deployment

When models need to run across diverse environments

Deep learning capabilities

For computer vision, NLP, or other deep learning applications requiring specific architecture design of a neural network

If you decide to use a SQL-native ML platform, choose a provider based on supported model types, integration, and operational constraints. Table 6-5 summarizes the current state of several providers.

Table 6-5. Comparison of platforms for ML

Platform	Supported capabilities	Syntax style	Key strengths	Limi
BigQuery ML	Regression, classification, clustering, time series, deep learning	`CREATE MODEL` statements	Most mature, extensive model library, AutoML integration	Goog ecosy lock-
Snowflake ML	Regression, classification, time series	`CREATE SNOWFLAK E.ML.FORECAST`	Strong data sharing, Python integration via Snowpark	New offer limit type
Amazon Redshift ML	Regression, classification, clustering	`CREATE MODEL` with SageMaker integration	Deep AWS ecosystem integration, automatic SageMaker training	Requ Sage know adva featu
Databricks SQL	Regression, classification, clustering, deep learning	Spark SQL with ML functions	Unified analytics platform, strong MLOps	More setup know helpf
Azure Synapse	Classification, regression (via integrations)	`PREDICT` functions	Strong Microsoft ecosystem integration	Limi SQL capa

BigQuery ML excels in ease of use and model variety. Its integration with Google Cloud's AI Platform enables automatic hyperparameter tuning and model explanation features. The platform's columnar storage and distributed processing make it particularly effective for training on large datasets.

Snowflake ML leverages the platform's unique architecture for secure data sharing, enabling collaborative ML workflows across organizations. Its Python integration through Snowpark allows for more complex feature engineering while maintaining the SQL interface for model management.

Redshift ML benefits from tight integration with Amazon SageMaker, automatically selecting appropriate algorithms and handling model deployment. This hybrid approach combines SQL simplicity with SageMaker's advanced capabilities.

Advanced Feature Engineering Patterns

In "Data Processing and Analysis", we saw how SQL can be used to perform essential preparatory tasks: profiling the dataset, studying correlations, and even detecting outliers. These capabilities naturally extend into broader data quality operations such as deduplication, correcting data type inconsistencies, and identifying or imputing missing values. Together, these steps ensure that the raw dataset is both trustworthy and consistent—a prerequisite for any meaningful machine learning model.

But once the foundation is laid, the next challenge is feature engineering: transforming raw data into representations that a model can learn from effectively. The following sections introduce some of the most impactful feature engineering strategies that can be directly implemented in SQL. While the techniques may vary depending on the dataset and problem domain, these patterns are widely applicable across industries and use cases.

Categorical encoding

Categorical encoding converts string/ID fields into numeric signals that models can consume, since some models rely on numerical data to be applied. Different techniques can be performed to discretize the values, depending on the number of distinct categories involved and the allowed dataset column size.

One-hot encoding

In one-hot encoding, a variable of *n* categories is split into *n* columns, where each column references each category and only one column out of *n* can have value 1 for each record: the remaining ones are 0. This indicates that for a specific row only one category is allowed, and this is flagged by the 1 in one of the encoding columns (Example 6-22).

Example 6-22. One-hot encoding

```
SELECT
  usertype,
  IF(usertype = 'Customer',   1, 0) AS usertype_customer,
  IF(usertype = 'Subscriber', 1, 0) AS usertype_subscriber
FROM `bigquery-public-data.new_york_citibike.citibike_trips`
WHERE usertype IS NOT NULL
LIMIT 100;
```

This creates one binary indicator per category. It's easy to interpret in linear models (coefficients read as per-category effects) and avoids spurious ordering that would come from naively casting strings to integers. Use this method when there are just a few distinct values, since it significantly increases the dataset width.

Frequency encoding

Frequency encoding maps each category to how often it appears in the training data—either a raw count (`COUNT(*)`) or a normalized proportion (`COUNT(*)/total`). This method preserves information in a single dense numeric value, scales well to high-cardinality features, and tends to play nicely with tree-based models like XGBoost. To avoid leakage on ML modeling, compute frequencies on the training split (or pre-cutoff data), consider smoothing/clipping for rare categories, and fall back to a global prior for unseen categories at inference (Example 6-23).

Example 6-23. Frequency encoding

```
WITH freq AS (
  SELECT category, COUNT(*) AS category_freq
  FROM `bigquery-public-data.thelook_ecommerce.products`
  GROUP BY category
)
SELECT
  id,
  p.category,
  f.category_freq
FROM `bigquery-public-data.thelook_ecommerce.products` p
JOIN freq f USING (category)
LIMIT 100;
```

This query builds a compact "count-encoding" feature. The CTE `freq` groups products by `category` and computes how many rows fall into each (`category_freq`). The main `SELECT` then joins that table back to the original products via `USING` (`category`), so every product row inherits the frequency of its category. The result is one numeric feature per row—`category_freq`—without creating a wide one-hot matrix.

Feature hashing

Hashing (feature hashing) maps each category string to a fixed number of buckets using a stable hash (Example 6-24). We trade exact identity for a compact, constant-width representation that naturally handles unseen values at inference—no dictionary or re-fits needed. Collisions (the same value for different observations) are expected so pick enough buckets to keep them rare (often 2–10× the expected unique values of the variable). For multi-label fields, it is typical to hash each token separately and then aggregate (sum/mean) per row.

Example 6-24. Feature hashing (hashing trick)

```
SELECT
  id AS question_id,
  tags,
  ABS(MOD(FARM_FINGERPRINT(tags), 1024)) AS tag_bucket_1024
FROM bigquery-public-data.stackoverflow.posts_questions
WHERE tags IS NOT NULL
LIMIT 100;
```

This query uses `FARM_FINGERPRINT` to produce a stable 64-bit hash per tags value, then `MOD(..., 1024)` to map it into 1,024 buckets; `ABS` ensures nonnegative indices. Collisions are allowed, but the gain is a fixed-width feature that will not grow as new tags appear.

> **WARNING**
>
> This hashes the entire tags string. To have one bucket per individual tag, first split and `UNNEST` the tokens before hashing, then aggregate back to the question level.

Text feature engineering

Free-form text sometimes must be shaped into numeric signals before models can use it. This is a more complex task due to the degree of volatility inherent in free-form text. Users can have typos, mix lowercase and uppercase, and omit or add punctuation, among others, exponentially increasing the number of distinct values. As such, encoding free-form text can become a cumbersome task that requires different approaches.

Tokenization

Tokenization breaks a string into substrings (tokens), and stopword removal drops high-frequency *glue* words (like *the*, *and*) that are low signal (Example 6-25). There are several different methods to tokenize text, the most common being to split on whitespace. The goal is a compact set of interpretable terms that can be counted per document and later weighted, for example, with Term Frequency–Inverse Document Frequency (TF-IDF). Keep the normalization applied here—lowercasing and basic punctuation stripping, for example—identical at training and inference so that the model sees the same vocabulary rules in production, maintaining the cohesion of the tokenization process.

Example 6-25. Tokenization with minimal stopword removal

```
WITH base AS (
  SELECT
    id,
```

```
    LOWER(REGEXP_REPLACE(title, r'[^a-z0-9 ]', '')) AS clean
  FROM `bigquery-public-data.stackoverflow.posts_questions`
  WHERE title IS NOT NULL
),
tokens AS (
  SELECT id, token
  FROM base, UNNEST(SPLIT(clean, ' ')) AS token
  WHERE token != ''
)
SELECT id, token
FROM tokens
WHERE token NOT IN
('the','a','and','or','to','of','in','on','for','with','is','it')
LIMIT 100;
```

The `base` CTE lowercases each title and strips punctuation, leaving letters, digits, and spaces in `clean`. The `tokens` step splits `clean` on spaces and flattens the result so each `(id, token)` appears on its own row. The outer query removes a short stopword list, keeping only semantically useful terms. From here, we can `GROUP BY id, token` to compute per-title term counts, or aggregate across documents to build document frequencies for TF–IDF.

Character n-grams

Character n-grams represent text as overlapping slices of fixed length. Using trigrams (Example 6-26 is an example of length 3) often strikes a good balance: they are short enough to match misspellings and morphological variants (e.g., *json*, *jsonp*, *json5* share many trigrams), yet long enough to carry meaning. Trigrams shine on short, noisy strings—like titles, filenames, or code identifiers—and pair well with hashing if you need to cap dimensionality.

Example 6-26. Character 3-grams (subword features)

```
WITH base AS (
  SELECT
    id,
    LOWER(REGEXP_REPLACE(title, r'[^a-z0-9]', '')) AS t
  FROM `bigquery-public-data.stackoverflow.posts_questions`
  WHERE title IS NOT NULL
)
SELECT
```

```
  id,
  SUBSTR(t, pos, 3) AS trigram
FROM base,
UNNEST(GENERATE_ARRAY(1, GREATEST(LENGTH(t) - 2, 0))) AS pos
WHERE LENGTH(t) >= 3
LIMIT 200;
```

First, remove non-alphanumerics so trigrams aren't broken by punctuation, storing the cleaned string in `t`. For each row, `GENERATE_ARRAY` produces all valid starting positions for a 3-character window. `SUBSTR(t, pos, 3)` then emits every overlapping trigram for that title. To turn these into model inputs, aggregate by `(id, trigram)` to get counts; if the feature space grows too large, hash each `trigram` into a fixed number of buckets and aggregate counts per bucket.

In practice, choose *tokenization* when your text is relatively clean and a few words carry most of the meaning—you want interpretable features, easy error analysis, and the option to add phrase information with bigrams or trigrams. Choose *character n-grams* when strings are short or noisy, when you expect typos or code-like identifiers, or when language boundaries blur. It is common to combine both: tokens provide semantics for longer titles or descriptions, while char-grams act as a robust backstop. If dimensionality becomes a concern, hash either representation to a fixed number of buckets and aggregate counts per bucket.

You can also explore other SQL-friendly encodings:

- Word n-grams and their hashed variants
- TF–IDF or even BM25-style weighting via aggregate statistics
- Regex-derived flags and dictionary/gazetteer joins for domain terms
- Simple orthographic features (length, digit/emoji ratios, case patterns) and similarity shingles with hashed sets for near-duplicate detection

Finally, modern workflows often leverage large language models to map free text into structured labels, tags, or summaries—useful when we need business-specific categories rather than raw tokens. Chapter 7 returns to that approach.

Quantile binning

Quantile binning discretizes a skewed continuous feature into ordered categories. In *equal-frequency* binning, each bin contains roughly the same number of rows, so interval widths typically differ. In *equal-width* binning, all bins share the same numeric span, which can yield very different counts per bin. As data become sparser and more skewed, these approaches diverge. Both are useful: equal-frequency adapts to skew and resists outliers; equal-width preserves constant spans that are easy to explain and compare over time.

Equal-frequency (deciles)

Equal-frequency bins place approximately the same number of rows in each bucket, adapting to skew automatically. This is a strong default for modeling: it linearizes monotone effects, is resilient to outliers, and yields intuitive ordinal categories.

The first step (Example 6-27) computes the exact cut points using `PERCENTILE_CONT`, which calculates percentiles with interpolation at the desired probabilities (10%, 20%, …, 90%).

Example 6-27. Compute exact decile cut points

```
WITH bounds AS (
  SELECT
    PERCENTILE_CONT(tripduration, 0.1) OVER() AS p10,
    PERCENTILE_CONT(tripduration, 0.2) OVER() AS p20,
    PERCENTILE_CONT(tripduration, 0.3) OVER() AS p30,
    PERCENTILE_CONT(tripduration, 0.4) OVER() AS p40,
    PERCENTILE_CONT(tripduration, 0.5) OVER() AS p50,
    PERCENTILE_CONT(tripduration, 0.6) OVER() AS p60,
    PERCENTILE_CONT(tripduration, 0.7) OVER() AS p70,
    PERCENTILE_CONT(tripduration, 0.8) OVER() AS p80,
    PERCENTILE_CONT(tripduration, 0.9) OVER() AS p90
  FROM `bigquery-public-data.new_york_citibike.citibike_trips`
  WHERE tripduration IS NOT NULL
```

```
  LIMIT 1
)
SELECT * FROM bounds;
```

With the cut points materialized, we then assign each row to a bin (1–10) using a `CASE` expression (Example 6-28). Each threshold comparison places the value into the appropriate decile.

Example 6-28. Assign rows to decile bins

```
-- Continuing from the bounds CTE above
SELECT
  t.tripduration,
  CASE
    WHEN t.tripduration <  b.p10 THEN 1
    WHEN t.tripduration <  b.p20 THEN 2
    WHEN t.tripduration <  b.p30 THEN 3
    WHEN t.tripduration <  b.p40 THEN 4
    WHEN t.tripduration <  b.p50 THEN 5
    WHEN t.tripduration <  b.p60 THEN 6
    WHEN t.tripduration <  b.p70 THEN 7
    WHEN t.tripduration <  b.p80 THEN 8
    WHEN t.tripduration <  b.p90 THEN 9
    ELSE 10
  END AS duration_decile_eqf
FROM `bigquery-public-data.new_york_citibike.citibike_trips` AS t
CROSS JOIN bounds AS b
WHERE t.tripduration IS NOT NULL
LIMIT 200;
```

Due to ties and discreteness, counts per bin may differ slightly, but that is expected with real-world data.

Equal-width (deciles)

Equal-width bins divide the numeric range into fixed spans (Example 6-29). They are easy to explain and stable for dashboards or reporting, but they can be sensitive to outliers: a long tail may push most rows into the first bin or two. It is usually preferable to leverage equal-width when physical thresholds matter (e.g., speed limits, business price bands) or when consistent spans aid interpretation across time.

Example 6-29. Equal-width deciles from min/max

```sql
-- Step A: compute min/max and the bin width
WITH stats AS (
  SELECT
    MIN(tripduration) AS vmin,
    MAX(tripduration) AS vmax
  FROM `bigquery-public-data.new_york_citibike.citibike_trips`
  WHERE tripduration IS NOT NULL
),
params AS (
  SELECT
    vmin,
    vmax,
    SAFE_DIVIDE(vmax - vmin, 10) AS width
  FROM stats
)

-- Step B: assign each row to a bin 1..10 using width
SELECT
  t.tripduration,
  CASE
    WHEN p.width IS NULL OR p.width = 0 THEN 1
    ELSE LEAST(
      10,
      1 + CAST(FLOOR( (t.tripduration - p.vmin) / p.width ) AS INT64)
    )
  END AS duration_decile_eqw
FROM `bigquery-public-data.new_york_citibike.citibike_trips` AS t
CROSS JOIN params AS p
WHERE t.tripduration IS NOT NULL
LIMIT 200;
```

The query first computes `vmin` and `vmax` over non-null `tripduration`. In `params`, `width = SAFE_DIVIDE(vmax - vmin, 10)` derives the span of each decile with `SAFE_DIVIDE` yielding `NULL` on zero-division. On the outer query, the guard in `CASE` assigns all rows to bin 1 when `NULL` or when the range is degenerate (all values equal). For typical ranges, we transform each value into a *0-based* index via `FLOOR((value - vmin)/width)` where values equal to `vmin` map to index 0 (bin 1), while values equal to `vmax` map to index 10. `LEAST(10, 1 + index)` caps the maximum so `vmax` lands in bin 10. This creates *left-closed, right-open* intervals for bins 1 to 9 — `[vmin + (k-1)·width, vmin + k·width)`— and a *closed* final bin up to `vmax`.

> **WARNING**
>
> Because equal-width uses the raw range, extreme outliers can stretch `width` and compress the early bins. In these cases, consider outlier removal as seen on "Data Processing and Analysis" on a trimmed range (e.g., between `p1` and `p99`).

Target encoding (leakage-aware) with K-fold out-of-fold mean

Target encoding replaces each category with a statistic of the target—typically the mean for regression tasks or positive rate for classification tasks. It shines on *high-cardinality* features because it condenses thousands of levels into a single scalar with real signal. The catch is *leakage*: if a row's own label helps compute its encoded value, the model overfits.

In K-fold out-of-fold mean, we first split the data into K folds with a stable hash so every row is deterministically assigned to a fold. Next, we preaggregate the target per (category, fold) to get a sum and a count for every category inside each fold. These summaries let us compute a leakage-safe mean for any row without touching its own label: for a row in fold k with category c, take the category totals across all folds and subtract the contribution of fold k; divide the remaining sum by the remaining count. That gives the category's mean built from other folds only, which mimics test-time conditions.

Going step-by-step, the first (Example 6-30) creates a stable 0–4 `fold` id via a hash of `order_id`, in this case using `FARM_FINGERPRINT`, then summarizes each `(category, fold)` with a sum and a count. The deterministic fold keeps splits reproducible across runs.

Example 6-30. Build folds and per-(category, fold) aggregates

```
WITH items AS (
  SELECT
    oi.order_id,
    oi.id,
    oi.sale_price,
    p.category,
    -- Deterministic 0..4 fold id from order_id
    MOD(ABS(FARM_FINGERPRINT(CAST(oi.order_id AS STRING))), 5) AS fold
```

```
  FROM `bigquery-public-data.thelook_ecommerce.order_items` AS oi
  JOIN `bigquery-public-data.thelook_ecommerce.products`    AS p  USING (id)
  WHERE oi.sale_price IS NOT NULL
),

-- Step B: per-(category, fold) sums and counts
per_cat_fold AS (
  SELECT
    category,
    fold,
    SUM(sale_price) AS sum_price,
    COUNT(*)        AS n
  FROM items
  GROUP BY category, fold
),
```

To finish the out-of-fold setup we need to calculate two rollups, as shown in Example 6-31. First are per-category totals across all folds (`per_cat_all`). These give the full sum and count for each category. In step C, we remove the portion that comes from the current fold so each row is compared only to the rest of the data. Second are global totals per fold and overall (`global_fold, global_all`). These let us compute a fold-specific global average that excludes the current fold as well; this average will be used as a stabilizing prior for rare categories.

Example 6-31. Category and global totals (ingredients for out-of-fold prior)

```
-- Step C: global totals per fold and overall (to form an out-of-fold prior)
global_fold AS (
  SELECT
    fold,
    SUM(sale_price) AS sum_fold,
    COUNT(*)        AS n_fold
  FROM items
  GROUP BY fold
),
global_all AS (
  SELECT
    SUM(sale_price) AS sum_all,
    COUNT(*)        AS n_all
  FROM items
)
```

In Example 6-32 we first add `per_cat_all`, which holds the category totals across every fold. The final `SELECT` then stitches all pieces together for a leakage-safe value. For each row, it looks up the row's category totals in `per_cat_all` and removes the contribution of the row's own fold via the left join to `per_cat_fold`; this yields the category mean computed from other folds only. In parallel, it forms a fold-specific global prior by taking the overall totals from `global_all` and subtracting the current fold's totals from `global_fold`, so even the prior excludes the row's fold. Finally, it blends the category mean with that prior using shrinkage (here, weight `alpha = 20`): intuitively, we are adding some pull toward the out-of-fold global average when the category is rare, and relying mostly on the category's own out-of-fold evidence when it is well represented. `COALESCE` handles categories that don't appear in the current fold, and `SAFE_DIVIDE` protects against division by zero. The result is a single, dense feature per row that carries real signal while avoiding any use of the row's own label.

Example 6-32. Out-of-fold encoded value with shrinkage (final mapping)

```
-- Step D: per-category totals across all folds (to subtract current fold)
per_cat_all AS (
  SELECT
    category,
    SUM(sale_price) AS sum_all,
    COUNT(*)        AS n_all
  FROM items
  GROUP BY category
),

-- Step E: leakage-aware encoding from OTHER folds + shrinkage
SELECT
  i.*,
  SAFE_DIVIDE(
    (pc_all.sum_all - COALESCE(pc_fold.sum_price, 0))
    + 20 * SAFE_DIVIDE(
            (glob_all.sum_all - COALESCE(glob_fold.sum_fold, 0)),
            NULLIF(glob_all.n_all - COALESCE(glob_fold.n_fold, 0), 0)
          ),
    (pc_all.n_all - COALESCE(pc_fold.n, 0)) + 20
  ) AS category_target_enc
FROM items AS i
JOIN per_cat_all     AS pc_all   ON pc_all.category = i.category
```

```
LEFT JOIN per_cat_fold AS pc_fold ON pc_fold.category = i.category
  AND pc_fold.fold = i.fold
LEFT JOIN global_fold  AS glob_fold ON glob_fold.fold = i.fold
CROSS JOIN global_all  AS glob_all
LIMIT 1000;
```

Principal component analysis

Principal component analysis (PCA) projects correlated numeric features into a smaller set of orthogonal components that capture most of the variance. The benefits are dimensionality reduction, decorrelation (useful for linear models and regularization), and a degree of noise smoothing. The trade-off is interpretability as the components are linear combinations, not the original fields, which hinders interpretability.

In BigQuery ML, a PCA model can be trained and then applied to score rows with component values. Another option is to inspect loadings to understand which inputs drive each component as a proxy understanding of feature importance. PCA is sensitive to scale, so make sure features are comparable before training by standardizing or normalizing the features beforehand.

Using the Citibike public dataset in Example 6-33, we start by assembling a compact, numeric feature matrix: raw duration, hour-of-day, and geodesic distance between start and end points. As in earlier examples, writing data to our own dataset enables model training and function creation because public tables are read-only.

Example 6-33. Create feature table

```
CREATE OR REPLACE TABLE `oreilly.citibike_features` AS
SELECT
  tripduration,
  EXTRACT(HOUR FROM starttime) AS start_hour,
  ST_DISTANCE(
    ST_GEOGPOINT(start_station_longitude, start_station_latitude),
    ST_GEOGPOINT(end_station_longitude, end_station_latitude)
  ) AS distance_m
FROM `bigquery-public-data.new_york_citibike.citibike_trips`
WHERE start_station_latitude  IS NOT NULL
  AND end_station_latitude  IS NOT NULL
  AND start_station_longitude IS NOT NULL
  AND end_station_longitude IS NOT NULL;
```

This yields three clean numeric columns per trip: duration, hour, and straight-line distance in meters. These fields often covary (e.g., distance and duration), which makes them good candidates for PCA.

Then, in Example 6-34, we fit a PCA model that automatically orders components by explained variance and retains as many as needed to reach the chosen threshold; in this case, it is 95%. To do so, we leverage `CREATE OR REPLACE MODEL` to specify the creation of the model followed by `OPTIONS`, which account for the hyperparameter specification we want our models to have. Lastly, the `SELECT` statement refers to the dataset used as input for this task. This is a standardized approach for all model creation on BigQuery ML, as we will see in the following sections. There are, however, multiple parameters to choose that range from how to select the number of components (either via `PCA_EXPLAINED_VARIANCE_RATIO` or `NUM_PRINCIPAL_COMPONENTS`) to defining the solver (`PCA_SOLVER`) used during the computation of PCA. For more detail, see BigQuery's documentation.

Example 6-34. Train PCA with a variance threshold

```
CREATE OR REPLACE MODEL `oreilly.pca_citibike`
OPTIONS(
  MODEL_TYPE = 'PCA',
  PCA_EXPLAINED_VARIANCE_RATIO = 0.95
  -- NUM_PRINCIPAL_COMPONENTS = 2
) AS
SELECT tripduration, start_hour, distance_m
FROM `oreilly.citibike_features`;
```

With a small feature set like this, the threshold typically yields one to three components. It is advisable to keep this trained model fixed so future projections remain comparable over time.

Lastly, we can apply the PCA model to produce per-row scores on PC1, PC2, and so on, as shown in Example 6-35. These scores are dense, decorrelated features that can be fed into downstream models or plots.

Example 6-35. Score rows with principal components

```
SELECT *
FROM ML.PREDICT(
```

```
  MODEL `oreilly.pca_citibike`,
  TABLE `oreilly.citibike_features`
)
LIMIT 100;
```

In parallel, `ML.PRINCIPAL_COMPONENT_INFO` can be used to visualize both the eigenvalue and the explained variance corresponding to each principal component as in Example 6-36. For the individual contribution of each variable to each principal component, the process is identical, just switch `ML.PRINCIPAL_COMPONENT_INFO` for `ML.PRINCIPAL_COMPONENTS` in the query below.

Example 6-36. View component loadings and variance info

```
SELECT *
FROM ML.PRINCIPAL_COMPONENT_INFO(MODEL `oreilly.pca_citibike`);
```

Model Creation and Training

The foundation of a successful machine learning project lies in the systematic creation and training of models that can learn meaningful patterns from data. Such patterns are then leveraged to perform several actions ranging from classification tasks to forecasting and recommendation processes.

In traditional ML workflows, this stage (if not earlier) is often where the transition of data from database to a training instance machine happens—local if small use cases or cloud instances for large applications. For this step, we will look into how to select models for each kind of task, how to train them in BigQuery ML, and how to use them, leveraging SQL-like syntax for all of these steps.

While our examples focus on BigQuery ML because of its mature SQL-native machine learning capabilities, similar model creation patterns are emerging in other cloud platforms, including Snowflake ML, Amazon Redshift ML, and Azure Synapse Analytics. Below are some common approaches for different use cases, though each model type offers many more options and parameter tuning possibilities.

Supervised learning models

Supervised learning models learn from labeled training data to make predictions on new, unseen examples. These models form the backbone of most business machine learning applications, from predicting customer churn to forecasting sales revenue. Tree-based models like Random Forest and Boosted Trees excel at capturing nonlinear patterns and feature interactions without requiring extensive feature engineering. They handle mixed data types naturally, provide feature importance rankings, and often achieve strong performance out-of-the-box.

Next, train a Random Forest classifier (Example 6-37) to predict user type from trip characteristics. The target variable `usertype` distinguishes between `Customer` and `Subscriber`, making this a binary classification problem well-suited to tree ensembles.

Example 6-37. Prepare classification training data

```
CREATE OR REPLACE TABLE `oreilly.usertype_classification_features` AS
SELECT
  usertype,
  tripduration,
  starttime,
  EXTRACT(HOUR FROM starttime) AS start_hour,
  EXTRACT(DAYOFWEEK FROM starttime) AS day_of_week,
  start_station_name,
  ST_DISTANCE(
    ST_GEOGPOINT(start_station_longitude, start_station_latitude),
    ST_GEOGPOINT(end_station_longitude, end_station_latitude)
  ) AS distance_m
FROM `bigquery-public-data.new_york_citibike.citibike_trips`
WHERE usertype IS NOT NULL
  AND tripduration IS NOT NULL
  AND tripduration BETWEEN 60 AND 7200
  AND start_station_name IS NOT NULL
  AND start_station_latitude IS NOT NULL
  AND end_station_latitude IS NOT NULL;
```

This feature table includes both numeric features (duration, distance) and categorical features (station name, temporal patterns) that tree-based models can leverage effectively. The filtering removes extreme outliers and ensures data quality for reliable model training.

The Random Forest in Example 6-38 uses 100 trees (`NUM_PARALLEL_TREE`) with maximum depth of 10 levels. `SUBSAMPLE = 0.8` means each tree trains on 80% of the data, while `MIN_SPLIT_LOSS` prevents splits that don't significantly improve the model. These ensemble techniques reduce overfitting compared to single decision trees. `INPUT_LABEL_COLS` is the parameter that defines the target of the problem at hand.

Example 6-38. Train a random forest classifier

```
CREATE OR REPLACE MODEL `oreilly.usertype_forest`
OPTIONS(
  MODEL_TYPE = 'RANDOM_FOREST_CLASSIFIER',
  INPUT_LABEL_COLS = ['usertype'],
  NUM_PARALLEL_TREE = 100,
  MAX_TREE_DEPTH = 10,
  SUBSAMPLE = 0.8,
  MIN_SPLIT_LOSS = 0.1
) AS
SELECT
  usertype,
  tripduration,
  start_hour,
  day_of_week,
  start_station_name,
  distance_m,
  starttime
FROM `oreilly.usertype_classification_features`;
```

To apply this model, use `ML.PREDICT` to generate class predictions and probability scores for each prediction (Example 6-39). The `predicted_usertype_probs` array contains probability scores for each class on `predicted_usertype`, allowing you to understand model confidence and implement custom thresholds for business decisions.

Example 6-39. Generate predictions on new data

```
SELECT
  predicted_usertype,
  predicted_usertype_probs,
  tripduration,
  start_hour,
  start_station_name
FROM ML.PREDICT(MODEL `oreilly.usertype_forest`,
  (
```

```
    SELECT *
    FROM `oreilly.usertype_classification_features`
    LIMIT 1000
   )
 )
ORDER BY predicted_usertype_probs[OFFSET(1)].prob DESC
LIMIT 20;
```

Unsupervised learning models

Unsupervised learning discovers hidden patterns and structures in data without labeled targets. These techniques are invaluable for exploratory data analysis, customer segmentation, and identifying natural groupings in data: a prior example is PCA.

As another example, *k-means clustering* groups similar data points into *k* clusters by minimizing within-cluster distances to centroids, making it particularly useful for market research and understanding customer behavior patterns.

Using Citibike trip patterns, we'll segment stations based on usage characteristics to identify different types of stations in the bike-sharing network. This analysis can reveal insights such as commuter-focused stations compared to leisure-oriented locations (Example 6-40).

Example 6-40. Prepare station clustering features

```
CREATE OR REPLACE TABLE `oreilly.station_clustering_features` AS
SELECT
  start_station_name,
  COUNT(*) AS total_trips,
  AVG(tripduration) AS avg_duration,
  STDDEV(tripduration) AS stddev_duration,
  COUNT(DISTINCT EXTRACT(DATE FROM starttime)) AS active_days,
  AVG(EXTRACT(HOUR FROM starttime)) AS avg_start_hour,
  COUNT(DISTINCT usertype) AS user_types
FROM `bigquery-public-data.new_york_citibike.citibike_trips`
WHERE start_station_name IS NOT NULL
  AND tripduration IS NOT NULL
  AND tripduration BETWEEN 60 AND 7200
GROUP BY start_station_name
HAVING COUNT(*) >= 100;  -- Filter stations with sufficient data
```

This aggregation creates station-level features that capture usage patterns: total trip volume, average trip characteristics, temporal patterns, and user diversity. The `HAVING` clause ensures we only cluster stations with substantial trip history, to enable reliable pattern detection.

The model in Example 6-41 creates 5 clusters using the `k-means++` initialization method in `KMEANS_INIT_METHOD`, which provides better starting centroids than random initialization. `STANDARDIZE_FEATURES = TRUE` ensures all features contribute equally regardless of their scales, while `DISTANCE_TYPE = 'EUCLIDEAN'` defines how similarity is measured between stations.

Example 6-41. Train k-means clustering model

```
CREATE OR REPLACE MODEL `oreilly.station_clusters`
OPTIONS(
  MODEL_TYPE = 'KMEANS',
  NUM_CLUSTERS = 5,
  STANDARDIZE_FEATURES = TRUE,
  MAX_ITERATIONS = 50,
  KMEANS_INIT_METHOD = 'KMEANS++',
  DISTANCE_TYPE = 'EUCLIDEAN'
) AS
SELECT
  total_trips,
  avg_duration,
  stddev_duration,
  active_days,
  avg_start_hour,
  user_types
FROM `oreilly.station_clustering_features`;
```

To analyze the clustering algorithm that is now trained, we leverage the predictions achieved through `ML.PREDICT`, and we join the original features with cluster assignments to understand what each cluster represents (Example 6-42). By examining the average characteristics within each cluster, we can, for example, identify patterns such as “high-volume versus low-volume commuter stations.”

Example 6-42. Analyze cluster characteristics

```
WITH predictions AS (
  SELECT
```

```
    s.start_station_name,
    s.total_trips,
    s.avg_duration,
    p.CENTROID_ID as cluster_id
  FROM `oreilly.station_clustering_features` s
  JOIN ML.PREDICT(MODEL `oreilly.station_clusters`,
                  TABLE `oreilly.station_clustering_features`) p
    ON s.start_station_name = p.start_station_name
)
SELECT
  cluster_id,
  COUNT(*) as stations_in_cluster,
  ROUND(AVG(total_trips)) as avg_trips_per_station,
  ROUND(AVG(avg_duration)) as avg_trip_duration
FROM predictions
GROUP BY cluster_id
ORDER BY cluster_id;
```

Anomaly detection

Anomaly detection identifies data points that deviate significantly from normal patterns, making it valuable for fraud detection, system monitoring, and quality control. BigQuery ML provides autoencoder-based anomaly detection that learns to reconstruct normal patterns, flagging instances with high reconstruction errors as potential anomalies.

Autoencoders are neural networks trained to reconstruct their input through a compressed representation. When applied to normal data, they learn efficient encodings of typical patterns. Anomalous data points that do not fit these learned patterns will have higher reconstruction errors, enabling their identification. Example 6-43 shows how to train an autoencoder for anomaly detection.

Example 6-43. Train autoencoder for anomaly detection

```
CREATE OR REPLACE MODEL `oreilly.trip_anomaly_detector`
OPTIONS(
  MODEL_TYPE = 'AUTOENCODER',
  ACTIVATION_FN = 'RELU',
  BATCH_SIZE = 1000,
  DROPOUT = 0.2,
  HIDDEN_UNITS = [10, 4, 2, 4, 10],  -- Encoding-decoding structure
  MAX_ITERATIONS = 100,
  LEARN_RATE = 0.001,
```

```
  OPTIMIZER = 'ADAM'
) AS
SELECT
  tripduration,
  start_hour,
  day_of_week,
  distance_m
FROM `oreilly.usertype_classification_features`;
```

The autoencoder uses a symmetric architecture that compresses the four input features into two dimensions in the middle layer, then reconstructs them back to four outputs as seen on the hyperparameter `HIDDEN_UNITS`. This bottleneck forces the model to learn efficient representations of normal trip patterns, making reconstruction difficult for unusual combinations.

To evaluate the autoencoder, compute each row's reconstruction error with `ML.RECONSTRUCTION_LOSS`. As shown in Example 6-44, flag the top 5% of errors, estimated via `APPROX_QUANTILES` as anomalies. High-error rows typically reflect unusual trip patterns, such as very long rides at off-hours or atypical distance–duration combinations.

Example 6-44. Identify anomalous trips

```
WITH losses AS (
  SELECT
    *,
    mean_squared_error AS reconstruction_error
  FROM ML.RECONSTRUCTION_LOSS(
    MODEL `oreilly.trip_anomaly_detector`,
    (
      SELECT
        tripduration,
        start_hour,
        day_of_week,
        distance_m
      FROM `oreilly.usertype_classification_features`
      LIMIT 10000
    )
  )
),
p95 AS (
  SELECT APPROX_QUANTILES(reconstruction_error, 100)[OFFSET(95)] AS thresh
  FROM losses
)
SELECT
```

```
  tripduration,
  start_hour,
  distance_m,
  reconstruction_error,
  CASE
    WHEN reconstruction_error > p95.thresh THEN 'ANOMALY'
    ELSE 'NORMAL'
  END AS anomaly_flag
FROM losses, p95
ORDER BY reconstruction_error DESC
LIMIT 20;
```

> **TIP**
>
> This form assumes that the top 5% will always be the anomalous errors; in reality, it can be much more or much less. Using previous methods such as those seen in "Outlier Detection" can make this process much more robust and flexible, since the threshold varies with the data instead of using a strict *x*% threshold.

Recommendation systems

Recommendation systems personalize choices by estimating how much each user will like each item based on past behavior. A compact, effective approach is *matrix factorization*, which learns low-dimensional vectors for users and items so that their dot product reproduces observed interactions and generalizes to unseen pairs. In BigQuery ML, this works well with both explicit ratings and implicit signals such as purchases or visits.

We first assemble a user–station interaction table (see Example 6-45). This creates synthetic user IDs by hashing bike ID and station combinations, assuming bikes represent individual users. The trip count becomes an implicit rating (capped at 5), representing how much a "user" prefers each station based on usage frequency.

Example 6-45. Prepare user–station interaction data

```
CREATE OR REPLACE TABLE `oreilly.user_station_interactions` AS
WITH user_trips AS (
  SELECT
    CONCAT('user_', MOD(ABS(FARM_FINGERPRINT(
      CONCAT(CAST(bikeid AS STRING), start_station_name))), 10000)) AS user_id,
```

```
    start_station_name AS station_id,
    COUNT(*) AS trip_count
  FROM `bigquery-public-data.new_york_citibike.citibike_trips`
  WHERE start_station_name IS NOT NULL
    AND bikeid IS NOT NULL
  GROUP BY user_id, station_id
  HAVING trip_count >= 2  -- Focus on repeat usage
)
SELECT
  user_id,
  station_id,
  LEAST(trip_count, 5) AS rating  -- Cap at 5 for rating scale
FROM user_trips;
```

Applying the `MATRIX_FACTORIZATION` model type, we fit 50 latent factors for users and stations (`NUM_FACTORS = 50`) so their dot product reconstructs the observed ratings (see Example 6-46). `L2_REG = 0.1` adds ridge regularization to curb overfitting on sparse data, and `MAX_ITERATIONS = 30` controls optimization passes.

Example 6-46. Train matrix factorization model

```
CREATE OR REPLACE MODEL `oreilly.station_recommender`
OPTIONS(
  MODEL_TYPE = 'MATRIX_FACTORIZATION',
  USER_COL = 'user_id',
  ITEM_COL = 'station_id',
  RATING_COL = 'rating',
  NUM_FACTORS = 50,
  L2_REG = 0.1,
  MAX_ITERATIONS = 30
) AS
SELECT
  user_id,
  station_id,
  rating
FROM `oreilly.user_station_interactions`;
```

To generate the final recommendations, `ML.RECOMMEND` generates predictions for all unobserved user–station pairs for the specified user (Example 6-47). The highest predicted ratings represent stations the model believes this user would prefer based on patterns learned from similar users and station characteristics.

Example 6-47. Generate station recommendations

```
SELECT
  user_id,
  station_id,
  predicted_rating
FROM ML.RECOMMEND(
  MODEL `oreilly.station_recommender`,
  (SELECT 'user_1234' as user_id)
 )  -- Recommend for specific user
WHERE predicted_rating IS NOT NULL
ORDER BY predicted_rating DESC
LIMIT 10;
```

Time-series forecasting

Time-series forecasting predicts future values based on historical temporal patterns, seasonal trends, and external factors. The key distinctions are:

Classic models

> These models rely on analyzing the timeseries data to study behavior and predict the next X values. These can include AutoRegressive Integrated Moving Average (ARIMA) models and exponential smoothing models from Simple Exponential Smoothing to Holt–Winters (Triple Exponential) Smoothing.

General-purpose models

> These are typical regression machine learning models. For these models, the dataset has to be changed into a tabular format through feature engineering while mitigating leakage (a typical occurrence in this area). This is particularly useful when we want to account for data other than the time series (for example, flag a Black Friday event).

Hybrid models

> These models extend the classic approach for time-series modeling and combine it with additional factors such as exogenous variables, for example, Seasonal AutoRegressive Integrated Moving Average with eXogenous regressors (SARIMAX).

In BigQuery ML, `ARIMA_PLUS` is a SARIMAX-style model—seasonality plus optional exogenous variables—with features such as holiday effects. Use it to forecast daily bike trip volumes from historical Citibike data (Example 6-48).

Example 6-48. Prepare daily trip volume data

```
CREATE OR REPLACE TABLE `oreilly.daily_trip_volume` AS
SELECT
  DATE(starttime) AS trip_date,
  COUNT(*) AS daily_trips,
  AVG(tripduration) AS avg_duration,
  COUNT(DISTINCT start_station_name) AS active_stations,
  MAX(
      CASE
          WHEN EXTRACT(DAYOFWEEK FROM starttime) IN (1, 7) THEN 1
          ELSE 0
      END
  ) AS is_weekend
FROM `bigquery-public-data.new_york_citibike.citibike_trips`
WHERE starttime >= '2017-01-01'
  AND starttime < '2018-01-01'
GROUP BY DATE(starttime)
ORDER BY trip_date;
```

This aggregation creates daily-level features including total trip volume (our forecast target), average trip duration, station use, and weekend indicators. The time range focuses on a single year to ensure consistent seasonal patterns without long-term structural changes.

Triggering the creation of an ARIMA model, BigQuery ML automatically selects orders with `AUTO_ARIMA = TRUE` and can incorporate holiday effects via `HOLIDAY_REGION = 'US'`. For a single daily series of Citibike trips, the plain `ARIMA_PLUS` variant is enough: you provide the timestamp and the value to forecast, and the model learns seasonality and trend directly from history (Example 6-49).

Example 6-49. Train ARIMA_PLUS forecasting model

```
CREATE OR REPLACE MODEL `oreilly.trip_volume_forecast`
OPTIONS(
  MODEL_TYPE = 'ARIMA_PLUS',
  TIME_SERIES_TIMESTAMP_COL = 'trip_date',
  TIME_SERIES_DATA_COL     = 'daily_trips',
```

```
  HOLIDAY_REGION           = 'US',
  AUTO_ARIMA               = TRUE,
  DATA_FREQUENCY           = 'DAILY'
) AS
SELECT trip_date, daily_trips
FROM `oreilly.daily_trip_volume`;
```

With `ML.FORECAST` we can generate predictions for the next *n* days over a confidence interval—in this case, a 30-day horizon with 95% confidence bounds (Example 6-50). The `forecast_timestamp` is that date (or timestamp), and `forecast_value` is the model's best point estimate of your target at that moment—for example, the expected number of trips.

Example 6-50. Generate future predictions

```
SELECT
  forecast_timestamp,
  forecast_value,
  standard_error,
  confidence_level,
  prediction_interval_lower_bound,
  prediction_interval_upper_bound
FROM ML.FORECAST(
  MODEL `oreilly.trip_volume_forecast`,
  STRUCT(30 AS horizon, 0.95 AS confidence_level)
)
ORDER BY forecast_timestamp;
```

The `standard_error` expresses how uncertain that point estimate is. It is the square root of the forecast variance, and it usually grows as you look further ahead.

The remaining fields describe uncertainty as an interval where `confidence_level` simply repeats the coverage you requested (for example, 0.95). Given that level, `prediction_interval_lower_bound` and `prediction_interval_upper_bound` form the range within which the actual future value is expected to fall. Longer horizons and higher confidence levels produce wider intervals, reflecting increasing uncertainty.

> **TIP**
>
> If you want to add external drivers (e.g., `avg_duration`, `is_weekend`), switch to `ARIMA_PLUS_XREG`. In that case `ML.FORECAST` requires a third argument: a table of *future* regressor values covering your forecast horizon as, without those future values, the model cannot produce exogenous-aware forecasts.

Hyperparameter tuning

Hyperparameter tuning searches model settings (*learning rate*, *tree depth*, or *subsampling*) that aren't learned during training so that the final model performs better on a chosen objective. In BigQuery ML you turn this on by adding options to `CREATE MODEL`. The service runs multiple trials, each with a different configuration, and keeps the one that scores best. We can then inspect all trials to understand what worked.

A compact example

In Example 6-51 we tune three high-leverage knobs for a boosted-tree classifier. We ask for 20 trials (two in parallel), let Vizier—Google's optimization and hyperparameter tuning system—explore the space, and optimize for `ROC AUC` using the existing `usertype_classification_features` defined earlier.

Example 6-51. Train with automated tuning

```
CREATE OR REPLACE MODEL `oreilly.usertype_xgb_tuned`
OPTIONS(
  MODEL_TYPE = 'BOOSTED_TREE_CLASSIFIER',
  INPUT_LABEL_COLS = ['usertype'],
  NUM_TRIALS = 20,
  MAX_PARALLEL_TRIALS = 2,
  HPARAM_TUNING_ALGORITHM = 'VIZIER_DEFAULT',
  HPARAM_TUNING_OBJECTIVES = ['ROC_AUC'],
  LEARN_RATE = HPARAM_RANGE(0.01, 0.3),
  MAX_TREE_DEPTH = HPARAM_CANDIDATES([4, 6, 8, 10]),
  SUBSAMPLE = HPARAM_RANGE(0.6, 1.0)
) AS
SELECT
  usertype,
  tripduration,
```

```
  start_hour,
  day_of_week,
  start_station_name,
  distance_m
FROM `oreilly.usertype_classification_features`;
```

With `ML.TRIAL_INFO` (Example 6-52) we can check how each configuration scored and which settings the winner used. If several trials cluster near the top with similar scores, prefer the simpler configuration (shallower trees, higher subsample) for robustness.

Example 6-52. Inspect trials and the chosen hyperparameters

```
SELECT *
FROM ML.TRIAL_INFO(MODEL `oreilly.usertype_xgb_tuned`)
ORDER BY training_loss asc
LIMIT 20;
```

Tuning objectives and multi-objective optimization

BigQuery ML supports multiple objectives for hyperparameter tuning. For classification tasks, you can optimize for metrics like precision, recall, accuracy, F1 score, log loss, or ROC AUC. Regression models can be tuned to minimize various error metrics such as mean absolute/squared error, or maximize R-squared values. When using multiple objectives (Example 6-53), the system finds models that balance the trade-offs between them.

Example 6-53. Multi-objective tuning for trip anomaly detection

```
CREATE OR REPLACE MODEL `oreilly.trip_anomaly_multi_tuned`
OPTIONS(
  MODEL_TYPE = 'BOOSTED_TREE_CLASSIFIER',
  INPUT_LABEL_COLS = ['is_anomaly'],
  NUM_TRIALS = 15,
  MAX_PARALLEL_TRIALS = 3,
  HPARAM_TUNING_ALGORITHM = 'VIZIER_DEFAULT',
  HPARAM_TUNING_OBJECTIVES = ['PRECISION', 'RECALL'],
  MAX_TREE_DEPTH = HPARAM_CANDIDATES([3, 4, 5, 6]),
  MIN_TREE_CHILD_WEIGHT = HPARAM_RANGE(1, 5),
  REG_ALPHA = HPARAM_RANGE(0, 2)
) AS
SELECT * FROM `oreilly.trip_anomaly_features`
WHERE split_col = 'TRAIN';
```

Best practices and cost considerations

Performing hyperparameter tuning is essentially analyzing a trade-off between maximum performance and cost. To ensure that the process is as efficient as possible, consider these principles:

Choose appropriate objectives

Select objectives that align with your business goals. For fraud detection, where false positives are costly, prioritize precision. For medical diagnosis, where missing cases is critical, focus on recall. For balanced performance across different metrics, F1 score provides a good middle ground, while ROC AUC is excellent for ranking and probability calibration tasks.

Set reasonable trial counts

Balance exploration with computational cost by choosing appropriate trial counts. Start with 10–20 trials for initial experiments, scale up to 50–100 trials for production models, and consider the trade-off between parallel trials (faster results) and sequential trials (better optimization).

Use domain knowledge for ranges

Leverage your understanding of the problem to set sensible hyperparameter ranges. For small datasets, limit tree depth to prevent overfitting. For large datasets, allow deeper trees and more complex models. For noisy data, increase regularization parameters to improve generalization.

Multi-objective optimization

When using multiple objectives, understand that the system will find models that balance the trade-offs between them. This is particularly useful when you need to optimize for precision and recall simultaneously.

Hyperparameter tuning increases computational costs since each trial trains a complete model. A practical strategy is to start with a smaller sample of

your data for initial exploration, then refine the hyperparameter ranges and run the final tuning on the full dataset. In Example 6-54, we are quickly exploring whether to perform a k-means clustering or multiple trials but only focusing on 20% (`WHERE RAND() < 0.2`) of the data as a sample guided approach.

Example 6-54. Cost-effective two-phase tuning approach

```
CREATE OR REPLACE MODEL `oreilly.station_clusters_explore`
OPTIONS(
  MODEL_TYPE = 'KMEANS',
  NUM_CLUSTERS = HPARAM_CANDIDATES([3, 5, 7, 10]),
  NUM_TRIALS = 8,
  HPARAM_TUNING_OBJECTIVES = ['DAVIES_BOULDIN_INDEX']
) AS
SELECT * FROM `oreilly.station_clustering_features`
WHERE RAND() < 0.2;
```

Next, we can again use `ML.TRIAL_INFO` to understand which configurations performed best and see if there are significant performance variations with these combinations. Afterwards, if we decide to continue the tuning, we can remove the filtering of 20% and expand it to the full dataset.

When configuring your hyperparameter search, keep in mind that larger `MAX_PARALLEL_TRIALS` values speed things up but can slightly reduce search quality with Vizier's default algorithm. Since trials consume normal training resources, treat the number of trials as your budget. Once tuning is complete, keep the final tuned model fixed for reproducible scoring. Finally, when selecting your optimization objective, choose metrics that align with your business goals: precision for fraud detection, recall for medical diagnosis, or F1 score for balanced performance.

> **TIP**
>
> For a deeper exploration of hyperparameter tuning techniques beyond what SQL-based ML platforms offer, see *Hyperparameter Tuning with Python* by Louis Owen (Packt Publishing), which covers advanced optimization strategies and best practices.

Model Evaluation and Validation

Evaluating a model means more than quoting a single number. It requires summarizing overall performance, inspecting threshold trade-offs (for classifiers), and ensuring estimates will generalize.

Evaluation metrics

In BigQuery ML, `ML.EVALUATE` gives standard metrics on reserved evaluation/test data. This returns headline classification metrics such as accuracy, precision, recall, F1 score, `log_loss`, and `roc_auc`. For regression models, it focuses on mean absolute error, or MAE, which shows typical absolute error, root mean square error (RMSE), which exacts a heavier penalty on large errors, and r2_score, which explains the variance. Consistency between training and evaluation metrics signals healthy generalization.

For classification thresholds, `ML.ROC_CURVE` shows operating points with the ROC table listing many candidate thresholds, each with its recall and false-positive rate, plus supporting counts. This allows scanning to shortlist thresholds that match risk tolerance (for example, preferring higher recall in safety-critical screening).

Moreover, `ML.CONFUSION_MATRIX` can also be leveraged to understand the true/false positives and negatives. Either of these can be used easily by applying them directly on top of trained models, as shown in Example 6-55.

Example 6-55. Different evaluation metrics computation for model evaluation

```
SELECT *
FROM ML.EVALUATE(MODEL `oreilly.usertype_forest`)
-- FROM ML.ROC_CURVE(MODEL `oreilly.usertype_forest`)
-- FROM ML.CONFUSION_MATRIX(MODEL `oreilly.usertype_forest`);
```

Cross-validation

When you run a simple `CREATE MODEL … AS SELECT` in BigQuery ML, the system will, by default, hold out part of your data for evaluation. Concretely, if you do not override split options, BigQuery ML uses

`DATA_SPLIT_METHOD='AUTO_SPLIT'` and randomly reserves a portion of rows as an evaluation set. The size of this evaluation set can be controlled using `DATA_SPLIT_EVAL_FRACTION`, and the training happens on the remaining rows. Previous calls such as `ML.EVALUATE(MODEL …)` will, unless a custom evaluation table is provided, report metrics computed on that reserved evaluation and test data. This is convenient and often sufficient for early iterations.

However, a single random split can also be brittle. To get a more reliable, lower-variance estimate of generalization, you can run k-fold cross-validation (CV). In k-fold CV, you partition the data into *k* equal folds, train *k* separate models (each time leaving one fold out), evaluate each model on its held-out fold and then compute the average/median of the metrics. There is no one-liner for CV in BigQuery ML, but a small amount of SQL—plus a BigQuery Scripting loop—does the job cleanly and reproducibly.

First, divide the dataset into *k* folds as in Example 6-56. In this case, every row is assigned to exactly one of five folds ($k = 5$) using a stable hash of `taxi_id`. We now have a reproducible partition key that can be reused across experiments.

Example 6-56. Splitting the dataset into folds

```
CREATE OR REPLACE TABLE `oreilly.taxi_with_folds` AS
SELECT
  CAST(tips > 0 AS INT64) AS tipped,
  trip_seconds,
  trip_miles,
  payment_type,
  EXTRACT(HOUR FROM trip_start_timestamp) AS pickup_hour,
  -- Stable 5-way partition based on an ID
  ABS(MOD(FARM_FINGERPRINT(CAST(taxi_id AS STRING)), 5)) AS fold
FROM `bigquery-public-data.chicago_taxi_trips.taxi_trips`
WHERE trip_seconds IS NOT NULL
  AND trip_miles  IS NOT NULL
  AND tips        IS NOT NULL
  AND trip_start_timestamp IS NOT NULL
  AND taxi_id     IS NOT NULL;
```

Next we need to implement k-fold CV using a BigQuery Scripting `WHILE` loop. It begins by declaring two variables: `k`, the number of folds, and `f`, the

current fold index starting at zero. The loop will run while `f < k`. In each iteration, the script trains a new model on all rows except those in the current fold `f` and then evaluates that model exclusively on fold `f` rows. The first iteration creates a `results` table, while subsequent iterations append to it. Lastly, it should advance the loop by incrementing `f`. At the end, you have trained `k` independent models, each evaluated on its own held-out fold, and `k` rows of metrics collected in a single table for aggregation.

Starting with the training of all folds except `f`, Example 6-57 creates or replaces a model with a name suffixed by the current fold index. The query selects the label and features from `oreilly.taxi_with_folds` but excludes the current fold via `WHERE fold != f`. Here, the model is trained with `DATA_SPLIT_METHOD = 'NO_SPLIT'`, which prevents BigQuery ML from reserving its own internal holdout. In cross-validation the only holdout is the fold you explicitly set aside.

Example 6-57. Step A: train model on folds ≠ `f`

```
EXECUTE IMMEDIATE FORMAT("""
  CREATE OR REPLACE MODEL `oreilly.taxi_tipped_lr_cv_%d`
  OPTIONS(
    model_type = 'logistic_reg',
    input_label_cols = ['tipped'],
    data_split_method = 'NO_SPLIT'
  ) AS
  SELECT
    tipped,
    trip_seconds,
    trip_miles,
    payment_type,
    pickup_hour
  FROM `oreilly.taxi_with_folds`
  WHERE fold != %d
""", f, f);
```

When `f = 0`, we evaluate the freshly trained model only on rows from `fold = 0` and create the results table `oreilly.cv_results`. This avoids any accidental overwrite in later iterations: you create once and then append thereafter. The evaluation relation must project exactly the label and

features used for training so that scoring and metrics are computed on a consistent schema (Example 6-58).

Example 6-58. Step B: create results table from evaluation on fold `f`

```
EXECUTE IMMEDIATE FORMAT("""
  CREATE OR REPLACE TABLE `oreilly.cv_results` AS
  SELECT %d AS fold, *
  FROM ML.EVALUATE(
    MODEL `oreilly.taxi_tipped_lr_cv_%d`,
    (
      SELECT
        tipped,
        trip_seconds,
        trip_miles,
        payment_type,
        pickup_hour
      FROM `oreilly.taxi_with_folds`
      WHERE fold = %d
    )
  )
""", f, f, f);
```

For `f = 1, 2, 3, 4`, we repeat the evaluation but insert the new metrics into `oreilly.cv_results`. This produces one row per fold (each with precision, recall, F1, `roc_auc`, and related fields) ready to be averaged after the loop completes (Example 6-59). As before, keep the evaluation relation aligned with the training schema.

Example 6-59. Step C: append evaluation metrics for fold `f`

```
EXECUTE IMMEDIATE FORMAT("""
  INSERT INTO `oreilly.cv_results`
  SELECT %d AS fold, *
  FROM ML.EVALUATE(
    MODEL `oreilly.taxi_tipped_lr_cv_%d`,
    (
      SELECT
        tipped,
        trip_seconds,
        trip_miles,
        payment_type,
        pickup_hour
      FROM `oreilly.taxi_with_folds`
      WHERE fold = %d
    )
```

```
  )
""", f, f, f);
```

To wrap all of this into the full initial logic, we can insert each of these steps into the final query (Example 6-60).

Example 6-60. Full cross-validation run

```
DECLARE k INT64 DEFAULT 5;
DECLARE f INT64 DEFAULT 0;

WHILE f < k DO
  -- Step A
  EXECUTE IMMEDIATE FORMAT("...Step A SQL...", f, f);

  IF f = 0 THEN
    -- Step B
    EXECUTE IMMEDIATE FORMAT("...Step B SQL...", f, f, f);
  ELSE
    -- Step C
    EXECUTE IMMEDIATE FORMAT("...Step C SQL...", f, f, f);
  END IF;

  SET f = f + 1;
END WHILE;
```

Finally, we can compute the aggregated results on top of `oreilly.cv_results` (Example 6-61).

Example 6-61. Compute aggregated results of CV

```
SELECT
  AVG(precision)  AS avg_precision,
  AVG(recall)     AS avg_recall,
  AVG(f1_score)   AS avg_f1,
  AVG(roc_auc)    AS avg_roc_auc,
  STDDEV(roc_auc) AS sd_roc_auc
FROM `oreilly.cv_results`;
```

Interpretation of results

The value of a model is increased if its interpretability is high, with a clear understanding of its behavior. This is particularly important when exposing the rationale to business stakeholders, facilitating easier adoption, especially in new cases of implementing autonomous approaches like these. One way to achieve this is to use `ML.EXPLAIN_PREDICT` as seen in

Example 6-62. This returns each row alongside feature-level contributions to its predicted score. We can use it to debug surprising cases ("why did this ride score so high?"), support human review, or generate user-facing rationales where appropriate.

Example 6-62. Explain individual predictions (local attributions)

```
SELECT *
FROM ML.EXPLAIN_PREDICT(
  MODEL `oreilly.taxi_tipped_lr`,
  (SELECT
     CAST(tips > 0 AS INT64) AS tipped,
     trip_seconds,
     trip_miles,
     EXTRACT(HOUR FROM trip_start_timestamp) AS pickup_hour,
     payment_type
   FROM `bigquery-public-data.chicago_taxi_trips.taxi_trips`
   WHERE trip_seconds IS NOT NULL
     AND trip_miles  IS NOT NULL
     AND tips        IS NOT NULL
     AND trip_start_timestamp IS NOT NULL
   LIMIT 50)
);
```

From this single interface, you can obtain other interpretability views by adjusting only the function:

Global explanations

If we trained with `ENABLE_GLOBAL_EXPLAIN=TRUE`, we can switch to `ML.GLOBAL_EXPLAIN(MODEL …)` to aggregate absolute attributions across a dataset. This produces a ranked list of global drivers that complements local explanations.

Coefficients for linear/logistic models

Likewise, we can use `ML.WEIGHTS(MODEL …)` instead to inspect signed coefficients. Large absolute weights indicate strong influence and the sign shows direction (positive or negative effect).

Feature importance for tree-based or imported XGBoost models

Use `ML.FEATURE_IMPORTANCE(MODEL …)` to see split-based importance scores. This allows us to compare the ranking with domain expectations to guard against spurious proxies.

Together, these variants let you move smoothly from individual cases (local) to overall drivers (global), and to nonlinear summaries (importance), without changing how you assemble data for prediction.

Model Deployment and Monitoring

After a model is trained, the last stage is to deploy it so that other users or services can access it and consume the predictions. After deployment, you need to establish processes that enable performance monitoring to ensure a reactive response once the model starts degrading.

Making models available for predictions

Once a model is trained, the most direct route to value is to score rows with SQL. We can run ad-hoc predictions in a `SELECT`, wrap that in a view for self-serve consumption, or write the scores into a managed table for downstream jobs and business dashboards (Example 6-63). If the application later requires online inference, export the model for serving on Vertex AI—your scoring logic doesn't change; only the hosting surface does.

Example 6-63. Explain individual predictions (local attributions)

```
CREATE OR REPLACE VIEW `oreilly.usertype_scores_latest` AS
SELECT
  'Subscriber' AS label,
  subscriber_prob AS predicted_probability,
  CASE
    WHEN subscriber_prob >= 0.40 THEN 'Subscriber'
    ELSE 'Customer'
  END AS business_decision,
  tripduration,
  distance_m,
  start_station_name,
  start_hour
FROM (
```

```
  SELECT
    tripduration,
    distance_m,
    start_station_name,
    start_hour,
    (SELECT prob
     FROM UNNEST(predicted_usertype_probs)
     WHERE label = 'Subscriber') AS subscriber_prob
  FROM ML.PREDICT(
    MODEL `oreilly.usertype_forest`,
    (
      SELECT
        tripduration,
        distance_m,
        start_station_name,
        start_hour,
        day_of_week
      FROM `oreilly.usertype_classification_features`
      WHERE tripduration IS NOT NULL
        AND distance_m IS NOT NULL
        AND start_station_name IS NOT NULL
    )
  )
);
```

In this query, we execute `ML.PREDICT` to generate predictions and wrap that within another `SELECT` statement to `UNNEST` the results. In this case, given that it is a binary classification problem, we unnest and retrieve only one of the labels because we can derive the probability of the other class (`P(Subscriber) = 1 - P(Customer)`). With that, the final view is created with the particularity of manually defining the threshold (in this case at 40%) directly on the SQL code instead of using the default 50/50 of the algorithm.

Another format of making the model available is to directly export it to Google Cloud Storage (Example 6-64) so that it can be used beyond SQL-based predictions. This is useful if you need to deploy the model for real-time serving, for example through Vertex AI or your own API, since BigQuery itself is geared to batch and analytical use cases. It also makes the model portable as it enables the storage of versions in Cloud Storage, archives old ones, or moves them between projects for reproducibility and auditing. Once exported, the model can be integrated with other tools, such

as TensorFlow or XGBoost-based environments, where you might want to analyze it more deeply, run explainability workflows, or incorporate it into a larger ML pipeline.

Example 6-64. Export model to Google Cloud Storage

```
EXPORT MODEL `oreilly.usertype_forest`
OPTIONS (uri = 'gs://oreilly-ml-models/usertype_forest/');
```

Scheduling batch predictions

Operationally, batch scoring is a recurring `INSERT … SELECT` over fresh rows that writes into a partitioned destination (for cost and hygiene). In this case we would need to create `oreilly.predictions_usertype_daily` beforehand. Scheduling is configured with BigQuery Scheduled Queries (UI, API, or BQ). In practice, we author a single SQL body that filters input rows by the run time and then set the cadence (hourly/daily) in the scheduler. For testing and debugging, we can easily switch the `CURRENT_DATE` by any other date like `DATE('2016-08-24')` (see Example 6-65).

Example 6-65. Scheduling batch predictions

```
DECLARE run_date DATE DEFAULT CURRENT_DATE();

INSERT INTO `oreilly.predictions_usertype_daily`  -- partitioned by score_date
DATE
  (score_date, trip_key, predicted_label, predicted_probability)
SELECT
  run_date AS score_date,
  GENERATE_UUID() AS trip_key,
  business_decision AS predicted_label,
  predicted_probability
FROM (
  SELECT
    CASE
      WHEN subscriber_prob >= 0.40 THEN 'Subscriber'
      ELSE 'Customer'
    END AS business_decision,
    subscriber_prob AS predicted_probability
  FROM (
    SELECT
      -- pull the model's probability for the 'Subscriber' class
      (SELECT prob
```

```
      FROM UNNEST(predicted_usertype_probs)
      WHERE label = 'Subscriber') AS subscriber_prob
  FROM ML.PREDICT(
    MODEL `oreilly.usertype_forest`,
    (
      SELECT
        tripduration,
        distance_m,
        start_station_name,
        start_hour,
        day_of_week
      FROM `oreilly.usertype_classification_features`
      WHERE DATE(starttime) = run_date
        AND tripduration IS NOT NULL
        AND distance_m IS NOT NULL
        AND start_station_name IS NOT NULL
    )
  )
)
);
```

Model monitoring and retraining strategies

In production, two monitoring questions matter most. *Skew* checks whether today's serving features differ materially from the training distribution. *Drift* checks whether serving features are changing over time, even if they still resemble training at a given instant. `ML.VALIDATE_DATA_SKEW` compares a single serving slice to the model's stored training statistics; `ML.VALIDATE_DATA_DRIFT` compares a study window to a baseline window. Both return one row per feature with a divergence score and an `is_anomaly` flag based on configurable thresholds.

Skew

In Example 6-66, this query answers the question: did today's inputs shift relative to what the model learned? It supplies today's rows to `ML.VALIDATE_DATA_SKEW`, which contrasts each feature with the model's training distribution. Flags on categorical features often indicate unseen values (e.g., station names) or rebalanced frequencies; flags on numeric features often indicate range shifts (e.g., durations inflated by an ingest issue). Recommended responses include validating upstream data,

broadening category handling if the change is legitimate, adjusting thresholds temporarily, or scheduling retraining.

Example 6-66. Skew: compare today's serving data to the model's training stats

```
SELECT *
FROM ML.VALIDATE_DATA_SKEW(
  MODEL `oreilly.usertype_forest`,
  (
    SELECT
      tripduration,
      distance_m,
      start_station_name,
      start_hour,
    FROM `oreilly.usertype_classification_features`
    WHERE DATE(starttime) = CURRENT_DATE()
      AND tripduration IS NOT NULL
      AND distance_m   IS NOT NULL
      AND start_station_name IS NOT NULL
  ),
  STRUCT(0.2 AS categorical_default_threshold)
);
```

Drift

Drift focuses on trend rather than one-off shocks. Example 6-67 compares the last seven days on `base` to the prior seven on `study`. A feature marked as `is_anomaly = TRUE` is moving faster than the tolerance allows. Suitable responses include increasing retraining frequency, enriching features to explain the movement (seasonality, weather, campaigns), or temporarily adjusting serving thresholds while additional data accumulates. For low-volume segments, wider windows stabilize the estimate, while for multi-tenant contexts, per-segment runs prevent local issues from being averaged away.

Example 6-67. Drift: last 7 days versus previous 7 days

```
DECLARE run_date DATE DEFAULT CURRENT_DATE();

WITH base AS (
  SELECT
    tripduration,
    distance_m,
    start_station_name,
```

```
    start_hour,
  FROM `oreilly.usertype_classification_features`
  WHERE DATE(starttime) BETWEEN DATE_SUB(run_date, INTERVAL 14 DAY)
                            AND DATE_SUB(run_date, INTERVAL 8 DAY)
    AND tripduration IS NOT NULL AND distance_m IS NOT NULL
),
study AS (
  SELECT
    tripduration,
    distance_m,
    start_station_name,
    start_hour,
  FROM `oreilly.usertype_classification_features`
  WHERE DATE(starttime) BETWEEN DATE_SUB(run_date, INTERVAL 7 DAY)
                            AND DATE_SUB(run_date, INTERVAL 1 DAY)
    AND tripduration IS NOT NULL AND distance_m IS NOT NULL
)
SELECT *
FROM ML.VALIDATE_DATA_DRIFT(
  (SELECT * FROM base),
  (SELECT * FROM study),
  STRUCT(0.2 AS categorical_default_threshold)
);
```

Skew checks function best as go-no-go gates for a given scoring run, while drift diagnostics inform the cadence of retraining and the evolution of the feature set. In practice, skew alerts prevent suspect batches from propagating, and drift trends provide early signals that model assumptions are becoming stale even when headline metrics have not yet degraded.

Thresholds benefit from empirical tuning. A sensible approach is to begin with defaults, then adjust based on several weeks of observed incidents and near-misses, aiming to minimize both false alarms and missed degradations. Sensitivity may differ across features and segments; categorical features with rapid taxonomy growth often warrant more tolerant thresholds than stable numerical measurements.

Results from `ML.VALIDATE_DATA_SKEW` and `ML.VALIDATE_DATA_DRIFT` are best persisted in a small, partitioned monitoring table. Charting divergence values and anomaly flags over time makes spikes and slow upward drifts visible at a glance, and simple rules can trigger alerts on either abrupt excursions or sustained trends. When an alert fires, the remediation—data

correction, encoding update, retraining, or threshold change—should be recorded alongside the event. That record becomes a durable playbook, improving future thresholds and shortening the time from detection to resolution.

Summary

SQL's place in data science is broader than as a staging lane for Python or R. In cloud warehouses, the language reaches from first look to final model: you can reshape messy tables, profile distributions, check assumptions, compare groups, and train models without hauling data elsewhere. That power comes from scale and proximity—engines optimized for columnar, distributed execution—and from SQL's declarative style, which keeps analyses concise and reproducible inside the platform's governance boundaries. The trade-offs are familiar: iterative algorithms feel awkward, the modeling catalog is shallower than open source ecosystems, and visualization remains basic. The craft is knowing when those limits matter.

Across the chapter, we moved from quick diagnostics to defensible inference and then to production `ML.UNPIVOT`s and windows yield compact summaries; correlations and "regression from aggregates" let you reason with only summaries in hand; z-scores and percentile trims surface outliers that would warp conclusions. Normality checks, confidence intervals, t-tests and Mann–Whitney, hand-rolled ANOVA, and Kruskal–Wallis showed that rigorous testing is possible in pure SQL or with small UDFs. Feature engineering stays close to the data: one-hot for small cardinals, frequency and hashing for large ones, tokenization and character n-grams for text, quantile bins for skew, leakage-aware target encoding for stability, and PCA when you need compact, decorrelated signals.

On the modeling side, we trained and explained classifiers, segmented with k-means, flagged anomalies with autoencoders, built recommenders with matrix factorization, and forecasted with ARIMA-family models—tuned, evaluated, and cross-validated in place. Deployment is just another query: wrap `PREDICT` in a view, schedule batch scores, export when you need

online serving, and watch data health with skew and drift checks. The takeaway is pragmatic: keep more of the loop in SQL when scale, governance, and speed-to-insight dominate; switch to Python/R when you truly need their breadth. Used deliberately, SQL can be both launchpad and finish line.

The machine learning landscape continues to evolve. While this chapter focused on classic ML—classification, clustering, forecasting—a new wave of capabilities has emerged: large language models, embeddings, and generative AI. These tools don't just predict labels. They generate text, understand semantics, and retrieve knowledge. In the next chapter, we explore how modern SQL platforms are bringing these generative AI capabilities directly into the warehouse, and how SQL itself becomes a powerful lever for agentic workflows—where autonomous systems reason, retrieve, and act over governed data using the same declarative language we've been refining throughout this book.

Part III. SQL in Emerging Technologies

Chapter 7. Generative AI in SQL

In Chapter 6, we focused on SQL as a first-class environment for analytics and classic machine learning. We saw how platforms like BigQuery, Snowflake, and Redshift can train, evaluate, and serve models directly in the warehouse without shipping data out. Over the last few years, a new capability has joined that toolbox: generative AI applied as close as possible to governed data and existing analytical workloads.

This chapter focuses on GenAI workloads in SQL, primarily powered by large language models (LLMs): generating and interpreting text, working with embeddings and vector search, and orchestrating multistep retrieval and reasoning workflows. These capabilities change both what we build and how we operate it. Instead of predicting a numeric label or probability, we are often generating free-form text, grounding answers in recent data, and calling out to powerful external models. This introduces concerns such as prompt design, hallucination control, evaluation, and safety that touch architecture, governance, and cost as much as they touch modeling. The emphasis is on practical patterns and guardrails to run these workloads reliably from SQL over governed data.

Warehouses are rapidly adding native support for LLMs, embeddings, and vector search. Instead of exporting data into ad hoc scripts or separate services, teams can now summarize support tickets, classify feedback, extract entities, retrieve relevant documents, and even generate draft responses using SQL queries that run next to the source tables. This shift turns the warehouse into more than a system of record: it becomes the central engine for both analytical and GenAI workloads.

Where these workloads run also matters. SQL-native designs keep GenAI close to governed warehouse data, while hybrid and external approaches trade that tight coupling for additional flexibility and different latency profiles. A recurring theme in this chapter is deciding when SQL should be

the home for GenAI logic and when to lean on services outside the warehouse.

Two related terms are used throughout. *SQL-native GenAI* refers to LLMs, embeddings, and vector search invoked inside SQL queries. *Warehouse-native GenAI* refers to stateful, auditable workflows built from tables, jobs, and metadata (operational artifacts defined in "Productionizing Generative AI in the Data Warehouse"), even when model calls run through remote endpoints.

Because vendor features evolve especially quickly in this field, we also make some light assumptions about the warehouse capabilities available to you. The concrete syntax in examples leans toward BigQuery-style SQL because of its mature support for in-database machine learning and LLMs, but the ideas generalize to other modern warehouses. This chapter uses some combination of the following capabilities. If your platform only provides a subset, you can still reuse most patterns with minor adaptations:

LLM functions in SQL

: Built-in or externalized functions that can generate, summarize, classify, or extract structured fields from text directly in SQL. Examples include BigQuery's `ML.GENERATE_TEXT` or warehouse functions that wrap managed LLM endpoints. Where native functions are not available, similar behavior can often be exposed via user-defined functions, stored procedures, or remote services that read from and write back to warehouse tables.

Embedding generation

: The ability to turn text into fixed-length numeric vectors using SQL-accessible functions such as `ML.GENERATE_EMBEDDING` or equivalent model endpoints. In BigQuery, these vectors are often stored in regular tables as `ARRAY<FLOAT64>` columns. This is BigQuery's `ARRAY<T>` syntax, where `T` is the element type, so `ARRAY<FLOAT64>` means "an array of 64-bit floating-point values"; other platforms may use different type names or syntax.

Vector search/approximate nearest neighbor

> Functions or extensions that support top-k similarity search over embeddings inside, or tightly coupled to, the warehouse. BigQuery's `VECTOR_SEARCH`, Snowflake's and Redshift's vector capabilities, or vendor-managed approximate nearest neighbor (ANN) indexes all fall into this category. At smaller scales, even a brute-force distance computation in SQL can be sufficient.

External model calls

> A way to call out to managed LLM or embedding services while keeping data and orchestration anchored in SQL. This includes remote UDFs, federated queries, or pipelines that materialize results back into warehouse tables. It is important both for platforms that lack native LLM features and for scenarios where specialized or proprietary models live outside the warehouse.

Where we introduce example functions such as `ML.GENERATE_TEXT`, `ML.GENERATE_EMBEDDING`, or `VECTOR_SEARCH`, treat them as representative of the broader class of LLM, embedding, and vector search functions in warehouse platforms. Exact syntax and options differ by vendor, but the surrounding SQL patterns (CTEs, joins, filters, and aggregations) are portable.

BigQuery is not the only serious platform for these patterns. Snowflake exposes a broad SQL-native AI surface through Cortex AI functions such as `AI_COMPLETE`, `AI_CLASSIFY`, `AI_EXTRACT`, and `AI_EMBED`, plus retrieval features such as Cortex Search. Databricks offers AI functions such as `ai_query`, `ai_extract`, and `ai_classify`, along with SQL access to Mosaic AI Vector Search. Microsoft Fabric also now exposes built-in AI functions directly in Warehouse SQL, though that surface is newer. Redshift supports adjacent generative workflows as well, especially text-to-SQL through Amazon Q and external-function patterns through Lambda, but its experience is currently less centered on BigQuery-style inline LLM calls in

ordinary SQL queries. We therefore use BigQuery as the running syntax—not because it is uniquely capable, but because its combination of remote models, embeddings, and vector search maps especially cleanly to the end-to-end examples in this chapter.

To keep the discussion concrete and to easily compare these different approaches, we reuse a single running example: a customer-support knowledge base and product catalog managed in the warehouse. The goal is to improve support quality and efficiency using GenAI patterns that largely stay inside SQL, while respecting the same governance and observability practices applied to other analytics. Assume three core tables: `support_tickets` (raw customer interactions), `support_kb_articles` (curated knowledge base content), and `product_catalog` (stable product metadata used for routing and analysis). Throughout this chapter, `kb` in table names is shorthand for "knowledge base.""

These tables, highlighted in Table 7-1, anchor a single running example for the rest of the chapter. Each pattern introduced in "Generative AI Inside SQL" is grounded in this schema, with derived tables such as embeddings and operational artifacts added as needed. Keeping the workload tied to the same entities makes it easier to compare designs, reuse SQL, and understand how to apply the ideas to your own warehouse.

Table 7-1. Running example schema (minimal)

Table	Key columns (illustrative)
`support_tickets`	`ticket_id`, `created_at`, `product_id`, `channel`, `status`, `subject`, `body`
`support_kb_articles`	`article_id`, `updated_at`, `language`, `product_ids`, `title`, `body`
`product_catalog`	`product_id`, `product_name`, `category`, `support_tier`, `is_active`

A typical ticket row might look like Example 7-1.

Example 7-1. Demo of a ticket row

```
{
  "ticket_id": "tkt_1002",
  "product_id": "prod_001",
  "created_at": "2026-01-01T16:22:00Z",
  "subject": "Login timeout after update",
  "body": "After upgrading to v2.3, we see repeated timeouts..."
}
```

> **NOTE**
>
> All code for this chapter is in the book's GitHub repository. The Chapter 7 examples are organized under chapter_07/: 01_setup/, 02_primitives/, 03_patterns/, and 04_production/, matching the flow of the chapter from setup through production patterns. Following that structure makes it easier to build the examples step-by-step and reuse the same setup across the chapter.

Generative AI Inside SQL

Running GenAI workloads directly in SQL over warehouse data is less about having one more API to call and more about deciding where data moves and how teams collaborate. When LLM and embedding calls live inside SQL, sensitive tables do not need to be exported to ad hoc scripts, existing scheduling and monitoring can be reused, and model outputs become first-class columns that downstream queries and dashboards can join on.

One useful mental model is to treat these calls as ordinary SQL transformations. LLM invocations add new columns to existing rows (e.g., summaries, classifications, extracted entities) much like any other computed field. Embedding functions replace or augment text with numeric vectors, providing an alternative representation of the same underlying entities. Vector search then acts as a similarity join where, instead of "match rows where keys are equal," we ask "find rows that are close in vector space."

With this perspective, LLMs extend existing analytical entities rather than replacing them. A ticket in `support_tickets` can carry both its original text and a short summary. A catalog item in `product_catalog` can carry both a name and an embedding. A knowledge base article can be joined to tickets either on foreign keys or on semantic similarity.

This also changes the kind of questions the warehouse can help answer. Traditional SQL is excellent at telling you *what* moved: ticket volume is up 18 percent, refund rate doubled, or renewal conversions fell after a release. GenAI adds a first-pass layer for *why* it may have moved by synthesizing likely drivers from the surrounding context such as seasonality, product launches, support themes, or policy changes reflected in text. It does not prove causality, and it does not remove the need for careful analyst review, but it can compress work that previously required a data scientist or business analyst to manually connect metrics to narrative explanations. Once you have stable aggregate tables, those explanation fields can be attached at the grain of each row, segment, or time window and stored back in the warehouse as queryable historical artifacts rather than comments scattered across documents or slide decks. This preserves institutional memory and lets analysts spend less time on repeated narrative analysis and more time on deeper causal and correlational work.

Because these operations are expressed as SQL queries, they can run in batch alongside nightly ETL, be materialized as views or incremental tables, and be reasoned about with the same mental models you already use for performance, cost, and reliability. GenAI workloads become "just another transformation step" in a pipeline (powerful, but governed) rather than a black box bolted onto the side. Table 7-2 summarizes the trade-offs between SQL-native and external GenAI approaches.

Table 7-2. SQL-native compared to external GenAI approaches

Aspect	SQL-native GenAI (inside SQL over warehouse data)	External or app-level GenAI
Data movement	Data stays in governed tables, only prompts/results flow through functions.	Often requires exporting data or building custom connectors and caches.
Orchestration and monitoring	Reuses existing SQL jobs, schedulers, and logs. Easy to trace lineage.	Separate job systems and observability, harder to align with warehouse SLAs.
Governance and security	Benefits from row-level security, masking, and audit trails already in place.	Must reimplement or mirror governance controls in external services.
Latency profile	Optimized for batch or scheduled enrichment, seconds to minutes per job.	Better suited for low-latency, interactive experiences and UI-driven flows.
Cost visibility	LLM usage can be tied to SQL jobs and tables, making attribution easier.	LLM spend often appears as a separate service bill, harder to map to datasets.

Core Patterns of SQL + GenAI

Although vendor features differ, most SQL-based GenAI systems follow a small set of recurring workflows that combine the capabilities listed earlier.

Inline LLM enrichment

Inline enrichment uses LLM functions directly in queries to summarize, classify, or extract structured fields from existing rows. These calls are composed with joins, filters, and CTEs like any other function, and their outputs become new columns on the same tables. This is the natural choice when each row (or a small group of rows) can be enriched independently and the primary output is an updated table.

Semantic search with embeddings

Embedding-based search generates numeric vectors for text columns and stores them in vector-capable columns, for example as `ARRAY<FLOAT64>` in BigQuery and then uses vector similarity functions to find semantically related rows. Because embeddings place similar pieces of text close together in a high-dimensional space, queries can retrieve "nearby" articles or tickets even when they share few keywords. This pattern is most useful when keyword search or simple filters miss semantically similar content and you need robust similarity joins, clustering, or deduplication.

Retrieval-augmented generation (RAG)

Retrieval-augmented generation (RAG) chains embedding generation, vector search, and LLM calls so that prompts are grounded in retrieved warehouse data. For example, a pipeline might retrieve relevant `support_kb_articles` by similarity to a ticket and then pass both the ticket and the articles into `ML.GENERATE_TEXT` to propose a draft response. RAG is appropriate when answers must be anchored in governed warehouse content and you want the model to reason primarily over retrieved facts rather than its own training data.

Text-to-SQL

Text-to-SQL patterns translate natural-language questions into SQL queries, typically using schema-aware prompting and validation loops. The warehouse executes (or validates) the generated query, enabling copilot-style interfaces where users ask questions in plain language while still benefiting from the full power of SQL. This is a good fit when users want natural-language interfaces but you need strict control over which schemas and operations are allowed.

SQL as an agent tool

Agent patterns orchestrate multistep reasoning where an LLM plans, calls SQL tools (analytic queries, vector searches, schema inspection), interprets results, and iterates. SQL remains the execution environment. The agent decides when and how to use it, treating queries and vector searches as tools alongside others such as HTTP calls or document parsers. This is the right shape when workflows require iteration, planning, or combining multiple tools beyond a single query.

Two questions cut across these patterns: where the work runs, and how people interact with the result. The next subsection provides a decision framework.

TIP

Inline LLM patterns in SQL are at their best when used for batch or scheduled enrichment jobs, for example annotating tickets or articles overnight. They are not suited for millisecond-level chat latency for end users. Interactive experiences typically rely on hybrid or external architectures that still call back into the warehouse when needed:

```
Inline LLM:    table → LLM() → enriched table
Embeddings:    table → EMBEDDING() → vector table
SQL RAG:       embed → search → LLM() → answer table
Agents:        LLM ⇄ SQL (many turns)
```

These patterns and interaction modes form the backbone of the rest of the chapter. Subsequent sections show concrete SQL implementations, cost and latency guardrails, and production variants for each.

AI-Ready Data

Model quality matters but for retrieval-heavy use cases, data readiness often matters more. GenAI systems perform better when the warehouse is prepared for retrieval, grounding, and tool use. This is less about teaching the model new facts than about giving it a clean, current, and authoritative context surface at runtime. In practice, AI-ready data improves text-to-SQL accuracy, makes RAG retrieval more precise, and gives agents better context for choosing and interpreting queries.

Useful AI-enablement work usually looks familiar to warehouse teams:

- Document tables and columns with business meaning, not just technical names and types
- Curate a small set of representative SQL queries, metric definitions, and common join paths
- Store authoritative knowledge sources such as approved knowledge-base articles, runbooks, and prior DS or BA analyses in retrievable form
- Keep those sources current and versioned so retrieval reflects recent product, policy, and process changes
- Separate reviewed, authoritative context from drafts or untrusted text before it can reach prompts

These assets do not retrain the model. They improve grounding at inference time. That distinction matters. A well-documented warehouse with curated retrieval sources will usually outperform a poorly documented one even when both use the same model.

When SQL Should Run GenAI Workloads

There is no single "correct" place to run GenAI workloads, but you can usually start by looking at the artifact you are trying to ship and the latency you need.

If the deliverable is a table or view (summaries, triage labels, embeddings), keep the pipeline SQL-native. Do filtering and joins in the warehouse, and call LLM or embedding functions only on curated rows. This tends to be the simplest approach to schedule and govern.

If the deliverable is an interactive assistant or UI, treat SQL as a tool in a hybrid architecture. Store state (tables, embeddings) in the warehouse, but run multistep orchestration, caching, and integration in the application layer.

If you require subsecond latency, cross-system retrieval, or model hosting that the warehouse cannot provide, you will often move the GenAI runtime to be external and use SQL for curated inputs and grounding content.

The next sections turn these ideas into concrete SQL examples for inline LLMs, embeddings, and small SQL-native RAG workflows.

Primitives: LLMs, Embeddings, and Search

In Chapter 6, we treated SQL as a serious environment for machine learning: we created models, evaluated them, and served predictions without exporting data into separate training scripts. SQL-native GenAI extends that idea, but with different ergonomics. Instead of producing a single numeric prediction, we are often producing free-form text or turning text into vectors that can be searched.

This can feel like "just another API call," but the shift is bigger than that. When GenAI primitives are available inside SQL, they become part of a governed data workflow. Inputs are tables, outputs are columns, and the process can be versioned, audited, and scheduled like any other transformation.

To keep this section concrete, we continue with the running support example and work through three tasks that show up in almost every support organization: summarize recent tickets by product, extract triage labels as typed columns for routing and reporting, and retrieve the most relevant knowledge base passages for a ticket (optionally followed by a grounded draft reply).

Although the model calls are new, the surrounding design follows standard warehouse patterns. Use CTEs to shape data, aggregation to control cost, and schema design to keep outputs reliable.

LLM Functions Inside SQL

Most warehouses expose text generation through models or functions that you register once and then reuse from queries. In BigQuery, for example, you can define a remote model that connects directly to Vertex AI's Gemini models without any training step in the warehouse itself. Other platforms expose similar capabilities as built-in functions or external UDFs that proxy to managed LLM endpoints.

An LLM call is an expensive, probabilistic transformation. The SQL engine cannot push predicates into an external model call the way it pushes filters into a scan, so we must shape the input set before invoking the model.

We will start with a "human-facing" output (summaries) and then move to the more warehouse-friendly pattern: structured extraction into columns.

Example 7-2 shows a onetime setup for a remote model connected to Gemini that will be used in later examples by simply referring to that model from `ML.GENERATE_TEXT` calls.

Example 7-2. Creating a remote model connected to Gemini

```
CREATE OR REPLACE MODEL `oreilly_support.gemini_flash`
REMOTE WITH CONNECTION `us.vertex_connection`
OPTIONS(ENDPOINT = 'gemini-2.5-flash');
```

This statement creates a remote model named `gemini_flash` in the `oreilly_support` dataset. The `REMOTE WITH CONNECTION` clause specifies

a Cloud resource connection that BigQuery uses to communicate with Vertex AI. The `ENDPOINT` option identifies the specific Gemini model to use, in this case `gemini-2.5-flash`, a fast and cost-effective model suitable for summarization and classification tasks. For a complete list of available Gemini models and their capabilities, see the Vertex AI model documentation.

Remote models do not run inside the warehouse execution engine: each call invokes an external service, with latency and token pricing governed by the provider. They still feel SQL-native because orchestration, permissions, and logging are handled by the warehouse, and the inputs and outputs are regular tables and columns. The connection `us.vertex_connection` must be created once per project using the BigQuery console or `bq mk` command. It requires the Vertex AI User role to call Gemini models. For details on creating connections, see the BigQuery documentation and Vertex AI permissions documentation.

Generating free-form text

If you are introducing GenAI into a warehouse team, summarization is a good first workload because it is easy to evaluate qualitatively: does the summary capture what a support manager would notice when skimming tickets? It is also a good fit for scheduled execution. Many teams start by generating a "weekly issues" table per product and joining it into existing dashboards, keeping the output close to the analysts and stakeholders who already live in SQL.

With the model registered, the workflow has two steps: shape the inputs into one row per product (Example 7-3) and then call the model once per product to generate a small summary artifact (Example 7-4).

Example 7-3. Summarizing support tickets: shape inputs (one row per product)

```
WITH tickets_for_summary AS (
  SELECT ticket_id, product_id, created_at, subject, body
  FROM `oreilly_support.support_tickets`
  WHERE created_at >= TIMESTAMP_SUB('2024-01-14', INTERVAL 7 DAY)
),
```

```
grouped AS (
  SELECT
    product_id,
    COUNT(*) AS ticket_count,
    STRING_AGG(
      CONCAT('Subject: ', subject, '\nBody: ', body),
      '\n\n---\n\n'
      ORDER BY created_at DESC
    ) AS ticket_text
  FROM tickets_for_summary
  GROUP BY product_id
)
```

This step does the warehouse-native work such as filter, group, and shape raw tickets into a prompt-friendly input table. Keeping it separate makes it easy to inspect the exact text that will be sent to the model.

Example 7-4. Summarizing support tickets: generate summaries (one model call per product)

```
SELECT
  product_id,
  ticket_count,
  (SELECT ml_generate_text_llm_result FROM ML.GENERATE_TEXT(
    MODEL `oreilly_support.gemini_flash`,
    (SELECT FORMAT(
      CONCAT(
        'You are a support triage assistant. ',
        'Summarize the main issues customers report for product %s ',
        'over the last week in 3 bullet points.\n\n',
        'Tickets:\n%s'
      ),
      product_id,
      ticket_text
    ) AS prompt),
    STRUCT(256 AS max_output_tokens, TRUE AS flatten_json_output)
  )) AS issue_summary
FROM grouped;
```

On this second step, we generate summaries by injecting `product_id` and `ticket_text` into a prompt template. The `ML.GENERATE_TEXT` call is a scalar function that returns a single string per input row. In this case, we are calling it once per product, so the output is one summary per product.

The third argument is written as `STRUCT(...)`, which is BigQuery's syntax for constructing an inline typed record. You can read `STRUCT(256 AS max_output_tokens, TRUE AS flatten_json_output)` as "one small record with two named fields." This style is common in BigQuery and some newer analytical engines, but it is not universal SQL syntax: other platforms may express the same idea with named parameters, JSON/object literals, or separate option clauses.

Prompt length directly affects cost, latency, and model limits. The example above aggregates all ticket text per product for clarity, but real pipelines typically cap the number of tickets, truncate long bodies, or sample representative subsets before calling the model. Treat LLM-heavy queries like other expensive warehouse transformations: schedule them thoughtfully, monitor usage, and apply limits to avoid runaway costs.

The examples above are deliberately written like a warehouse transformation with one expensive step: the LLM call. SQL does the high-cardinality work (filtering, grouping), and the model runs on only a small number of aggregated rows, producing an artifact you can join into existing dashboards.

Some design choices to follow include:

Input shaping

: Selects only needed columns and restricts to a recent time window

Bounded cost

: Caps tickets per product so prompt size stays predictable

Operational artifact

: Materializes the output keyed by `(product_id, run_date)` so you can trend summaries over time and reproduce what the model saw

Structured outputs

Summaries are useful, but warehouses earn their keep when they produce structured datasets. A support operation needs more than a digest. It needs to route work. A dashboard of open tickets by category, escalation rules (e.g., “high severity notification issues go to the on-call”), and trend quantification (e.g., “login timeouts tripled after the last release”) all rely on stable, typed columns that SQL can group and filter.

This is where structured LLM outputs matter. The model can still read messy text, but instead of asking it for prose, we ask it to fill a small schema that SQL can store as typed columns. Where your warehouse supports it, prefer schema-guided generation (for example BigQuery’s `AI.GENERATE_TABLE`) because it reduces parsing fragility and gives you a cleaner place to attach validation.

Two simple heuristics go a long way in making this reliable. First, constrain the output space: an explicit schema and small allowlists in the body of the prompt. Second, reduce randomness by lowering the temperature and shortening the output. Temperature is the dial for creativity, so prefer low values (0.2 or below) when you need consistency. For enrichment jobs, you typically want consistency over creativity, hence the low temperature. Shorter outputs are also more predictable and less likely to exceed token limits.

The workflow has four steps. First, create a “safe” input view for model calls (Example 7-5). Then, build prompts (Example 7-6), generate typed fields with an output schema (Example 7-7), and enforce allowlists and defaults in SQL (Example 7-8).

Example 7-5. A safe input view for ticket triage

```
CREATE OR REPLACE VIEW `oreilly_support.support_tickets_llm_input` AS
SELECT
  ticket_id,
  created_at,
  channel,
  product_id,
  status,
  subject,
  REGEXP_REPLACE(
    body,
```

```
    r'[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}',
    '[REDACTED_EMAIL]'
  ) AS body
FROM `oreilly_support.support_tickets`;
```

The view `support_tickets_llm_input` is the contract for what the model is allowed to see. It keeps the triage job focused (only the needed columns) and gives you a clear place to apply lightweight redaction before any model call. Treat this as a convenience guardrail, not a substitute for row-level security, column masking, or dataset-level access controls.

Example 7-6. Ticket triage: building `prompt_data`

```
CREATE TEMP TABLE triage_prompt_data AS
SELECT
  ticket_id,
  CONCAT(
    'You are a support triage system. ',
    'Analyze the ticket and respond with ONLY valid JSON.\n\n',
    'Required fields:\n',
    '- issue_category: must be exactly one of: ',
    '"login_timeout", "performance", "notifications", "other"\n',
    '- severity: must be exactly one of: "low", "medium", "high"\n',
    '- needs_engineering: must be true or false\n',
    '- short_rationale: a brief one-sentence explanation ',
    '(keep under 100 characters)\n\n',
    'Ticket Details:\n',
    'Product ID: ',
    REGEXP_REPLACE(IFNULL(product_id, 'N/A'), r'[\n\r\t"]', ' '),
    '\n',
    'Subject: ', REGEXP_REPLACE(IFNULL(subject, 'N/A'), r'[\n\r\t"]', ' '), '\n',
    'Body: ', REGEXP_REPLACE(IFNULL(body, 'N/A'), r'[\n\r\t"]', ' '), '\n\n',
    'Respond with only the JSON object, no additional text.'
  ) AS prompt
FROM `oreilly_support.support_tickets_llm_input`
WHERE status IN ('open', 'in_progress');
```

Example 7-6 makes the model call inspectable. You can sample prompts, check for accidental data leakage, and tune the instruction format without running the model. Sanitizing newlines and quotes reduces accidental JSON breakage and keeps the prompt shape stable across messy ticket text.

Example 7-7. Ticket triage: generating typed rows with `AI.GENERATE_TABLE`

```
CREATE TEMP TABLE triage_generated AS
SELECT *
FROM AI.GENERATE_TABLE(
  MODEL `oreilly_support.gemini_flash`,
  (SELECT ticket_id, prompt FROM triage_prompt_data),
  STRUCT(
    '''issue_category STRING, severity STRING,
    needs_engineering BOOL, short_rationale STRING''' as output_schema,
    512 AS max_output_tokens,
    0.1 AS temperature
  )
);
```

`AI.GENERATE_TABLE` is useful when you want structured outputs without fragile JSON parsing. The model still makes mistakes, but the contract is clearer as it returns four fields with the requested types. You still keep the temperature low because this is a routing signal, not a creative writing task.

Example 7-8. Ticket triage: enforcing allowlists and defaults in SQL

```
SELECT
  ticket_id,
  IF(
    issue_category IN ('login_timeout','performance','notifications','other'),
    issue_category,
    'other'
  ) AS issue_category,
  IF(
    severity IN ('low','medium','high'),
    severity,
    'medium'
  ) AS severity,
  IFNULL(needs_engineering, FALSE) AS needs_engineering,
  SUBSTR(
    IFNULL(short_rationale, 'Error processing ticket'),
    1,
    100
  ) AS short_rationale
FROM triage_generated;
```

The last step is where SQL earns its keep. Even if the model returns a surprise value, the allowlists ensure your warehouse tables contain only known tags. Downstream logic becomes predictable again when dashboards group cleanly, routing rules do not fail on unexpected strings, and you can treat “defaulted” rows as an observable error mode to monitor.

A `CREATE TEMP TABLE` statement builds the table but produces no visible output. To inspect the results, run a follow-up query such as `SELECT * FROM triage_prompt_data LIMIT 10;` in the same session.

Structured outputs are only as reliable as the constraints around them. Even with an output schema, models can return `NULL` or values outside your allowlists. In production, treat the model result as an intermediate artifact. Enforce allowlists in SQL, log failures for review, and use sensible defaults so downstream workflows do not break.

> **TIP**
>
> These patterns share a few practical guardrails. Minimize what you send to the model by filtering early, capping text, and aggregating so you call the model once per entity rather than once per event. Treat every model response as untrusted. Prefer low temperature, short outputs, and explicit schemas. Parse defensively with `SAFE_CAST`. Log raw outputs and define fallbacks so downstream workflows stay stable when parsing fails.

Embeddings, Search, and Retrieval

LLM calls produce text, but they work best when you already know which text to put in the prompt. The challenge is finding that text in the first place. Keyword search helps when the words match, but often the most relevant knowledge base article uses different phrasing from the customer's ticket. Embeddings solve this by letting you search by meaning rather than by exact terms. In a warehouse, that search becomes just another SQL join.

Embeddings and retrieval are the connective tissue between messy text and SQL. This section moves from mental model to data model to query-time workflow. First, we treat embeddings as join keys so similarity behaves like a SQL primitive. Next, we design a stable embeddings table so retrieval is reproducible and cheap to refresh. Then we show query-time retrieval, the step that decides what text a model will see and what you can cache and reuse (see "Logging, Monitoring, and Evaluating GenAI Outputs" for logging). Read these subsections as a pipeline: weak chunking or sloppy metadata will surface as noisy retrieval later.

Embeddings as semantic join keys

A human support agent finds the right knowledge base article by memory and intuition ("this ticket sounds like the login timeout issue."). Embeddings approximate that process by using an embedding model that converts a piece of text into a fixed-length vector of numbers. You can think of that vector as a compact "semantic fingerprint." Texts with similar meaning tend to land near each other in embedding space even when the exact words differ. Once you can represent text as vectors, the warehouse can do something very SQL-like with it by joining and ranking by similarity.

Keep this mental model in mind. Vector search behaves like a top-k join where the join condition is "closest vectors," and the output is a small table of nearest neighbors. From there, the rest is standard SQL. You can filter by metadata, join back to source tables, aggregate results, and materialize them for downstream use.

In a warehouse, embeddings become powerful because they are data, not just model inputs. Once you store an `embedding` column alongside IDs and metadata, you can build semantic search views, perform similarity joins (e.g., tickets ↔ articles), cluster or deduplicate noisy text, and keep the results governed and queryable like any other dataset.

Most modern warehouses represent embeddings as an array or vector of floats and provide a vector search primitive that can efficiently return nearest neighbors for a query vector. In BigQuery, for example, `ARRAY<FLOAT64>` means "an array whose elements are `FLOAT64`." When those primitives are not available, you can still implement the same logic at smaller scales by computing a distance function and ordering by it. The concept stays the same; only the performance changes.

As with remote LLM models, embedding generation typically starts by registering a remote model endpoint once. Example 7-9 creates a remote model connected to an embedding endpoint. You will reuse it for both "offline" embedding pipelines (embed all knowledge base [kb] articles) and "query-time" embedding (embed a new ticket or user query).

Example 7-9. Creating a remote embedding model

```
CREATE OR REPLACE MODEL `oreilly_support.text_embedding`
REMOTE WITH CONNECTION `us.vertex_connection`
OPTIONS(ENDPOINT = 'text-embedding-004');
```

The pattern mirrors Example 7-2, but the endpoint is an embedding model rather than a generative model. The key difference in practice is cost and usage: embedding pipelines are usually batch jobs that you run when documents change, whereas generation is often triggered by downstream workflows (like RAG, and copilots) and needs tighter prompt controls.

Embeddings are not interchangeable across models. A vector produced by one embedding model should not be compared to a vector produced by a different model, even if both are "text embedding" models. Store the embedding model identifier alongside the vector and treat model upgrades as data migrations, not as invisible swaps.

Designing an embeddings table

Before generating embeddings, you need to decide what a "document" is from the model's perspective. For retrieval, the typical unit is not an entire article but a chunk of text that is small enough to fit into an answer context.

Chunk size and overlap drive the main precision-versus-cost trade-off. Larger chunks reduce storage and indexing overhead, but they can dilute relevance because each vector represents more content. Smaller chunks improve retrieval precision but increase table size and refresh cost. A useful starting point is "a few hundred tokens per chunk," with a small overlap (for example 10–20% of chunk length) to reduce boundary effects where a relevant sentence falls between chunks.

Once you pick a chunking strategy, make it operational. Give each chunk a stable `chunk_id` so you can rerun the pipeline idempotently, store a content hash so you only re-embed when the chunk changes, and record `embedding_model` so model upgrades are explicit migrations rather than invisible swaps.

Finally, think about physical layout. For small corpora, brute-force search can work, but at larger scales you will want vendor-provided ANN/vector indexes and a table layout (clustering or partitioning) that matches how you filter retrieval by document ID, language, tenant, or model version. However, you can also do it with open source approaches like Faiss or HNSWlib if your warehouse does not natively support vector search.

The prototype has two steps: chunk kb articles into stable retrieval units (Example 7-10), then embed those chunks and materialize a single embeddings table you can query with vector search (Example 7-11). "Productionizing Generative AI in the Data Warehouse" shows how to productionize the same workflow with incremental refresh, logging, and evaluation.

Example 7-10. Chunking knowledge base articles into stable chunks (prototype)

```
CREATE TEMP TABLE kb_article_chunks AS
WITH source_articles AS (
  SELECT article_id, body, language, updated_at
  FROM `oreilly_support.support_kb_articles`
  WHERE language = 'en'
)
SELECT
  article_id,
  language,
  updated_at,
  offset AS chunk_index,
  CONCAT(article_id, ':', LPAD(CAST(offset AS STRING), 4, '0')) AS chunk_id,
  TRIM(paragraph) AS chunk_text,
  TO_HEX(MD5(TRIM(paragraph))) AS chunk_hash
FROM source_articles,
  UNNEST(SPLIT(body, '\n\n')) AS paragraph WITH OFFSET
WHERE TRIM(paragraph) != '';
```

The `UNNEST(SPLIT(body, '\n\n')) ... WITH OFFSET` expression splits each article body on double-newlines and returns each paragraph as a separate row, with `offset` providing the zero-based position. The `CONCAT(..., LPAD(...))` builds a sortable chunk identifier by zero-padding the offset to four digits (e.g., `article_123:0007`), ensuring chunks sort lexicographically in the correct order. This chunking step makes the

retrieval unit explicit. You can inspect `kb_article_chunks`, tune chunk boundaries, and verify that identifiers and hashes behave as expected before incurring embedding costs.

Example 7-11. Embedding kb chunks and materializing `kb_article_embeddings` (prototype full refresh)

```
CREATE OR REPLACE TABLE `oreilly_support.kb_article_embeddings`
CLUSTER BY article_id, embedding_model AS
SELECT
  article_id,
  chunk_id,
  chunk_index,
  content,
  chunk_hash,
  language,
  updated_at,
  ml_generate_embedding_result AS embedding,
  'text-embedding-004' AS embedding_model,
  CURRENT_TIMESTAMP() AS embedded_at
FROM ML.GENERATE_EMBEDDING(
  MODEL `oreilly_support.text_embedding`,
  (SELECT
    article_id,
    chunk_id,
    chunk_index,
    chunk_text as content,
    chunk_hash,
    language,
    updated_at
  FROM kb_article_chunks)
);
```

This second step is intentionally prototype-friendly. It prioritizes clarity over refresh efficiency by rebuilding the English kb embeddings in one job. That is often the fastest path to a useful semantic search view, and it keeps the moving pieces small while you tune chunking and filters.

Here are three design choices:

Stable chunk IDs

> `chunk_id = article_id:chunk_index` makes retrieval results traceable and makes reruns idempotent.

Hashes as change detectors

`chunk_hash` gives you a cheap signal for incremental refresh later.

Explicit versioning

Storing `embedding_model` makes model upgrades visible data migrations, not silent swaps.

Chunking is where retrieval quality is won or lost. Paragraphs are a reasonable starting point for short documents, but long articles usually need fixed-size chunks (often a few hundred tokens) with overlap. The best chunking strategy depends on your content (tickets, docs, chat logs), your latency and cost budget.

Query-time semantic retrieval with vector search

Query-time retrieval is the operational boundary between your corpus and any downstream generation. It decides what the model can see, so it governs answer quality, latency, and compliance. Treat retrieval as its own logged artifact, not a hidden subquery.

With embeddings materialized, semantic retrieval becomes a normal query. The workflow has three steps: define the ticket and its query vector (Example 7-12), run vector search to retrieve candidate chunks (Example 7-13), then materialize a small retrieval result table you can join to downstream jobs (Example 7-14).

Example 7-12. Vector search: ticket and query vector

```
CREATE TEMP TABLE ticket AS
SELECT
  ticket_id,
  product_id,
  CONCAT(subject, '\n', body) AS content
FROM `oreilly_support.support_tickets`
WHERE ticket_id = 'tkt_1002';

CREATE TEMP TABLE query_vec AS
SELECT
  (SELECT ticket_id FROM ticket) AS query_id,
```

```
  ml_generate_embedding_result AS embedding
FROM ML.GENERATE_EMBEDDING(
  MODEL `oreilly_support.text_embedding`,
  (SELECT content FROM ticket)
);
```

This first step keeps the "query" explicit. The ticket becomes a single row with a `content` field, and `query_vec` becomes a one-row table with an embedding you can reuse. That reuse matters in production, where you often cache query embeddings or log them alongside retrieval runs.

Example 7-13. Vector search: retrieve candidate chunks

```
CREATE TEMP TABLE retrieved AS
SELECT
  query.query_id AS query_id,
  base.article_id AS article_id,
  base.chunk_id AS chunk_id,
  base.content AS content,
  distance
FROM VECTOR_SEARCH(
  (SELECT article_id, chunk_id, content, embedding
   FROM `oreilly_support.kb_article_embeddings`
   WHERE embedding_model = 'text-embedding-004'
     AND language = 'en'),
  'embedding',
  (SELECT query_id, embedding FROM query_vec),
  'embedding',
  top_k => 50,
  distance_type => 'COSINE'
);
```

`VECTOR_SEARCH` takes two table expressions, one for the corpus (here, the kb embeddings filtered to English and the current model) and one for the query vector, and returns the nearest neighbors along with a `distance` score. The second and fourth arguments (`'embedding'`) name the embedding column in each table. The `top_k` parameter controls how many neighbors to return per query row, and `distance_type` specifies the similarity metric (cosine distance in this case, where lower values mean higher similarity). This step produces a table of candidates keyed by `query_id`. Keeping `top_k` larger than the final display size leaves room to

apply additional SQL filters (product scoping, tenant boundaries, approvals) without accidentally filtering away all nearest neighbors.

Example 7-14. Vector search: retrieval result table (ready for joins)

```
CREATE OR REPLACE TABLE `oreilly_support.ticket_kb_retrieval` AS
WITH params AS (SELECT GENERATE_UUID() AS run_id)
SELECT
  (SELECT run_id FROM params) AS run_id,
  (SELECT ticket_id FROM ticket) AS ticket_id,
  (SELECT product_id FROM ticket) AS product_id,
  r.article_id,
  r.chunk_id,
  r.distance,
  r.content
FROM retrieved r
JOIN `oreilly_support.support_kb_articles` a USING (article_id)
WHERE (SELECT product_id FROM ticket) IN UNNEST(a.product_ids)
ORDER BY r.distance
LIMIT 5;
```

Retrieval produces a table. Once you have `ticket_kb_retrieval`, the output is joinable and loggable like any other dataset, and it has the shape you need for downstream generation and evaluation. The final table should be something similar to Example 7-15.

Example 7-15. Demo of a retrieval result table

```
run_id | ticket_id | product_id | article_id | chunk_id     | distance |
chunk_text
...    | tkt_1002  | prod_001   | kb_0442    | kb_0442:0003 | 0.12     | ...login
ti
```

In this result, `distance = 0.12` indicates high similarity, meaning the chunk likely addresses the ticket's issue directly. Cosine distances above 0.4 are often noise. Filtering to `distance < 0.35` typically improves precision without losing relevant matches.

Retrieval is usually the part worth iterating on. Pure vector similarity is a strong default, but it can miss cases where exact terms matter. Error codes, product names, and version strings are often better handled with a hybrid signal.

Hybrid search (lexical + vector + reranking)

Hybrid search combines two different notions of relevance. Lexical signals use exact term matches or keyword overlap, which is strong for error codes, product names, and version strings. Vector signals use embeddings to capture semantic similarity, which is strong for paraphrases and fuzzy phrasing. In practice, lexical signals come from simple term-based scoring like “how many query words appear in this chunk” or “does this exact phrase appear,” while vector signals come from embedding distance or similarity. A hybrid approach uses both so you do not have to choose between exactness and meaning.

There are multiple shapes for hybrid search. Some systems fuse scores from two retrievers, while others use one signal as a filter and another as a reranker. In SQL, the most practical shape is a cheap lexical step that narrows the search space, followed by vector similarity, then a lightweight score that blends the two. That is the pattern shown here.

Hybrid retrieval usually has three stages. A cheap lexical step builds a bounded candidate set, vector search ranks within that set, and a simple reranker combines the signals into a final ordering.

Example 7-16 shows the lexical step as two short, inspectable queries where we normalize a query into tokens. Example 7-17 then scores kb chunks using token overlap and keeps the top candidates.

Example 7-16. Hybrid retrieval: stage 1a, normalizing query tokens

```
CREATE TEMP TABLE query_tokens AS
WITH params AS (
  SELECT 'login timeout after update' AS query_text
)
SELECT token
FROM UNNEST(
  SPLIT(
    REGEXP_REPLACE(
      LOWER((SELECT query_text FROM params)),
      r'[^a-z0-9 ]',
      ' '
    ),
    ' '
  )
) AS token
WHERE token != '' AND LENGTH(token) >= 3;
```

The `query_tokens` table is a small normalization step. It lowercases the query, strips punctuation, splits on spaces, and drops short tokens. This is not meant to be a perfect tokenizer, just a stable set of terms that are cheap to reuse across queries and easy to inspect when results look surprising.

Example 7-17. Hybrid retrieval: stage 1b, lexical candidates and score

```
CREATE TEMP TABLE lexical_candidates AS
SELECT
  e.article_id,
  e.chunk_id,
  e.content,
  (
    SELECT COUNTIF(
      REGEXP_CONTAINS(
        LOWER(e.content),
        CONCAT(
          r'(^|[^a-z0-9])',
          REGEXP_REPLACE(
            t.token,
            r'([\\.^$|?*+(){}\[\]-])',
            r'\\\1'
          ),
          r'([^a-z0-9]|$)'
        )
      )
    )
    FROM query_tokens t
  ) AS lexical_score
FROM `oreilly_support.kb_article_embeddings` e
JOIN `oreilly_support.support_kb_articles` a
  ON a.article_id = e.article_id
WHERE e.language = 'en'
  AND e.embedding_model = 'text-embedding-004'
;
```

The `lexical_candidates` table then scores chunks by counting how many query tokens appear as whole-word matches. This is intentionally crude, but it works well for terms such as product names and error strings. The score gives you a simple knob: you can carry all rows forward, or you can filter and cap a candidate set (for example `lexical_score > 0` and `LIMIT`) to keep the next step predictable in cost and latency.

Once you have lexical scores (and, optionally, a bounded candidate set), vector similarity can be computed within it and then combined with lexical score. The next three listings keep each step explicit: build the query embedding (Example 7-18), run vector search within the candidate set (Example 7-19), then rerank with a blended score (Example 7-20).

Example 7-18. Hybrid retrieval: stage 2a, building the query embedding

```
CREATE TEMP TABLE query_embedding AS
SELECT ml_generate_embedding_result AS embedding
FROM ML.GENERATE_EMBEDDING(
  MODEL `oreilly_support.text_embedding`,
  (SELECT 'login timeout after update' AS content)
);
```

The `query_embedding` table makes the query vector explicit so it can be logged, cached, or reused. It is also a natural place to attach request metadata such as the ticket ID, language, or model version.

Example 7-19. Hybrid retrieval: stage 2b, vector search within candidates

```
CREATE TEMP TABLE candidate_embeddings AS
SELECT e.chunk_id, e.embedding
FROM `oreilly_support.kb_article_embeddings` e
JOIN lexical_candidates c USING (chunk_id)
WHERE e.embedding_model = 'text-embedding-004'
  AND e.language = 'en';

CREATE TEMP TABLE vector_hits AS
SELECT base.chunk_id, distance
FROM VECTOR_SEARCH(
  TABLE candidate_embeddings,
  'embedding',
  (SELECT embedding FROM query_embedding),
  top_k => 20,
  distance_type => 'COSINE'
);
```

`candidate_embeddings` materializes the narrowed search space so `VECTOR_SEARCH` can take a simple table argument. The query vector comes from the single-row `query_embedding` table as a scalar subquery, which keeps the call readable and easy to reuse.

Example 7-20. Hybrid retrieval: stage 2c, reranking with blended score

```
SELECT
  c.article_id,
  c.chunk_id,
  0.7 * (1.0 / (1.0 + v.distance))
    + 0.3 * (LEAST(c.lexical_score, 10) / 10.0) AS hybrid_score,
  c.lexical_score,
  v.distance,
  c.content
FROM vector_hits v
JOIN lexical_candidates c USING (chunk_id)
ORDER BY hybrid_score DESC
LIMIT 5;
```

The final `SELECT` turns two independent signals into a single ordering. The distance is transformed so higher means better then combined with a capped lexical score. The `0.7` and `0.3` weights are a simple starting ratio that slightly favors semantic similarity while still letting exact-term matches move results up. This is a cheap reranker that often improves precision. When the retrieval corpus grows, a stronger lexical index or a model-based reranker can replace the scoring without changing the overall query shape.

A `hybrid_score` near 1.0 means the chunk matched well on both signals, with high semantic similarity and multiple keyword hits. Scores below 0.5 usually indicate weak matches on at least one signal. When reviewing results, check whether the top hits have high lexical scores (exact term matches) or high vector similarity (semantic matches). This tells you which signal is doing the work for a given query.

The weights in the final `hybrid_score` expression are not special. They are a starting point. In practice, tune them with a small evaluation set and log the top results so it is obvious when a change improves one query and harms another.

Multimodal embeddings

Text is not the only content worth searching by meaning. Support tickets often include screenshots of error dialogs, product photos, or scanned documents. Multimodal embedding models can convert all of these formats

into vectors just like text, which means the same retrieval patterns apply: store the embedding, run `VECTOR_SEARCH`, join the results.

The model registration follows the same pattern as Example 7-9, but with a multimodal endpoint such as `multimodalembedding@001` instead of `text-embedding-004`. The key constraint is that images must live in Google Cloud Storage in a bucket within the same project as your BigQuery instance and remote model connection.

In Example 7-21, we embed an image stored in Cloud Storage by passing its URI as the `content` column.

Example 7-21. Generating embeddings for images

```
SELECT
  'img_001' AS image_id,
  ml_generate_embedding_result AS embedding
FROM ML.GENERATE_EMBEDDING(
  MODEL `oreilly_support.multimodal_embedding`,
  (SELECT 'gs://advanced-sql/O_Reilly_Media_logo.svg.png' AS content)
);
```

From here, retrieval works exactly as shown in Examples 7-12 and 7-13. The `VECTOR_SEARCH` call, distance filtering, and result shape are unchanged. Only the source of the embedding differs. The added benefit of this approach is that it allows you to search images by text.

Multimodal inputs are often more sensitive than text. Screenshots and PDFs can carry account IDs, invoices, and other regulated data. Treat embedding calls as data egress and apply the same masking, scoping, and access controls used for LLM calls.

With retrieval producing a ranked table of chunks, the next section shows how to turn that table into a grounded LLM response.

Patterns: RAG, Agents, and MCPs

The previous section gave you the building blocks. LLM calls generate and classify text, embeddings turn text/images into vectors, and vector search

retrieves by meaning. This section combines them into two patterns that show up repeatedly in production systems.

Retrieval-augmented generation (RAG) fetches a small, relevant slice of data and passes it to a model so the answer is grounded in your warehouse content rather than the model's training data. Agents go further. Instead of a single retrieve-then-generate step, an agent runs a loop, calling tools, interpreting results, and deciding what to do next, with SQL often serving as its primary tool.

Both patterns share a core idea. The model is a reasoning step, not the whole system. The warehouse handles data selection, governance, and logging. The model handles synthesis and language. Getting that division of labor right is what makes these patterns reliable at scale.

Grounded Generation (the "G" in RAG)

LLMs are confident even when they are wrong. Ask a model about your product's return policy and it will give you an answer, just not necessarily "your" answer. RAG fixes this by injecting retrieved context into the prompt so the model reasons over facts you control rather than facts it imagines.

RAG has two steps: retrieval ("R")—as seen in the previous section—and generation ("G"). After embedding the knowledge base and retrieving the top-k chunks at query time, the next step is to pack those chunks into a prompt, call the model, and log the result alongside the retrieved IDs so the output is reproducible and reviewable.

To understand the "G" in RAG, we start with a simple example of drafting a grounded reply from the `ticket_kb_retrieval` table we built in Example 7-14. The mini-RAG query has three parts: pack the retrieval table into a prompt-ready context string (Example 7-22), pull the ticket text for the same ticket (Example 7-23), then call the model and return the draft alongside the retrieved IDs (Example 7-24).

Example 7-22. Mini-RAG: context CTE (packing retrieved chunks)

```
WITH context AS (
  SELECT
    run_id,
    ticket_id,
    product_id,
    STRING_AGG(
      CONCAT('[', article_id, ' | ', chunk_id, ']\n', content),
      '\n\n---\n\n'
      ORDER BY distance
    ) AS context_text,
    ARRAY_AGG(chunk_id ORDER BY distance) AS retrieved_chunk_ids,
    ARRAY_AGG(DISTINCT article_id ORDER BY article_id) AS retrieved_article_ids
  FROM `oreilly_support.ticket_kb_retrieval`
  GROUP BY run_id, ticket_id, product_id
),
```

The CTE `context` is the bridge from retrieval to generation. `STRING_AGG` concatenates each chunk's metadata header and text into a single `context_text` block, separated by `---` dividers and ordered by similarity so the most relevant chunks appear first. Meanwhile, `ARRAY_AGG` collects chunk and article IDs into structured arrays (`retrieved_chunk_ids`, `retrieved_article_ids`) that travel alongside the prompt. Those arrays are what you log and evaluate later, since they let you trace exactly which sources the model saw, independent of how the prompt was formatted.

Example 7-23. Mini-RAG: ticket CTE (attaching the ticket text)

```
ticket AS (
  SELECT CONCAT(subject, '\n', body) AS ticket_text
  FROM `oreilly_support.support_tickets`
  WHERE ticket_id = (SELECT ticket_id FROM context)
)
```

Keeping the ticket text in its own CTE makes the join point explicit: the ticket you draft for is the same ticket that produced the retrieval run.

Example 7-24. Mini-RAG: generating a grounded draft reply (final `SELECT`)

```
SELECT
  c.run_id,
  c.ticket_id,
  (SELECT ml_generate_text_llm_result FROM ML.GENERATE_TEXT(
    MODEL `oreilly_support.gemini_flash`,
    (SELECT FORMAT(
      CONCAT(
```

```
        'Draft a helpful support reply using ONLY the context.\n\n',
        'Ticket:\n%s\n\n',
        'Context:\n%s\n\n',
        'Requirements:\n',
        '- 3-6 sentences max.\n',
        '- End with: \"Sources: <comma-separated chunk_id list>\".\n'
      ),
      (SELECT ticket_text FROM ticket),
      c.context_text
    ) AS prompt),
    STRUCT(
      0.2 AS temperature,
      256 AS max_output_tokens,
      TRUE AS flatten_json_output
    )
  )) AS draft_reply,
  c.retrieved_chunk_ids,
  c.retrieved_article_ids
FROM context c;
```

The model call is then just another column transformation. `ML.GENERATE_TEXT` receives a prompt built from the ticket text and the aggregated context, and the instruction explicitly requires a `Sources:` line listing chunk IDs. Because those IDs come straight from the retrieval table, you have a concrete, checkable claim: the draft should cite only chunks that were actually retrieved. After generation, you can compare the cited IDs against `retrieved_chunk_ids` to flag hallucinated or missing citations. No external tooling required.

Some situations when this may be enough:

- Batch drafting or analyst-scale workflows where seconds of latency are acceptable
- Small corpora with stable document structure and clear product scoping
- Human review loops where drafts are suggestions, not final customer-facing output

Situations when this may fail:

- Retrieval needs hybrid search, reranking, or stronger governance gates on what can be retrieved.
- You need strict citation enforcement and evaluation harnesses that run continuously.
- You need low-latency, multiturn state, or retrieval that spans systems beyond the warehouse.

"Productionizing Generative AI in the Data Warehouse" expands this idea into operational guardrails: logging, evaluation, and cost and security controls. The next pattern uses the same primitives differently. Instead of a single query, an agent runs a short loop and treats SQL as a tool it can call repeatedly.

SQL-Based Agents and MCPs

So far, our patterns have been mostly "one-shot" where a query retrieves context, calls a model, and produces a table row. Agents are different. An agent is a multistep loop where an LLM decides which tools to use, runs them, interprets results, and iterates until it can answer the user or complete a task.

SQL is often the most important tool in that loop. It is how the agent inspects schemas, retrieves facts, computes aggregates, and (when you have embeddings) performs semantic retrieval. The core design question is not "can the model write SQL?" but "how do we build agents that generate and execute SQL safely and predictably?"

This section builds a SQL-generating agent in Python using the OpenAI Agents SDK, progressively adding guardrails and schema context (though any other agent framework could be used). It then explores the *Model Context Protocol (MCP)* as an alternative architecture for exposing SQL capabilities to agents.

Before the examples, set up a local PostgreSQL instance with Docker so the code runs as is. Example 7-25 creates a persistent volume and runs any

`.sql` or `.sh` files in `./initdb` on first boot (a good place to create the `oreilly_support` schema and load sample data, similar to what we created previously on BigQuery).

Example 7-25. Local PostgreSQL setup for the agent examples

```
docker run --name local-postgres \
  -e POSTGRES_USER=postgres \
  -e POSTGRES_PASSWORD=postgres \
  -e POSTGRES_DB=app \
  -p 5432:5432 \
  -v postgres_data:/var/lib/postgresql/data \
  -v "$PWD/initdb":/docker-entrypoint-initdb.d \
  -d postgres:16
```

The next round of examples assume `postgresql://postgres:postgres@localhost:5432/app`. The `POSTGRES_*` variables create the default role and database on first boot, `-p 5432:5432` publishes the container on localhost (change the left side to use a different host port), `postgres_data` persists the data directory, and `./initdb` scripts run once at initialization.

The workflow has five parts:

1. Build a minimal agent that executes SQL (Examples 7-26 and 7-27).
2. Add read-only and tenant-scoping guardrails (Examples 7-28 and 7-29).
3. Inject schema context so the model knows what tables exist (Examples 7-31 and 7-32).
4. Handle failures with rich error messages (Example 7-33).
5. Connect agents to SQL capabilities via MCP (Example 7-35).

Building a SQL-generating agent

The simplest useful agent pattern is text-to-SQL where we take a natural-language question, generate SQL, execute it, and return the results

(Examples 7-26 and 7-27). The next example builds this in Python using the OpenAI Agents SDK (`openai-agents`), connecting to the PostgreSQL database we've just created. The same `oreilly_support` schema used throughout this chapter serves as the target.

The agent has two responsibilities: (1) decide what SQL to generate based on the user's question, and (2) call a tool that executes that SQL and returns results. The tool is where we enforce safety: the agent generates, but the tool validates and executes. This separation mirrors the pattern from Example 7-5 where we created a "safe input view" as a contract for what the model is allowed to see.

Example 7-26. SQL execution tool for the agent

```
import asyncio
import psycopg
from agents import Agent, Runner, function_tool
from dotenv import load_dotenv

load_dotenv()
DATABASE_URL = "postgresql://postgres:postgres@localhost:5432/app"
MAX_ROWS = 50

@function_tool
async def execute_sql(query: str) -> str:
    """Execute a SQL query against the support database and return results."""
    print("execute_sql:", query)
    async with await psycopg.AsyncConnection.connect(DATABASE_URL) as conn:
        async with conn.cursor() as cur:
            await cur.execute(query)
            if cur.description is None:
                return "Query executed successfully but returned no rows."
            columns = [desc[0] for desc in cur.description]
            rows = await cur.fetchmany(MAX_ROWS + 1)
            if not rows:
                return "Query executed successfully but returned no rows."
            has_more = len(rows) > MAX_ROWS
            rows = rows[:MAX_ROWS]
            result = " | ".join(columns) + "\n"
            result += "\n".join(" | ".join(str(v) for v in row) for row in rows)
            if has_more:
                result += "\n... (more rows not shown)"
            return result
```

The `@function_tool` decorator from OpenAI Agents SDK exposes `execute_sql` as a callable tool. When the agent decides to run a query, the SDK extracts the `query` argument from the model's output, calls the function, and returns the result string as the tool's response. The function connects to PostgreSQL, runs the query, formats column headers with rows, fetches at most 50 rows (plus one to detect truncation), and returns a short notice when results are empty or truncated to avoid overwhelming the context window.

Example 7-27. Agent definition and execution loop

```
agent = Agent(
    name="sql_agent",
    instructions="""You are a SQL assistant for the oreilly_support database.
    When asked a question, generate a PostgreSQL query to answer it.
    Use the execute_sql tool to run queries, the schema is 'oreilly_support'.
    If a query fails, read the error and try again with a corrected query.""",
    tools=[execute_sql],
)

async def main():
    result = await Runner.run(
        agent,
        input=[{
          "role": "user",
          "content": "How many open tickets are there per product?"
        }],
    )
    print(result.final_output)

if __name__ == "__main__":
    asyncio.run(main())
```

The `Agent` receives the user's question, generates SQL, calls `execute_sql`, and returns the result. If the query fails (syntax error, missing column), the agent sees the error message and can retry with a corrected query. This feedback loop distinguishes agents from one-shot prompts.

> **NOTE**
>
> These examples use the OpenAI Agents SDK's default model and load the API key from a `.env` file via `load_dotenv()`, which reads `OPENAI_API_KEY`. If you prefer alternatives to OpenAI, you can configure a different model by passing the `model` parameter to `Agent()`. The SDK also supports other providers.

Adding guardrails

Production SQL agents need multiple layers of protection. Add three: read-only verification at the query level, required filters in tool signatures for tenant isolation, and database-level permissions as a final backstop. These layers complement each other. Defense in depth means any single layer can fail without catastrophic consequences.

Read-only verification

The first guardrail is ensuring all queries are read-only (Example 7-28). Python libraries like `pglast` (a PostgreSQL parser) can parse SQL and detect whether it modifies data. Unlike regex matching, validation based on an abstract syntax tree (AST) catches edge cases: `INSERT ... RETURNING`, data-modifying CTEs, `SELECT INTO`, and transaction control statements like `COMMIT` that could escape a read-only transaction.

Example 7-28. Adding read-only query validation with pglast

```
import pglast
from pglast import ast, parse_sql

WRITE_NODES = (
    ast.InsertStmt, ast.UpdateStmt, ast.DeleteStmt,
    ast.DropStmt, ast.TruncateStmt,  # extend as needed
)


def is_read_only(sql: str) -> tuple[bool, str]:
    """Check if SQL is read-only. Returns (is_safe, reason)."""
    try:
        for raw_stmt in parse_sql(sql):
            if isinstance(raw_stmt.stmt, WRITE_NODES):
                return False, f"{type(raw_stmt.stmt).__name__} not allowed"
```

```
        return True, "OK"
    except pglast.parser.ParseError as e:
        return False, f"Parse error: {e}"
```

The `is_read_only` function parses the SQL into an abstract syntax tree and checks whether the top-level statement is a write-capable node. A query like `DELETE FROM support_tickets` returns `(False, "DeleteStmt not allowed")`, which the agent sees as a rejection message. This feedback lets the agent self-correct: it knows the query was rejected and why.

This example covers the most common write statements. Production implementations should extend `WRITE_NODES` to include additional statements like `CREATE`, `ALTER`, `GRANT`, and transaction control statements based on your security requirements.

For defense in depth, also connect to the database using a PostgreSQL user with read-only permissions. Create a dedicated user that has only `SELECT` privileges and only on the schemas/tables/views the agent needs.

Required filters in tool signatures

A more subtle guardrail is forcing the agent to provide certain parameters (Examples 7-29 and 7-30). By making `product_id` a required argument in the tool signature, the agent must specify it and you can validate that the generated SQL actually uses it. This pattern mirrors the product-scoping logic from Example 7-14 where retrieval results were filtered to match the ticket's `product_id`.

Example 7-29. AST-based tenant filter validation

```
from pglast import ast, parse_sql
from pglast.enums import A_Expr_Kind
from pglast.visitors import Visitor

class FilterChecker(Visitor):
    """Walk the AST to find equality filters on a specific column."""

    def __init__(self, column: str, value: str):
        self.column, self.value, self.found = column, value, False

    def visit_A_Expr(self, ancestors, node):
        if node.kind == A_Expr_Kind.AEXPR_OP:
```

```
            op = node.name[0].sval if node.name else None
            if op == "=" and self._matches(node.lexpr, node.rexpr):
                self.found = True

    def _matches(self, left, right):
        col = val = None
        if isinstance(left, ast.ColumnRef):
            col = left.fields[-1].sval if left.fields else None
        if isinstance(right, ast.A_Const) and hasattr(right.val, "sval"):
            val = right.val.sval
        return col == self.column and val == self.value
```

The `FilterChecker` visitor walks the parsed AST looking for equality expressions (`=`). When it finds one, it checks whether the left side is a `ColumnRef` matching the required column and the right side is a string constant matching the required value. This catches `product_id = 'prod_001'` but correctly rejects `product_id IN ('prod_001')` or subquery-based bypasses.

Example 7-30. Using the filter checker in the tool

```
def has_required_filter(sql: str, column: str, value: str) -> bool:
    """Check if SQL contains an equality filter on column = value."""
    try:
        tree = parse_sql(sql)
        checker = FilterChecker(column, value)
        checker(tree)
        return checker.found
    except Exception as e:
        print(f"Filter validation error: {e}")
        return False

@function_tool
async def execute_sql(query: str, product_id: str) -> str:
    """Execute a SQL query scoped to a specific product."""
    is_safe, reason = is_read_only(query)
    if not is_safe:
        return f"Query rejected: {reason}"
    if not has_required_filter(query, "product_id", product_id):
        return f"Query must filter by product_id = '{product_id}'"
    # ... execute query
```

The tool signature requires both `query` and `product_id`. When the agent calls `execute_sql`, the tool parses the SQL and walks the AST to verify the

filter is present. This is more robust than string matching. It handles whitespace variations, quoted identifiers, and rejects attempts to bypass the filter with `IN` clauses or subqueries.

Here are three design choices:

Validation before execution

> Check read-only and filter requirements before connecting to the database, so rejected queries never touch the connection pool.

Informative rejections

> Return the reason for rejection so the agent can self-correct. A message like `"Query must filter by product_id = 'prod_001'"` tells the agent exactly what to fix.

Defense in depth

> Combine AST validation (pglast) with database permissions (read-only user) so no single layer is a single point of failure.

With safety guardrails in place, the next challenge is accuracy: how does the agent know which tables and columns exist?

Enhancing accuracy with schema context

The biggest accuracy improvement for SQL-generating agents is providing schema context. Without it, the model guesses table and column names based on its training data. With it, the model has a map. This is analogous to the schema description in Table 7-1. The agent needs to know what the `support_tickets`, `support_kb_articles`, and `product_catalog` tables contain.

The workflow has two parts: fetch schema information from the database (Example 7-31), then inject it into the agent's instructions (Example 7-32) so every query benefits from accurate metadata.

Example 7-31. Fetching schema context from PostgreSQL

```
async def get_schema_context(conn, schema_name: str = "oreilly_support") -> str:
    """Fetch schema information from PostgreSQL."""
    query = """
    SELECT table_name, column_name, data_type, is_nullable
    FROM information_schema.columns
    WHERE table_schema = %s
    ORDER BY table_name, ordinal_position
    """
    async with conn.cursor() as cur:
        await cur.execute(query, (schema_name,))
        rows = await cur.fetchall()

    # Format as a readable schema description
    tables = {}
    for table, column, dtype, nullable in rows:
        qualified = f"{schema_name}.{table}"
        if qualified not in tables:
            tables[qualified] = []
        null_str = "" if nullable == "YES" else " NOT NULL"
        tables[qualified].append(f"  {column}: {dtype}{null_str}")

    schema_text = "Available tables:\n\n"
    for table, columns in tables.items():
        schema_text += f"{table}:\n" + "\n".join(columns) + "\n\n"
    return schema_text
```

The `get_schema_context` function queries PostgreSQL's `information_schema.columns` view, which contains metadata about all columns in all tables. The result is formatted as a readable text block that becomes part of the agent's system prompt.

Example 7-32. Injecting schema context into agent instructions

```
schema_context = await get_schema_context(conn)
agent = Agent(
    name="sql_agent",
    instructions=f"""\
You are a SQL assistant for the oreilly_support database.

DATABASE SCHEMA:
{schema_context}
Generate queries using only the tables/columns above.
Always filter by product_id. If a query fails, fix it and retry.""",
    tools=[execute_sql],
)
```

With this context, the agent no longer guesses column names. It can see that `status` is a column in `support_tickets`, that `product_id` links tickets to products, and that the schema uses `character varying` rather than `varchar`.

For large schemas, you may need to filter which tables to include or summarize the schema. Some approaches are:

- Only include tables relevant to the user's question (determined by a preliminary classification step).
- Include table names and primary/foreign keys, but omit less-used columns.
- Cache the schema and refresh periodically rather than fetching on every request.

Schema context increases prompt length and therefore cost. For a schema with 50 tables and 500 columns, the context might add 2,000–3,000 tokens per request. Balance accuracy against cost by including only the tables the agent is likely to need. Another option can be to have `get_schema_context` as a tool that the agent can call only when it needs more detailed information.

Handling failures gracefully

Agents will eventually generate invalid SQL. The key is making failures recoverable. When the tool returns an error, the agent should see enough context to fix it. This is the same principle behind the allowlist enforcement in Example 7-8, where we handled unexpected model outputs with sensible defaults rather than failing silently.

Example 7-33. Rich error messages for agent recovery

```
@function_tool
async def execute_sql(query: str, product_id: str) -> str:
    """Execute a SQL query scoped to a specific product."""
    # ... Verify query is read-only and has required filter ...
    try:
        async with await psycopg.AsyncConnection.connect(DATABASE_URL) as conn:
```

```
            async with conn.cursor() as cur:
                await cur.execute(query)
                if cur.description is None:
                    return "Query executed successfully but returned no rows."
                columns = [desc[0] for desc in cur.description]
                rows = await cur.fetchmany(MAX_ROWS + 1)
                if not rows:
                    return "Query executed successfully but returned no rows."
                has_more = len(rows) > MAX_ROWS
                rows = rows[:MAX_ROWS]
                result = " | ".join(columns) + "\n"
                result += "\n".join(" | ".join(str(v) for v in row) for row in
rows)
                if has_more:
                    result += "\n... (more rows not shown)"
                return result
    except psycopg.errors.UndefinedColumn as e:
        return f"ERROR: Unknown column. {e}\nCheck the schema and try again."
    except psycopg.errors.UndefinedTable as e:
        return (
            f"ERROR: Unknown table. {e}\n"
            "Tables: support_tickets, support_kb_articles, product_catalog"
        )
    except psycopg.errors.SyntaxError as e:
        return f"ERROR: SQL syntax error. {e}\nPlease fix the query syntax."
    except Exception as e:
        return f"ERROR: {type(e).__name__}: {e}"
```

Each error type returns a message that tells the agent what went wrong and hints at how to fix it. An `UndefinedColumn` error includes the actual exception message (which names the bad column) plus a reminder to check the schema. An `UndefinedTable` error lists the available tables so the agent can pick the right one.

With descriptive error messages, the agent can often self-correct. In testing, agents typically succeed within 1–3 attempts when given clear error feedback.

Here are four design choices:

Catch specific exceptions

> Different error types get different guidance. A syntax error needs different advice from what a missing table needs.

Include the original error

: The agent benefits from seeing the exact PostgreSQL error message, not just a generic "something went wrong."

Suggest next steps

: Phrases like "Check the schema and try again" or "Available tables: …" give the agent actionable hints.

Truncate large results

: Returning 10,000 rows floods the context window. Cap results and indicate that additional rows were omitted.

SQL agents via MCP

The previous sections built SQL tools directly into the agent code. That works well for single-purpose agents, but as you deploy more agents that need SQL access, you start duplicating connection logic, validation rules, and security policies. A model context protocol (MCP) offers an alternative where instead of embedding SQL capabilities in each agent, you run a separate MCP server that provides SQL tools, and agents connect to it.

What is MCP?

MCP is an open standard for connecting AI agents to external tools and data sources. An MCP server exposes three types of capabilities:

Tools

: Functions the agent can call (like `execute_sql` or `list_tables`)

Resources

: Data the agent can read (like schema definitions or configuration)

Prompts

: Templates for common tasks (like "analyze this table")

An agent connects to one or more MCP servers and gains access to their capabilities without custom integration code. The benefit is separation of concerns, with the MCP server handling database connections, query validation, and security, and the agent just calling tools. This is especially valuable when multiple agents or applications need SQL access. You configure security once in the MCP server rather than in each agent.

Using postgres-mcp

Several MCP servers are available. In this case we will use the postgres-mcp server since it provides a production-ready MCP interface for PostgreSQL. It includes read-only mode, query validation using `pglast` (the same library from Example 7-28), and additional tools for database health and index analysis.

Example 7-34 shows how to run the server using Docker. The `DATABASE_URI` environment variable points to your PostgreSQL instance, and the server exposes a Server-Sent Events (SSE) endpoint on port 8000.

Example 7-34. Running postgres-mcp with Docker

```
docker run -p 8000:8000 \
  -e DATABASE_URI=postgresql://postgres:postgres@localhost:5432/app \
  crystaldba/postgres-mcp --access-mode=restricted --transport=sse
```

The `--access-mode=restricted` flag enables read-only mode with query validation. The server parses all SQL using `pglast` and rejects any statements that could modify data, including `COMMIT` and `ROLLBACK` that might escape a read-only transaction. This is the same defense-in-depth approach we implemented manually in Example 7-28, but configured once at the server level.

Connecting an agent to MCP

Among many agent frameworks to leverage, OpenAI Agents SDK supports MCP servers connection and usage directly (Example 7-35).

Example 7-35. Connecting an agent to postgres-mcp

```
import asyncio
from agents import Agent, Runner
```

```
from agents.mcp import MCPServerSse
from dotenv import load_dotenv

load_dotenv()

async def main():
    async with MCPServerSse(
        params={"url": "http://localhost:8000/sse"},
        name="postgres",
    ) as mcp_server:

        agent = Agent(
            name="sql_agent",
            instructions="""You are a SQL assistant for a PostgreSQL database.
            Use the available tools to explore the schema and run queries.
            Always check the schema before writing queries.""",
            mcp_servers=[mcp_server],
        )

        result = await Runner.run(
            agent,
            input=[{
                "role": "user",
                "content": "What tables are available and "
                           "how many tickets are open?",
            }],
        )
        print(result.final_output)


if __name__ == "__main__":
    asyncio.run(main())
```

The `MCPServerSse` class establishes a connection to the MCP server using SSE as the transport. Once connected, the agent automatically discovers the tools exposed by postgres-mcp (including `execute_sql`, `list_schemas`, `list_objects`, and `get_object_details`) without any additional configuration. The MCP server handles connection pooling, query validation, and security. Your agent code stays focused on the task.

When to use MCP versus custom tools

MCP excels when you want standardized, reusable database access across multiple agents or applications. Use MCP when:

- Multiple agents need SQL access with consistent security policies.
- You want to separate database concerns from agent logic.
- You are deploying agents that connect to databases you do not control.
- You want the additional capabilities postgres-mcp provides (health checks, index analysis, explain plans).

Use custom tools when:

- You need fine-grained control over query validation logic.
- Your tool needs to do more than just execute SQL (for example, transform results, call other services, or enforce business rules).
- You are building a single-purpose agent where the overhead of MCP is not justified.

Table 7-3 exposes the main differences between using custom tools and an MCP server.

Table 7-3. SQL agent architecture comparison

Dimension	Custom tools	MCP server
Security config	In agent code	In MCP server (centralized)
Reusability	Per-agent	Shared across agents
Additional capabilities	Build yourself	Often included (health, explain, etc.)
Complexity	Lower for simple cases	Lower for multi-agent deployments
Debugging	Direct stack traces	Requires MCP server logs

You also can combine both approaches. Use an MCP server for standard SQL access, but add custom tools for domain-specific operations like "find similar tickets" that combine vector search with business logic.

Productionizing Generative AI in the Data Warehouse

The patterns in "Primitives: LLMs, Embeddings, and Search" and "Patterns: RAG, Agents, and MCPs" are deliberately buildable. Productionizing them means treating the external step (model calls, tool calls, or both) like any other operational dependency: stable inputs, versioned artifacts, reproducible runs, and clear accountability. Two questions show up on day one: "What happened?" and "Can we reproduce it?"

This section builds the operational spine for those answers: versioned artifacts (prompts and retrieval configs), core run logs, evaluation and review loops, and the guardrails that keep cost and risk bounded. Exhaustive metadata is not the aim. A small set of shared tables and views that every GenAI workflow can reuse is shown in Figure 7-1.

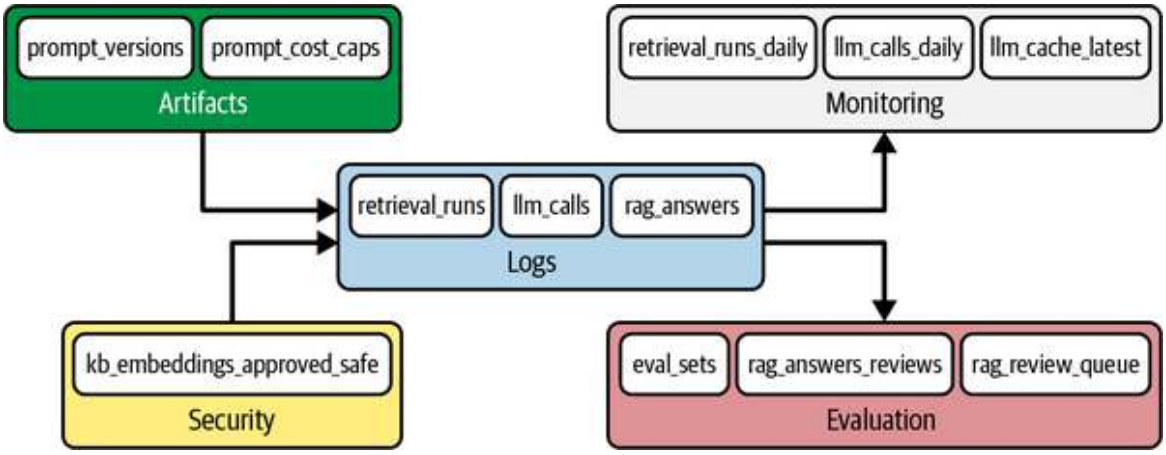


Figure 7-1. Production GenAI artifact flow

These five interconnected components form a complete production surface:

Artifacts

Versioned registries for prompts and cost caps. These are the stable identifiers that make every run reproducible. When something breaks, you trace back to a `prompt_id` and `prompt_version`, not to a string someone edited in a notebook.

Security

Views that gate what data can reach a model. The `kb_embeddings_approved_safe` view enforces approval flags and filters before any text enters a prompt. This is where you prevent data leakage and block obvious injection patterns.

Logs

The core run tables (`retrieval_runs`, `llm_calls`, and `rag_answers`) that capture what happened. Each row ties an output to its inputs,

configuration, and lineage. These are the audit trail and the foundation for everything downstream.

Monitoring

Daily rollup views and a cache layer built on top of logs. `llm_calls_daily` and `retrieval_runs_daily` power dashboards and alerts. `llm_cache_latest` enables cheap deduplication. This is where you catch drift, errors, and cost overruns before users do.

Evaluation

Stable test sets and human review loops. `eval_sets` anchors regression tests. `rag_answer_reviews` and `rag_review_queue` capture structured feedback from reviewers. This is how you measure whether the system is getting better or worse.

The rest of this section explains each component with concrete table definitions and queries you can adapt to your own warehouse, as well as practical deployment architectures for GenAI in SQL.

Logging, Monitoring, and Evaluating GenAI Outputs

If you take only one lesson from production RAG and agents, make it this: do not treat model outputs as ephemeral strings. Treat them as data products. That means you materialize outputs into tables, track versions, and build metrics that tell you when the system changes. This section starts with the versioned artifacts that make runs reproducible, then adds logs, monitoring views, and evaluation loops.

The operational questions you must be able to answer are simple and nonnegotiable:

- Which prompt template and model produced this output?
- What retrieval configuration and sources were used?

- Did the run succeed, how long did it take, and what did it cost?
- Are quality and relevance improving or drifting?

Run lineage and versioned artifacts

The first time a production RAG pipeline produces a bad answer, someone will ask "what prompt did we use?" If the answer is "whatever was in the code at the time," you have already lost. Reproducibility starts with versioned artifacts. Every run must point to an immutable prompt version and retrieval configuration, not to a mutable string that changes between deploys.

Here, *retrieval configuration* means the retrieval-side recipe, not the generation-side settings. It includes choices such as:

- Which corpus, table, or approved view was searched
- Which embedding model and embedding version produced the vectors
- Which chunking strategy or chunk version was in use
- What filters were applied (for example language, tenant, product, or approval status)
- How many neighbors were requested (`top_k`)
- Which distance metric was used
- Whether any reranking or post-filtering step was applied

Settings such as temperature, thinking budget, or max output tokens belong to the LLM call configuration instead and should be logged separately with the model invocation.

The simplest implementation is an append-only registry table. Each row captures a prompt template, its version number, and the owner who approved it. When you log a run, you record `prompt_id` and `prompt_version` as foreign keys. Later, when you need to debug or

compare outputs, you join back to the registry and see exactly what the model received.

Example 7-36 shows a minimal prompt registry. The `is_active` flag lets you deprecate old versions without deleting them, and the `owner` field establishes accountability.

Example 7-36. Prompt version registry table

```
CREATE TABLE IF NOT EXISTS
  `oreilly_support.prompt_versions` (
  prompt_id STRING,
  prompt_version INT64,
  description STRING,
  template_text STRING,
  created_at TIMESTAMP,
  owner STRING,
  is_active BOOL
);
```

The discipline is simple. Never mutate a row. If you need to change a prompt, insert a new version. That guarantees you can always reproduce a past run by looking up its version, even months later, assuming the model provider hasn't deprecated the model in the meantime.

Prompt templates can contain sensitive context. If you store `template_text`, put the table in a restricted dataset or split the text into a locked-down companion table with short retention.

Core run logs and daily observability

With versioned artifacts in place, the next layer is running logs. Every model call and every retrieval lookup should produce a row that ties the output back to its inputs, configuration, and timing. This is not optional instrumentation; it is the foundation for debugging, cost tracking, and regression detection.

A good log table answers several questions at once. What prompt and model produced this output? How long did it take and how many tokens did it consume? Did it succeed or fail? And critically, can this call be linked to the retrieval step that fed it context?

Example 7-37 shows a core `llm_calls` table designed around these questions. The `parent_run_id` field lets you trace multistep agent workflows where one model call triggers another. The `input_hash` field enables caching and deduplication without storing the full prompt text in every row.

Example 7-37. Core logging table for LLM calls

```
CREATE TABLE IF NOT EXISTS
  `oreilly_support.llm_calls` (
  run_id STRING,
  parent_run_id STRING,
  created_at TIMESTAMP,
  prompt_id STRING,
  prompt_version INT64,
  model STRING,
  params_json STRING,
  input_hash STRING,
  retrieval_run_id STRING,
  tokens_in INT64,
  tokens_out INT64,
  latency_ms INT64,
  status STRING,
  output_text STRING
);
```

Retrieval needs its own log table with the same philosophy: tie every lookup to a configuration, record what came back, and make it joinable to downstream model calls. Example 7-38 shows a minimal retrieval log. The `retrieval_config_id` should identify an immutable retrieval recipe. The `retrieved` column uses the same `ARRAY<STRUCT<...>>` pattern introduced earlier to store each retrieved `chunk_id` and its `distance` score, so you can later ask "what context did the model actually see?" without flattening every retrieval result into separate rows.

Example 7-38. Core logging table for retrieval runs

```
CREATE TABLE IF NOT EXISTS
  `oreilly_support.retrieval_runs` (
  retrieval_run_id STRING,
  created_at TIMESTAMP,
  retrieval_config_id STRING,
  embedding_model STRING,
  embedding_version STRING,
```

```
  query_hash STRING,
  retrieved ARRAY<STRUCT<
    chunk_id STRING,
    distance FLOAT64
  >>
);
```

This table keeps the retrieval output compact while preserving the essentials like configuration identifiers, a stable query hash, and the list of retrieved chunks with distances. That is enough to diagnose drift and reproduce downstream prompts without storing full query text.

Raw logs are essential for debugging, but nobody wants to query millions of rows to answer "are we having a bad day?" The next step is to build thin daily views that aggregate the signals you actually alert on. These views power dashboards, feed scheduled alert queries, and give you a quick pulse check without scanning the full log.

Example 7-39 shows a daily rollup for model calls. It tracks call counts, error rates, token usage, and latency. That is enough to catch cost spikes and reliability issues at a glance.

Example 7-39. Daily monitoring view for LLM calls

```
CREATE OR REPLACE VIEW
  `oreilly_support.llm_calls_daily` AS
SELECT
  DATE(created_at) AS run_date,
  prompt_id,
  prompt_version,
  model,
  COUNT(*) AS call_count,
  COUNTIF(COALESCE(status, 'ok') != 'ok') AS error_count,
  SUM(COALESCE(tokens_in, 0)) AS tokens_in,
  SUM(COALESCE(tokens_out, 0)) AS tokens_out,
  AVG(latency_ms) AS avg_latency_ms
FROM `oreilly_support.llm_calls`
GROUP BY run_date, prompt_id, prompt_version, model;
```

This view is intentionally compact; it is the basis for SLAs and cost tracking without overfitting to one workflow. Because it is keyed by prompt and model, you can compare versions side by side and catch regressions quickly.

It is advisable to keep the metrics narrow. If you need richer diagnostics, join back to the raw log table for drill-down rather than bloating the daily view.

Example 7-40 shows a similar rollup for retrieval behavior. The `avg_best_distance` metric is a cheap drift signal that captures whether the nearest match is getting weaker on average or getting stronger.

Example 7-40. Daily monitoring view for retrieval runs

```
CREATE OR REPLACE VIEW
  `oreilly_support.retrieval_runs_daily` AS
SELECT
  DATE(created_at) AS run_date,
  retrieval_config_id,
  embedding_model,
  embedding_version,
  COUNT(*) AS run_count,
  AVG(ARRAY_LENGTH(retrieved)) AS avg_top_k,
  AVG((
    SELECT MIN(distance)
    FROM UNNEST(retrieved)
  )) AS avg_best_distance
FROM `oreilly_support.retrieval_runs`
GROUP BY
  run_date,
  retrieval_config_id,
  embedding_model,
  embedding_version;
```

This view tells you whether retrieval is changing even when downstream answers look fine. A rising best distance often signals stale embeddings, a broken filter, or drift in the underlying corpus.

Use a small number of stable metrics first. You can always add more later, but removing noisy metrics after teams depend on them is harder.

Example 7-41 shows two minimal alert queries built on the daily views.

Example 7-41. Alert queries for failures and retrieval drift

```
SELECT *
FROM `oreilly_support.llm_calls_daily`
WHERE run_date = CURRENT_DATE()
  AND error_count > 0;
```

```sql
SELECT *
FROM `oreilly_support.retrieval_runs_daily`
WHERE run_date = CURRENT_DATE()
  AND avg_best_distance > 0.35;
```

These checks are intentionally blunt. They are easy to schedule, cheap to run, and they surface the first-order signals that something changed.

Tune thresholds empirically and keep the alert logic in SQL where it is easy to review. A single daily alert is better than ten noisy ones that people ignore.

Monitoring is only half the story; the other half is evaluation. A practical starting point is to evaluate retrieval directly because it is cheaper and more deterministic than evaluating generated text. You can then extend that approach to generation with a layered strategy:

- Cheap automated checks (format, length, “Sources:” line present + subset-validated, JSON parse success)
- Human review for a small, representative sample (especially for customer-facing output)
- Rubric-based scoring for regression tests, ideally with repeatable prompts and stable eval sets

Evaluation and review loops

Monitoring tells you that something changed. Evaluation tells you whether the change made things better or worse. Without stable evaluation data, you are left arguing about anecdotes. “I think the new prompt is better” versus “but this one answer was wrong.” The fix is to treat evaluation as a dataset, not a one-off exercise.

For retrieval-heavy systems, a strong first metric is `recall@k`: given a query with one or more known-relevant chunks, did the retrieval step return those chunks in its top k results? It is a good starting point because it measures the basic prerequisite for a grounded answer. If the relevant

context never enters the prompt, downstream prompt tuning or model changes cannot recover it.

The simplest starting point is a curated set of test inputs with known-good outputs. You run each new prompt version against this set, compare the results, and flag regressions before they reach production. This is not glamorous work, but it is the difference between "we think it's fine" and "we measured it."

Example 7-42 shows a minimal eval set table. Each row is a test case with an input, an expected output (as JSON for structured checks), and a stable ID you can reference across runs.

Example 7-42. Minimal evaluation set table

```
CREATE TABLE IF NOT EXISTS
  `oreilly_support.eval_sets` (
  eval_set_id STRING,
  row_id STRING,
  input_text STRING,
  expected_json STRING,
  created_at TIMESTAMP
);
```

Keep the eval set small and representative. A hundred well-chosen examples will catch more regressions than a thousand random ones. Update it when you find new failure modes, but resist the temptation to add every edge case. That leads to a test suite nobody can maintain.

Building and evolving eval suites

Do not force every GenAI workflow into a single "accuracy" number. Retrieval, structured extraction, text-to-SQL, and grounded generation fail in different ways, so they need different checks. A useful eval suite combines three buckets of cases:

Representative cases

> The common questions, tickets, and retrieval patterns that make up most production traffic

Known failure modes

Examples taken from incidents, reviewer complaints, hallucinations, or malformed outputs that you do not want to reintroduce

Recent-change cases

Examples affected by new products, policy changes, schema changes, seasonality, or other shifts in the underlying data

The most practical pattern is to keep two eval sets. Maintain a small stable core set for regression testing and a smaller rolling set that you refresh from recent production events. The core set tells you whether a new prompt or model regressed on known-important cases. The rolling set tells you whether the system still handles what users are asking right now.

The right metric depends on the task:

Retrieval

`recall@k` asks whether the known-relevant chunk or chunks appear in the top `k` retrieved results. This is usually the best first retrieval metric because it is simple, interpretable, and directly tied to RAG usefulness. Then add hit rate on known-relevant chunks, average rank of the first relevant result, citation coverage, and drift signals such as `avg_best_distance`.

Structured extraction or classification

Exact match, field-level precision and recall, JSON parse success, null rate, and allowlist violation rate.

Text-to-SQL and agent tool use

Execution success rate, result correctness against known-good answers, retry count, and safety rejection rate.

Grounded generation

Citation-validity rate, rubric score, human approval rate, and the rate of unsupported claims.

Thresholds should match the type of check. Deterministic checks such as JSON parse success, schema validation, or citation subset validation should usually be near 100 percent. Human-facing quality metrics are better used as comparative gates against the current approved version, for example “do not ship if approval rate drops materially” rather than pretending there is one universal pass mark.

To make these checks durable, log eval outcomes as data instead of leaving them in notebooks or CI output. Example 7-43 shows a minimal table that records one metric result per eval row. The `retrieval_config_id` column ties each result back to the retrieval settings in play, so you can compare prompt changes and retrieval changes independently.

Example 7-43. Minimal evaluation results table

```
CREATE TABLE IF NOT EXISTS
  `oreilly_support.eval_results` (
  eval_run_id STRING,
  eval_set_id STRING,
  row_id STRING,
  metric_name STRING,
  metric_value FLOAT64,
  passed BOOL,
  prompt_id STRING,
  prompt_version INT64,
  retrieval_config_id STRING,
  model STRING,
  created_at TIMESTAMP
)
CLUSTER BY eval_set_id, metric_name, prompt_id;
```

As production patterns shift, the eval suite should shift with them. Add new cases when the review queue finds a failure, when drift alerts fire, when analysts report a new query pattern, or when product and policy changes alter the expected answers. Evaluation is not a fixed benchmark you write once. It is a maintained dataset that evolves with the system.

Automated checks go only so far. For customer-facing outputs, you also need human review. Reviewing everything does not scale, but reviewing a consistent sample and capturing the decisions as structured data does. Over

time, those decisions become a signal. If approval rates drop after a prompt change, you have a problem.

Example 7-44 shows the three pieces of a lightweight review loop. One table holds generated answers, another holds reviewer decisions, and a view surfaces the next batch of unreviewed outputs.

Example 7-44. Minimal human review tables for RAG outputs

```
CREATE TABLE IF NOT EXISTS
  `oreilly_support.rag_answers` (
  run_id STRING,
  ticket_id STRING,
  created_at TIMESTAMP,
  answer_text STRING,
  retrieved_chunk_ids ARRAY<STRING>,
  prompt_id STRING,
  prompt_version INT64,
  model STRING
);

CREATE TABLE IF NOT EXISTS
  `oreilly_support.rag_answer_reviews` (
  run_id STRING,
  reviewer STRING,
  decision STRING,
  notes STRING,
  created_at TIMESTAMP
)
CLUSTER BY decision;

CREATE OR REPLACE VIEW
  `oreilly_support.rag_review_queue` AS
SELECT
  a.run_id,
  a.ticket_id,
  a.created_at,
  a.answer_text,
  a.retrieved_chunk_ids,
  a.prompt_id,
  a.prompt_version,
  a.model
FROM `oreilly_support.rag_answers` a
LEFT JOIN `oreilly_support.rag_answer_reviews` r
  USING (run_id)
WHERE r.run_id IS NULL
```

```
ORDER BY a.created_at DESC
LIMIT 100;
```

The `rag_answers` table is intentionally narrow. It captures the output, lineage fields, and the retrieved chunk IDs needed for audit and review. That keeps the review queue simple and makes it easy to join back to logging tables.

This review loop makes evaluation continuous instead of episodic. The queue is small, the decisions are structured, and the results can be joined back to prompt versions and retrieval configs. Human review does not replace automated checks. It complements them by giving you a real-world acceptance signal on a small sample without overwhelming reviewers.

> **WARNING**
>
> Logging can create a second data leak. Prompts and retrieved context often contain sensitive text. A common production mistake is to "solve observability" by storing raw prompts everywhere. A safer pattern is to log hashes by default (`input_hash`) and store full prompt text only in a restricted dataset with a short retention window and explicit access controls.

Cost and Performance Engineering

Cost surprises are rare when you treat GenAI as a transformation pipeline and shape inputs, bound output size, and measure usage. The new dials are tokens, latency, and external pricing, but the discipline is the same. Two ideas keep spend predictable: incremental refresh for expensive artifacts, and explicit cost controls and caching for model calls.

Incremental refresh and embedding versioning

The first time you run a full embedding job on a large corpus, you will notice the cost. The second time, you will ask, "Why are we re-embedding documents that haven't changed?" That question leads directly to incremental refresh so track what moved, embed only the delta, and merge the results back into the main table.

This is not premature optimization. Embedding costs scale with token count, and a naive daily rebuild can easily exceed the cost of the model calls it supports. The fix is the same pattern you would use for any expensive transformation. Maintain a change log, process the delta, and version the output so downstream consumers know when the corpus changed.

The pattern has three steps. Example 7-45 identifies articles updated in the last day and rechunks them into a delta table.

Example 7-45. Identify changed articles and build a chunk delta

```
CREATE TEMP TABLE kb_article_chunks_delta AS
WITH source_articles AS (
  SELECT article_id, body, language, updated_at
  FROM `oreilly_support.support_kb_articles`
  WHERE language = 'en'
    AND updated_at >= TIMESTAMP_SUB(CURRENT_TIMESTAMP(), INTERVAL 1 DAY)
)
SELECT
  article_id,
  CONCAT(article_id, ':', LPAD(CAST(offset AS STRING), 4, '0')) AS chunk_id,
  TRIM(paragraph) AS content,
  updated_at
FROM source_articles,
  UNNEST(SPLIT(body, '\n\n')) AS paragraph WITH OFFSET
WHERE TRIM(paragraph) != '';
```

Example 7-46 embeds only the changed chunks. This is where you save money. Embedding calls scale with the delta, not the full corpus.

Example 7-46. Embed only the changed chunks

```
CREATE TEMP TABLE kb_embeddings_delta AS
SELECT
  d.article_id,
  d.chunk_id,
  d.content,
  d.updated_at,
  ml_generate_embedding_result AS embedding
FROM ML.GENERATE_EMBEDDING(
  MODEL `oreilly_support.text_embedding`,
  (SELECT article_id, chunk_id, content, updated_at FROM kb_article_chunks_delta)
) AS d;
```

Example 7-47 merges the delta into the main embeddings table. Updated chunks get new embeddings, and new chunks are inserted.

Example 7-47. Merge delta embeddings into the main table

```
MERGE `oreilly_support.kb_article_embeddings` t
USING kb_embeddings_delta s
ON t.chunk_id = s.chunk_id
WHEN MATCHED THEN
  UPDATE SET
    embedding = s.embedding,
    content = s.content,
    updated_at = s.updated_at,
    embedded_at = CURRENT_TIMESTAMP()
WHEN NOT MATCHED THEN
  INSERT (article_id, chunk_id, content, updated_at, embedding, embedded_at)
  VALUES (s.article_id, s.chunk_id, s.content, s.updated_at, s.embedding,
    CURRENT_TIMESTAMP());
```

This pattern scopes embedding work to a delta set and merges the results into the main table. It keeps daily refreshes cheap and makes it easy to backfill if you ever need a full rebuild.

Pair the delta refresh with an `embedding_version` signal (for example, a semantic version or a max `updated_at` snapshot) so retrieval logs and caches can detect when the corpus changed.

If your kb changes rapidly, keep a short “hot” window of frequent refreshes and a slower cadence for the long tail.

Cost controls and caching

Model costs have a way of surprising teams. A prompt that costs pennies in development can cost hundreds of dollars when it runs against a full production table. The fix is not to guess. Set explicit budgets, measure against them, and make overages visible before they become invoice line items.

The two primary cost drivers are call count and prompt size. The best optimizations are therefore often boring. Reduce calls by aggregating to business entities, filter early, and prefer bounded outputs. Explicit budgets make those decisions visible and give you a trip wire before costs spiral.

Example 7-48 shows a simple token cap table and a daily overage query. The cap table records a budget per prompt version. The query flags any version that exceeded its budget today.

Example 7-48. Token budget caps per prompt

```
CREATE TABLE IF NOT EXISTS
  `oreilly_support.prompt_cost_caps` (
  prompt_id STRING,
  prompt_version INT64,
  daily_token_cap INT64,
  owner STRING,
  created_at TIMESTAMP
);

SELECT
  d.run_date,
  d.prompt_id,
  d.prompt_version,
  d.tokens_in + d.tokens_out AS tokens_total,
  c.daily_token_cap
FROM `oreilly_support.llm_calls_daily` d
JOIN `oreilly_support.prompt_cost_caps` c
  USING (prompt_id, prompt_version)
WHERE d.run_date = CURRENT_DATE()
  AND d.tokens_in + d.tokens_out > c.daily_token_cap;
```

Note that this is a lightweight guardrail, not an accounting system. It identifies the prompt versions that exceeded their token budgets so you can triage and tighten them quickly.

Teams typically tune caps per prompt and use ownership fields for routing. Accountability and fast feedback matter more than perfect cost attribution.

Budgets tell you when you are spending too much. Caching helps you spend less in the first place. If you call a model with the same prompt twice, you should get the same answer from a cache instead of paying for a second call. This sounds obvious, but getting caching right requires discipline. You need explicit cache keys, clear invalidation rules, and a way to detect when the cache is stale.

A safe cache key must include everything that affects the output. That means `prompt_id`, `prompt_version`, model identifier, model parameters,

and a content hash of the input. For RAG outputs, it often also needs a data version signal such as `embedding_version` or a retrieval snapshot. If any of these change, the cache entry is invalid.

The simplest implementation is a view over your existing `llm_calls` table. Example 7-49 shows a deduplicated cache view that returns the most recent successful output for each unique combination of prompt, model, and input.

Example 7-49. Cache view for latest successful outputs

```
CREATE OR REPLACE VIEW
  `oreilly_support.llm_cache_latest` AS
SELECT
  prompt_id,
  prompt_version,
  model,
  params_json,
  input_hash,
  output_text,
  created_at
FROM `oreilly_support.llm_calls`
WHERE COALESCE(status, 'ok') = 'ok'
QUALIFY ROW_NUMBER() OVER (
  PARTITION BY
    prompt_id,
    prompt_version,
    model,
    params_json,
    input_hash
  ORDER BY created_at DESC
) = 1;
```

This view turns `llm_calls` into a cache you can join against cheaply. It also gives you one place to enforce "only successful rows are cacheable." Because the cache key includes prompt and model parameters, you can safely reuse outputs across runs without silently mixing incompatible versions.

Having a cache view is only half the pattern. The other half is using it correctly. Before you call a model, check the cache, identify the misses, and generate only for those. This keeps your model calls minimal and makes the cost savings explicit in your query plan.

Example 7-50 shows the first step. Compute stable hashes for your inputs and identify which ones are not in the cache.

Example 7-50. Cache miss identification before generation

```
WITH prompts AS (
  SELECT
    ticket_id,
    CONCAT('Summarize ticket: ', subject, '\n', body) AS prompt,
    TO_HEX(SHA256(
      CONCAT('Summarize ticket: ', subject, '\n', body)
    )) AS input_hash
  FROM `oreilly_support.support_tickets_llm_input`
  WHERE status IN ('open', 'in_progress')
  LIMIT 100
),
cache AS (
  SELECT input_hash, output_text
  FROM `oreilly_support.llm_cache_latest`
  WHERE prompt_id = 'ticket_summary'
    AND prompt_version = 1
    AND model = 'gemini-2.5-flash'
    AND params_json = TO_JSON_STRING(
      STRUCT(0.2 AS temperature, 256 AS max_output_tokens)
    )
),
misses AS (
  SELECT p.ticket_id, p.prompt, p.input_hash
  FROM prompts p
  LEFT JOIN cache c USING (input_hash)
  WHERE c.input_hash IS NULL
)
SELECT * FROM misses;
```

This example uses a stable `input_hash` as the join key and filters the cache by the full configuration. The `misses` CTE gives you a clear surface for rate limiting, batching, or deferring work if budgets are tight.

The final step is to generate only for the misses and merge the results with cached outputs. Example 7-51 continues the CTE chain from above. The `generated` CTE calls the model only for rows in `misses`, so you pay for exactly the prompts that were not already cached. The final `SELECT` joins all three sources (`prompts` for the full input set, `cache` for hits, and `generated` for misses) and uses `COALESCE` to pick whichever output exists.

Downstream consumers see a single result set and do not need to know which rows came from the cache versus a fresh model call.

Example 7-51. Generate for misses and merge with cache (continues previous CTE)

```
generated AS (
  SELECT input_hash, ml_generate_text_llm_result AS output_text
  FROM ML.GENERATE_TEXT(
    MODEL `oreilly_support.gemini_flash`,
    (SELECT input_hash, prompt FROM misses),
    STRUCT(
      0.2 AS temperature,
      256 AS max_output_tokens,
      TRUE AS flatten_json_output
    )
  )
)
SELECT
  p.ticket_id,
  COALESCE(c.output_text, g.output_text) AS summary_text
FROM prompts p
LEFT JOIN cache c USING (input_hash)
LEFT JOIN generated g USING (input_hash);
```

This pattern keeps the cache lookup cheap and makes cache misses explicit. It also lets you materialize `generated` into `llm_calls` for reuse and auditing.

For production, wrap this in a scheduled job and tag each run with a `run_id` so outputs are traceable to a specific batch.

> **WARNING**
>
> Caches can silently make your system wrong. If a prompt depends on "current" data (latest incidents, evolving kb articles), caching by prompt text alone can return stale answers. Treat caching as a policy decision: define when it is allowed, include the relevant version signals in the key, and keep TTL values short for volatile workflows.

Security and Privacy Controls

GenAI inside a warehouse changes the meaning of data movement. The data may never leave a table in the traditional sense, but it can leave the warehouse through model calls. That makes the LLM invocation path a privileged data egress route, and it should be treated as such. This section focuses on schema-level guardrails that keep that path safe.

The core pattern is to put guardrails in the schema, not in prose:

- Views define what text is allowed to reach a model (redaction, allowlisted columns).
- Row-level security scopes retrieval and prevents cross-tenant leakage.
- Approval flags gate which chunks can be retrieved and passed to the model.
- Allowlisted datasets and dry runs constrain what agents can query.

Prompt injection and retrieval poisoning are the canonical failures in RAG systems. The content you retrieve is untrusted input, and it can contain adversarial instructions. The mitigations are mostly about separation and gating: keep system instructions separate from context, constrain retrieval to approved sources, and add detection hooks that can flag suspicious chunks for review.

Example 7-52 shows a pragmatic belt-and-suspenders view with explicit chunk approval plus a conservative text filter. The example assumes you have added an `is_approved` column to your embeddings table (or joined to an approval table) to gate which chunks can reach a model.

Example 7-52. Conservative safe-retrieval view for approved chunks

```
CREATE OR REPLACE VIEW
  `oreilly_support.kb_embeddings_approved_safe` AS
SELECT *
FROM `oreilly_support.kb_article_embeddings`
WHERE is_approved = TRUE
  AND NOT REGEXP_CONTAINS(
    LOWER(content),
    CONCAT(
```

```
      r'(ignore\s+previous\s+instructions|system\s+prompt|',
      r'exfiltrate|password|api\s*key)'
    )
  );
```

This view enforces two gates: an explicit approval flag and a conservative filter for obvious injection cues. The approval flag is the primary control. The filter is a trip wire. In practice, you would add `is_approved` as a column on the embeddings table or maintain a separate approval table that you join here.

Use the view as the only source for retrieval in production jobs. That way you can tighten filters or approvals without touching prompt logic.

The filter above is intentionally blunt. Do not rely on regexes as your primary defense. The primary defense is architectural: least-privilege access, explicit approval gates, and logging that lets you detect and trace bad retrieval sources. Use heuristic filters as a secondary trip wire that routes content to review, not as a replacement for governance.

Treat logs as sensitive data. Apply the same masking, row-level security, and retention limits you would use for the source tables.

Practical Deployment Architectures

There is no single correct deployment architecture for warehouse GenAI. The right answer depends on latency, governance, and how tightly the workflow needs to integrate with applications. "Generative AI Inside SQL" provides the decision framework. Here we summarize a few practical patterns and the warehouse artifacts they typically rely on.

Table 7-4 sketches four common architectures and the artifacts you should expect to maintain for each.

Table 7-4. Reference deployment architectures for warehouse GenAI

Pattern	Where it runs	Key warehouse artifacts	Notes
SQL-native enrichment	Scheduled SQL jobs	Enrichment tables + `llm_calls` logs	Best for batch summaries, classifications, and "human in the loop" dashboards
Warehouse RAG drafts	Scheduled SQL + retrieval tables	`kb_article_embeddings`, `retrieval_runs`, `rag_answers`, `eval_results`	Good for drafting replies or internal tools where seconds of latency is acceptable
Hybrid interactive RAG	App service + warehouse state	Warehouse stores embeddings/logs, app assembles prompts and calls LLM	Typical for product features: app handles low latency and caching, warehouse preserves governance and auditability
Agent copilots (SQL tool use)	App agent loop + allowlisted SQL tools	Allowlisted views, `agent_schema`, `agent_tool_calls`, cost caps	Treat SQL as a tool with strict policies, keep prompts and outputs versioned and logged

The shared theme is that state stays in the warehouse: documents, embeddings, run logs, and eval results remain governed even when orchestration moves outside. That is what makes audits and regressions possible.

One operational recommendation applies across all of these areas: treat prompts, embedding versions, and retrieval policies as versioned artifacts with explicit ownership. If you can answer "which version produced this output?" you can debug regressions and roll forward safely. If you cannot, you will end up chasing anecdotal failures in chat logs.

Production Readiness Checklist

Every production GenAI workflow should be able to pass a short readiness checklist. This is intentionally small and practical so teams can use it as a gate before launch:

Versioned artifacts

: Are `prompt_id`, `prompt_version`, and retrieval configs recorded?

Run lineage

: Do `run_id` and `parent_run_id` tie outputs to inputs and tools?

Safe inputs

: Do model calls read from allowlisted views with redaction?

Explicit budgets

: Are per-prompt or per-job caps defined and monitored?

Caching policy

: Do cache keys include version and data freshness signals?

Evaluation data

: Do stable eval sets and logged scores exist for regressions?

Human review

Does a queue exist for high-risk outputs?

Alerting

Are daily error and drift alerts active and owned?

Retention

Do logs have explicit TTLs and access controls?

Ownership

Does each prompt and pipeline have a named owner?

Summary

GenAI inside the warehouse extends the SQL workflows from Chapter 6 into unstructured and semantic workloads without giving up governance. When LLM calls, embeddings, and retrieval live next to governed tables, the most valuable shift is operational. Inputs and outputs become datasets with lineage, and expensive model calls become bounded transformation steps you can schedule, audit, and improve. The running support example shows the shape of these workloads. We started with controlled enrichment and then layered on retrieval and generation, and we constantly asked the same question from "Generative AI Inside SQL". Should this run in SQL, in a hybrid application, or outside the warehouse?

Across the chapter, we treated LLMs and embeddings as primitives that combine well with the relational tools you already use. Inline LLM patterns produce summaries and structured labels that behave like normal columns. Embeddings turn text into joinable vectors. Retrieval-augmented generation becomes a pipeline that fetches a small, authoritative slice of data and then asks the model to synthesize. Retrieval is broader than vector search. Many effective RAG systems rely on metadata joins, keyword filters, and scoped

lookups, using vectors as one signal among several rather than as the entire definition of RAG.

That combination matters because it extends SQL from describing *what* happened to helping draft a grounded explanation of *why* it likely happened. Metrics still come from queries, joins, and aggregations. The model adds synthesis across structured signals and text, for example surfacing seasonality, recurring complaint themes, or rollout effects that would otherwise require manual analyst interpretation. When those explanations are stored at the grain of an aggregate row or time window, they become durable historical context rather than analyst notes that disappear into documents.

Production reliability comes from artifacts, not clever prompts. Versioned templates, call logs, retrieval logs, eval sets, and review queues turn GenAI into a system you can monitor, debug, and regress-test. Chunk approval, safe views, and allowlisted tool surfaces reduce security and cost surprises. Hybrid retrieval and multimodal extensions widen what you can ground on, and agents highlight why tool design matters: safe, testable tool use with explicit fallbacks, not just "SQL generation." Keep GenAI in SQL when the deliverable is a governed table and batch latency is acceptable. Move to hybrid or external designs when you need low-latency interactivity, cross-system retrieval, or multistep orchestration.

Key takeaways:

- Treat LLM and embedding calls as expensive transformations. Shape inputs, bound outputs, and log everything you might need to reproduce a run.
- Use SQL-native primitives for enrichment, retrieval, and mini-RAG. Refactor into warehouse-native workflows when you need refresh jobs, approvals, evaluation, and review loops.
- Treat retrieved text as untrusted input. Separate instructions from context, gate chunks with approvals, and validate citations against retrieved IDs.

- Make evaluation a first-class dataset. Start with retrieval metrics like recall@k and layer in human review and regression checks for generation.
- Control cost with boring mechanics. Limit call counts, keep prompts small, cache with version signals, and alert on error spikes or retrieval drift.

Artifacts you should now recognize and be able to adapt to your own warehouse:

- Safe input views like `support_tickets_llm_input`
- Retrieval state like `kb_article_embeddings` with incremental refresh via delta tables
- Versioned prompts and logs from "Productionizing Generative AI in the Data Warehouse" including `prompt_versions`, `retrieval_runs`, `llm_calls`
- RAG outputs and review loops including `rag_answers`, `rag_answer_reviews`, `eval_sets`, `eval_results`
- Agent safety surface including `agent_allowlist`, `agent_schema`, and `agent_tool_calls`

If this chapter showed SQL absorbing new AI primitives, Chapter 8 turns to the language itself. We will look at evolving syntax, pipe-style query design, and emerging SQL-adjacent languages that suggest where SQL may be heading next.

Chapter 8. Innovations in SQL Syntax and the Future of SQL

For decades, SQL has served as the backbone of data processing, analysis, and transformation. Its declarative nature, broad tooling support, and adaptability across platforms have made it one of the most durable and impactful languages in computing history.

But SQL longevity is not just due to its original brilliance; it is also because of its ability to evolve. In recent years, SQL has undergone a wave of innovations. These changes aim to address long-standing pain points in readability, maintainability, and expressiveness, particularly for modern analytics workflows that span cloud warehouses, streaming architectures, and semantic layers.

In this chapter, we explore several of these innovations, not as isolated features but as reflections of broader shifts in how we use and think about SQL. From Google's Pipe SQL syntax, which brings a linear and pipeline-like style to SQL expression, to composable query abstractions like PRQL and Malloy, which reframe SQL as a target rather than a starting point, we trace how the language is being reimagined for clarity, composability, and semantic expressiveness.

Finally, we close with a forward-looking view. While it's impossible to predict SQL's next 50 years, we examine the durable trends already underway—from AI-assisted query generation to the experimental frontier of quantum-accelerated optimizers—and offer a reflection on SQL's role in the evolving data ecosystem.

Together, these patterns represent a step forward in SQL's maturity: not a replacement of the old but a thoughtful extension of it, designed to meet the demands of modern, governed, large-scale data work.

GoogleSQL Pipe Syntax

GoogleSQL pipe syntax is a new way to write SQL queries in BigQuery (and related Google Cloud products) using a linear, step-by-step structure. Instead of the rigid clause order of traditional SQL `SELECT ... FROM ... WHERE ... GROUP BY ...`, pipe syntax lets you chain operations with a pipe operator `|>`, passing the result of one step as the input to the next. The goal is to make queries easier to read, write, and maintain by expressing the logical flow of data transformations in a top-down sequence.

You may wonder why Google changed something that had been successful for over the past 50 years, but the reality is that SQL isn't perfect. Classic SQL has a few well-known pain points that can make complex queries hard to write and understand. The most common ones include the following:

Fixed clause order

> In standard SQL, clauses must appear in a specific order (`SELECT`, `FROM`, `WHERE`, `GROUP BY`, etc.), which sometimes forces a query structure that doesn't match the logical thought process. Deviating from this order often requires nested subqueries or common table expressions.

Inside-out query logic

> The way you build complex logic in SQL is often inside-out. For example, to filter after an aggregation or to reuse a computed value, you might need to wrap a subquery. This makes queries more verbose and harder to follow.

Repetition of expressions

> Standard SQL frequently requires repeating the same expressions in multiple clauses. A column used in `SELECT` might have to be repeated in `GROUP BY` and again in `ORDER BY`. This leads to redundancy and potential errors if you forget to update one place.

Pipe syntax was introduced to address these issues from within SQL itself. Rather than inventing an entirely new language or tool, Google extended the SQL dialect to incorporate a pipeline style syntax. This means pipe syntax is not an entirely separate language—it's an optional feature within GoogleSQL.

If you already know SQL, you'll find pipe syntax easy to learn because it uses the same operations and semantics as standard SQL, just written in a different order. You can adopt it incrementally: standard SQL and pipe syntax can be mixed in the same query, and existing SQL queries remain valid. There's no performance penalty either—queries using pipe syntax are optimized and executed just like their standard SQL equivalents.

Pipe syntax works like this: you start a query, often with a `FROM` clause, then use the pipe operator `|>` followed by a pipe operator name (such as `SELECT`, `WHERE`, `AGGREGATE`, etc.) and its arguments. Each `|>` operator takes the table from the previous step as input and produces a new table as output.

You can then add another `|>` to further transform the data, and so on, stringing together multiple transformations in logical order. This linear, top-down flow means that the order of the operations in your query matches the order of your thinking—you can start from raw data and progressively refine it step-by-step.

To illustrate the contrast, consider a scenario: suppose we want to find the total sales and number of sales for each product item, but only for certain categories and excluding a particular item. In standard SQL, you might write a query similar to Example 8-1.

Example 8-1. Simple SQL query

```
SELECT item,
       COUNT(*) AS num_items,
       SUM(sales) AS total_sales
FROM Sales
WHERE item != 'bananas'
  AND category IN ('fruit', 'nut')
GROUP BY item
ORDER BY item DESC;
```

In pipe syntax, the same logic is expressed as a sequence of transformations, starting with the `FROM` clause and adding filters and aggregations in order, as in Example 8-2.

Example 8-2. Simple Pipe SQL query

```
FROM Sales
|> WHERE
     item != 'bananas'
     AND category IN ('fruit', 'nut')
|> AGGREGATE
     COUNT(*) AS num_items,
     SUM(sales) AS total_sales
     GROUP BY item
|> ORDER BY item DESC;
```

Both queries produce the same result; however, the pipe version reads like a logical recipe: *"Start from `sales`, filter out bananas and non-fruit categories, aggregate by item, and then sort by item."* There is no need for a subquery or to repeat the `item` column name in multiple places—each transformation is applied in place and the query flows step-by-step.

Core Syntax and Semantics of Pipe Queries

Pipe syntax extends GoogleSQL with minimal changes to fundamental semantics. A query written with pipes is still an ordinary SQL query under the hood—it yields the same results and the query optimizer treats it the same as the equivalent standard SQL. What changes is how you write and structure the query. Key characteristics of pipe syntax:

Linear sequence of operations

> Instead of nesting subqueries or juggling multiple CTEs, you can append one transformation after another using `|>`. Any operation can be applied in any order, as many times as needed. For example, you could filter, then aggregate, then filter again, then join—whatever suits the logic—whereas in standard SQL, doing a filter after an aggregation would force a subquery or `HAVING` clause.

Top-down readability

Because you typically start with the `FROM` (the data source) and then add operations, the query reads from top to bottom in the same order that the data flows. This is more intuitive for many, especially when exploring data. In fact, any prefix of a pipe query (up to a `|>` symbol) is itself a valid query, which means you can run a pipe query step-by-step to see intermediate results. This greatly aids debugging and incremental query building.

Each step is self-contained

Each pipe operator "sees" the output of only the previous step. Columns from earlier steps are not directly accessible unless they were carried forward in the data. This means, for instance, if you compute a new column and then drop it in a later step, you can't reference it afterward (unless you recompute it)—which encourages a clear flow of transformations. In formal terms, a pipe operator can reference only columns from its immediate input table. Note that this doesn't prevent using outer query references inside subqueries if needed—normal SQL scoping rules for correlated subqueries still apply.

Same operators, new flexibility

Pipe syntax supports essentially all the same operations as standard SQL—selecting columns, filtering (`WHERE`), joining tables, aggregating (`GROUP BY`), sorting (`ORDER BY`), limiting results, set operations (`UNION/INTERSECT/EXCEPT`), and even advanced operations like `PIVOT`—but you're not forced to use them in the fixed order dictated by standard SQL. You can perform an aggregation, then join the result with another table, then aggregate again, for example, all within one continuous query. Under standard syntax, that scenario would typically require multiple nested queries or temporary tables.

No repetition of grouping columns

One of the most appreciated differences is how pipe syntax handles aggregation. In standard SQL, when you use `GROUP BY`, you often must

repeat the grouped-by columns in the `SELECT` clause and sometimes in `ORDER BY`. In pipe syntax, the grouping expressions are specified once and the output automatically includes them. The result table from an `AGGREGATE` step will include all the group-by columns plus the aggregate calculations, without needing to list the group keys again in a separate select clause. This eliminates a lot of repetition and reduces errors.

Familiar semantics

None of these changes alter the meaning of your queries. A filter in pipe syntax works just like a `WHERE` clause in standard SQL; a join works like a normal SQL `JOIN`, and so on. The declarative, set-based nature of SQL is preserved—the query engine can still optimize and reorder operations under the hood. In fact, Google notes that pipe syntax queries typically have exactly the same performance and cost as the equivalent standard SQL, since they compile to the same execution plan.

Mixing with standard SQL

You don't have to choose between standard and pipe syntax exclusively—they can be combined. For example, you might use a pipe sequence to build a complex subquery, and then use that subquery in a standard SQL outer query, or vice versa. A view written in standard SQL can be queried using pipe syntax and vice versa. This flexibility means teams can adopt pipe syntax gradually without breaking existing workflows.

In the next sections, we'll explore each of the pipe operators that you can use after the `|>` symbol. We'll explain what each operator does, how it compares to traditional SQL, and provide examples to illustrate usage.

Pipe Operators Overview

GoogleSQL's pipe syntax introduces a number of pipe operators, each corresponding to a particular kind of SQL operation or clause (see Table 8-

1). Most of these operators share their name and behavior with a standard SQL clause (for example, `SELECT` or `WHERE`), while a few are new keywords introduced for pipeline-specific functionality (like `EXTEND` and `SET`).

Table 8-1. Pipe operators overview

Command	Description
`SELECT`	Produces a new table with only the listed columns (like a projection)
`EXTEND`	Propagates all columns from the input table and adds new computed columns
`SET`	Updates or replaces the values of existing columns in the current table
`DROP`	Removes specified columns from the current table
`RENAME`	Renames specified columns in the current table
`AS`	Assigns a new table alias to the result of the previous stage (useful especially before joins)
`WHERE`	Filters rows based on a boolean condition (same as a standard `WHERE`, and also replaces `HAVING`/`QUALIFY`)
`LIMIT`	Limits the number of rows (optionally with an `OFFSET`)
`AGGREGATE`	Performs aggregation (optionally grouping by one or more columns)
`ORDER BY`	Sorts the rows by specified expressions (like standard `ORDER BY`)

Command	Description
`UNION`/`INTERSECT`/`EXCEPT`	Combines results of queries (set operations, same semantics as standard `UNION`/`INTERSECT`/`EXCEPT`)
`JOIN`	Joins the piped input table with another table or subquery (supports all join types, `ON`/`USING` clauses, etc.)
`CALL`	Invokes a table-valued function, using the current pipe output as an argument (for advanced use cases)
`TABLESAMPLE`	Randomly samples rows from the input (similar to `TABLE SAMPLE` clause in standard SQL)
`PIVOT`/`UNPIVOT`	Rotates rows to columns, and vice versa, for pivoting datasets

This list may look long, but remember that most of these are exactly the same operations you already know from SQL, just used in a different position in the query. In fact, the BigQuery documentation emphasizes that pipe syntax "supports the same operations as existing GoogleSQL query syntax." The main additions that have no direct standard SQL clause equivalent are `EXTEND`, `SET`, `DROP`, `RENAME`, and `AS`: these provide more convenient ways to manipulate columns and aliases as part of your pipeline.

The next sections cover the major pipe operators. We focus on the most commonly used ones (`SELECT`, `EXTEND`, `WHERE`, `AGGREGATE`) and then touch on joins and set operations. Examples often show a pipe syntax query alongside its traditional SQL counterpart for clarity.

The SELECT operator

The pipe `SELECT` operator is used to pick a subset of columns or expressions from the current result, similar to a `SELECT` clause in a subquery. In pipeline terms, `|> SELECT ...` takes the input table and produces a new table containing only the columns or expressions listed. Any columns not listed are dropped at that point. You can also use aliases in a `SELECT` operator to rename the output columns (just as you would in a normal `SELECT` clause).

A key difference from conventional SQL is that the column list does not have to appear at the start of the query. In pipe syntax, a `SELECT` can appear at the end or in the middle of your pipeline, and you can use multiple `SELECT` steps if needed. A pipe query doesn't even require a final `SELECT`—if you omit it, the output contains whatever columns remain after the last transformation. Starting with `FROM <table>` without a subsequent `SELECT` is similar to `SELECT * FROM <table>`.

Common use cases include choosing final output columns, especially after doing various transformations where you might have extra intermediate columns you no longer need, or extracting just a few fields from a wider table early in the pipeline to focus on them. It's also useful if you want to rearrange column order or eliminate columns before passing data to a later stage.

Suppose we have an `Orders` table with many columns—order IDs, customer info, product details, timestamps—and we want a summary of just `order_id` and `total_amount`. In pipe syntax, you can do it as shown in Example 8-3.

Example 8-3. Simple Pipe SQL SELECT query

```
FROM Orders
|> EXTEND quantity * price AS line_value   -- compute something
|> AGGREGATE
     SUM(line_value) AS total_amount
     GROUP BY order_id
|> SELECT order_id, total_amount;
```

In this pipeline, the final `SELECT` ensures that only `order_id` and `total_amount` columns appear in the output. In standard SQL, the

equivalent query would need to list `order_id` and the sum expression in the outer `SELECT` and also repeat `order_id` in the `GROUP BY` clause.

With pipe syntax, the grouping key `order_id` was listed once in the `AGGREGATE` step and automatically carried through, so we only had to select it at the end for output clarity.

The pipe `SELECT` supports most of the same features as a standard `SELECT`, including expressions, aliases, and window functions. However, you cannot do an aggregation function directly in a `SELECT` pipe (use an `AGGREGATE` operator for that), and `SELECT DISTINCT` is generally handled by using a `UNION DISTINCT` or an `AGGREGATE` to group unique values, since the pipeline encourages clarity about whether you are doing aggregation or not.

The EXTEND operator

The `EXTEND` operator is one of the powerful new additions in pipe syntax. `EXTEND` is used to add new computed columns to your data without removing the existing columns. It essentially extends each row with additional fields derived from that row.

When you write `|> EXTEND expression AS new_col`, the pipeline will produce a table that has all the original columns plus a new column `new_col` computed by evaluating the expression for each row. You can list multiple expressions, separated by commas, to add several new columns in one `EXTEND` step.

In regular SQL, adding a computed column often means you either incorporate the expression in a `SELECT`, thereby producing it in place of or in addition to other columns or you use a subquery/CTE to first calculate it. For example, you might do `SELECT *, (quantity*price) AS line_value FROM Orders` as a subquery, then aggregate the results. Pipe syntax eliminates the need for a subquery just to add a column—you simply extend the current pipeline. The `EXTEND` operator is similar to doing `SELECT *, ...` inside a subquery, but with a clearer intent and without having to repeat all columns manually.

This operator shines whenever you need to derive intermediate values, computing a normalized or transformed version of a column, like converting units, calculating a tax or discount, parsing a JSON string into fields, or even performing a calculation that will be used in a subsequent aggregation or filtering step (e.g., `EXTEND quantity * price AS line_value` before summing up revenue).

Adding flags or categories to each row (e.g., `EXTEND (CASE WHEN amount < 0 THEN 'refund' ELSE 'sale' END) AS transaction_type`). One key design philosophy with pipe syntax is to favor adding new columns (with `EXTEND`) rather than modifying or reusing existing ones in place. This ties into the idea of *immutability* and easier debugging: by keeping the original data intact and layering new results, you can trace how each new column is derived. A common style recommendation is to prefer `EXTEND` over `SET` when you might otherwise overwrite a column.

Suppose we have a table `Transactions(amount, tax_rate)` and we want to calculate the after-tax amount for each transaction and then filter for high-value sales, as in Example 8-4.

Example 8-4. Using `EXTEND` in Pipe SQL

```
FROM Transactions
|> EXTEND amount * (1 + tax_rate) AS total_with_tax
|> WHERE total_with_tax > 1000;
```

This will output all columns from `Transactions` plus the new `total_with_tax` and then the `WHERE` filters on that new column. In standard SQL, you'd likely do this in a subquery or repeat the expression in the `WHERE` clause, as in Example 8-5.

Example 8-5. Equivalent to `EXTEND` in standard SQL

```
WITH t AS (
  SELECT *, amount * (1 + tax_rate) AS total_with_tax
  FROM Transactions
)
SELECT *
FROM t
WHERE total_with_tax > 1000;
```

The pipe syntax version is more linear: it extends the data with a new field and immediately filters on it without extra scaffolding.

The SET operator

The `SET` operator allows in-place modification of existing columns in the pipeline. It's like an update to the current result set: for each row, it evaluates the given expressions and assigns them to specified columns. The syntax is `|> SET column_name = expression [, column2 = expr2, ...]`. After a `SET` step, the named columns will hold the new values computed from the old values of that row, or constants.

In Pipe SQL, the `EXTEND` operator creates a new column and leaves existing columns intact (preserving the original data), while `SET` updates an existing column in place (mutating its values). For example, `|> SET price = price * 1.1` overwrites the `price` column, whereas `|> EXTEND price * 1.1 AS new_price` adds a new column and retains the original `price`. Pipelines typically favor `EXTEND` for lineage-preserving transformations and use `SET` only when an in-place schema mutation is truly needed (such as to meet a fixed schema or conserve resources).

In standard SQL `SELECT` queries, there's actually no direct equivalent to `SET` (since `SELECT`s don't update values in place). The way you "set" a column in a `SELECT` is by selecting an expression as that same column name, or by using a subquery to compute it first. Pipe syntax makes it explicit with a dedicated operator.

Use `SET` when you want to transform or normalize data in place, especially if the original version of the column is not needed afterward. For example, if you have inconsistent casing in a text field, you might do `|> SET name = LOWER(name)`. Or to cleanse data, `|> SET email = TRIM(email)`. Another use is to simplify values such as `|> SET status = 'closed' WHERE status = 'shipped'`, though conditional updates might require a `CASE` expression within the `SET`.

However, as noted earlier, many transformations can be done by adding new columns instead. The best practice is to reserve `SET` for cases where overwriting is truly needed (for instance, to conserve memory when adding many columns, or to comply with an expected schema). Overusing `SET` can make the pipeline harder to debug, since you lose the original values, and it may not be obvious how a column was changed if you set it far upstream.

The *immutable data philosophy* suggests preferring `EXTEND (new columns)` most of the time. One scenario where `SET` is quite useful, though, is performing the same operation on many columns (like trimming all string columns or filling `NULL`s in multiple columns)—doing that via `EXTEND` would require adding new columns for each, whereas `SET` can mutate them in one step.

Suppose we have a `Users` table with a column `full_name` that sometimes has leading/trailing spaces. We can clean it up and standardize it to lowercase, as in Example 8-6.

Example 8-6. Using `SET` in Pipe SQL

```
FROM Users
|> SET full_name = LOWER(TRIM(full_name));
```

After this, `full_name` in the pipeline will be cleaned. In standard SQL, you'd achieve this by selecting the expression `LOWER(TRIM(full_name)) as full_name` (alias to the same name) in a subquery or final select, which isn't as clearly an *in place* notion even though its function is effectively similar.

The DROP and RENAME operators

These two straightforward operators are handy for managing columns in your pipeline:

`DROP`

> Removes one or more columns from the current table. For example, `|> DROP temp_var` would eliminate the column `temp_var` from the pipeline output going forward.

`RENAME`

: Changes the name of one or more columns. For example, `|> RENAME old_name AS new_name` (you can omit the `AS` keyword or, if you prefer, `RENAME old_name new_name` works too).

In traditional SQL, dropping or renaming columns typically happens by adjusting what you `SELECT`. If you didn't want a column, you simply wouldn't select it. If you wanted to rename it, you'd use an alias in the `SELECT`. In pipe syntax, since you might carry many columns through multiple transformations, it's convenient to be able to drop or rename at a specific point without having to list all columns.

Use should use `DROP` after you've added some intermediate columns that you no longer need for subsequent steps. For example, if you used `EXTEND` to add a detailed calculation only to filter by it, you might drop that detail column afterward to keep the dataset slim.

Use `RENAME` when a column name is too generic or conflicting. A common scenario is after a `JOIN` where both tables had a column with the same name, for example, both have an ID or date column. You can rename one of them to disambiguate. Another scenario is to give more meaningful names to aggregated results if you didn't already alias them in the `AGGREGATE` step.

Imagine you have a pipeline where you join two tables and both have a status column. After joining, you might do as shown in Example 8-7.

Example 8-7. Using `RENAME` in Pipe SQL

```
|> RENAME status AS order_status,
         status_2 AS customer_status;
```

This assumes perhaps you aliased the second table's status as `status_2` when joining; you can then rename it to something clearer. Alternatively, you could rename columns prior to join using `RENAME` or alias them in the join clause. Pipe syntax gives you flexibility to rename either before or after operations.

Another example use case, focused on the `DROP` operator, would be if you extended a table with a rank using a window function for an intermediate calculation. If you don't want that rank column in your final output, you can drop it, as in Example 8-8.

Example 8-8. Using `DROP` in Pipe SQL

```
FROM Sales
|> EXTEND DENSE_RANK() OVER (ORDER BY amount DESC) AS amount_rank
|> WHERE amount_rank <= 10
|> DROP amount_rank;
```

After filtering to the top 10 by amount, we dropped the `amount_rank` column since it's no longer needed. In standard SQL, that `amount_rank` would likely exist only in a subquery or you would simply not select it in the final projection. Pipe syntax made it explicit: we added it, used it, then removed it.

The AS operator

The `AS` operator is used to assign a table alias to the result of the previous pipeline stage. This might sound a bit abstract if you haven't encountered it, but it's very useful in certain situations, especially with joins and self joins.

In standard SQL, we often use aliases after subqueries or table names so that we can refer to those tables or their columns unambiguously (e.g., `FROM Orders AS o JOIN Customers AS c ON o.cust_id = c.id`). In pipe syntax, when you do `|> AS alias`, you're essentially saying, "Treat the output of all the previous pipes as a table named `alias` from this point onward."

One reason for using this operator would be when joining with another table. When you use the `JOIN` pipe operator, the pipeline input itself needs an alias so you can refer to its columns in the join condition. If your pipeline input was originally a table with an alias, but you lost that alias due to transformations, `AS` gives you a new handle to use.

Another use is disambiguating column names in complex queries. BigQuery might require a new alias after certain operations like

aggregation, since the original table name is no longer in scope once you aggregate.

For example, if we aggregated a production table by item, the resulting table doesn't really have a single table name: it's a derived result. If we want to join that aggregated result with something else, we can do so, as in Example 8-9.

Example 8-9. Using `AS` in Pipe SQL

```
FROM Production
|> AGGREGATE SUM(sales) AS total_sales, COUNT(*) AS num_sales GROUP BY item
|> AS agg  -- give an alias to the aggregated result
|> JOIN SomeOtherTable AS t ON agg.item = t.item
```

In this snippet, `|> AS agg` lets us refer to `agg.item` in the join condition. Without the `AS`, we wouldn't have a straightforward way to reference the item coming out of the aggregation stage because the original table alias would have gone out of scope.

Use `AS` in a pipeline when you need to name the current result for use in subsequent operations (particularly `JOIN`s or when calling a table function via `CALL`). It doesn't change the data in any way; it's purely a labeling mechanism.

An important note is that the `AS` in pipe syntax is not used for column names; you still alias columns using the `AS` alias inside `SELECT`, `EXTEND`, `AGGREGATE`, or any other compatible operator.

The WHERE operator

The pipe `WHERE` acts as a pipeline step rather than a fixed clause keyword. You can place `|> WHERE` after any prior operation (`FROM`, `AGGREGATE`, window functions, etc.) to filter the current result. Using `|> WHERE` immediately after `FROM` is the same as a standard `WHERE`, but by placing it after an aggregation it functions like `HAVING` (and after a window it functions like `QUALIFY`). Each `|> WHERE` simply filters whatever data has

flowed through the pipeline up to that point, making the query's order explicit instead of relying on strict clause sequence.

In pipe syntax, `|> WHERE` condition can be used after any stage of the query. It behaves the same as a normal `WHERE` if used right after a `FROM` (filtering raw data), but it has additional flexibility. Namely, you can use `WHERE` after an aggregation or even after a window function computation, effectively taking the role of SQL's `HAVING` or `QUALIFY` clauses.

In fact, BigQuery's implementation explicitly notes that pipe `WHERE` replaces the need for `HAVING` and `QUALIFY` in pipeline queries. If you do an `AGGREGATE` and then want to filter aggregated groups, just do another `|> WHERE ...` (this is equivalent to `HAVING` in standard SQL). Similarly, if you have a window function result and want to filter on it (something you'd normally do with `QUALIFY` in BigQuery SQL), you can also use `WHERE` for that purpose in pipe syntax.

Example: using `WHERE` as `HAVING`. Using our `Production` table, suppose we want the total sales per item but only for items that have total sales >= 5. In standard SQL, you'd use a `HAVING` clause such as in Example 8-10.

Example 8-10. Using `HAVING` in standard SQL

```
SELECT item, SUM(sales) AS total_sales
FROM Production
GROUP BY item
HAVING SUM(sales) >= 5;
```

In pipe syntax, you could do as in Example 8-11.

Example 8-11. Using `WHERE` in Pipe SQL

```
FROM Production
|> AGGREGATE SUM(sales) AS total_sales GROUP BY item
|> WHERE total_sales >= 5;
```

This is more naturally readable as "after aggregating, filter the results to where `total_sales >= 5`." And indeed, BigQuery's pipe syntax expects you to do exactly that, since there's no separate `HAVING` clause. The result would include only the items meeting the criterion.

Another example focused on window functions would be the way BigQuery SQL provides `QUALIFY` to filter on window function outputs. In pipe syntax, you might do as in Example 8-12.

Example 8-12. Using `WHERE` with window functions in Pipe SQL

```
FROM Sales
|> EXTEND RANK() OVER (ORDER BY amount DESC) AS r
|> WHERE r <= 10;
```

This would keep only the top 10 rows by amount (per whatever partition if you add `PARTITION BY` in the window). It parallels the use of `QUALIFY r <= 10` in standard syntax, but using the common `WHERE` keyword is possible because the pipeline allows placing the filter after computing the window.

If you have multiple conditions to apply stepwise, you can chain multiple `WHERE` pipes. For instance, you might first filter out invalid data, then later filter based on some computed value. Chaining `WHERE` like this can sometimes improve readability compared to one giant condition with `AND` by breaking the logic into separate stages. The style guide suggests this filter chaining as a way to handle complex predicates more clearly, essentially treating each `WHERE` as a distinct rule being applied.

The LIMIT operator

The `LIMIT` pipe operator limits the number of rows in the result, optionally with an `OFFSET` to skip some rows. It works just like a standard `LIMIT n [OFFSET m]` clause.

In pipe syntax, you can use `|> LIMIT 100` at the end of your query to get at most 100 rows. Or you could even use it mid-pipeline if needed.

There's not much semantic difference here from standard SQL; it's mostly a positional difference; you're using it in sequence with other operations rather than at the very end of the query text. Keep in mind that if you apply `LIMIT` in the middle of a pipeline, subsequent transformations will see only that limited subset of rows, which is usually not what you want for final

analysis results, but it can be useful for debugging or if intentionally sampling data for further processing, as in Example 8-13.

Example 8-13. Using `LIMIT` in Pipe SQL

```
FROM HugeTable
|> ORDER BY created_at DESC
|> LIMIT 100
|> ... further analysis on the latest 100 rows ...
```

This would sort a large table by date and take the latest 100 entries and then continue pipeline processing on just those 100, perhaps to do some detailed examination. In standard SQL, accomplishing the same midstream limit would require a subquery or temp table of the limited result before further operations.

Typically, though, `LIMIT` is used at the end for output row limiting, just as you'd use it in standard SQL. If you need deterministic ordering with the limit, remember to use `ORDER BY` before the `LIMIT` as shown in Example 8-13, because otherwise the 100 rows returned might be arbitrary.

The AGGREGATE operator

The `AGGREGATE` operator is one of the most significant features of pipe syntax, as it subsumes the functionality of the `GROUP BY` clause and part of `SELECT` for aggregate expressions in standard SQL. It is used to group rows and calculate aggregate values such as sums, counts, averages, over each group, or to perform full-table aggregation without grouping. The syntax looks like Example 8-14.

Example 8-14. `AGGREGATE` syntax in Pipe SQL

```
|> AGGREGATE aggregate_expr1 AS alias1, aggregate_expr2 AS alias2, ...
    GROUP BY grouping_expr1 [, grouping_expr2, ...]
```

You can also omit the `GROUP BY` to do a full-table aggregation, which produces a single row. And BigQuery's pipe syntax even allows a shorthand `GROUP AND ORDER BY` in one step for convenience. With this advanced usage you can sort the groups as part of the aggregation, but you can also just do a separate `|> ORDER BY` afterward.

In standard SQL, your `SELECT` list contains the aggregate computations, and the `GROUP BY` clause lists all the grouping keys. In pipe syntax, the grouping keys are listed once in the `GROUP BY` part of the `AGGREGATE` operator, and they will automatically be included as output columns. You do not list the group keys as separate expressions in the aggregate expression list. This eliminates duplication as you specify each grouping column one time, and BigQuery will output it alongside your aggregates.

The `AGGREGATE` operator's list should contain only aggregate expressions like `SUM(x)` or more complex expressions that include aggregates. Pure detail columns generally belong in the `GROUP BY` section. This is similar to the rule in standard SQL that nonaggregated columns in `SELECT` must appear in `GROUP BY`.

There is no separate `HAVING` clause. If you want to filter groups after aggregation, as mentioned, use a `WHERE` pipe following the aggregate. Window functions are not allowed inside `AGGREGATE` since window functions conceptually operate on the detail rows, and once you're aggregating you've collapsed rows. If you need to use a window function in an aggregated query, the workaround is to compute the window in an earlier pipeline stage (using `EXTEND`) before aggregating.

Example: total sales per region. Suppose we have an `Orders` table with columns (`region`, `amount`), and we want the total sales per region. Example 8-15 shows the standard SQL format.

Example 8-15. `AGGREGATE` in standard SQL with `GROUP BY`

```
SELECT region, SUM(amount) AS total_sales
FROM Orders
GROUP BY region;
```

Example 8-16 shows the pipe syntax.

Example 8-16. Using `AGGREGATE` in Pipe SQL

```
FROM Orders
|> AGGREGATE SUM(amount) AS total_sales GROUP BY region;
```

The result will have two columns, `region` and `total_sales`. We did not have to explicitly `SELECT` region as an output by grouping on it. We automatically get it in the result. This makes the query a bit more concise and reduces repetition. Imagine grouping by multiple columns; you'd list them once instead of in both `SELECT` and `GROUP BY`.

The power of pipe syntax becomes more apparent in multistep transformations. For instance, consider a scenario where you want to do a two-level aggregation: first you sum up daily sales for each product and then take those results and compute an average across days for each product category. In standard SQL, this often means a subquery or CTE for the first aggregation and then a second aggregation on the results. In pipe syntax, you can chain two `AGGREGATE` operations, as in Example 8-17.

Example 8-17. Using chained `AGGREGATE` operations in Pipe SQL

```
FROM Sales  -- Assume Sales has columns product_id, category, sale_date, amount
|> AGGREGATE SUM(amount) AS total_sales, COUNT(*) AS sale_count
     GROUP BY product_id, category, EXTRACT(Date FROM sale_date) AS sale_day
|> AGGREGATE AVG(total_sales) AS avg_daily_sales,
             MAX(total_sales) AS max_daily_sales
     GROUP BY category;
```

The first `AGGREGATE` groups by product and day (and category), giving daily totals per product. The second `AGGREGATE` then groups by category to compute some metrics across all days/products in that category. Pipe SQL allows us to do this in one continuous flow, without any subqueries. Each aggregation step's output flows into the next. In standard SQL, you'd likely have one subquery for the daily aggregation and an outer query for the category aggregation.

Also notice the use of multiple expressions in the `AGGREGATE` list with aliases. It's similar to having multiple aggregate columns in a `SELECT`. You can alias each aggregate result (as done with `avg_daily_sales` etc.). BigQuery's pipe syntax even allows you to alias grouping expressions; for example, if we wanted to alias that extracted date as `sale_day` in the first aggregate's `GROUP BY`, we could, though the alias is mostly for readability since it will appear as a column name.

The JOIN operator

Joining tables in pipe syntax is done via the `JOIN` pipe operator. It functions similarly to a standard SQL `JOIN` except that one side of the join is implicitly the current pipeline result, and the other side is provided as an argument to the operator. The general form is shown in Example 8-18.

Example 8-18. The `JOIN` operator in Pipe SQL

```
|> [JOIN_TYPE] JOIN <table_or_subquery> [AS alias] ON condition
```

Example 8-19 shows a simple join: the `Orders` table is the input from the `FROM` clause, and we join it with `Customers`. We could alias `Customers` as well (`... JOIN Customers AS c ON ...`).

Example 8-19. Using the `JOIN` operator in Pipe SQL

```
FROM Orders
|> JOIN Customers ON Orders.customer_id = Customers.id;
```

Also, it's good practice to have an alias for the pipeline input at this point. If `Orders` was given an alias in the `FROM` (like `FROM Orders AS o`), that alias carries through. If not, BigQuery might treat the table name as the implicit alias. In more complex pipelines, you might explicitly put `|> AS o` before the join to give a name to the stream (especially if the previous step was something like an aggregate or other operation that lost the original alias). Then you would write `|> JOIN Customers AS c ON o.customer_id = c.id`.

All standard join types are supported: `INNER` (default), `LEFT OUTER`, `RIGHT OUTER`, `FULL OUTER`, as well as semijoins like `LEFT JOIN ... ON ...` with a subsequent filter or using `EXISTS` inside `WHERE`. You can also do `CROSS JOIN` or `JOIN ... USING (col)` just like in standard SQL.

Compared to standard SQL they're conceptually the same, but you have flexibility to join at different points in the pipeline. In the former, all joins for a query are typically declared together in the `FROM`/`JOIN` clauses at the start. In pipe syntax, you could do some transformations on one table, then join, then do more transformations, and then maybe join another table,

intermixing as needed. This can make a multistep process clearer. It is somewhat analogous to doing sequential `JOIN` operations with intermediate CTEs in standard SQL, but the pipe makes it a continuous flow.

Another difference is that when joining in pipe syntax, the left side of the join is whatever came through the pipe, and the right side you specify explicitly. If you need to join two raw tables, you have choices; either you could do that in the initial `FROM` clause (as standard) or you could start from one table and use a pipe join for the other. Both approaches work. For instance, `FROM A JOIN B ON ... |> ...` is equivalent to `FROM A |> JOIN B ON ... |> ...`. In the first case the join happens before the pipe starts, and in the second, the pipeline starts with `A` and then the first operation is a join with `B`. The result is the same. So you can choose which style reads better.

If your query is primarily a pipeline with multiple steps, you might choose to also express the initial join as part of the pipeline for consistency.

It's also possible to filter and then join. Suppose we have `Orders` and `Customers` tables, and we want to filter orders for a certain product category and then join customer info to those orders before selecting output. In standard SQL it would look like Example 8-20.

Example 8-20. Applying the `JOIN` operator in standard SQL

```
SELECT o.order_id, o.date, c.customer_name, c.region
FROM Orders o
JOIN Customers c ON o.customer_id = c.id
WHERE o.product_category = 'Electronics';
```

In pipe syntax, one way to write this is shown in Example 8-21.

Example 8-21. Applying the `JOIN` operator in Pipe SQL

```
FROM Orders
|> WHERE product_category = 'Electronics'
|> JOIN Customers AS c ON Orders.customer_id = c.id
|> SELECT order_id, date, c.customer_name, c.region;
```

Notice here we used `Orders.customer_id` in the join `ON` clause. BigQuery might allow that since `Orders` was the original table name. Alternatively,

we could alias `Orders AS o` either in the `FROM` or via `AS o` right before the join and then use `o.customer_id = c.id`. The result will be the same.

This pipeline reads as *start from orders, filter to electronics, join customers, then pick the columns to output*. One important consideration is that after a join, you have all columns from both sides in your pipeline. You may want to use `SELECT` to narrow down, or `DROP` to remove some, or `RENAME` if there are overlapping names. This is analogous to how in standard SQL you would typically alias tables and refer only to needed columns. In pipe syntax, you have explicit operators to manage columns post-join.

Combining results with UNION, INTERSECT, EXCEPT

Pipe syntax also supports set operations to combine query results. These work similarly to the standard SQL set operators:

- `UNION` to combine two query results, for eliminating duplicates by default, or using `UNION ALL` to keep all
- `INTERSECT` to get only the rows common to both results (BigQuery's standard SQL supports only `INTERSECT DISTINCT`, which is the default)
- `EXCEPT` to get rows from the first result that are not present in the second (BigQuery supports `EXCEPT DISTINCT` similarly)

In pipe syntax, these are expressed as pipe operators that take another query as an argument. Example 8-22 takes all rows from `TableA` with `x>10` and all rows from `TableB` with `x>10` (because of `UNION ALL`, there is no deduplication) and then sorts them together.

Notice how `UNION` is used in the middle of the pipeline. The left side of the `|> UNION` is the pipeline's current result (`TableA` filtered), and on the right of `UNION` we provide a standalone query in parentheses. We could list multiple queries separated by commas if we wanted to union more than two sets.

Example 8-22. Combining results with `UNION ALL` in Pipe SQL

```
FROM TableA
|> WHERE x > 10
|> UNION ALL (SELECT * FROM TableB WHERE x > 10)
|> ORDER BY x;
```

The equivalent standard SQL is shown in Example 8-23. The pipe version is similar in effect but integrates the union into the pipeline.

Example 8-23. Combining results with `UNION ALL` in standard SQL

```
SELECT ... FROM A WHERE ...
UNION ALL
SELECT ... FROM B WHERE ...
ORDER BY ....
```

`INTERSECT` and `EXCEPT` work the same way, as shown in Examples 8-24 and 8-25.

Example 8-24. Using `INTERSECT` in Pipe SQL

```
FROM TableA
|> INTERSECT DISTINCT (SELECT * FROM TableB);
```

Example 8-24 would give rows that are in A and in B, with duplicates removed.

Example 8-25. Using `EXCEPT` in Pipe SQL

```
FROM TableA
|> EXCEPT DISTINCT (SELECT * FROM TableB);
```

Example 8-25 gives rows in A that are not in B.

BigQuery also allows an optional `BY NAME` modifier for `INTERSECT/EXCEPT` if you want to match columns by name instead of position. This is an advanced detail as, normally, you would ensure the two sides have the same columns in the same order for the set operation, just like in standard SQL.

For most readers focusing on learning pipe syntax, understanding `SELECT`, `EXTEND`, `WHERE`, and `AGGREGATE`, will cover the majority of use cases. You can always refer to BigQuery's documentation for the exact syntax of these advanced operators if you need them.

Applied Pipe SQL by Examples

To solidify the concepts, compare a few classic SQL queries with their pipe syntax equivalents. These examples demonstrate typical scenarios where pipe syntax can simplify things.

Filtering and selecting

Say you want to select and filter in one query. In standard SQL, you might retrieve data from a `Users` table and apply a filter in the `WHERE` clause, as in Example 8-26.

Example 8-26. Filtering and selecting with standard SQL

```
SELECT id, name, status
FROM Users
WHERE status = 'active' AND region = 'US';
```

The same process in Pipe SQL is shown in Example 8-27.

Example 8-27. Filtering and selecting with Pipe SQL

```
FROM Users
|> WHERE status = 'active' AND region = 'US'
|> SELECT id, name, status;
```

In this simple case, the difference is mainly stylistic: both queries do the same filtering. The pipe version starts with `FROM Users` and then filters, which some find more natural than "get users, then filter criteria." The `SELECT` at the end picks the three columns to output; if we omitted it, we'd get all columns by default in this case.

Using an intermediate calculation

Suppose you want to compute a derived value and then filter on it. In standard SQL, this often requires either repeating the expression in the `WHERE` clause or introducing an extra subquery, as in Example 8-28.

Example 8-28. Using an intermediate calculation with standard SQL

```
-- This repeats the expression in the WHERE because it's not selected separately.
SELECT order_id, quantity * unit_price AS line_value
FROM Orders
WHERE (quantity * unit_price) > 1000;
```

```
-- We can also use a subquery
SELECT order_id, line_value
FROM (
  SELECT order_id, quantity * unit_price AS line_value
  FROM Orders
)
WHERE line_value > 1000;
```

Pipe syntax makes the sequence of operations clearer by first creating the derived column, then applying the filter, and finally selecting the desired output, as in Example 8-29.

Example 8-29. Using an intermediate calculation with Pipe SQL

```
FROM Orders
|> EXTEND quantity * unit_price AS line_value
|> WHERE line_value > 1000
|> SELECT order_id, line_value;
```

Here, pipe syntax clearly separates the steps. It first adds a new column `line_value`, then filters by it, and then selects the output. This improves clarity, especially if the expression is complex or used in multiple places. In standard SQL, without a subquery you'd have to repeat `quantity * unit_price` in the `WHERE` clause. With a subquery, you introduce extra nesting. The pipe version avoids both repetition and nesting.

Aggregation and post-filtering

Next, we consider how to aggregate data and then filter the aggregated results. In standard SQL this involves a `HAVING` clause, which can sometimes be confusing because it applies after grouping, unlike `WHERE`, as you can see in Example 8-30.

Example 8-30. Aggregation and post-filtering with standard SQL

```
SELECT item, SUM(sales) AS total_sales
FROM Produce
WHERE category = 'fruit'
GROUP BY item
HAVING SUM(sales) >= 5;
```

In pipe syntax, the same logic is expressed as two clear steps: first aggregate and then filter the aggregated output, as in Example 8-31.

Example 8-31. Aggregation and post-filtering with Pipe SQL

```
FROM Produce
|> WHERE category = 'fruit'
|> AGGREGATE SUM(sales) AS total_sales GROUP BY item
|> WHERE total_sales >= 5;
```

The pipe query first filters to fruits, then aggregates by item, and then filters out groups with `total_sales < 5`. The `HAVING` clause from standard SQL is replaced by a second `WHERE` in the pipeline. This is quite straightforward to read: *filter raw data, aggregate, then filter the aggregated results*.

In standard SQL, mixing `WHERE` and `HAVING` in one query can sometimes confuse newcomers since `WHERE` applies before grouping and `HAVING` applies after. The pipe syntax makes the timing explicit by just using two filtered steps in sequence.

Multistep aggregation (eliminating subqueries)

Another example is hierarchical aggregation, where we apply one level of grouping and then aggregate again on the result. In standard SQL, this typically requires nesting queries, since the intermediate grouped result must be materialized before the second aggregation.

Suppose we have a `Trips` table with columns (`city`, `year`, `num_trips`), and we want to find the average number of trips per year per city. In standard SQL, we might need a subquery, as in Example 8-32.

Example 8-32. Multistep aggregation in standard SQL

```
SELECT city, AVG(yearly_trips) AS avg_trips_per_year
FROM (
  SELECT city, year, SUM(num_trips) AS yearly_trips
  FROM Trips
  GROUP BY city, year
)
GROUP BY city;
```

In pipe syntax, we can chain two AGGREGATE steps. Each stage produces an intermediate table that flows into the next, avoiding subqueries and keeping the ultilevel logic in a single pipeline. Example 8-33 shows this.

Example 8-33. Multistep aggregation in Pipe SQL

```
FROM Trips
|> AGGREGATE SUM(num_trips) AS yearly_trips GROUP BY city, year
|> AGGREGATE AVG(yearly_trips) AS avg_trips_per_year GROUP BY city;
```

In Example 8-33 the first AGGREGATE grouped by city and year (summing trips per year per city). The second AGGREGATE took that result and grouped by city to average those yearly sums.

Pipe syntax also makes multiple aggregations straightforward. Each aggregation is its own step, and the intermediate result doesn't need a named subquery—it's simply the output of the previous stage.

Joining after a transformation

Another common scenario is combining aggregated results with reference data through a join. In standard SQL this usually requires a subquery or a CTE to hold the aggregated table before it can be joined with another table. In pipe syntax, the aggregation flows directly into the join: you can compute totals, alias the result, then join with the lookup table. This avoids an extra subquery layer and keeps the logic as a single pipeline.

Suppose you have a `Sales` table (with `region` and `amount` as columns) and a `Regions` table (with `region` and `manager` as columns). If you want the total sales per region along with the `manager` name, you'd typically aggregate sales and then join the regions, as in Example 8-34. In this case we use a CTE, but you can do a subquery directly.

Example 8-34. Joining after a transformation in standard SQL

```
WITH region_sales AS (
  SELECT region, SUM(amount) AS total_sales
  FROM Sales
  GROUP BY region
)
SELECT rs.region, rs.total_sales, r.manager
```

```
FROM region_sales rs
JOIN Regions r ON rs.region = r.region;
```

In Example 8-35, we first aggregate `Sales` by region and then use `|> AS rs` to alias that result (after aggregation, the original table alias is gone). We then join with `Regions AS r`.

Example 8-35. Joining after a transformation in Pipe SQL

```
FROM Sales
|> AGGREGATE SUM(amount) AS total_sales GROUP BY region
|> AS rs
|> JOIN Regions AS r ON rs.region = r.region
|> SELECT rs.region, total_sales, r.manager;
```

Finally, we select the fields we want. This pipeline reads almost like the problem statement *aggregate sales by region, call that result `rs`, join with `Regions` info `r`, then select `region`, `total_sales`, `manager`.* It's clear and avoids having to formally break the logic into a separate CTE or subquery.

These examples show a pattern: whenever standard SQL would force you to introduce a nested query or repeat yourself, pipe syntax offers a cleaner, stepwise alternative. Even in simpler cases, some people prefer the top-down readability of pipe syntax. Of course, for very simple queries (one `SELECT` from one table with maybe one filter), the difference is minor. Pipe syntax shines as query complexity grows.

Real-World Use Cases and Applications of Pipe SQL

Pipe syntax is versatile and can be used anywhere you'd use standard SQL queries. Here are some practical scenarios where it particularly proves its worth:

Exploratory data analysis

> When you are interactively exploring a dataset, you often start with a broad query and then refine it step-by-step. With pipe syntax, you can start with `FROM my_table` to see some rows, then add `|> WHERE …` to zoom in, then maybe `|> AGGREGATE …` to summarize, and so on. At each step, you can execute the query to see intermediate results (since

any prefix of the pipeline is a valid query). This makes the "write a bit and test" workflow much smoother for analysts. It's almost like working with a DataFrame in pandas or R's dplyr—you can iteratively build transformations.

Data transformation pipelines

In data engineering, it's common to have to transform raw data through multiple stages—cleaning, enriching, aggregating, pivoting, etc. Pipe syntax essentially gives SQL a built-in way to express these pipelines clearly. Instead of juggling many CTEs (which can be fine, but often ends up with the logic split across different parts of the query), you express the pipeline in one continuous flow. This linearization can help with maintainability, because it's easy to comment or adjust one step without too much context switching. And as Google's team noted, it can replace many cases where people might otherwise reach for external data processing tools (like writing Google Dataflow or using multiple queries).

Complex analytical queries

For advanced analytics, you often need multistep queries such as *filter this subset, then calculate these metrics, then join to another subset, then filter again, then rank results,* etc. Each of those steps might correspond to a subquery or temp table in traditional SQL. With pipe syntax, you can do them in order. This not only avoids writing extra boilerplate, but it can reduce cognitive load—you can focus on one transformation at a time. If something is wrong, you can test the pipeline up to that point. Teams at large companies found that using pipe syntax made their complex report queries significantly more readable and reduced repetition by ~80% in some cases (for example, when working with heavily nested JSON data, they could incrementally extract and filter fields).

Log data analysis

Interestingly, Google enabled pipe syntax not just in BigQuery but also in Cloud Logging's query interface. Logs often require filtering and parsing fields and aggregating counts. The linear pipeline form is very natural for those tasks (many logging query languages in other systems have a pipelike syntax, which made adopting SQL pipes attractive for users familiar with those).

Improved collaboration and code generation

Because pipe syntax tends to be easier to read, it can help when sharing queries with others or reviewing code. Fewer mental gymnastics are required to follow the flow of data. It can also be easier for tools (including AI assistants) to generate or modify pipe queries because each step is localized. For instance, an LLM might find it simpler to produce a correct pipe-structured query from a natural language description because the logical steps map more directly to the query structure. This is an emerging area, but it's a fascinating side benefit.

Pipe syntax is most helpful when standard SQL becomes unwieldy—deep nesting, repeated expressions, or incremental changes that force another subquery. It's not that pipe syntax enables fundamentally new query results (anything you can do with pipes, you can do with standard SQL), but it enables writing them in a way that mirrors good procedural decomposition.

That said, pipe syntax is also perfectly fine for simple queries. You won't lose anything by writing a one-liner like `FROM table |> WHERE ... |> SELECT ...` instead of the equivalent standard form, if you prefer the style. BigQuery executes them the same way.

Best Practices and Style Guidelines for Pipe SQL Syntax

Like any powerful feature, pipe syntax is most effective when used along with good practices. Here are some recommendations to keep your pipeline queries clean, readable, and efficient:

Write queries as a series of logical steps

Think of your query as a sequence of transformations on the data. Each `|>` ideally represents one logical step. Try to ensure each pipe operator is doing one job. For example, don't combine too many unrelated actions in a single `EXTEND` or a single `WHERE` if it can be sensibly split.

Filter chaining

If you have a complex set of filter conditions, consider applying multiple `WHERE` steps for clarity. For instance, filter for a date range in one step and then filter for status in the next. This separation can document the intent of each filter and make the query easier to tweak: you can comment out or modify one condition without touching the others.

Separate transformation from aggregation

Where possible, do any needed calculations (`EXTEND/SET`) in the pipeline before the `AGGREGATE` step, rather than embedding expressions inside `AGGREGATE`. This makes the aggregation step solely about *here's how we're grouping and summing/counting/etc.* and the prior steps about preparing the data for that. For example, if you need to apply a formula to a column before summing it, do an `EXTEND` for the formula, then `AGGREGATE` on the extended column (as shown in an earlier example with `amount_with_tax`).

Keep each step focused

Resist the urge to do too much at once. If a step starts to look long or encompass multiple tasks, see if it logically splits into two steps. The query might be a few lines longer, but each query will be simpler. This also aids debugging—you can inspect the output after each stage to ensure it's what you expect.

Use immutable-style transformations when possible

Prefer adding new columns (`EXTEND`) over modifying or reusing existing ones (`SET`), unless there is a clear benefit to overwriting. Creating new columns for derived data helps preserve the original data for reference and reduces confusion. For example, rather than using the query in Example 8-36, it might be clearer to write it as in Example 8-37.

Example 8-36. Immutable-style transformations in Pipe SQL

```
|> SET status = 'closed' WHERE status = 'shipped'
```

In Example 8-37, you keep the original status if needed, and then use `normalized_status` going forward. You can always drop the old status column later if it's truly not needed.

Example 8-37. Immutable-style transformations in Pipe SQL—better option

```
|> EXTEND (CASE WHEN status = 'shipped'
THEN 'closed' ELSE status END) AS normalized_status
```

The note from earlier stays the same. The `SET` operator is not bad by itself; it's there for a reason such as for mass normalization of multiple columns. Just use it carefully and document via comments what you're doing when you use it, since it's essentially an in-place mutation, which can be less transparent than adding a new field.

Formatting and readability tips

GoogleSQL pipe syntax is often more readable by design, but good formatting will make it even better. One key formatting best practice is *vertical alignment* of operators. This suggests putting each `|>` on a new line, aligned vertically down the page, which creates a clear visual pipeline, as shown in Example 8-38.

Example 8-38. Vertical alignment of code in Pipe SQL

```
FROM TableName
|> WHERE ...
|> EXTEND ...
|> AGGREGATE ...
|> ORDER BY ...;
```

Another recommendation is to use *indentation*. After using a `|>` and the operator name, if you have multipart arguments such as multiple columns in `SELECT`/`EXTEND`, or a multiline `JOIN` condition, indent those continued lines a few spaces for clarity.

Typically, you might indent by 2 or 4 spaces from the `|>` level (Example 8-39).

Example 8-39. Indentation of code in Pipe SQL

```
|> AGGREGATE
     SUM(CASE WHEN type = 'A' THEN 1 ELSE 0 END) AS count_a,
     SUM(amount) AS total_amount,
   GROUP BY
     category,
     region,
|> JOIN other_table
     ON table.key = other_table.key
     AND other_table.flag = TRUE
```

This style, as shown in Example 8-39, lists each aggregate on its own line and each group-by key on its own line, making it easy to scan. Similarly, long `JOIN` conditions are broken into lines after the `ON`.

Another interesting best practice relates to using *blank lines* to group logic. If your pipeline has logical sections such as first some filtering, then some calculations, and then an aggregation, you can insert a blank line in the query to visually separate them, as seen in Example 8-40. The blank lines help to emphasize the stages (In this case, filtering, extending, and then aggregating).

Example 8-40. Blank lines to separate logic Pipe SQL

```
FROM raw_data
|> WHERE date >= '2023-01-01'

|> EXTEND col1 ` col2 AS combined,
|> SET combined = combined * 2

|> AGGREGATE SUM(combined) AS total ... GROUP BY category
```

Last but not least, use *trailing commas*. When listing multiple expressions (in `SELECT`, `EXTEND`, `AGGREGATE`, etc.), consider ending each line with a

comma. This way, if you add another expression later or reorder them, you touch only one line in the diff. This practice also avoids a missing comma error if you forget to add one on the previous line. An example of such style can be observed in Example 8-41. It's a minor stylistic choice, but it's common in writing SQL for better maintainability.

Example 8-41. Trailing commas between expressions in Pipe SQL

```
|> SELECT
     col1,
     col2,
     col3,
```

Some people also prefer writing `|>` followed by exactly one space, and then the operator, while others might put the operator on a new line below the pipe. The key is to choose a style and stick to it. For example, consistently using two spaces after the pipe before the operator can align multiline blocks nicely. The key is that it doesn't hurt readability.

Efficiency considerations

From a performance perspective, pipe syntax doesn't inherently make your query faster or slower; it's about the same as standard SQL. But you should still be mindful of usual SQL best practices such as filtering early to reduce data volume and avoiding extremely large `JOIN`s or aggregations if they are not necessary.

One advantage of pipe syntax is that because you can filter and aggregate in any order, you have more freedom to push filters before aggregations or joins where appropriate. At the same time, BigQuery's optimizer will often rearrange operations for you when it can, but not always across the pipe boundaries if it's not semantically safe to do so.

In general, write the query in the order that makes logical sense and trust the optimizer, but also apply common sense (for example, if joining a huge table with a small one, that's fine, but if you need only a subset of the huge table, put a `WHERE` before the join to limit rows).

If a pipeline gets extremely long or if you find yourself reusing a particular sub-pipeline in multiple queries, consider encapsulating pieces as views or table-valued functions. Pipe syntax plays well with these: you could define a view with a pipeline query and then query that view. This is similar to modularizing code.

When not to use pipe syntax

While pipe syntax is powerful, there are situations where teams may want to pause before adopting it broadly. For example, when working in a team where not everyone is familiar with pipe syntax, there may be an initial learning curve. Pipe syntax is meant to be easy to learn for SQL users, but it's new, so some colleagues might not immediately recognize the `|>` notation in code reviews.

Debugging/troubleshooting can also be challenging if not all SQL IDEs or clients have full support for syntax highlighting or formatting of `|>`, since it's new. This will improve over time.

You may need to port your query to a system that doesn't support pipe syntax. Most other SQL engines as of 2026 don't (a few like Spark SQL on Databricks and Firebolt have their own variants). In those cases, you'd have to rewrite in standard SQL anyway.

A related consideration is AI-assisted development. Most LLMs have been trained on significantly more examples of traditional SQL than pipe-style queries. As a result, when generating queries automatically, LLMs may default to standard SQL patterns even when pipe syntax is available. This does not affect correctness, but it may require additional prompting or manual refactoring to express the query in pipe syntax.

Google and others are advocating for pipe syntax to become more standard across SQL dialects, but until that happens, be mindful of portability. BigQuery's official stance is that pipe syntax represents a potential future direction for improving SQL usability, and the platform encourages its adoption. However, it is important to note that the largest relational database ecosystems, including SQL Server, PostgreSQL, MySQL, and

Oracle, do not currently support this syntax. For practitioners working primarily on those platforms, pipe syntax may feel more like a dialect-specific improvement rather than a broadly portable SQL pattern.

For readers interested in exploring pipe syntax in more depth, the official GoogleSQL pipe syntax documentation provides a comprehensive reference covering syntax, supported operators, and practical examples. Additionally, the Google Research article "SQL Has Problems. We Can Fix Them: Pipe Syntax in SQL" offers valuable insight into the motivation and design principles behind this approach.

Parallel Innovations in SQL Expression

While Pipe SQL introduces a new way to structure queries within GoogleSQL itself, it's not the only innovation reshaping how we express logic in SQL workflows. In parallel, projects like PRQL and Malloy reimagine the query experience at a different layer: they operate as composable languages that compile to SQL, aiming to make queries more modular, readable, and reusable. These tools don't compete with SQL; instead, they build on it. By abstracting away repetitive clauses, enabling *semantic modeling*, or structuring transformations as pipelines, they reflect a broader movement toward ergonomics and clarity in data work. In this section, we highlight what they bring to the SQL ecosystem and why they matter.

PRQL: A Pipelined Query Language

Outside of BigQuery, the ideas behind pipelined SQL have inspired entirely new query languages. One example is the Pipelined Relational Query Language (PRQL), an open source project that calls itself "a simple, powerful, pipelined SQL replacement." PRQL is designed as a standalone query language that compiles to standard SQL. It emphasizes a small set of orthogonal transformations (like `FROM`, `FILTER`, `GROUP`, `DERIVE`, `SELECT`, `JOIN`) that form a linear pipeline. Each line of a PRQL query is applied to the results of the previous line, making queries very readable. As its

designers put it in the PRQL website: “each line of the query is a transformation of the previous line’s result. This makes it easy to read, and simple to write.”

PRQL was built with several modern features that SQL lacks. For instance, it has built-in syntax for literals (dates like `@2024-01-28`, ranges with `..`, and even Python-style f-strings for string interpolation). It supports variables, functions, and strong naming in pipeline stages, allowing more abstraction than SQL. Critically, PRQL is database-agnostic: it compiles to the SQL dialect of your choice (PostgreSQL, BigQuery, DuckDB, etc.), so you can write a query once and target multiple engines. The PRQL team also emphasizes composability: they use a small number of powerful and orthogonal primitives so there is exactly one way to express each logical step. This orthogonality makes PRQL queries easy to extend and refactor without unintended side effects.

In practice, PRQL feels more like a programming language layered on SQL. It avoids SQL’s verbosity to focus on transformations, as shown in Example 8-42.

Example 8-42. PRQL example

```
from invoices
filter invoice_date >= @2024-01-01
join customers [customer_id]
filter customers.region == "west"
derive {
  net_income = total - tax,
  high_value = net_income > 1000
}
group customer_id (
  aggregate {
    total_net = sum net_income,
    invoice_count = count this,
    high_value_count = sum high_value
  }
)
sort total_net desc
take 20
```

PRQL compiles to equivalent SQL with familiar constructs like `ORDER BY` and `LIMIT`, but its pipeline style lets you express filters, joins, calculations,

and aggregations in a top-down, readable sequence. For example, a typical query might start from a list of invoices, join to customer records, filter by region, compute derived values like `net_income`, group by customer, calculate aggregates, and return the top customers by total value. All these steps happen in a logical flow that avoids nesting or clause repetition, as in Example 8-42. Because PRQL compiles to SQL, it treats SQL as the *assembly language* of databases: it aims to simplify query authoring without changing what the engine does. In this way, PRQL reflects a thoughtful design innovation, providing a clear, composable syntax that improves readability while still compiling down to standard SQL for execution.

Malloy: A Semantic Modeling Language

Another noteworthy innovation is Malloy, an open source language designed by Looker as a semantic modeling and query language. Malloy treats data as a network of relationships and builds queries declaratively on that model. In essence, Malloy is both a *semantic layer* (defining dimensions, measures, and relationships) and a query syntax that operates on that model. Queries and calculations written in Malloy ultimately compile to SQL for execution, so they run on standard databases like BigQuery, Postgres, or DuckDB, providing a more expressive and composable modeling experience while still producing optimized SQL under the hood.

The key idea in Malloy is *declarative modeling*. Instead of writing explicit joins, groupings, or aggregations in every query, you define logic like metrics and derived fields once, at the source, and reuse them across queries. For example, in Malloy, a model for invoice data might be declared like in Example 8-43.

Example 8-43. Malloy example

```
source: invoices is table('data.invoices') {
  measure: net_income is total - tax
  measure: total_net is sum(net_income)
}
```

```
run: invoices -> {
  where: invoice_date >= @2024-01-01
  group_by: customer_id
  aggregate: total_net
  having: total_net > 1000
}
```

This query operates on top of the predefined model and focuses only on what varies: the filter, grouping, and threshold logic. Measures like `net_income` and `total_net` are defined once, upstream in the source, and reused consistently. The structure is readable, composable, and declarative. It's closer to describing an analysis than scripting a step-by-step plan.

Malloy strongly emphasizes composability and reuse. It treats data models like function libraries: once you define a calculation or relationship (say, a business metric like `customer_lifetime_value`), it can be reused across many queries. This avoids the repetition common in raw SQL. Calculations, relationships, and query patterns can be defined once and reused across different analytical contexts. This greatly improves maintainability: when business logic changes, you update it in one place and all dependent queries pick up the new definition.

From a technical standpoint, Malloy remains declarative and mathematical. You describe the data you want, not the exact procedural steps to get it. The language automatically handles things like preventing join fanout or double-counting, based on the defined semantic relationships. It also natively supports nested or ultilevel aggregation, which is very handy for drill-down queries. All of these features—automatic joins, reusable definitions, richer metadata—are delivered via Malloy's higher-level syntax. But because Malloy queries compile into SQL that runs directly on the warehouse, you still leverage SQL's performance and familiarity. In short, Malloy is a SQL ecosystem innovation that raises the level of abstraction, focusing on the meaning of data. It acknowledges that SQL itself is the assembly language of data platforms, and it builds a semantic layer on top of it to make analysts more productive.

The Future of SQL

As we look toward the horizon, the evolution of SQL will no longer be defined merely by syntactic sugar or new keywords; instead it will evolve in concert with broader technology trends. For instance, the "pipe revolution" discussed earlier in this chapter solves the human-computer interface problem, making SQL readable and writable. However, the next decade of innovation will tackle the fundamental limitations of the engines that execute SQL. The convergence of artificial intelligence, quantum computing, and specialized hardware is beginning to challenge the practical boundaries of classical relational algebra, opening the door to database systems that are more autonomous, probabilistic, and selectively accelerated by nonclassic hardware.

AI and Autonomous Databases

Artificial intelligence and machine learning are rapidly entering the world of data platforms. One visible impact is on query authoring: LLMs can now translate natural language or partial specifications into SQL. For example, Uber's QueryGPT project uses generative AI and vector search to turn English questions into complex SQL queries. In production, it can cut query authoring time from about 10 minutes to 3 minutes by automating the query generation process. Similarly, many tools now offer AI-powered auto-complete and error correction for SQL, helping analysts write queries without memorizing every syntax detail. We should expect *text-to-SQL* and *conversational interfaces* (like those powered by GPT-style models) to become commonplace aids for database users.

AI is also affecting databases themselves. The concept of the autonomous database (as promoted by vendors like Oracle) is that many traditional DBA tasks like tuning, indexing, provisioning, patching, or even security configuration can be automated by AI/ML. For example, Oracle defines an autonomous database as "a cloud database that uses AI to automate tuning, security, backups, updates, and other routine management activities." In practice, this means future SQL engines may self-optimize: they could learn

from query patterns, automatically suggest indexes, or even rewrite queries for better performance without human intervention. In essence, we are moving toward database systems that are *self-managing* and *self-securing*. In the near term, these features appear as tooling around SQL (auto-tuners, ML-driven optimizers). Longer term, the boundary between a SQL engine and an AI agent may blur, but SQL will still serve as the *lingua franca* through which users express intent.

Overall, the rise of AI suggests that writing SQL will become easier and more automated and that running SQL will become more managed and autonomous. Human data professionals will spend less time on boilerplate, using tools that handle routine patterns. But the core remains: we will still need a language (likely an evolved SQL dialect) to express queries, and we will still need expertise to guide and oversee the process. AI augments rather than replaces the human role in analytics, and SQL will be the interface between humans, AI tools, and the data.

The Quantum Horizon: Quantum-Assisted Optimization

A more speculative frontier is the use of quantum computing to solve database problems. Query optimization (especially join-order selection) is a nondeterministic polynomial-time hard (NP-hard) problem, and some researchers are investigating whether quantum algorithms could help. Recent work like the Q²O project has demonstrated hybrid quantum-classical optimizers that tackle join ordering and report speedups in prototype settings. In principle, quantum annealers or circuits could explore many candidate query plans in parallel, possibly finding near-optimal plans more quickly than classic heuristics.

It is too early to say whether quantum systems will reshape mainstream SQL. Current quantum hardware remains limited in scale and prone to error, and all existing quantum optimization methods are at the research stage. No commercial database product today uses a quantum optimizer. However, as quantum technology matures, it could influence SQL engines indirectly. For example, some future optimizer might include a quantum-inspired subroutine for join ordering or indexing. At the very least, SQL

query planners will need to keep an eye on these developments, since they represent a fundamentally new way to tackle combinatorial query problems. For the engineering-minded practitioner, the key takeaway is that quantum query optimizers are possible in theory and are under active research, but they remain experimental. In the near future, we will see more proofs of concept, but mass deployment is likely years away.

Declarative Governance and Policies

Another trend is toward declaratively embedding data governance and policies within the data platform. In an era of data privacy laws and complex data meshes, organizations want to ensure policies are enforced consistently. This has led to ideas such as *policy as code* and *declarative access controls*. In SQL, we already see features like row-level security policies, column masking, and audit triggers. Going forward, we are likely to see richer language constructs for governance. For example, data catalog tools might allow you to tag tables with sensitivity labels, and SQL engines could automatically filter or encrypt columns at query time based on those tags. Cloud platforms are adopting *declarative policies* (for instance, AWS allows organization-wide policy rules that block certain queries or data exposures). Although these examples are outside SQL, the same mindset could influence SQL-based workflows. We can imagine SQL engines that support declarative metadata, for instance, specifying data contracts or lineage within the query, which the system then enforces.

This focus on governance is driven by engineering requirements: quality, compliance, and security in complex pipelines. Declarative governance could use SQL's extensibility (extensions or annotations) or surrounding tools that generate guarded SQL. The main point is that data practitioners will increasingly treat governance as an integral part of the language and the workflow, as recommended throughout this book, rather than as an afterthought. SQL may gain more formal constructs (or frameworks) to support these policies, even as we avoid speculative predictions. In practice, we should expect incremental advances: better schema management,

integrated lineage tracking, and policy frameworks that work hand in hand with SQL query planning.

Summary

This chapter explored how SQL is evolving to better match the realities of modern data work. Rather than redefining SQL's declarative foundations, current innovations focus on improving how logic is expressed, composed, and reasoned as queries grow in size and complexity. Across syntax, structure, and execution models, the common goal is to make SQL more readable, maintainable, and aligned with how data transformations are actually designed.

GoogleSQL pipe syntax provides a concrete example of this evolution. By expressing queries as a linear sequence of transformations, pipe syntax aligns SQL with a data-flow mental model that emphasizes progression and intent. Filters, aggregations, joins, and projections become explicit steps in a pipeline, reducing the reliance on nested subqueries, repeated expressions, and indirect reasoning about execution order. As demonstrated throughout the chapter, this restructuring does not change SQL semantics or performance characteristics, but it significantly improves clarity, debuggability, and composability.

Beyond pipe syntax, the chapter examined a broader set of parallel innovations reshaping SQL expression across platforms and ecosystems. Modular SQL functions, incremental materializations, composable query layers, and dataframe-inspired syntax bridges all reflect the same underlying shift. SQL is increasingly being used not just to retrieve data but to define transformation logic as reusable, evolvable building blocks within larger analytical systems. These patterns reduce coupling, encourage reuse, and make complex pipelines easier to reason about over time.

Looking forward, the future of SQL is less about replacing the language and more about extending its expressive surface. SQL is absorbing ideas from functional programming, dataframes, and stream processing while remaining declarative at its core. At the same time, advances in query

optimization, adaptive execution, and tighter integration with cloud native platforms continue to abstract away physical execution concerns, allowing practitioners to focus on intent rather than mechanics.

Across these developments, SQL remains a central abstraction for analytics and data engineering. Its longevity comes from evolving without losing its core strengths as new workloads and collaboration models emerge. The innovations discussed in this chapter aim to make SQL more natural to write, read, and maintain as data systems continue to scale.

Index

Symbols

3NF (see third normal form (3NF))

A

abstract syntax tree (AST)

- agent-specified parameters, Required filters in tool signatures
- database permissions with AST validation, Required filters in tool signatures
- read-only guardrail for agents, Read-only verification

access control, Security and access control: Guardrails, not roadblocks

ACID-compliant transactions

- Apache Iceberg, The Data Lakehouse
- DuckDB, Embedded Analytics with DuckDB

advanced data modeling

- choosing right abstraction for each problem, Summary
- knowledge graphs and semantic modeling, Knowledge Graphs and Semantic Modeling: The New Frontier for SQL-Querying with Cypher Versus SQL
  - RDF and OWL, RDF and OWL: The Foundations of Semantic Modeling

semantic models and ontologies, Semantic Models and Ontologies-PostgreSQL and GraphDB

physical models, Bridging Semantics to Execution

physical models versus semantic models, Physical models versus semantic models in practice-Physical models versus semantic models in practice

semantic models, Semantic Models and Ontologies, Bridging Semantics to Execution

(see also semantic modeling)

SQL as a modeling language, SQL as a modeling language

best practices, SQL as a modeling language

traditional SQL data modeling, Traditional SQL Data Modeling Techniques-Summary of Traditional Models

choosing star or snowflake, Snowflake Schemas: Adding Normalization to Dimensions

lakehouse modeling and ELT workflows, Lakehouse Modeling and ELT Workflows

one big table, One Big Table: The Wide Denormalized Table Approach

snowflake schemas, Snowflake Schemas: Adding Normalization to Dimensions-Snowflake Schemas: Adding Normalization to Dimensions

snowflake versus star schemas, Snowflake Schemas: Adding Normalization to Dimensions

star schema, Dimensional Modeling: Star Schema for Analytics-Dimensional Modeling: Star Schema for Analytics

summary of traditional models, Summary of Traditional Models-Summary of Traditional Models

third normal form, Third Normal Form—Structured, Relational Modeling-Third Normal Form—Structured, Relational Modeling

advanced feature engineering patterns, Advanced Feature Engineering Patterns-Principal component analysis

categorical encoding, Categorical encoding-Feature hashing

about categorical encoding, Categorical encoding

feature hashing, Feature hashing

frequency encoding, Frequency encoding

one-hot encoding, One-hot encoding

principal component analysis, Principal component analysis-Principal component analysis

about PCA, Principal component analysis

quantile binning, Quantile binning-Equal-width (deciles)

about quantile binning, Quantile binning

equal-frequency binning, Equal-frequency (deciles)

equal-width binning, Equal-width (deciles)

target encoding with K-fold out-of-fold mean, Target encoding (leakage-aware) with K-fold out-of-fold mean-Target encoding (leakage-aware) with K-fold out-of-fold mean

about K-fold out-of-fold mean, Target encoding (leakage-aware) with K-fold out-of-fold mean

about target encoding, Target encoding (leakage-aware) with K-fold out-of-fold mean

text feature engineering, Text feature engineering-Character n-grams
about encoding free-form text, Text feature engineering
character n-grams, Character n-grams
tokenization, Tokenization
advanced SQL techniques
CTEs, Common Table Expressions and Subquery Factoring
recursive CTEs, Recursive CTEs for Hierarchical and Iterative Queries-Recursive CTEs for Hierarchical and Iterative Queries
termination of recursive CTEs, Recursive CTEs for Hierarchical and Iterative Queries
window functions, Advanced Window Functions for Analytic Queries-Advanced Window Functions for Analytic Queries
about window functions, Advanced Window Functions for Analytic Queries
common categories of, Advanced Window Functions for Analytic Queries
comparing rows without self joins, Advanced Window Functions for Analytic Queries
handled by SQL, Extended Analytical and OLAP Functions
importance of window functions, Advanced Window Functions for Analytic Queries
ranking, Advanced Window Functions for Analytic Queries
value window functions, Advanced Window Functions for Analytic Queries

advanced statistical analysis, Advanced Statistical Analysis-Kruskal—Wallis test
about SQL capabilities, Advanced Statistical Analysis
confidence intervals, Confidence Intervals: Estimating the Mean-Confidence Intervals: Estimating the Mean
data processing and analysis, Data Processing and Analysis-Outlier Detection
correlation and association, Correlation and Association
dataset statistical description, Dataset Statistical Description-Dataset Statistical Description
outlier detection, Outlier Detection
regression from aggregates, Regression from Aggregates
multiple groups testing
Kruskal—Wallis test, Kruskal—Wallis test
one-way analysis of variance, One-way ANOVA: A handmade analysis-One-way ANOVA: A handmade analysis
non-parametric statistical tests, Mann–Whitney U
normality testing, Normality Testing
p-values, Mann–Whitney U
calculating, One-way ANOVA: A handmade analysis
two-sample tests
Mann–Whitney U test, Mann–Whitney U-Multiple Groups Testing
two-sample t-test, Two-sample t-test-Two-sample t-test

agents generating SQL, SQL-Based Agents and MCPs-When to use MCP versus custom tools

about agents, SQL as an agent tool, SQL-Based Agents and MCPs

building a SQL-generating agent, Building a SQL-generating agent-Building a SQL-generating agent

failures handled gracefully, Handling failures gracefully

PostgreSQL local instance with Docker, SQL-Based Agents and MCPs

schema context provided to agents, Enhancing accuracy with schema context-Enhancing accuracy with schema context

guardrails added, Adding guardrails-Required filters in tool signatures

agent-specified parameters, Required filters in tool signatures

database permissions with AST validation, Required filters in tool signatures

read-only verification, Read-only verification

MCP server with SQL tools, SQL agents via MCP

connecting an agent to MCP, Connecting an agent to MCP

description of model context protocol, What is MCP?

postgres-mcp server, Using postgres-mcp

when to use MCP versus custom tools, When to use MCP versus custom tools

OpenAI Agents SDK, SQL-Based Agents and MCPs

@function_tool decorator, Building a SQL-generating agent

loading the API key, Building a SQL-generating agent

AI and autonomous databases, AI and Autonomous Databases

AI assistance with SQL query generation, Preface-Preface

(see also generative AI (GenAI) in SQL; machine learning (ML))

American National Standards Institute (ANSI), Standard SQL-86: Laying the Foundations

analytic functions (window functions), Advanced Window Functions for Analytic Queries-Advanced Window Functions for Analytic Queries

handled by SQL, Extended Analytical and OLAP Functions

analytics

embedded analytics with DuckDB, Embedded Analytics with DuckDB

star schema, Dimensional Modeling: Star Schema for Analytics-Dimensional Modeling: Star Schema for Analytics

anomaly detection, Anomaly detection-Anomaly detection

ANOVA (analysis of variance), one-way, One-way ANOVA: A handmade analysis-One-way ANOVA: A handmade analysis

degrees of freedom, One-way ANOVA: A handmade analysis

mean squares, One-way ANOVA: A handmade analysis

p-value calculation, One-way ANOVA: A handmade analysis

sums of squares, One-way ANOVA: A handmade analysis

ANSI (American National Standards Institute), Standard SQL-86: Laying the Foundations

anti-patterns to avoid, Avoiding Common SQL Anti-Patterns-Neglecting to analyze/update statistics

architectural anti-patterns, Avoiding Common Architectural Anti-Patterns-Ignoring Governance and Security

ignoring governance and security, Ignoring Governance and Security

missing lineage and monitoring, Missing Lineage and Monitoring

monolithic “all-in-one” scripts, Monolithic “All-in-One” SQL Scripts

not designing for evolution, Not Designing for Evolution

over-coupling to technologies, Over-Coupling to Specific Technologies

skipping tests, Skipping Tests for SQL Models

tight coupling between pipeline components, Tight Coupling Between Pipeline Components

data not filtered early, Not filtering data early

ignoring EXPLAIN or EXPLAIN ANALYZE, Ignoring query plan feedback

indexes ignored or not right, Ignoring indexes (or not having the right index)

large transactions or improper batching, Large transactions or improper batching

misusing DISTINCT or UNION, Misusing DISTINCT or UNION

prepared statements not used, Not using prepared statements (when executing many similar queries)

row-by-agonizing-row processing, RBAR (row-by-agonizing-row) processing

selecting all columns needlessly, Selecting all columns (SELECT *) unnecessarily

table statistics not updated, Neglecting to analyze/update statistics

wildcard searches without proper indexing, Wildcard searches without full-text indexing

Apache AGE, PostgreSQL and GraphDB

Apache Airflow, Alternative Orchestration Options

Apache Flink SQL, Apache Flink SQL

Apache Iceberg

ACID-compliant transactions, The Data Lakehouse

data file compaction, Data Storage and Management with Apache Iceberg

manifest-based pruning accelerating reads, The Data Lakehouse

partition evolution, Data Storage and Management with Apache Iceberg

schema evolution, Iceberg schema evolution-Iceberg schema evolution

snapshot expiration, Data Storage and Management with Apache Iceberg

storage for ecommerce data pipeline, Pipeline Overview and Design Principles

BigLake metastore setup, Set up BigLake (Iceberg) metastore

BigQuery compatibility, Open table formats, Data Storage and Management with Apache Iceberg, Iceberg schema evolution

BigQuery connection to BigLake catalog, Create a BigQuery dataset and BigLake connection

creating GCS bucket, Create a cloud storage bucket

creating GCS bucket billing error, Create a cloud storage bucket

design principle, Pipeline Overview and Design Principles

GCS to Iceberg via Spark batch, GCS to Iceberg with Spark batch processing (Bronze layer)-GCS to Iceberg with Spark batch processing (Bronze layer)

Iceberg data storage and management, Data Storage and Management with Apache Iceberg-Auditing and recovery: Time travel with SQL

Iceberg managing raw data, Storage Layer

open table formats, Open table formats

time travel to query historical data, Auditing and recovery: Time travel with SQL-Auditing and recovery: Time travel with SQL

architecting SQL-driven solutions

about principles behind, Architecting SQL-Driven Solutions

architectural anti-patterns to avoid, Avoiding Common Architectural Anti-Patterns-Ignoring Governance and Security

architectural patterns for data pipelines, Architectural Patterns for Data Pipelines-Data Mesh and Data Products

batch versus streaming, Batch Versus Streaming: Choosing the Right SQL Pipeline Approach-Batch Versus Streaming: Choosing the Right SQL Pipeline Approach

data mesh and data as a product, Data Mesh and Data Products-Data Mesh and Data Products

data sharing, Beyond ETL/ELT and Streaming: Zero-ETL and Data Sharing, Beyond ETL/ELT and Streaming: Zero-ETL and Data Sharing

ELT and ETL transformations, Transformation Patterns: ETL, ELT, and Streaming-Transformation Patterns: ETL, ELT, and Streaming

ELT for most modern data pipelines, Transformation patterns

streaming transformations, Transformation Patterns: ETL, ELT, and Streaming

zero-ETL, Beyond ETL/ELT and Streaming: Zero-ETL and Data Sharing, Beyond ETL/ELT and Streaming: Zero-ETL and Data Sharing

data contracts, Formalizing Data Agreements: Data Contracts-Formalizing Data Agreements: Data Contracts, Data Mesh and Data Products

data pipeline definition, Architecting SQL-Driven Solutions

designing modular SQL data pipelines, Foundational Design Principles for SQL Data Pipelines-Service-level agreements and data freshness

automation and orchestration, Automation and orchestration

clean SQL pipeline organization, Maintainability and readability

data lineage and dependency management, Data lineage and dependency management

idempotency and rerunnability, Idempotency and rerunnability

layering SQL models, Maintainability and readability

LEGO-style pipeline, Modularity and composability

maintainability and readability, Maintainability and readability

modularity and composability, Modularity and composability

service-level agreements and data freshness, Service-level agreements and data freshness

layered architecture, Layered Architecture: Separating Storage, Transformation, and Orchestration-Connecting back to the architecture

about layered architecture, Layered Architecture: Separating Storage, Transformation, and Orchestration

consumption layer, Consumption layer

data ingestion, Data ingestion

orchestration and scheduling, Orchestration and scheduling-Orchestration and scheduling

storage and processing, Storage and processing

testing strategies per layer, Implementing testing strategies

transformation and modeling, Transformation and modeling-SQL as a modeling language

medallion architecture for levels of data maturity, Storage and processing

observability, Observability: Knowing what's working (and what's not)

scalability, Architectures for Scalability-Hybrid and Multi-Cloud Patterns

cloud native scaling, Cloud Native Scaling

comparison chart of architectures, Hybrid and Multi-Cloud Patterns

concurrency and isolation in practice, Concurrency and Isolation in Practice

hybrid and multi-cloud patterns, Hybrid and Multi-Cloud Patterns-Hybrid and Multi-Cloud Patterns

traditional RDBMS scaling, Traditional RDBMS Scaling

scalability built and operationalized, Building and Operationalizing Scalable SQL Data Platforms-Avoiding Common Architectural Anti-Patterns

continuous integration and continuous delivery, Continuous integration and continuous delivery for SQL pipelines-Continuous integration and continuous delivery for SQL pipelines

Infrastructure as Code, Infrastructure as Code (IaC) in scaling and for data platforms-Avoiding Common Architectural Anti-Patterns

layered architecture, Layered Architecture: Separating Storage, Transformation, and Orchestration-Connecting back to the architecture

scaling from design to production, From Design to Production: Scaling SQL Data Solutions-Avoiding Common Architectural Anti-Patterns

security and governance in data pipeline, Security and Governance Considerations-Security Versus Semantics: Consistent Definitions as Control

dataset and schema boundaries, Dataset and Schema Boundaries as Governance

identity and execution boundaries, Identity and Execution Boundaries

AST (see abstract syntax tree (AST))

audit glance use case, Auditing and recovery: Time travel with SQL

auto-increment keys in dialects of SQL, Comparison of SQL Dialects over Time

autoencoder-based anomaly detection, Anomaly detection-Anomaly detection

about autoencoders, Anomaly detection

automation

automated workflow for SQL data pipeline, Orchestration and Workflow Management-Alternative Orchestration Options

data pipelines, Automation and orchestration

Infrastructure as Code with CI/CD, Infrastructure as Code (IaC) in scaling and for data platforms

B

B-tree indexes, Indexing and Efficient Data Access

ideal for typical queries, Indexing and Efficient Data Access

batch ingestion, Data ingestion

ecommerce SQL data pipeline, Batch Ingestion for Bulk Data-GCS to Iceberg with Spark batch processing (Bronze layer)

API to Google Cloud Storage, API to GCS (landing zone)-API to GCS (landing zone)

GCS to Iceberg and Spark (Bronze), GCS to Iceberg with Spark batch processing (Bronze layer)-GCS to Iceberg with Spark batch processing (Bronze layer)

implementation approach, Batch Ingestion for Bulk Data

batch versus streaming data pipelines, Batch Versus Streaming: Choosing the Right SQL Pipeline Approach-Batch Versus Streaming: Choosing the

Right SQL Pipeline Approach

hybrid approach, Batch Versus Streaming: Choosing the Right SQL Pipeline Approach

Kappa architecture, Batch Versus Streaming: Choosing the Right SQL Pipeline Approach

Lambda architecture, Batch Versus Streaming: Choosing the Right SQL Pipeline Approach

BigLake (Google) metastore for Iceberg tables, Set up BigLake (Iceberg) metastore

BigQuery connection, Create a BigQuery dataset and BigLake connection

BigQuery (Google)

Apache Iceberg

compatibility, Open table formats, Data Storage and Management with Apache Iceberg

latest snapshot, Iceberg schema evolution

time travel, Auditing and recovery: Time travel with SQL-Auditing and recovery: Time travel with SQL

BigLake catalog connection, Create a BigQuery dataset and BigLake connection

BigQueryML (see BigQueryML (Google))

cost-based optimizers, Query Planning and Cost-Based Optimization

execution plan as DAG, Query Planning and Cost-Based Optimization

dry-run feature before query execution, BigQuery: Navigating cost-effective analysis

ELT pattern, Transformation patterns

federated query engine EXTERNAL_QUERY, Hybrid and Multi-Cloud Patterns

GenAI platform, Generative AI in SQL

- connecting to Gemini models, LLM Functions Inside SQL
- documentation online, LLM Functions Inside SQL

Iceberg tables queried, Iceberg schema evolution

indexing versus partitioning and clustering, Indexing and Efficient Data Access, Indexing and Efficient Data Access

Infrastructure as Code example, Infrastructure as Code (IaC) in scaling and for data platforms

materialized views, Query Rewriting, Caching, and Materialized Views

- automatic maintenance, Query Rewriting, Caching, and Materialized Views

performance tuning, Tuning in Cloud Data Warehouses: BigQuery Versus Snowflake-Snowflake: Monitoring and profiling queries

- about differences from OLTP, Tuning in Cloud Data Warehouses: BigQuery Versus Snowflake
- concurrency and workload isolation, Concurrency and workload isolation-Snowflake concurrency and isolation best practices
- concurrency and workload isolation best practices, BigQuery concurrency and isolation best practices
- concurrency and workload isolation versus Snowflake, Snowflake concurrency and isolation best practices

cost optimization, BigQuery: Navigating cost-effective analysis

cost optimization best practices, BigQuery cost optimization best practices

execution graph, BigQuery: Monitoring and profiling queries

INFORMATION_SCHEMA.JOBS view, BigQuery: Monitoring and profiling queries

iterative nature of, BigQuery: Monitoring and profiling queries

monitoring and profiling queries, BigQuery: Monitoring and profiling queries

Query Insights feature, BigQuery: Monitoring and profiling queries

pricing on-demand versus flat-rate, BigQuery: Navigating cost-effective analysis

recursive CTEs, Recursive CTEs for Hierarchical and Iterative Queries

documentation online, Recursive CTEs for Hierarchical and Iterative Queries

result cache, Query Rewriting, Caching, and Materialized Views

query exactly consistent, Query Rewriting, Caching, and Materialized Views

slots managing compute resources, Concurrency and workload isolation

INFORMATION_SCHEMA.JOBS view, BigQuery: Monitoring and profiling queries

workload isolation in practice, Concurrency and Isolation in Practice

summary statistics, Dataset Statistical Description

vector search, Integrating ML and LLMs in SQL Platforms

window functions, Advanced Window Functions for Analytic Queries

BigQuery ML (Google)

autoencoder-based anomaly detection, Anomaly detection-Anomaly detection

hyperparameter tuning, Hyperparameter tuning-Best practices and cost considerations

enabling hyperparameter tuning, Hyperparameter tuning

multiple objectives for tuning, Tuning objectives and multi-objective optimization

model creation and training

anomaly detection, Anomaly detection-Anomaly detection

applying the model, Supervised learning models

data portion held for evaluation, Cross-validation

hyperparameter tuning, Hyperparameter tuning-Best practices and cost considerations

recommendation systems, Recommendation systems

SELECT input dataset, Principal component analysis

supervised learning models, Supervised learning models-Supervised learning models

time-series forecasting, Time-series forecasting-Time-series forecasting

training a Random Forest classifier, Supervised learning models

unsupervised learning models, Unsupervised learning models-Unsupervised learning models

model deployment and monitoring, Model Deployment and Monitoring-Drift

predictions from models, Making models available for predictions-Making models available for predictions

model evaluation and validation, Model Evaluation and Validation

cross-validation, Cross-validation-Interpretation of results

evaluation metrics, Evaluation metrics

principal component analysis model, Principal component analysis

RAG with CTEs, Grounded Generation (the “G” in RAG)-Grounded Generation (the “G” in RAG)

SQL integration, Integrating ML and LLMs in SQL Platforms-Integrating ML and LLMs in SQL Platforms

block range indexes (BRIN), Indexing and Efficient Data Access

book supplemental material online, Using Code Examples

book web page, How to Contact Us

Boyce, Raymond, Historical Overview: From Relational to Modern SQL

C

caching query results, Query Rewriting, Caching, and Materialized Views

materialized views versus, Query Rewriting, Caching, and Materialized Views

categorical encoding, Categorical encoding-Feature hashing

about categorical encoding, Categorical encoding

feature hashing, Feature hashing

frequency encoding, Frequency encoding

one-hot encoding, One-hot encoding

Chamberlin, Donald, Historical Overview: From Relational to Modern SQL

character n-grams, Character n-grams

when to use, Character n-grams

CI/CD (continuous integration and continuous delivery), Continuous integration and continuous delivery for SQL pipelines-Continuous integration and continuous delivery for SQL pipelines

cloud data warehouses

BigQuery (see BigQuery (Google))

Infrastructure as Code (IaC), Infrastructure as Code (IaC) in scaling and for data platforms-Avoiding Common Architectural Anti-Patterns

cloud-agnostic, Infrastructure as Code (IaC) in scaling and for data platforms

pay-as-you-go model, Cost-optimization strategies

BigQuery on-demand versus flat rate, BigQuery: Navigating cost-effective analysis

Snowflake Auto-Suspend, Snowflake: Architecting for cost-aware computation

Snowflake compute credits, Snowflake: Architecting for cost-aware computation

Snowflake per-second compute and storage billing, Snowflake cost optimization best practices

performance optimization, Tuning in Cloud Data Warehouses: BigQuery Versus Snowflake-Snowflake: Monitoring and profiling queries

about differences from OLTP, Tuning in Cloud Data Warehouses: BigQuery Versus Snowflake

concurrency and workload isolation, Concurrency and workload isolation-Snowflake concurrency and isolation best practices

cost-optimization strategies, Cost-optimization strategies-Snowflake cost optimization best practices

monitoring and profiling queries, Monitoring and profiling query performance-Snowflake: Monitoring and profiling queries

scalability, Cloud Native Scaling-Hybrid and Multi-Cloud Patterns

cloud native scaling, Cloud Native Scaling

comparison chart of architectures, Hybrid and Multi-Cloud Patterns

concurrency and isolation in practice, Concurrency and Isolation in Practice

hybrid and multi-cloud patterns, Hybrid and Multi-Cloud Patterns-Hybrid and Multi-Cloud Patterns

traditional RDBMS scaling, Traditional RDBMS Scaling

scalability built and operationalized, Building and Operationalizing Scalable SQL Data Platforms-Avoiding Common Architectural Anti-Patterns

continuous integration and continuous delivery, Continuous integration and continuous delivery for SQL pipelines-Continuous integration and continuous delivery for SQL pipelines

Infrastructure as Code, Infrastructure as Code (IaC) in scaling and for data platforms-Avoiding Common Architectural Anti-Patterns

layered architecture, Layered Architecture: Separating Storage, Transformation, and Orchestration-Connecting back to the architecture

scaling from design to production, From Design to Production: Scaling SQL Data Solutions-Avoiding Common Architectural Anti-Patterns

Snowflake (see Snowflake)

clustering

BigQuery, Indexing and Efficient Data Access, Indexing and Efficient Data Access

indexing alternative, Indexing and Efficient Data Access

Snowflake, Indexing and Efficient Data Access, Indexing and Efficient Data Access

Codd, Edgar F., Historical Overview: From Relational to Modern SQL

comments explaining why, Maintainability and readability

common table expressions (see CTEs (common table expressions))

compliance

data contract specifications, Formalizing Data Agreements: Data Contracts, Formalizing Data Agreements: Data Contracts

data lineage, Data lineage and dependency management

managing data changes over time, Temporal Data Management

one big table and, One Big Table: The Wide Denormalized Table Approach

compliance testing, Implementing testing strategies

concurrency and workload isolation

BigQuery and Snowflake in practice, Concurrency and Isolation in Practice

performance tuning, Concurrency and workload isolation-Snowflake concurrency and isolation best practices

BigQuery best practices, BigQuery concurrency and isolation best practices

BigQuery versus Snowflake, Snowflake concurrency and isolation best practices

Snowflake best practices, Snowflake concurrency and isolation best practices

confidence intervals, Confidence Intervals: Estimating the Mean-Confidence Intervals: Estimating the Mean

consumption layer of layered architecture, Consumption layer

best practices, Best practices for the consumption layer

ecommerce SQL data pipeline, Consume Layer-Live Data in Consume Layer

bridging semantics to execution, Bridging Semantics to Execution-Physical models versus semantic models in practice

dbt Semantic layer, The dbt Semantic layer (MetricFlow)-The dbt Semantic layer (MetricFlow)

live data in the consume layer, Live Data in Consume Layer-Live Data in Consume Layer

operationalizing the semantic layer, Operationalizing the Semantic layer

semantic contract tests, Semantic Contracts are Testable Contracts

semantic joins and inferred logic example, Example: Semantic joins and inferred logic

SQL-first semantic execution, SQL-First Semantic Execution-The dbt Semantic layer (MetricFlow)

stable interface for data consumption, A Stable Interface for Data Consumption, SQL-First Semantic Execution

semantic models, governance rules on top of tables, Bridging Semantics to Execution

continuous integration and continuous delivery (CI/CD), Continuous integration and continuous delivery for SQL pipelines-Continuous integration and continuous delivery for SQL pipelines

correlation and association

Pearson correlation, Correlation and Association

Spearman correlation, Correlation and Association

cost-based optimization

BigQuery, Query Planning and Cost-Based Optimization

execution plan as DAG, Query Planning and Cost-Based Optimization

query planning and, Query Planning and Cost-Based Optimization-Query Planning and Cost-Based Optimization

basic information, Query Planning and Cost-Based Optimization

EXPLAIN ANALYZE, Query Planning and Cost-Based Optimization

EXPLAIN command, Query Planning and Cost-Based Optimization

Snowflake, Query Planning and Cost-Based Optimization

Query Profile tool, Query Planning and Cost-Based Optimization

“speaking the optimizer’s language”, Query Planning and Cost-Based Optimization

costs

cloud data warehouse pay-as-you-go, Cost-optimization strategies

BigQuery on-demand versus flat rate, BigQuery: Navigating cost-effective analysis

Snowflake Auto-Suspend, Snowflake: Architecting for cost-aware computation

Snowflake compute credits, Snowflake: Architecting for cost-aware computation

Snowflake per-second compute and storage billing, Snowflake cost optimization best practices

comparison chart of architecture cost models, Hybrid and Multi-Cloud Patterns

continuously running Spark stream, GCS to Iceberg and Spark structured streaming (Bronze layer)

cost efficiency of ecommerce data pipeline, Pipeline Overview and Design Principles

hyperparameter tuning, Best practices and cost considerations

input shaping before expensive LLM calls, LLM Functions Inside SQL

chunk size and overlap, Designing an embeddings table

tips for input and output, Structured outputs

modularity affecting, Modularity and composability

cross-validation (CV), Cross-validation-Interpretation of results

interpretation of results, Interpretation of results

CTEs (common table expressions), Common Table Expressions and Subquery Factoring

modularizing a query, Common Table Expressions and Subquery Factoring

RAG with context CTE, Grounded Generation (the “G” in RAG)

ticket CTE join point, Grounded Generation (the “G” in RAG)

recursive CTEs, Recursive CTEs for Hierarchical and Iterative Queries-Recursive CTEs for Hierarchical and Iterative Queries

termination, Recursive CTEs for Hierarchical and Iterative Queries

customer-support knowledge base built

about this GenAI example, Generative AI in SQL

agents generating SQL, SQL-Based Agents and MCPs-When to use MCP versus custom tools

about agents, SQL as an agent tool, SQL-Based Agents and MCPs

building a SQL-generating agent, Building a SQL-generating agent-Building a SQL-generating agent

connecting an agent to MCP, Connecting an agent to MCP

failures handled gracefully, Handling failures gracefully

guardrails added, Adding guardrails-Required filters in tool signatures

MCP server with SQL tools, SQL agents via MCP

OpenAI Agents SDK, SQL-Based Agents and MCPs, Building a SQL-generating agent, Building a SQL-generating agent

postgres-mcp server, Using postgres-mcp

PostgreSQL local instance with Docker, SQL-Based Agents and MCPs

schema context provided to agents, Enhancing accuracy with schema context-Enhancing accuracy with schema context

when to use MCP versus custom tools, When to use MCP versus custom tools

code online, Generative AI in SQL

embeddings, search, and retrieval, Embeddings, Search, and Retrieval-Multimodal embeddings

designing an embeddings table, Designing an embeddings table-Designing an embeddings table

embeddings as semantic join keys, Embeddings as semantic join keys

hybrid search of lexical, vector, reranking, Hybrid search (lexical + vector + reranking)-Hybrid search (lexical + vector + reranking)

multimodal embeddings, Multimodal embeddings

query-time semantic retrieval, Query-time semantic retrieval with vector search-Query-time semantic retrieval with vector search

LLMs extending not replacing, Generative AI Inside SQL

remote model connecting to Gemini, LLM Functions Inside SQL

retrieval-augmented generation, Grounded Generation (the “G” in RAG)-Grounded Generation (the “G” in RAG)

text generation

free-form text summarizing tickets, Generating free-form text-Generating free-form text

structured outputs for ticket triage, Structured outputs-Structured outputs

CV (cross-validation), Cross-validation-Interpretation of results

interpretation of results, Interpretation of results

Cypher language of Neo4j, What Is a Knowledge Graph?, Property Graphs and Cypher

hybrid with SQL via PostgreSQL, PostgreSQL and GraphDB

Neo4j and Cypher together, Graph Technologies: Neo4j and Cypher

readability of queries, Querying with Cypher Versus SQL

querying with Cypher versus SQL, Querying with Cypher Versus SQL-Querying with Cypher Versus SQL

SQL versus, Graph Technologies: Neo4j and Cypher

D

D-statistic, Normality Testing

DAGs (directed acyclic graphs)

BigQuery execution plans as, Query Planning and Cost-Based Optimization

dbt documenting pipeline, Lineage and documentation

lineage diagrams, Lineage and discovery: Seeing the forest and the trees

orchestration of data pipeline processes, Automation and orchestration, Orchestration and scheduling

Apache Airflow, Alternative Orchestration Options

Google Cloud Workflows, Orchestration and Workflow Management

data as a product, Formalizing Data Agreements: Data Contracts

data mesh and, Data Mesh and Data Products-Data Mesh and Data Products

data contracts, Formalizing Data Agreements: Data Contracts, Data Mesh and Data Products

data as a product, Formalizing Data Agreements: Data Contracts

tooling emerging around, Formalizing Data Agreements: Data Contracts

data engineering a SQL data pipeline

about the project, Building Data Engineering Solutions with SQL

architecture of data pipeline, Pipeline Overview and Design Principles

design principles, Pipeline Overview and Design Principles

code online, Building Data Engineering Solutions with SQL

dbt project, Tests and documentation

helper Cloud Run Job, GCS to Iceberg and Spark structured streaming (Bronze layer)

orchestration and scheduling configurations, Alternative Orchestration Options

Producers, Create a BigQuery dataset and BigLake connection

consume layer, Consume Layer-Live Data in Consume Layer

bridging semantics to execution, Bridging Semantics to Execution-Physical models versus semantic models in practice

live data in the consume layer, Live Data in Consume Layer-Live Data in Consume Layer

operationalizing the semantic layer, Operationalizing the Semantic layer

semantic contract tests, Semantic Contracts are Testable Contracts

SQL-first semantic execution, SQL-First Semantic Execution-The dbt Semantic layer (MetricFlow)

stable interface for data consumption, A Stable Interface for Data Consumption

data ingestion, Pipeline Overview and Design Principles, Data Ingestion in a SQL-Centric Architecture-GCS to Iceberg with Spark batch processing (Bronze layer)

about approach to data ingestion, Data Ingestion in a SQL-Centric Architecture

batch ingestion, Batch Ingestion for Bulk Data-GCS to Iceberg with Spark batch processing (Bronze layer)

creating Pub/Sub topic and subscription, Create Pub/Sub topic and subscription

streaming ingestion, Streaming Ingestion with Spark Structured Streaming and a Flavor of Apache Flink SQL-Apache Flink SQL

Google Cloud setup, Setting Up the GCP Environment-Create a BigQuery dataset and BigLake connection

BigLake (Iceberg) metastore setup, Set up BigLake (Iceberg) metastore

BigQuery dataset and BigLake connection, Create a BigQuery dataset and BigLake connection

creating a cloud storage bucket, Create a cloud storage bucket

creating a GCP project, Create a GCP project

creating Pub/Sub topic and subscription, Create Pub/Sub topic and subscription

credentials for authentication, Create a BigQuery dataset and BigLake connection

enabling required APIs, Enable required APIs

free tier setup, Setting Up the GCP Environment

Google Cloud Shell or gcloud CLI, Setting Up the GCP Environment

Lambda architecture, Building Data Engineering Solutions with SQL

data ingestion, Streaming Ingestion with Spark Structured Streaming and a Flavor of Apache Flink SQL

design principle, Pipeline Overview and Design Principles

why Lambda architecture, Pipeline Overview and Design Principles

orchestration and workflow management, Orchestration and Workflow Management-Alternative Orchestration Options

alternative orchestration options, Alternative Orchestration Options

Google Cloud Workflows, Orchestration and Workflow Management

Google Cloud Workflows orchestration pipeline, Cloud Workflows Orchestration Pipeline-Cloud Workflows Orchestration Pipeline

scheduling the workflows, Scheduling the Workflows

workflow YAML files defined, Orchestration and Workflow Management

security and governance, Security and Governance Considerations-Security Versus Semantics: Consistent Definitions as Control

dataset and schema boundaries, Dataset and Schema Boundaries as Governance

identity and execution boundaries, Identity and Execution Boundaries

semantic consistency, Security Versus Semantics: Consistent Definitions as Control

storage in data lakehouse, Pipeline Overview and Design Principles, Storage Layer-Auditing and recovery: Time travel with SQL

advantages of lakehouses, The Data Lakehouse

decoupling storage and compute, The Data Lakehouse

Iceberg data storage and management, Data Storage and Management with Apache Iceberg-Auditing and recovery: Time travel with SQL

Iceberg time travel to query historical data, Auditing and recovery: Time travel with SQL-Auditing and recovery: Time travel with SQL

open table formats, Open table formats

transformation and data modeling, Transformation and Data Modeling with SQL-Tests and documentation

dbt on BigQuery, Transformation and Data Modeling with SQL-Tests and documentation

dbt Silver and Gold model design, Silver and Gold Model Design in dbt: Building a Medallion Architecture-Tests and documentation

tests and documentation, Tests and documentation-Tests and documentation

data ingestion, Data ingestion

batch ingestion, Data ingestion, Batch Ingestion for Bulk Data-GCS to Iceberg with Spark batch processing (Bronze layer)

API to Google Cloud Storage, API to GCS (landing zone)-API to GCS (landing zone)

GCS to Iceberg and Spark (Bronze), GCS to Iceberg with Spark batch processing (Bronze layer)-GCS to Iceberg with Spark batch processing (Bronze layer)

implementation approach, Batch Ingestion for Bulk Data

ecommerce SQL data pipeline, Pipeline Overview and Design Principles, Data Ingestion in a SQL-Centric Architecture-GCS to Iceberg with Spark batch processing (Bronze layer)

about approach to data ingestion, Data Ingestion in a SQL-Centric Architecture

batch ingestion, Batch Ingestion for Bulk Data-GCS to Iceberg with Spark batch processing (Bronze layer)

creating Pub/Sub topic and subscription, Create Pub/Sub topic and subscription

streaming ingestion, Streaming Ingestion with Spark Structured Streaming and a Flavor of Apache Flink SQL-Apache Flink SQL

streaming ingestion, Data ingestion, Streaming Ingestion with Spark Structured Streaming and a Flavor of Apache Flink SQL-Apache Flink SQL

Apache Flink SQL instead of Spark, Apache Flink SQL

data file compaction, Data Storage and Management with Apache Iceberg

GCS to Iceberg and Spark (Bronze), GCS to Iceberg and Spark structured streaming (Bronze layer)-GCS to Iceberg and Spark structured streaming (Bronze layer)

implementation approach, Data Ingestion in a SQL-Centric Architecture

Lambda style ingestion pattern, Streaming Ingestion with Spark Structured Streaming and a Flavor of Apache Flink SQL

Pub/Sub to Google Cloud Storage, Pub/Sub to GCS (landing zone)

why Spark Structured Streaming, Why Spark in a SQL-centric architecture?

data lakehouses

advantages of lakehouses, The Data Lakehouse

open table formats, Open table formats

decoupling storage and compute, The Data Lakehouse

ELT workflows and lakehouse modeling, Lakehouse Modeling and ELT Workflows

storage for ecommerce data pipeline, Pipeline Overview and Design Principles, Storage Layer-Auditing and recovery: Time travel with SQL

creating GCS bucket, Create a cloud storage bucket

design principle, Pipeline Overview and Design Principles

data lineage, Lineage and discovery: Seeing the forest and the trees

dbt capturing, Lineage and documentation

dependency management, Data lineage and dependency management

lineage diagrams as DAGs, Lineage and discovery: Seeing the forest and the trees

logging GenAI outputs, Run lineage and versioned artifacts

missing lineage as an anti-pattern, Missing Lineage and Monitoring

data mesh and data as a product, Data Mesh and Data Products-Data Mesh and Data Products

data modeling with SQL (see modeling with SQL)

data pipelines

architectural patterns for data pipelines, Architectural Patterns for Data Pipelines

batch versus streaming, Batch Versus Streaming: Choosing the Right SQL Pipeline Approach-Batch Versus Streaming: Choosing the Right SQL Pipeline Approach

data mesh and data as a product, Data Mesh and Data Products-Data Mesh and Data Products

data sharing, Beyond ETL/ELT and Streaming: Zero-ETL and Data Sharing, Beyond ETL/ELT and Streaming: Zero-ETL and Data Sharing

ELT and ETL transformations, Transformation Patterns: ETL, ELT, and Streaming-Transformation Patterns: ETL, ELT, and Streaming

ELT for most modern data pipelines, Transformation patterns

streaming transformations, Transformation Patterns: ETL, ELT, and Streaming

zero-ETL, Beyond ETL/ELT and Streaming: Zero-ETL and Data Sharing, Beyond ETL/ELT and Streaming: Zero-ETL and Data Sharing

architectures for scalability, Architectures for Scalability-Hybrid and Multi-Cloud Patterns

scalability built and operationalized, Building and Operationalizing Scalable SQL Data Platforms-Avoiding Common Architectural Anti-Patterns

data contracts, Formalizing Data Agreements: Data Contracts-Formalizing Data Agreements: Data Contracts, Data Mesh and Data Products

definition of data pipeline, Architecting SQL-Driven Solutions

designing modular SQL data pipelines, Foundational Design Principles for SQL Data Pipelines-Service-level agreements and data freshness

automation and orchestration, Automation and orchestration

clean SQL pipeline organization, Maintainability and readability

data lineage and dependency management, Data lineage and dependency management

idempotency and rerunnability, Idempotency and rerunnability

layering SQL models, Maintainability and readability

LEGO-style pipeline, Modularity and composability

maintainability and readability, Maintainability and readability

modularity and composability, Modularity and composability

service-level agreements and data freshness, Service-level agreements and data freshness

end-to-end SQL data pipeline built

about the project, Building Data Engineering Solutions with SQL

architecture of data pipeline, Pipeline Overview and Design Principles

code online, Building Data Engineering Solutions with SQL

consume layer, Consume Layer-Live Data in Consume Layer

data ingestion, Pipeline Overview and Design Principles, Data Ingestion in a SQL-Centric Architecture-GCS to Iceberg with Spark batch processing (Bronze layer)

data ingestion via Pub/Sub, Create Pub/Sub topic and subscription

dbt project online, Tests and documentation

design principles, Pipeline Overview and Design Principles

Google Cloud setup, Setting Up the GCP Environment-Create a BigQuery dataset and BigLake connection

helper Cloud Run Job online, GCS to Iceberg and Spark structured streaming (Bronze layer)

orchestration and workflow management, Orchestration and Workflow Management-Alternative Orchestration Options

Producers online, Create a BigQuery dataset and BigLake connection

security and governance, Security and Governance Considerations-Security Versus Semantics: Consistent Definitions as Control

storage in data lakehouse, Pipeline Overview and Design Principles, Storage Layer-Auditing and recovery: Time travel with SQL

transformation and data modeling, Transformation and Data Modeling with SQL-Tests and documentation

observability, Observability: Knowing what's working (and what's not)

scalability, Architectures for Scalability-Hybrid and Multi-Cloud Patterns

cloud native scaling, Cloud Native Scaling

comparison chart of architectures, Hybrid and Multi-Cloud Patterns

concurrency and isolation in practice, Concurrency and Isolation in Practice

hybrid and multi-cloud patterns, Hybrid and Multi-Cloud Patterns-Hybrid and Multi-Cloud Patterns

traditional RDBMS scaling, Traditional RDBMS Scaling

scalability built and operationalized, Building and Operationalizing Scalable SQL Data Platforms-Avoiding Common Architectural Anti-Patterns

continuous integration and continuous delivery, Continuous integration and continuous delivery for SQL pipelines-Continuous integration and continuous delivery for SQL pipelines

Infrastructure as Code, Infrastructure as Code (IaC) in scaling and for data platforms-Avoiding Common Architectural Anti-Patterns

layered architecture, Layered Architecture: Separating Storage, Transformation, and Orchestration-Connecting back to the architecture

scaling from design to production, From Design to Production: Scaling SQL Data Solutions-Avoiding Common Architectural Anti-Patterns

simplified with one big table approach, One Big Table: The Wide Denormalized Table Approach

data quality testing, Implementing testing strategies

data science SQL

advanced statistical analysis, Advanced Statistical Analysis-Kruskal—Wallis test

about SQL capabilities, Advanced Statistical Analysis

data processing and analysis, Data Processing and Analysis-Outlier Detection

correlation and association, Correlation and Association

dataset statistical description, Dataset Statistical Description-Dataset Statistical Description

outlier detection, Outlier Detection

regression from aggregates, Regression from Aggregates

machine learning, Machine Learning-Drift

bringing ML to the data, Integrating ML and LLMs in SQL Platforms, Machine Learning

providers of SQL-native ML platforms, Machine Learning-Machine Learning

SQL integration, Integrating ML and LLMs in SQL Platforms-Integrating ML and LLMs in SQL Platforms

when to use, Machine Learning

modern SQL, SQL in Data Science

current SQL:2023, Evolution of SQL, The Graph Revolution with SQL:2023, Navigating Graph Structures in SQL

limitations, SQL in Data Science

strengths, SQL in Data Science

data sharing, Beyond ETL/ELT and Streaming: Zero-ETL and Data Sharing, Beyond ETL/ELT and Streaming: Zero-ETL and Data Sharing

data transformations (see transformation and modeling in layered architecture)

data types

JSON data type in SQL:2023, The Graph Revolution with SQL:2023

semi-structured data, Semi-Structured Data Handling in SQL

unstructured data, Dealing with Unstructured Data in SQL

vector embeddings in Snowflake, Integrating ML and LLMs in SQL Platforms

Databricks

Delta Sharing protocol, Beyond ETL/ELT and Streaming: Zero-ETL and Data Sharing

ELT pattern, Transformation patterns

GenAI platform, Generative AI in SQL

dataset statistical description, Dataset Statistical Description-Dataset Statistical Description

weather dataset setup script online, Dataset Statistical Description

datasets

AI-ready data, AI-Ready Data

Citibike public dataset

principal component analysis, Principal component analysis

recommendation systems, Recommendation systems

time-series forecasting, Time-series forecasting

unsupervised learning models, Unsupervised learning models-Unsupervised learning models

data portion held for evaluation, Cross-validation

statistical description, Dataset Statistical Description-Dataset Statistical Description

weather dataset

setup script online, Dataset Statistical Description

statistical description, Dataset Statistical Description-Dataset Statistical Description

date functions

SQL dialect comparison, Comparison of SQL Dialects over Time

window functions, Advanced Window Functions for Analytic Queries, Advanced Window Functions for Analytic Queries

dbt

best practices, Transformation and Data Modeling with SQL

BigQuery, Transformation and Data Modeling with SQL-Tests and documentation

exposures feature, Exposures and ownership

governance by, Governance via dbt (Documentation, Lineage, and Exposures)-Testing and quality rules

exposures and ownership, Exposures and ownership

lineage and documentation, Lineage and documentation

testing and quality rules, Testing and quality rules

live data in the consume layer, Live Data in Consume Layer-Live Data in Consume Layer

project online, Tests and documentation

semantic contract tests, Semantic Contracts are Testable Contracts

Semantic layer, The dbt Semantic layer (MetricFlow)-The dbt Semantic layer (MetricFlow)

service account, Identity and Execution Boundaries

setting up, Identity and Execution Boundaries

Silver and Gold tiers, Transformation and Data Modeling with SQL

Silver and Gold model design, Silver and Gold Model Design in dbt: Building a Medallion Architecture-Tests and documentation

tests and documentation, Tests and documentation-Tests and documentation

generic versus singular tests, Tests and documentation

debugging
daily views of logs, Core run logs and daily observability
dependency resolution and DAGs, Orchestration and scheduling
EXPLAIN ANALYZE, Query Planning and Cost-Based Optimization
GCS bucket error about billing, Create a cloud storage bucket
idempotency for easier recovery, Idempotency and rerunnability
time travel to query historical data via Iceberg, Auditing and recovery: Time travel with SQL-Auditing and recovery: Time travel with SQL
deciles
equal-frequency binning, Equal-frequency (deciles)
equal-width binning, Equal-width (deciles)
degrees of freedom, One-way ANOVA: A handmade analysis
Dehghani, Zhamak, Data Mesh and Data Products
dependencies
data lineage, Data lineage and dependency management
transitive dependency, Third Normal Form—Structured, Relational Modeling
deploying and monitoring models, Model Deployment and Monitoring-Drift
deployment architectures, Practical Deployment Architectures
chart comparing, Practical Deployment Architectures
dialect comparison over time, Comparison of SQL Dialects over Time-Comparison of SQL Dialects over Time
advanced features, Comparison of SQL Dialects over Time

auto-increment keys, Comparison of SQL Dialects over Time

convergence in modern SQL, Comparison of SQL Dialects over Time

date/time functions, Comparison of SQL Dialects over Time

GenAI platforms, Generative AI in SQL

GoogleSQL pipe syntax, GoogleSQL Pipe Syntax-When not to use pipe syntax

building queries, GoogleSQL Pipe Syntax

description of pipe syntax, GoogleSQL Pipe Syntax

operators, Pipe Operators Overview-Combining results with UNION, INTERSECT, EXCEPT

syntax and semantics of pipe queries, Core Syntax and Semantics of Pipe Queries

identifier case sensitivity, Comparison of SQL Dialects over Time

newer cloud platforms, Comparison of SQL Dialects over Time

string functions, Comparison of SQL Dialects over Time

vendor-specific extensions, Comparison of SQL Dialects over Time

window functions, Advanced Window Functions for Analytic Queries

directed acyclic graphs (see DAGs (directed acyclic graphs))

discovery, Lineage and discovery: Seeing the forest and the trees

Docker for agents generating SQL, SQL-Based Agents and MCPs

postgres-mcp running with Docker, Using postgres-mcp

documentation as best practice, Best practices for the consumption layer

dbt best practice, Transformation and Data Modeling with SQL

tests and documentation, Tests and documentation-Tests and documentation

dbt generating, Lineage and documentation

DOOMQL online, Recursive CTEs for Hierarchical and Iterative Queries

drift in deployed models, Model monitoring and retraining strategies, Drift

DuckDB embedded analytics, Embedded Analytics with DuckDB

DuckDB-Wasm browser extension, Embedded Analytics with DuckDB

E

ECDF (empirical cumulative distribution function), Normality Testing

ecommerce SQL data pipeline built

about the data pipeline project, Building Data Engineering Solutions with SQL

architecture of data pipeline, Pipeline Overview and Design Principles

design principles, Pipeline Overview and Design Principles

code online, Building Data Engineering Solutions with SQL

dbt project, Tests and documentation

helper Cloud Run Job, GCS to Iceberg and Spark structured streaming (Bronze layer)

orchestration and scheduling configurations, Alternative Orchestration Options

Producers, Create a BigQuery dataset and BigLake connection

consume layer, Consume Layer-Live Data in Consume Layer

bridging semantics to execution, Bridging Semantics to Execution-Physical models versus semantic models in practice

live data in the consume layer, Live Data in Consume Layer-Live Data in Consume Layer

operationalizing the semantic layer, Operationalizing the Semantic layer

semantic contract tests, Semantic Contracts are Testable Contracts

SQL-first semantic execution, SQL-First Semantic Execution-The dbt Semantic layer (MetricFlow)

stable interface for data consumption, A Stable Interface for Data Consumption

data ingestion, Pipeline Overview and Design Principles, Data Ingestion in a SQL-Centric Architecture-GCS to Iceberg with Spark batch processing (Bronze layer)

about approach to data ingestion, Data Ingestion in a SQL-Centric Architecture

batch ingestion, Batch Ingestion for Bulk Data-GCS to Iceberg with Spark batch processing (Bronze layer)

creating Pub/Sub topic and subscription, Create Pub/Sub topic and subscription

streaming ingestion, Streaming Ingestion with Spark Structured Streaming and a Flavor of Apache Flink SQL-Apache Flink SQL

Google Cloud setup, Setting Up the GCP Environment-Create a BigQuery dataset and BigLake connection

BigLake (Iceberg) metastore setup, Set up BigLake (Iceberg) metastore

BigQuery dataset and BigLake connection, Create a BigQuery dataset and BigLake connection

creating a cloud storage bucket, Create a cloud storage bucket

creating a GCP project, Create a GCP project

creating Pub/Sub topic and subscription, Create Pub/Sub topic and subscription

credentials for authentication, Create a BigQuery dataset and BigLake connection

enabling required APIs, Enable required APIs

free tier setup, Setting Up the GCP Environment

Google Cloud Shell or gcloud CLI, Setting Up the GCP Environment

Lambda architecture, Building Data Engineering Solutions with SQL

data ingestion, Streaming Ingestion with Spark Structured Streaming and a Flavor of Apache Flink SQL

design principle, Pipeline Overview and Design Principles

why Lambda architecture, Pipeline Overview and Design Principles

orchestration and workflow management, Orchestration and Workflow Management-Alternative Orchestration Options

alternative orchestration options, Alternative Orchestration Options

Google Cloud Workflows, Orchestration and Workflow Management

Google Cloud Workflows orchestration pipeline, Cloud Workflows Orchestration Pipeline-Cloud Workflows Orchestration Pipeline

scheduling the workflows, Scheduling the Workflows

workflow YAML files defined, Orchestration and Workflow Management

security and governance, Security and Governance Considerations-Security Versus Semantics: Consistent Definitions as Control

dataset and schema boundaries, Dataset and Schema Boundaries as Governance

identity and execution boundaries, Identity and Execution Boundaries

semantic consistency, Security Versus Semantics: Consistent Definitions as Control

storage in data lakehouse, Pipeline Overview and Design Principles, Storage Layer-Auditing and recovery: Time travel with SQL

advantages of lakehouses, The Data Lakehouse

decoupling storage and compute, The Data Lakehouse

Iceberg data storage and management, Data Storage and Management with Apache Iceberg-Auditing and recovery: Time travel with SQL

Iceberg time travel to query historical data, Auditing and recovery: Time travel with SQL-Auditing and recovery: Time travel with SQL

open table formats, Open table formats

transformation and data modeling, Transformation and Data Modeling with SQL-Tests and documentation

dbt on BigQuery, Transformation and Data Modeling with SQL-Tests and documentation

dbt Silver and Gold model design, Silver and Gold Model Design in dbt: Building a Medallion Architecture-Tests and documentation

tests and documentation, Tests and documentation-Tests and documentation

ELT (extract, load, transform), Transformation Patterns: ETL, ELT, and Streaming

choosing ETL or ELT, Transformation Patterns: ETL, ELT, and Streaming

lakehouse modeling and ELT workflows, Lakehouse Modeling and ELT Workflows

most modern data pipelines, Transformation patterns

streaming transformations, Transformation Patterns: ETL, ELT, and Streaming

embedding-based search, Semantic search with embeddings, Embeddings, Search, and Retrieval-Multimodal embeddings

designing an embeddings table, Designing an embeddings table-Designing an embeddings table

embeddings as semantic join keys, Embeddings as semantic join keys

hybrid search of lexical, vector, reranking, Hybrid search (lexical + vector + reranking)-Hybrid search (lexical + vector + reranking)

multimodal embeddings, Multimodal embeddings

query-time semantic retrieval, Query-time semantic retrieval with vector search-Query-time semantic retrieval with vector search

empirical cumulative distribution function (ECDF), Normality Testing

end-to-end testing, Implementing testing strategies

equal-frequency binning (deciles), Equal-frequency (deciles)

equal-width binning (deciles), Equal-width (deciles)

ETL (extract, transform, load), Transformation Patterns: ETL, ELT, and Streaming-Transformation Patterns: ETL, ELT, and Streaming

choosing ETL or ELT, Transformation Patterns: ETL, ELT, and Streaming

streaming transformations, Transformation Patterns: ETL, ELT, and Streaming

evaluating and validating models, Model Evaluation and Validation

evaluation metrics, Evaluation metrics

productionizing generative AI, Evaluation and review loops

building and evolving evaluation suites, Building and evolving eval suites

human reviews, Building and evolving eval suites, Building and evolving eval suites

logging evaluation outcomes, Building and evolving eval suites

evolution of SQL

about SQL, Evolution of SQL

core philosophy maintained, Evolution of SQL

current SQL:2023, Evolution of SQL, The Graph Revolution with SQL:2023, Navigating Graph Structures in SQL

dialect comparison, Comparison of SQL Dialects over Time-Comparison of SQL Dialects over Time

GenAI platforms, Generative AI in SQL

GoogleSQL pipe syntax, GoogleSQL Pipe Syntax-When not to use pipe syntax

historical overview

Dr. Codd's relational model, Historical Overview: From Relational to Modern SQL

SEQUEL, Historical Overview: From Relational to Modern SQL

SQL-86, Standard SQL-86: Laying the Foundations

SQL-89, Standard SQL-89: Adding Integrity and Security

SQL-92, Standard SQL-92: Adding Modeling Capabilities

SQL:1999, Standard SQL:1999—Handling Industry Needs

SQL:2003 to SQL 2008, From the SQL:2003 Standard to SQL:2008

SQL:2011 to SQL:2016, From the SQL:2011 Standard to SQL:2016

SQL:2023, The Graph Revolution with SQL:2023

EXPLAIN command

anti-pattern of ignoring, Ignoring query plan feedback

cost-based optimizer choices, Query Planning and Cost-Based Optimization

EXPLAIN ANALYZE, Query Planning and Cost-Based Optimization

using regularly, Query Planning and Cost-Based Optimization

monitoring and profiling queries, BigQuery: Monitoring and profiling queries

EXPLAIN ANALYZE, BigQuery: Monitoring and profiling queries

exposures feature of dbt, Exposures and ownership

EXTERNAL_QUERY federated query engine (BigQuery), Hybrid and Multi-Cloud Patterns

extract, load, transform (see ELT (extract, load, transform))

extract, transform, load (see ETL (extract, transform, load))

F

feature engineering patterns, Advanced Feature Engineering Patterns-Principal component analysis

categorical encoding, Categorical encoding-Feature hashing

about categorical encoding, Categorical encoding

feature hashing, Feature hashing

frequency encoding, Frequency encoding

one-hot encoding, One-hot encoding

principal component analysis, Principal component analysis-Principal component analysis

about PCA, Principal component analysis

quantile binning, Quantile binning-Equal-width (deciles)

about quantile binning, Quantile binning

equal-frequency binning, Equal-frequency (deciles)

equal-width binning, Equal-width (deciles)

target encoding with K-fold out-of-fold mean, Target encoding (leakage-aware) with K-fold out-of-fold mean-Target encoding (leakage-aware) with K-fold out-of-fold mean

about K-fold out-of-fold mean, Target encoding (leakage-aware) with K-fold out-of-fold mean

about target encoding, Target encoding (leakage-aware) with K-fold out-of-fold mean

text feature engineering, Text feature engineering-Character n-grams

about encoding free-form text, Text feature engineering

character n-grams, Character n-grams

tokenization, Tokenization

federated query engines, Hybrid and Multi-Cloud Patterns

flat tables (see one big table (OBT))

Flink SQL (see Apache Flink SQL)

frequency encoding, Frequency encoding

future of SQL, The Future of SQL-Declarative Governance and Policies

AI and autonomous databases, AI and Autonomous Databases

declarative governance and policies, Declarative Governance and Policies

quantum computing-assisted optimization, The Quantum Horizon: Quantum-Assisted Optimization

G

GCP (see Google Cloud Platform (GCP))

GCS (see Google Cloud Storage (GCS))

Gemini models (Vertex AI)

- BigQuery connecting to Gemini, LLM Functions Inside SQL
- list of models and capabilities online, LLM Functions Inside SQL
- remote model connecting to Gemini, LLM Functions Inside SQL
  - documentation online, LLM Functions Inside SQL

generalized inverted index (GIN), Indexing and Efficient Data Access

generative AI (GenAI) in SQL

- about GenAI potential and concerns, Generative AI in SQL
  - SQL-native GenAI definition, Generative AI in SQL
  - warehouse-native GenAI definition, Generative AI in SQL
- agents generating SQL, SQL-Based Agents and MCPs-When to use MCP versus custom tools
  - about agents, SQL as an agent tool, SQL-Based Agents and MCPs
  - building a SQL-generating agent, Building a SQL-generating agent-Building a SQL-generating agent
  - connecting an agent to MCP, Connecting an agent to MCP
  - failures handled gracefully, Handling failures gracefully
  - guardrails added, Adding guardrails-Required filters in tool signatures

MCP server with SQL tools, SQL agents via MCP

OpenAI Agents SDK, SQL-Based Agents and MCPs, Building a SQL-generating agent, Building a SQL-generating agent

postgres-mcp server, Using postgres-mcp

PostgreSQL local instance with Docker, SQL-Based Agents and MCPs

schema context provided to agents, Enhancing accuracy with schema context-Enhancing accuracy with schema context

when to use MCP versus custom tools, When to use MCP versus custom tools

customer-support knowledge base (see customer-support knowledge base built)

embeddings, search, and retrieval, Embeddings, Search, and Retrieval-Multimodal embeddings

designing an embeddings table, Designing an embeddings table-Designing an embeddings table

embeddings as semantic join keys, Embeddings as semantic join keys

hybrid search of lexical, vector, reranking, Hybrid search (lexical + vector + reranking)-Hybrid search (lexical + vector + reranking)

multimodal embeddings, Multimodal embeddings

query-time semantic retrieval, Query-time semantic retrieval with vector search-Query-time semantic retrieval with vector search

GenAI inside SQL, Generative AI Inside SQL-When SQL Should Run GenAI Workloads

AI-ready data, AI-Ready Data

chart comparing SQL to GenAI approaches, Generative AI Inside SQL

core SQL patterns plus GenAI, Core Patterns of SQL + GenAI-SQL as an agent tool

GenAI as “just another transformation”, Generative AI Inside SQL

inline LLM patterns best for batch or scheduled jobs, SQL as an agent tool

SQL giving what, GenAI giving why, Generative AI Inside SQL

LLM functions inside SQL, LLM Functions Inside SQL-Structured outputs

generating free-form text, Generating free-form text-Generating free-form text

generating structured outputs, Structured outputs-Structured outputs

input shaping before expensive LLM calls, LLM Functions Inside SQL

remote model connecting to Gemini, LLM Functions Inside SQL

tips for input and output, Structured outputs

tips for producing structured outputs, Structured outputs

productionizing, Productionizing Generative AI in the Data Warehouse-Production Readiness Checklist

artifact flow, Productionizing Generative AI in the Data Warehouse

building and evolving evaluation suites, Building and evolving eval suites

checklist for production readiness, Production Readiness Checklist

deployment architectures, Practical Deployment Architectures

five interconnected components, Productionizing Generative AI in the Data Warehouse

outputs as data products, Logging, Monitoring, and Evaluating GenAI Outputs-Core run logs and daily observability

retrieval configuration, Run lineage and versioned artifacts

security and privacy controls, Security and Privacy Controls

versioned artifacts with explicit ownership, Run lineage and versioned artifacts, Practical Deployment Architectures

retrieval-augmented generation, Grounded Generation (the “G” in RAG)-Grounded Generation (the “G” in RAG)

about RAG, Patterns: RAG, Agents, and MCPs

when SQL should run GenAI workloads, When SQL Should Run GenAI Workloads

where to run workloads, Generative AI in SQL

platform capabilities, Generative AI in SQL

platforms available, Generative AI in SQL

GIN (generalized inverted index), Indexing and Efficient Data Access

GiST indexes, Indexing and Efficient Data Access

GitHub

book supplemental material, Using Code Examples

customer-support knowledge base code, Generative AI in SQL

ecommerce SQL data pipeline code, Building Data Engineering Solutions with SQL

dbt project, Tests and documentation

helper Cloud Run Job, GCS to Iceberg and Spark structured streaming (Bronze layer)

orchestration and scheduling configurations, Alternative Orchestration Options

Producers, Create a BigQuery dataset and BigLake connection

Google BigLake (see BigLake (Google) metastore for Iceberg tables)

Google BigQuery (see BigQuery (Google))

Google BigQuery ML (see BigQuery ML (Google))

Google Cloud Platform (GCP)

Cloud Run

API to Google Cloud Storage, Batch Ingestion for Bulk Data-API to GCS (landing zone)

enabling, Enable required APIs

helper Cloud Run Job online, GCS to Iceberg and Spark structured streaming (Bronze layer)

managed service reducing operational burden, Pipeline Overview and Design Principles

Pub/Sub to Google Cloud Storage, Streaming Ingestion with Spark Structured Streaming and a Flavor of Apache Flink SQL, Pub/Sub to GCS (landing zone)

publishing dbt-generated documentation, Lineage and documentation

refreshing BigQuery external tables, GCS to Iceberg and Spark structured streaming (Bronze layer)

Google Cloud Scheduler, Scheduling the Workflows

Google Cloud Storage bucket created, Create a cloud storage bucket

error about billing, Create a cloud storage bucket

Google Cloud Workflows, Orchestration and Workflow Management

code uploaded to Google Cloud Storage, Cloud Workflows Orchestration Pipeline

DAG of tasks, Orchestration and Workflow Management

orchestration pipeline, Cloud Workflows Orchestration Pipeline-Cloud Workflows Orchestration Pipeline

scheduling the workflows, Scheduling the Workflows

workflow YAML files defined, Orchestration and Workflow Management

setting up ecommerce data pipeline, Setting Up the GCP Environment-Create a BigQuery dataset and BigLake connection

BigLake (Iceberg) metastore, Set up BigLake (Iceberg) metastore

credentials for authentication, Create a BigQuery dataset and BigLake connection

enabling required APIs, Enable required APIs

free tier setup, Setting Up the GCP Environment

Google Cloud Shell or gcloud CLI, Setting Up the GCP Environment

Google Cloud Storage (GCS)
API to GCS, API to GCS (landing zone)-API to GCS (landing zone)
creating bucket, Create a cloud storage bucket
error about billing, Create a cloud storage bucket
exporting model to, Making models available for predictions
Pub/Sub to GCS, Pub/Sub to GCS (landing zone)
Google Vizier, A compact example
GoogleSQL pipe syntax, GoogleSQL Pipe Syntax-When not to use pipe syntax
applied by examples
aggregation and post-filtering, Aggregation and post-filtering
filtering and selecting, Filtering and selecting
intermediate calculation, Using an intermediate calculation
joining after a transformation, Joining after a transformation
multistep aggregation without subqueries, Multistep aggregation (eliminating subqueries)
best practices, Best Practices and Style Guidelines for Pipe SQL Syntax
building queries, GoogleSQL Pipe Syntax
description of pipe syntax, GoogleSQL Pipe Syntax
documentation and research article online, When not to use pipe syntax
operators, Pipe Operators Overview-Combining results with UNION, INTERSECT, EXCEPT

AGGREGATE, The AGGREGATE operator

AS, The AS operator

chart overviewing operators, Pipe Operators Overview

DROP, The DROP and RENAME operators

EXTEND, The EXTEND operator-The EXTEND operator

JOIN, The JOIN operator-The JOIN operator

LIMIT, The LIMIT operator

RENAME, The DROP and RENAME operators

SELECT, The SELECT operator

SET, The SET operator

UNION, INTERSECT, EXCEPT, Combining results with UNION, INTERSECT, EXCEPT

WHERE, The WHERE operator-The WHERE operator

real-world use cases, Real-World Use Cases and Applications of Pipe SQL-Real-World Use Cases and Applications of Pipe SQL

style guidelines, Formatting and readability tips-When not to use pipe syntax

efficiency considerations, Efficiency considerations

formatting and readability, Formatting and readability tips-Formatting and readability tips

syntax and semantics of pipe queries, Core Syntax and Semantics of Pipe Queries

when not to use, When not to use pipe syntax

governance

about governance, Governance via dbt (Documentation, Lineage, and Exposures)

dbt for, Governance via dbt (Documentation, Lineage, and Exposures)-Testing and quality rules

exposures and ownership, Exposures and ownership

lineage and documentation, Lineage and documentation

testing and quality rules, Testing and quality rules

declarative governance in future SQL, Declarative Governance and Policies

ecommerce SQL data pipeline, Security and Governance Considerations-Security Versus Semantics: Consistent Definitions as Control

dataset and schema boundaries, Dataset and Schema Boundaries as Governance

identity and execution boundaries, Identity and Execution Boundaries

federated governance of data mesh, Data Mesh and Data Products, Data Mesh and Data Products

semantic consistency, Security Versus Semantics: Consistent Definitions as Control

graphs

knowledge graphs (see knowledge graphs)

SQL and graphs as complementary, SQL and Graphs: Complementary, Not Competing-SQL and Graphs: Complementary, Not Competing

SQL:2023 graph processing, The Graph Revolution with SQL:2023, Navigating Graph Structures in SQL

Property Graph Queries, Navigating Graph Structures in SQL

H

hash indexes in PostgreSQL, Indexing and Efficient Data Access

historical overview of SQL

dialect comparison, Comparison of SQL Dialects over Time-Comparison of SQL Dialects over Time

GenAI platforms, Generative AI in SQL

GoogleSQL pipe syntax, GoogleSQL Pipe Syntax-When not to use pipe syntax

Dr. Codd's relational model, Historical Overview: From Relational to Modern SQL

SEQUEL, Historical Overview: From Relational to Modern SQL

SQL-86, Standard SQL-86: Laying the Foundations

SQL-89, Standard SQL-89: Adding Integrity and Security

SQL-92, Standard SQL-92: Adding Modeling Capabilities

SQL:1999, Standard SQL:1999—Handling Industry Needs

SQL:2003 to SQL 2008, From the SQL:2003 Standard to SQL:2008

SQL:2011 to SQL:2016, From the SQL:2011 Standard to SQL:2016

SQL:2023, The Graph Revolution with SQL:2023

hybrid approach of batch and streaming, Batch Versus Streaming: Choosing the Right SQL Pipeline Approach

Kappa architecture, Batch Versus Streaming: Choosing the Right SQL Pipeline Approach

Lambda architecture, Batch Versus Streaming: Choosing the Right SQL Pipeline Approach

hybrid on-premises and cloud patterns, Hybrid and Multi-Cloud Patterns-Hybrid and Multi-Cloud Patterns

hybrid search of lexical, vector, reranking, Hybrid search (lexical + vector + reranking)-Hybrid search (lexical + vector + reranking)

hyperparameter tuning, Hyperparameter tuning-Best practices and cost considerations

about hyperparameter tuning, Hyperparameter tuning

best practices and cost considerations, Best practices and cost considerations

enabling in BigQuery ML, Hyperparameter tuning

example of training with automated tuning, A compact example

multiple objectives for tuning, Tuning objectives and multi-objective optimization

Hyperparameter Tuning with Python (Owen), Best practices and cost considerations

I

IaC (see Infrastructure as Code (IaC))

IAM, Identity and Execution Boundaries

BigQuery policies down to dataset level, Dataset and Schema Boundaries as Governance

division of labor, Identity and Execution Boundaries

Iceberg (see Apache Iceberg)

idempotency of data pipeline, Idempotency and rerunnability

identifier case sensitivity across SQL dialects, Comparison of SQL Dialects over Time

identity and execution boundaries, Identity and Execution Boundaries

indexes

- anti-patterns of indexes, Ignoring indexes (or not having the right index)
- B-tree indexes, Indexing and Efficient Data Access
  - ideal for typical queries, Indexing and Efficient Data Access
- partitioning and clustering as complements, Indexing and Efficient Data Access
- performance optimization via, Indexing and Efficient Data Access-Indexing and Efficient Data Access
  - performance cost on writes, Indexing and Efficient Data Access
- Snowflake and BigQuery, Indexing and Efficient Data Access, Indexing and Efficient Data Access
- tips for using, Indexing and Efficient Data Access
  - covering indexes, Indexing and Efficient Data Access
  - multicolumn indexes, Indexing and Efficient Data Access
  - partial indexes, Indexing and Efficient Data Access

Infrastructure as Code (IaC), Infrastructure as Code (IaC) in scaling and for data platforms-Avoiding Common Architectural Anti-Patterns

- BigQuery example, Infrastructure as Code (IaC) in scaling and for data platforms

data engineer responsibilities, Infrastructure as Code (IaC) in scaling and for data platforms

Snowflake example, Infrastructure as Code (IaC) in scaling and for data platforms

development versus production example, Infrastructure as Code (IaC) in scaling and for data platforms

inline enrichment, Inline LLM enrichment

integration testing, Implementing testing strategies

ISO (International Organization for Standardization), Standard SQL-86: Laying the Foundations

J

JSON

GenAI output

free-form text, Generating free-form text

structured outputs, Structured outputs

newline-delimited JSON (NDJSON)

API batch ingestion, API to GCS (landing zone)

code that writes to NDJSON file, Pub/Sub to GCS (landing zone)

Pub/Sub messages to GCS, Pub/Sub to GCS (landing zone)

semi-structured data handling by SQL, Semi-Structured Data Handling in SQL

SQL:2016 native support, From the SQL:2011 Standard to SQL:2016

SQL:2023 JSON data type and functions, The Graph Revolution with SQL:2023

K

k-fold cross-validation, Cross-validation-Interpretation of results

    interpretation of results, Interpretation of results

K-fold out-of-fold mean target encoding, Target encoding (leakage-aware) with K-fold out-of-fold mean-Target encoding (leakage-aware) with K-fold out-of-fold mean

    about K-fold out-of-fold mean, Target encoding (leakage-aware) with K-fold out-of-fold mean

    about target encoding, Target encoding (leakage-aware) with K-fold out-of-fold mean

k-means clustering, Unsupervised learning models-Unsupervised learning models

Kappa architecture, Batch Versus Streaming: Choosing the Right SQL Pipeline Approach

    layered architecture, Layered Architecture: Separating Storage, Transformation, and Orchestration

knowledge base example (see customer-support knowledge base built)

knowledge graphs

    description, Knowledge Graphs and Semantic Modeling: The New Frontier for SQL, What Is a Knowledge Graph?

        facts versus a fact table, What Is a Knowledge Graph?

    entity nodes and relationship edges, Knowledge Graphs and Semantic Modeling: The New Frontier for SQL, What Is a Knowledge Graph?

        property graphs, Property Graphs and Cypher

    Neo4j graph database, What Is a Knowledge Graph?

Cypher language, What Is a Knowledge Graph?

Neo4j and Cypher together, Graph Technologies: Neo4j and Cypher

querying with Cypher versus SQL, Querying with Cypher Versus SQL-Querying with Cypher Versus SQL

readability of queries, Querying with Cypher Versus SQL

SQL versus, Graph Technologies: Neo4j and Cypher

semantic modeling, Knowledge Graphs and Semantic Modeling: The New Frontier for SQL-Querying with Cypher Versus SQL

RDF and OWL, RDF and OWL: The Foundations of Semantic Modeling

semantic models and ontologies, Semantic Models and Ontologies-PostgreSQL and GraphDB

SQL and graphs as complementary, SQL and Graphs: Complementary, Not Competing-SQL and Graphs: Complementary, Not Competing

Web Ontology Language transition, Graph Technologies: Neo4j and Cypher

Kolmogorov-Smirnov (KS) test, Normality Testing

Kruskal—Wallis test, Kruskal—Wallis test

kurtosis, Dataset Statistical Description

L

Lambda architecture, Batch Versus Streaming: Choosing the Right SQL Pipeline Approach

data lakehouses capturing batch and streams, The Data Lakehouse

ecommerce SQL data pipeline, Building Data Engineering Solutions with SQL

data ingestion, Streaming Ingestion with Spark Structured Streaming and a Flavor of Apache Flink SQL

design principle, Pipeline Overview and Design Principles

why Lambda architecture, Pipeline Overview and Design Principles

layered architecture, Layered Architecture: Separating Storage, Transformation, and Orchestration

layered architecture, Layered Architecture: Separating Storage, Transformation, and Orchestration-Connecting back to the architecture

about layered architecture, Layered Architecture: Separating Storage, Transformation, and Orchestration

consumption layer, Consumption layer

best practices, Best practices for the consumption layer

Gold models and Output or Delivery phase, Connecting back to the architecture

data ingestion, Data ingestion

batch ingestion, Data ingestion

streaming ingestion, Data ingestion

layering SQL models, Maintainability and readability

orchestration and scheduling, Orchestration and scheduling-Orchestration and scheduling

storage and processing, Storage and processing

medallion architecture, Storage and processing

testing strategies per layer, Implementing testing strategies

transformation and modeling, Transformation and modeling-SQL as a modeling language

lexical signals in hybrid search, Hybrid search (lexical + vector + reranking)-Hybrid search (lexical + vector + reranking)

lineage (see data lineage)

LLMs

agents generating SQL, SQL-Based Agents and MCPs-When to use MCP versus custom tools

about agents, SQL as an agent tool, SQL-Based Agents and MCPs

building a SQL-generating agent, Building a SQL-generating agent-Building a SQL-generating agent

connecting an agent to MCP, Connecting an agent to MCP

failures handled gracefully, Handling failures gracefully

guardrails added, Adding guardrails-Required filters in tool signatures

MCP server with SQL tools, SQL agents via MCP

OpenAI Agents SDK, SQL-Based Agents and MCPs, Building a SQL-generating agent, Building a SQL-generating agent

postgres-mcp server, Using postgres-mcp

PostgreSQL local instance with Docker, SQL-Based Agents and MCPs

schema context provided to agents, Enhancing accuracy with schema context-Enhancing accuracy with schema context

when to use MCP versus custom tools, When to use MCP versus custom tools

bringing LLMs to the data, Integrating ML and LLMs in SQL Platforms, Machine Learning

embeddings, search, and retrieval, Embeddings, Search, and Retrieval-Multimodal embeddings

designing an embeddings table, Designing an embeddings table-Designing an embeddings table

embeddings as semantic join keys, Embeddings as semantic join keys

hybrid search of lexical, vector, reranking, Hybrid search (lexical + vector + reranking)-Hybrid search (lexical + vector + reranking)

multimodal embeddings, Multimodal embeddings

query-time semantic retrieval, Query-time semantic retrieval with vector search-Query-time semantic retrieval with vector search

extending not replacing analytical entities, Generative AI Inside SQL

GenAI inside SQL, Generative AI Inside SQL-When SQL Should Run GenAI Workloads

(see also generative AI (GenAI) in SQL)

inline LLM patterns best for batch or scheduled jobs, SQL as an agent tool

LLM functions inside SQL, LLM Functions Inside SQL-Structured outputs

generating free-form text, Generating free-form text-Generating free-form text

input shaping before expensive LLM calls, LLM Functions Inside SQL

remote model connecting to Gemini, LLM Functions Inside SQL

structured outputs, Structured outputs-Structured outputs

tips for input and output, Structured outputs

tips for producing structured outputs, Structured outputs

SQL integration, Integrating ML and LLMs in SQL Platforms-Integrating ML and LLMs in SQL Platforms

load testing, Implementing testing strategies

logging

data access and query behavior, Security and access control: Guardrails, not roadblocks

evaluation outcomes, Building and evolving eval suites

generative AI output, Logging, Monitoring, and Evaluating GenAI Outputs

core run logs and daily observability, Core run logs and daily observability-Core run logs and daily observability

running lineage and versioned artifacts, Run lineage and versioned artifacts

logs as sensitive data, Security and Privacy Controls

loose coupling of ecommerce data pipeline, Pipeline Overview and Design Principles

M

machine learning (ML), Machine Learning-Drift

bringing ML to the data, Integrating ML and LLMs in SQL Platforms, Machine Learning

feature engineering patterns, Advanced Feature Engineering Patterns-Principal component analysis

- categorical encoding, Categorical encoding-Feature hashing
- principal component analysis, Principal component analysis-Principal component analysis
- quantile binning, Quantile binning-Equal-width (deciles)
- target encoding with K-fold out-of-fold mean, Target encoding (leakage-aware) with K-fold out-of-fold mean-Target encoding (leakage-aware) with K-fold out-of-fold mean
- text feature engineering, Text feature engineering-Character n-grams

model creation and training, Model Creation and Training-Best practices and cost considerations

- about ML model creation and training, Model Creation and Training
- anomaly detection, Anomaly detection-Anomaly detection
- hyperparameter tuning, Hyperparameter tuning-Best practices and cost considerations
- recommendation systems, Recommendation systems
- supervised learning models, Supervised learning models-Supervised learning models
- time-series forecasting, Time-series forecasting-Time-series forecasting

unsupervised learning models, Unsupervised learning models-Unsupervised learning models

model deployment and monitoring, Model Deployment and Monitoring-Drift

exporting to Google Cloud Storage, Making models available for predictions

exporting to Vertex AI for online inference, Making models available for predictions

monitoring and retraining models, Model monitoring and retraining strategies-Drift

predictions from models, Making models available for predictions-Making models available for predictions

scheduling batch predictions, Scheduling batch predictions

model evaluation and validation, Model Evaluation and Validation

cross-validation, Cross-validation-Interpretation of results

evaluation metrics, Evaluation metrics

providers of SQL-native ML platforms, Machine Learning-Machine Learning

SQL integration, Integrating ML and LLMs in SQL Platforms-Integrating ML and LLMs in SQL Platforms

when to use, Machine Learning

Malloy semantic modeling language, Malloy: A Semantic Modeling Language

Mann–Whitney U test, Mann–Whitney U-Multiple Groups Testing

materialized views (MVs), Query Rewriting, Caching, and Materialized Views-Query Rewriting, Caching, and Materialized Views

example, Query Rewriting, Caching, and Materialized Views

query result caching versus, Query Rewriting, Caching, and Materialized Views

matrix factorization, Recommendation systems

MCP (model context protocol), SQL agents via MCP

connecting an agent to MCP, Connecting an agent to MCP

description of, What is MCP?

postgres-mcp server, Using postgres-mcp

when to use MCP versus custom tools, When to use MCP versus custom tools

mean squares, One-way ANOVA: A handmade analysis

medallion architecture, Storage and processing

bronze (raw) data, Storage and processing

batch ingestion, GCS to Iceberg with Spark batch processing (Bronze layer)-GCS to Iceberg with Spark batch processing (Bronze layer)

dbt bronze tables as sources, Transformation and Data Modeling with SQL

dbt on BigQuery to silver and gold, Transformation and Data Modeling with SQL-Tests and documentation

Iceberg data storage and management, Data Storage and Management with Apache Iceberg-Auditing and recovery: Time travel with SQL

live data in the consume layer, Live Data in Consume Layer-Live Data in Consume Layer

open table formats, Open table formats

storage in data lakehouse, Storage Layer

streaming ingestion, GCS to Iceberg and Spark structured streaming (Bronze layer)

dataset and schema boundaries as governance, Dataset and Schema Boundaries as Governance

ecommerce SQL data pipeline, Pipeline Overview and Design Principles, Transformation and Data Modeling with SQL

gold (curated) data, Storage and processing

consume layer on top of gold, Consume Layer

dbt on BigQuery Gold tier, Transformation and Data Modeling with SQL

dbt on BigQuery Gold tier model design, Silver and Gold Model Design in dbt: Building a Medallion Architecture-Tests and documentation

silver (refined) data, Storage and processing

dbt on BigQuery Silver tier, Transformation and Data Modeling with SQL

dbt on BigQuery Silver tier model design, Silver and Gold Model Design in dbt: Building a Medallion Architecture-Tests and documentation

staged modeling layers, Staged modeling layers

MERGE statements, Idempotency and rerunnability

metric drift, SQL-First Semantic Execution

MetricFlow, The dbt Semantic layer (MetricFlow)-The dbt Semantic layer (MetricFlow)

    dbt-metricflow adapter, The dbt Semantic layer (MetricFlow)

Microsoft Fabric GenAI platform, Generative AI in SQL

ML (see machine learning (ML))

model context protocol (see MCP (model context protocol))

modeling in layered architecture, Transformation and modeling-SQL as a modeling language

    description of modeling, SQL as a modeling language

    physical models, Bridging Semantics to Execution

    semantic models, Semantic Models and Ontologies, Bridging Semantics to Execution

    SQL as a modeling language, SQL as a modeling language

        best practices, SQL as a modeling language

        modeling with SQL (see modeling with SQL)

    staged modeling layers, Staged modeling layers

modeling in machine learning, Model Creation and Training-Drift

    creating and training models, Model Creation and Training-Best practices and cost considerations

        about ML model creation and training, Model Creation and Training

        anomaly detection, Anomaly detection-Anomaly detection

hyperparameter tuning, Hyperparameter tuning-Best practices and cost considerations

recommendation systems, Recommendation systems

supervised learning models, Supervised learning models-Supervised learning models

time-series forecasting, Time-series forecasting-Time-series forecasting

unsupervised learning models, Unsupervised learning models-Unsupervised learning models

deploying and monitoring models, Model Deployment and Monitoring-Drift

exporting to Google Cloud Storage, Making models available for predictions

exporting to Vertex AI for online inference, Making models available for predictions

monitoring and retraining models, Model monitoring and retraining strategies-Drift

predictions from models, Making models available for predictions-Making models available for predictions

scheduling batch predictions, Scheduling batch predictions

evaluating and validating models, Model Evaluation and Validation

cross-validation, Cross-validation-Interpretation of results

evaluation metrics, Evaluation metrics

modeling with SQL

choosing right abstraction for each problem, Summary

knowledge graphs and semantic modeling, Knowledge Graphs and Semantic Modeling: The New Frontier for SQL-Querying with Cypher Versus SQL

RDF and OWL, RDF and OWL: The Foundations of Semantic Modeling

semantic models and ontologies, Semantic Models and Ontologies-PostgreSQL and GraphDB

physical models, Bridging Semantics to Execution

physical models versus semantic models, Physical models versus semantic models in practice-Physical models versus semantic models in practice

semantic models, Semantic Models and Ontologies, Bridging Semantics to Execution

(see also semantic modeling)

SQL as a modeling language, SQL as a modeling language

best practices, SQL as a modeling language

traditional SQL data modeling, Traditional SQL Data Modeling Techniques-Summary of Traditional Models

choosing star or snowflake, Snowflake Schemas: Adding Normalization to Dimensions

lakehouse modeling and ELT workflows, Lakehouse Modeling and ELT Workflows

one big table, One Big Table: The Wide Denormalized Table Approach

snowflake schemas, Snowflake Schemas: Adding Normalization to Dimensions-Snowflake Schemas: Adding Normalization to

Dimensions

snowflake versus star schemas, Snowflake Schemas: Adding Normalization to Dimensions

star schema, Dimensional Modeling: Star Schema for Analytics-Dimensional Modeling: Star Schema for Analytics

summary of traditional models, Summary of Traditional Models-Summary of Traditional Models

third normal form, Third Normal Form—Structured, Relational Modeling-Third Normal Form—Structured, Relational Modeling

modularity

ecommerce data pipeline design principle, Pipeline Overview and Design Principles

LEGO-style pipeline, Modularity and composability

SQL script, Modularity and composability

monitoring

deployed models, Model monitoring and retraining strategies-Drift

drift, Model monitoring and retraining strategies, Drift

drift signal for monitoring, Core run logs and daily observability

skew, Model monitoring and retraining strategies, Drift

observability, Observability: Knowing what's working (and what's not), Missing Lineage and Monitoring

production run logs and daily monitoring, Core run logs and daily observability-Core run logs and daily observability

query performance, Monitoring and profiling query performance-Snowflake: Monitoring and profiling queries

BigQuery, BigQuery: Monitoring and profiling queries

Snowflake, Snowflake: Monitoring and profiling queries-Snowflake: Monitoring and profiling queries

multi-cloud patterns, Hybrid and Multi-Cloud Patterns-Hybrid and Multi-Cloud Patterns

multimodal embeddings, Multimodal embeddings

sensitive data in multimodal inputs, Multimodal embeddings

MV (see materialized views (MVs))

N

natural language into SQL, Text-to-SQL

NDJSON (see newline-delimited JSON (NDJSON))

near real-time and real-time, Batch Versus Streaming: Choosing the Right SQL Pipeline Approach

Neo4j, What Is a Knowledge Graph?

Cypher language, What Is a Knowledge Graph?, Property Graphs and Cypher

hybrid with SQL via PostgreSQL, PostgreSQL and GraphDB

Neo4j and Cypher together, Graph Technologies: Neo4j and Cypher

readability of queries, Querying with Cypher Versus SQL

as a property graph, Property Graphs and Cypher

querying with Cypher versus SQL, Querying with Cypher Versus SQL-Querying with Cypher Versus SQL

SQL versus, Graph Technologies: Neo4j and Cypher

Web Ontology Language transition, Graph Technologies: Neo4j and Cypher

newline-delimited JSON (NDJSON)

API batch ingestion, API to GCS (landing zone)

Pub/Sub messages to GCS, Pub/Sub to GCS (landing zone)

code that writes to NDJSON file, Pub/Sub to GCS (landing zone)

non-parametric statistical tests, Mann–Whitney U

normality testing, Normality Testing

normalized models ideal for OLTP, Third Normal Form—Structured, Relational Modeling

NULL handling clearer in SQL:2023, The Graph Revolution with SQL:2023

O

observability, Observability: Knowing what's working (and what's not), Missing Lineage and Monitoring

production run logs and daily monitoring, Core run logs and daily observability-Core run logs and daily observability

OBT (see one big table (OBT))

OLAP (online analytical processing) functions

DuckDB embedded analytics, Embedded Analytics with DuckDB

handled by SQL, Extended Analytical and OLAP Functions

OLTP databases (see online transaction processing (OLTP) databases)

one big table (OBT), One Big Table: The Wide Denormalized Table Approach

delivery artifacts rather than models, One Big Table: The Wide Denormalized Table Approach

redundancy trade-off, One Big Table: The Wide Denormalized Table Approach

summary of traditional models, Summary of Traditional Models

one-hot encoding, One-hot encoding

one-way ANOVA (analysis of variance), One-way ANOVA: A handmade analysis-One-way ANOVA: A handmade analysis

degrees of freedom, One-way ANOVA: A handmade analysis

mean squares, One-way ANOVA: A handmade analysis

p-value calculation, One-way ANOVA: A handmade analysis

sums of squares, One-way ANOVA: A handmade analysis

online book resources (see resources online)

online inference via Vertex AI, Making models available for predictions

online transaction processing (OLTP) databases

cloud data warehouses versus, Tuning in Cloud Data Warehouses: BigQuery Versus Snowflake

normalized models ideal for OLTP, Third Normal Form—Structured, Relational Modeling

ontologies, Semantic Models and Ontologies

semantic models, Semantic Models and Ontologies-PostgreSQL and GraphDB

open table formats, Open table formats

metadata layer on flat files, Open table formats

OpenAI Agents SDK, SQL-Based Agents and MCPs

building a SQL-generating agent, Building a SQL-generating agent-Building a SQL-generating agent

@function_tool decorator, Building a SQL-generating agent

loading the API key, Building a SQL-generating agent

operators for GoogleSQL pipe syntax, Pipe Operators Overview-Combining results with UNION, INTERSECT, EXCEPT

AGGREGATE, The AGGREGATE operator

applied by examples

aggregation and post-filtering, Aggregation and post-filtering

filtering and selecting, Filtering and selecting

intermediate calculation, Using an intermediate calculation

joining after a transformation, Joining after a transformation

multistep aggregation without subqueries, Multistep aggregation (eliminating subqueries)

AS, The AS operator

best practices, Best Practices and Style Guidelines for Pipe SQL Syntax

chart overviewing operators, Pipe Operators Overview

documentation and research article online, When not to use pipe syntax

DROP, The DROP and RENAME operators

EXTEND, The EXTEND operator-The EXTEND operator

JOIN, The JOIN operator-The JOIN operator

LIMIT, The LIMIT operator

real-world use cases, Real-World Use Cases and Applications of Pipe SQL-Real-World Use Cases and Applications of Pipe SQL

RENAME, The DROP and RENAME operators

SELECT, The SELECT operator

SET, The SET operator

style guidelines, Formatting and readability tips-When not to use pipe syntax

efficiency considerations, Efficiency considerations

formatting and readability, Formatting and readability tips-Formatting and readability tips

UNION, INTERSECT, EXCEPT, Combining results with UNION, INTERSECT, EXCEPT

when not to use pipe syntax, When not to use pipe syntax

WHERE, The WHERE operator-The WHERE operator

orchestration and scheduling in layered architecture, Orchestration and scheduling-Orchestration and scheduling

about orchestration, Orchestration and Workflow Management

alternative orchestration options, Alternative Orchestration Options

ecommerce SQL data pipeline, Orchestration and Workflow Management-Alternative Orchestration Options

Google Cloud Workflows, Orchestration and Workflow Management

Google Cloud Workflows orchestration pipeline, Cloud Workflows Orchestration Pipeline-Cloud Workflows

Orchestration Pipeline

orchestration and scheduling configurations online, Alternative Orchestration Options

scheduling the workflows, Scheduling the Workflows

workflow YAML files defined, Orchestration and Workflow Management

event-driven schedules, Orchestration and scheduling

pipelines represented as DAGs, Orchestration and scheduling, Orchestration and Workflow Management

time-based schedules, Orchestration and scheduling

tools for data pipeline processes, Automation and orchestration

outliers

detecting, Outlier Detection

equal-frequency binning resilience to, Equal-frequency (deciles)

equal-width binning sensitivity to, Equal-width (deciles), Equal-width (deciles)

Kruskal—Wallis test robustness to, Kruskal—Wallis test

Mann–Whitney U test robustness to, Mann–Whitney U

Spearman correlation less sensitivity to, Correlation and Association

Owen, Louis, Best practices and cost considerations

OWL (Web Ontology Language)

semantic modeling, RDF and OWL: The Foundations of Semantic Modeling

transitioning from graph models, Graph Technologies: Neo4j and Cypher

ownership

data mesh, Data Mesh and Data Products, Data Mesh and Data Products

dbt exposure feature, Exposures and ownership

P

p-values, Mann–Whitney U

calculating, One-way ANOVA: A handmade analysis

parameterized queries unused as anti-pattern, Not using prepared statements (when executing many similar queries)

partitioning

BigQuery, Indexing and Efficient Data Access, Indexing and Efficient Data Access

indexing alternative, Indexing and Efficient Data Access

Snowflake, Indexing and Efficient Data Access, Indexing and Efficient Data Access

paths in SQL:2023 graph processing, The Graph Revolution with SQL:2023, Navigating Graph Structures in SQL

PCA (see principal component analysis (PCA))

Pearson correlation, Correlation and Association

performance optimization

anti-patterns to avoid, Avoiding Common SQL Anti-Patterns-Neglecting to analyze/update statistics

caching query results, Query Rewriting, Caching, and Materialized Views

materialized views versus, Query Rewriting, Caching, and Materialized Views

query consistency, Query Rewriting, Caching, and Materialized Views

cloud data warehouses, Tuning in Cloud Data Warehouses: BigQuery Versus Snowflake-Snowflake: Monitoring and profiling queries

about differences from OLTP, Tuning in Cloud Data Warehouses: BigQuery Versus Snowflake

concurrency and workload isolation, Concurrency and workload isolation-Snowflake concurrency and isolation best practices

cost-optimization strategies, Cost-optimization strategies-Snowflake cost optimization best practices

monitoring and profiling queries, Monitoring and profiling query performance-Snowflake: Monitoring and profiling queries

indexes for efficient data access, Indexing and Efficient Data Access-Indexing and Efficient Data Access

materialized views, Query Rewriting, Caching, and Materialized Views-Query Rewriting, Caching, and Materialized Views

caching query results versus, Query Rewriting, Caching, and Materialized Views

modularity affecting performance, Modularity and composability

monitoring and profiling queries, Monitoring and profiling query performance-Snowflake: Monitoring and profiling queries

BigQuery, BigQuery: Monitoring and profiling queries

Snowflake, Snowflake: Monitoring and profiling queries-Snowflake: Monitoring and profiling queries

query performance via Cypher in Neo4j, What Is a Knowledge Graph?

query planning and cost-based optimization, Query Planning and Cost-Based Optimization-Query Planning and Cost-Based Optimization

query rewriting, Query Rewriting, Caching, and Materialized Views

performance testing, Implementing testing strategies

Personally Identifiable Information (PII)

correctable and removable, One Big Table: The Wide Denormalized Table Approach

data contract specifications, Formalizing Data Agreements: Data Contracts

multimodal inputs, Multimodal embeddings

security and access control, Security and access control: Guardrails, not roadblocks

physical models, Bridging Semantics to Execution

pipe syntax (GoogleSQL), GoogleSQL Pipe Syntax-When not to use pipe syntax

applied by examples

aggregation and post-filtering, Aggregation and post-filtering

filtering and selecting, Filtering and selecting

intermediate calculation, Using an intermediate calculation

joining after a transformation, Joining after a transformation

multistep aggregation without subqueries, Multistep aggregation (eliminating subqueries)

best practices, Best Practices and Style Guidelines for Pipe SQL Syntax

building queries, GoogleSQL Pipe Syntax

description of pipe syntax, GoogleSQL Pipe Syntax

documentation and research article online, When not to use pipe syntax

operators, Pipe Operators Overview-Combining results with UNION, INTERSECT, EXCEPT

AGGREGATE, The AGGREGATE operator

AS, The AS operator

chart overviewing operators, Pipe Operators Overview

DROP, The DROP and RENAME operators

EXTEND, The EXTEND operator-The EXTEND operator

JOIN, The JOIN operator-The JOIN operator

LIMIT, The LIMIT operator

RENAME, The DROP and RENAME operators

SELECT, The SELECT operator

SET, The SET operator

UNION, INTERSECT, EXCEPT, Combining results with UNION, INTERSECT, EXCEPT

WHERE, The WHERE operator-The WHERE operator

real-world use cases, Real-World Use Cases and Applications of Pipe SQL-Real-World Use Cases and Applications of Pipe SQL

style guidelines, Formatting and readability tips-When not to use pipe syntax

efficiency considerations, Efficiency considerations

formatting and readability, Formatting and readability tips-Formatting and readability tips

syntax and semantics of pipe queries, Core Syntax and Semantics of Pipe Queries

when not to use, When not to use pipe syntax

Pipelined Relational Query Language (PRQL), PRQL: A Pipelined Query Language-PRQL: A Pipelined Query Language

pipelines (see data pipelines)

postgres-mcp server, Using postgres-mcp

PostgresML (PGML), Integrating ML and LLMs in SQL Platforms

integration with LLMs and AI APIs, Integrating ML and LLMs in SQL Platforms

training and using models in Postgres, Integrating ML and LLMs in SQL Platforms

vector embeddings storage, Integrating ML and LLMs in SQL Platforms

PostgreSQL

agents generating SQL, SQL-Based Agents and MCPs

building a SQL-generating agent, Building a SQL-generating agent-Building a SQL-generating agent

postgres-mcp server, Using postgres-mcp

schema context provided to agents, Enhancing accuracy with schema context-Enhancing accuracy with schema context

AI functionality, Integrating ML and LLMs in SQL Platforms

cost-based optimizers documentation online, Query Planning and Cost-Based Optimization

hybrid of Cypher with SQL, PostgreSQL and GraphDB

Apache AGE, PostgreSQL and GraphDB

indexes, Indexing and Efficient Data Access

performance boosting, Indexing and Efficient Data Access

table partitioning, Indexing and Efficient Data Access

materialized views, Query Rewriting, Caching, and Materialized Views, Query Rewriting, Caching, and Materialized Views

window functions, Advanced Window Functions for Analytic Queries

predictions from models, Making models available for predictions-Making models available for predictions

scheduling batch predictions, Scheduling batch predictions

Vertex AI for online inference, Making models available for predictions

Prefect, Alternative Orchestration Options

prepared statements unused as anti-pattern, Not using prepared statements (when executing many similar queries)

principal component analysis (PCA), Principal component analysis-Principal component analysis

about principal component analysis, Principal component analysis

BigQuery ML for model training, Principal component analysis

privacy

data contract specifications, Formalizing Data Agreements: Data Contracts

lineage tracking for anonymizing data, Data lineage and dependency management

multimodal inputs, Multimodal embeddings

prompt templates as sensitive context, Run lineage and versioned artifacts, Building and evolving eval suites

security and access control, Security and access control: Guardrails, not roadblocks

sensitive data stored locally, Hybrid and Multi-Cloud Patterns

processing in layered architecture, Storage and processing

Producers for SQL data pipeline online, Create a BigQuery dataset and BigLake connection

product catalog and knowledge base (see customer-support knowledge base built)

production

generative AI productionized, Productionizing Generative AI in the Data Warehouse-Production Readiness Checklist

artifact flow, Productionizing Generative AI in the Data Warehouse

checklist for production readiness, Production Readiness Checklist

deployment architectures, Practical Deployment Architectures

describing productionizing, Productionizing Generative AI in the Data Warehouse

five interconnected components, Productionizing Generative AI in the Data Warehouse

outputs as data products, Logging, Monitoring, and Evaluating GenAI Outputs-Core run logs and daily observability

retrieval configuration, Run lineage and versioned artifacts

security and privacy controls, Security and Privacy Controls

versioned artifacts with explicit ownership, Run lineage and versioned artifacts, Practical Deployment Architectures

scaling from design to production, From Design to Production: Scaling SQL Data Solutions-Avoiding Common Architectural Anti-Patterns

continuous integration and continuous delivery, Continuous integration and continuous delivery for SQL pipelines-Continuous integration and continuous delivery for SQL pipelines

Infrastructure as Code, Infrastructure as Code (IaC) in scaling and for data platforms-Avoiding Common Architectural Anti-Patterns

operationalizing SQL pipelines, From Design to Production: Scaling SQL Data Solutions

supporting layers, Supporting layers

testing strategies, Implementing testing strategies

prompt injection, Security and Privacy Controls

prompt version registry table, Run lineage and versioned artifacts

prompt templates as sensitive context, Run lineage and versioned artifacts

human reviews, Building and evolving eval suites

Property Graph Queries, Navigating Graph Structures in SQL

PostgreSQL hybrid of Cypher and SQL, PostgreSQL and GraphDB

property graphs, Property Graphs and Cypher

Cypher query language, Property Graphs and Cypher

entity nodes and relationship edges, Property Graphs and Cypher

Neo4j as a leading property graph, Property Graphs and Cypher

(see also Neo4j)

PRQL (PipeSQLPipelined Relational Query Language), PRQL: A Pipelined Query Language-PRQL: A Pipelined Query Language

Pub/Sub in SQL data pipeline

creating Pub/Sub topic and subscription, Create Pub/Sub topic and subscription

Google Cloud Storage landing zone, Pub/Sub to GCS (landing zone)

Q

quantile binning, Quantile binning-Equal-width (deciles)

about quantile binning, Quantile binning

equal-frequency binning, Equal-frequency (deciles)

equal-width binning, Equal-width (deciles)

quantum computing-assisted optimization, The Quantum Horizon: Quantum-Assisted Optimization

queries by agents read-only, Read-only verification

query planning and cost-based optimization, Query Planning and Cost-Based Optimization-Query Planning and Cost-Based Optimization

query-time semantic retrieval, Query-time semantic retrieval with vector search-Query-time semantic retrieval with vector search

R

RAG (retrieval-augmented generation), Retrieval-augmented generation (RAG), Grounded Generation (the “G” in RAG)-Grounded Generation (the “G” in RAG)

about RAG, Patterns: RAG, Agents, and MCPs

Random Forest classifier training, Supervised learning models

applying the model, Supervised learning models

RANK() window function, Advanced Window Functions for Analytic Queries

RBAR (row-by-agonizing-row) processing, RBAR (row-by-agonizing-row) processing

how to avoid, RBAR (row-by-agonizing-row) processing

RDF (Resource Description Framework), RDF and OWL: The Foundations of Semantic Modeling

SPARQL query language, RDF and OWL: The Foundations of Semantic Modeling

read-only queries by agents, Read-only verification

real-time and near real-time, Batch Versus Streaming: Choosing the Right SQL Pipeline Approach

live data in the consume layer, Live Data in Consume Layer-Live Data in Consume Layer

recommendation systems, Recommendation systems

recursive CTEs, Recursive CTEs for Hierarchical and Iterative Queries-Recursive CTEs for Hierarchical and Iterative Queries

termination, Recursive CTEs for Hierarchical and Iterative Queries

Redshift GenAI platform, Generative AI in SQL

regression from aggregates, Regression from Aggregates

regression testing, Implementing testing strategies

relational model of Dr. Codd, Historical Overview: From Relational to Modern SQL

resources online

BigQuery recursive CTEs documentation, Recursive CTEs for Hierarchical and Iterative Queries

book supplemental material, Using Code Examples

book web page, How to Contact Us

customer-support knowledge base code, Generative AI in SQL

documentation

BigQuery connecting to Gemini models, LLM Functions Inside SQL

BigQuery recursive CTEs, Recursive CTEs for Hierarchical and Iterative Queries

GoogleSQL pipe syntax, When not to use pipe syntax

PostgreSQL cost-based optimizers, Query Planning and Cost-Based Optimization

Vertex AI permissions, LLM Functions Inside SQL

Gemini models and their capabilities, LLM Functions Inside SQL

Google BigLake Metastore page, Set up BigLake (Iceberg) metastore

GoogleSQL pipe syntax documentation and research article, When not to use pipe syntax

PostgreSQL cost-based optimizers documentation, Query Planning and Cost-Based Optimization

recursive CTEs

BigQuery documentation, Recursive CTEs for Hierarchical and Iterative Queries

DOOMQL, Recursive CTEs for Hierarchical and Iterative Queries

Sudoku solver, Recursive CTEs for Hierarchical and Iterative Queries

SQL data pipeline code, Building Data Engineering Solutions with SQL

dbt project, Tests and documentation

helper Cloud Run Job, GCS to Iceberg and Spark structured streaming (Bronze layer)

orchestration and scheduling configurations, Alternative Orchestration Options

Producers, Create a BigQuery dataset and BigLake connection

weather dataset setup script, Dataset Statistical Description

retail SQL data pipeline built
- about the project, Building Data Engineering Solutions with SQL
- architecture of data pipeline, Pipeline Overview and Design Principles
  - design principles, Pipeline Overview and Design Principles
- code online, Building Data Engineering Solutions with SQL
  - dbt project, Tests and documentation
  - helper Cloud Run Job, GCS to Iceberg and Spark structured streaming (Bronze layer)
  - orchestration and scheduling configurations, Alternative Orchestration Options
  - Producers, Create a BigQuery dataset and BigLake connection
- consume layer, Consume Layer-Live Data in Consume Layer
  - bridging semantics to execution, Bridging Semantics to Execution-Physical models versus semantic models in practice
  - live data in the consume layer, Live Data in Consume Layer-Live Data in Consume Layer
  - operationalizing the semantic layer, Operationalizing the Semantic layer
  - semantic contract tests, Semantic Contracts are Testable Contracts
  - SQL-first semantic execution, SQL-First Semantic Execution-The dbt Semantic layer (MetricFlow)
  - stable interface for data consumption, A Stable Interface for Data Consumption
- data ingestion, Pipeline Overview and Design Principles, Data Ingestion in a SQL-Centric Architecture-GCS to Iceberg with Spark

batch processing (Bronze layer)

about approach to data ingestion, Data Ingestion in a SQL-Centric Architecture

batch ingestion, Batch Ingestion for Bulk Data-GCS to Iceberg with Spark batch processing (Bronze layer)

creating Pub/Sub topic and subscription, Create Pub/Sub topic and subscription

streaming ingestion, Streaming Ingestion with Spark Structured Streaming and a Flavor of Apache Flink SQL-Apache Flink SQL

Google Cloud setup, Setting Up the GCP Environment-Create a BigQuery dataset and BigLake connection

BigLake (Iceberg) metastore setup, Set up BigLake (Iceberg) metastore

BigQuery dataset and BigLake connection, Create a BigQuery dataset and BigLake connection

creating a cloud storage bucket, Create a cloud storage bucket

creating a GCP project, Create a GCP project

creating Pub/Sub topic and subscription, Create Pub/Sub topic and subscription

credentials for authentication, Create a BigQuery dataset and BigLake connection

enabling required APIs, Enable required APIs

free tier setup, Setting Up the GCP Environment

Google Cloud Shell or gcloud CLI, Setting Up the GCP Environment

Lambda architecture, Building Data Engineering Solutions with SQL

data ingestion, Streaming Ingestion with Spark Structured Streaming and a Flavor of Apache Flink SQL

design principle, Pipeline Overview and Design Principles

why Lambda architecture, Pipeline Overview and Design Principles

orchestration and workflow management, Orchestration and Workflow Management-Alternative Orchestration Options

alternative orchestration options, Alternative Orchestration Options

Google Cloud Workflows, Orchestration and Workflow Management

Google Cloud Workflows orchestration pipeline, Cloud Workflows Orchestration Pipeline-Cloud Workflows Orchestration Pipeline

scheduling the workflows, Scheduling the Workflows

workflow YAML files defined, Orchestration and Workflow Management

security and governance, Security and Governance Considerations-Security Versus Semantics: Consistent Definitions as Control

dataset and schema boundaries, Dataset and Schema Boundaries as Governance

identity and execution boundaries, Identity and Execution Boundaries

semantic consistency, Security Versus Semantics: Consistent Definitions as Control

storage in data lakehouse, Pipeline Overview and Design Principles, Storage Layer-Auditing and recovery: Time travel with SQL

advantages of lakehouses, The Data Lakehouse

decoupling storage and compute, The Data Lakehouse

Iceberg data storage and management, Data Storage and Management with Apache Iceberg-Auditing and recovery: Time travel with SQL

Iceberg time travel to query historical data, Auditing and recovery: Time travel with SQL-Auditing and recovery: Time travel with SQL

open table formats, Open table formats

transformation and data modeling, Transformation and Data Modeling with SQL-Tests and documentation

dbt on BigQuery, Transformation and Data Modeling with SQL-Tests and documentation

dbt Silver and Gold model design, Silver and Gold Model Design in dbt: Building a Medallion Architecture-Tests and documentation

tests and documentation, Tests and documentation-Tests and documentation

retrieval configuration, Run lineage and versioned artifacts

retrieval poisoning, Security and Privacy Controls

retrieval-augmented generation (RAG), Retrieval-augmented generation (RAG), Grounded Generation (the “G” in RAG)-Grounded Generation (the “G” in RAG)

about RAG, Patterns: RAG, Agents, and MCPs

row-by-agonizing-row (RBAR) processing, RBAR (row-by-agonizing-row) processing

how to avoid, RBAR (row-by-agonizing-row) processing

S

sandbox environment created via IaC, Infrastructure as Code (IaC) in scaling and for data platforms

scalability

architectures for scalability, Architectures for Scalability-Hybrid and Multi-Cloud Patterns

cloud native scaling, Cloud Native Scaling

comparison chart of architectures, Hybrid and Multi-Cloud Patterns

concurrency and isolation in practice, Concurrency and Isolation in Practice

hybrid and multi-cloud patterns, Hybrid and Multi-Cloud Patterns-Hybrid and Multi-Cloud Patterns

traditional RDBMS scaling, Traditional RDBMS Scaling

automating data pipelines for, Automation and orchestration

building and operationalizing scalability, Building and Operationalizing Scalable SQL Data Platforms-Avoiding Common Architectural Anti-Patterns

continuous integration and continuous delivery, Continuous integration and continuous delivery for SQL pipelines-Continuous integration and continuous delivery for SQL pipelines

Infrastructure as Code, Infrastructure as Code (IaC) in scaling and for data platforms-Avoiding Common Architectural Anti-Patterns

layered architecture, Layered Architecture: Separating Storage, Transformation, and Orchestration-Connecting back to the architecture

scaling from design to production, From Design to Production: Scaling SQL Data Solutions-Avoiding Common Architectural Anti-Patterns

data lakehouse scalability of data lake, Storage Layer

horizontal scaling, Traditional RDBMS Scaling

vertical scaling, Traditional RDBMS Scaling

scheduling and orchestration in layered architecture, Orchestration and scheduling-Orchestration and scheduling

event-driven schedules, Orchestration and scheduling

pipelines represented as DAGs, Orchestration and scheduling

scheduling batch predictions, Scheduling batch predictions

time-based schedules, Orchestration and scheduling

schemas

agents provided with schema context, Enhancing accuracy with schema context-Enhancing accuracy with schema context

choosing star or snowflake, Snowflake Schemas: Adding Normalization to Dimensions

customer-support knowledge base, Generative AI in SQL

data lakehouse adapting to changes, The Data Lakehouse

ecommerce SQL data pipeline, Building Data Engineering Solutions with SQL

Spark Structured Streaming schema for JSON, GCS to Iceberg and Spark structured streaming (Bronze layer)

Iceberg schema evolution, Iceberg schema evolution-Iceberg schema evolution

ontologies as schemas for graphs, Semantic Models and Ontologies

schema designed in 3NF, Third Normal Form—Structured, Relational Modeling

snowflake schemas, Snowflake Schemas: Adding Normalization to Dimensions-Snowflake Schemas: Adding Normalization to Dimensions

snowflake versus star, Snowflake Schemas: Adding Normalization to Dimensions

star schema, Dimensional Modeling: Star Schema for Analytics-Dimensional Modeling: Star Schema for Analytics

summary of traditional models, Summary of Traditional Models

security, Security and access control: Guardrails, not roadblocks

data contract specifications, Formalizing Data Agreements: Data Contracts

ecommerce SQL data pipeline, Security and Governance Considerations-Security Versus Semantics: Consistent Definitions as Control

dataset and schema boundaries, Dataset and Schema Boundaries as Governance

identity and execution boundaries, Identity and Execution Boundaries

ignoring governance and security as anti-pattern, Ignoring Governance and Security

multimodal inputs, Multimodal embeddings

prompt templates as sensitive context, Run lineage and versioned artifacts, Building and evolving eval suites

semantic consistency, Security Versus Semantics: Consistent Definitions as Control

testing security, Implementing testing strategies

semantic consistency, Security Versus Semantics: Consistent Definitions as Control

semantic layer, SQL for Data Transformations

BI tools, SQL for Data Transformations, Consumption layer

dbt Semantic layer, The dbt Semantic layer (MetricFlow)-The dbt Semantic layer (MetricFlow)

design of ecommerce SQL data pipeline, Pipeline Overview and Design Principles

downstream consumption, Tight Coupling Between Pipeline Components

enforcing semantic alignment, Best practices for the consumption layer

example of semantic layer in action, Example: Semantic joins and inferred logic

operationalizing, Operationalizing the Semantic layer

semantic modeling

about semantic models, Semantic Models and Ontologies, Bridging Semantics to Execution

consume layer bridging semantics to execution, Bridging Semantics to Execution-Physical models versus semantic models in practice

- physical models, Bridging Semantics to Execution
- physical versus semantic models in practice, Physical models versus semantic models in practice-Physical models versus semantic models in practice
- semantic contract tests, Semantic Contracts are Testable Contracts
- semantic joins and inferred logic example, Example: Semantic joins and inferred logic
- semantic models, Bridging Semantics to Execution
- semantic models, governance rules on top of tables, Bridging Semantics to Execution
- SQL-first semantic execution, SQL-First Semantic Execution-The dbt Semantic layer (MetricFlow)

knowledge graphs, Knowledge Graphs and Semantic Modeling: The New Frontier for SQL-Querying with Cypher Versus SQL

Malloy semantic modeling language, Malloy: A Semantic Modeling Language

ontologies, Semantic Models and Ontologies-PostgreSQL and GraphDB

- property graphs and Cypher, Property Graphs and Cypher
- RDF and OWL, RDF and OWL: The Foundations of Semantic Modeling

semantic search with embeddings, Semantic search with embeddings, Embeddings, Search, and Retrieval-Multimodal embeddings

designing an embeddings table, Designing an embeddings table-Designing an embeddings table

embeddings as semantic join keys, Embeddings as semantic join keys

hybrid search of lexical, vector, reranking, Hybrid search (lexical + vector + reranking)-Hybrid search (lexical + vector + reranking)

multimodal embeddings, Multimodal embeddings

query-time semantic retrieval, Query-time semantic retrieval with vector search-Query-time semantic retrieval with vector search

semi-structured data, Semi-Structured Data Handling in SQL

SEQUEL (Structured English Query Language), Historical Overview: From Relational to Modern SQL

service-level agreements (SLAs) and data freshness, Service-level agreements and data freshness

single-table models (see one big table (OBT))

skew in deployed models, Model monitoring and retraining strategies, Drift

skewness, Dataset Statistical Description

SLAs (service-level agreements) and data freshness, Service-level agreements and data freshness

sliding windows, Batch Versus Streaming: Choosing the Right SQL Pipeline Approach

slots managing compute resources (BigQuery), Concurrency and workload isolation

INFORMATION_SCHEMA.JOBS view, BigQuery: Monitoring and profiling queries

workload isolation in practice, Concurrency and Isolation in Practice

Snowflake

compute credits cost structure, Snowflake: Architecting for cost-aware computation

Auto-Suspend shutting down inactive warehouses, Snowflake: Architecting for cost-aware computation

cost-based optimizers, Query Planning and Cost-Based Optimization

Query Profile tool, Query Planning and Cost-Based Optimization

data sharing feature, Beyond ETL/ELT and Streaming: Zero-ETL and Data Sharing

dedicated virtual warehouses for compute, Concurrency and workload isolation

dedicated virual warehouses for compute

workload isolation in practice, Concurrency and Isolation in Practice

ELT pattern, Transformation patterns

GenAI platform, Generative AI in SQL

ML and SQL integration, Integrating ML and LLMs in SQL Platforms

indexing versus partitioning and clustering, Indexing and Efficient Data Access, Indexing and Efficient Data Access

Infrastructure as Code example, Infrastructure as Code (IaC) in scaling and for data platforms

development versus production example, Infrastructure as Code (IaC) in scaling and for data platforms

materialized views, Query Rewriting, Caching, and Materialized Views

performance tuning, Tuning in Cloud Data Warehouses: BigQuery Versus Snowflake-Snowflake: Monitoring and profiling queries

about differences from OLTP, Tuning in Cloud Data Warehouses: BigQuery Versus Snowflake

concurrency and workload isolation, Concurrency and workload isolation-Snowflake concurrency and isolation best practices

concurrency and workload isolation best practices, Snowflake concurrency and isolation best practices

concurrency and workload isolation versus BigQuery, Snowflake concurrency and isolation best practices

cost-optimization, Snowflake: Architecting for cost-aware computation

cost-optimization best practices, Snowflake cost optimization best practices

monitoring and profiling queries, Snowflake: Monitoring and profiling queries-Snowflake: Monitoring and profiling queries

regular review cycles, Snowflake: Monitoring and profiling queries

result cache, Query Rewriting, Caching, and Materialized Views

query consistency flexibility, Query Rewriting, Caching, and Materialized Views

Snowpark, Integrating ML and LLMs in SQL Platforms

Snowsight UI for Query Profile view, Snowflake: Monitoring and profiling queries

vector embeddings data type via Snowpark ML, Integrating ML and LLMs in SQL Platforms

window functions, Advanced Window Functions for Analytic Queries

snowflake schemas, Snowflake Schemas: Adding Normalization to Dimensions-Snowflake Schemas: Adding Normalization to Dimensions

choosing star or snowflake, Snowflake Schemas: Adding Normalization to Dimensions

star schema versus, Snowflake Schemas: Adding Normalization to Dimensions

summary of traditional models, Summary of Traditional Models

Spark DataFrameWriter API, GCS to Iceberg with Spark batch processing (Bronze layer)-GCS to Iceberg with Spark batch processing (Bronze layer)

Spark Structured Streaming, GCS to Iceberg and Spark structured streaming (Bronze layer)-GCS to Iceberg and Spark structured streaming (Bronze layer)

costs of continuously running Spark stream, GCS to Iceberg and Spark structured streaming (Bronze layer)

why Spark Structured Streaming, Why Spark in a SQL-centric architecture?

SPARQL query language of RDF data, RDF and OWL: The Foundations of Semantic Modeling

Spearman correlation, Correlation and Association

SQL

about, Evolution of SQL

agents generating SQL, SQL-Based Agents and MCPs-When to use MCP versus custom tools

about agents, SQL as an agent tool, SQL-Based Agents and MCPs

all scales from big data to local analysis, Embedded Analytics with DuckDB

anti-patterns to avoid, Avoiding Common SQL Anti-Patterns-Neglecting to analyze/update statistics

architectural anti-patterns to avoid, Avoiding Common Architectural Anti-Patterns-Ignoring Governance and Security

backbone of modern data platforms, Summary

current SQL:2023, Evolution of SQL, The Graph Revolution with SQL:2023, Navigating Graph Structures in SQL

dialect comparison, Comparison of SQL Dialects over Time-Comparison of SQL Dialects over Time

advanced features, Comparison of SQL Dialects over Time

auto-increment keys, Comparison of SQL Dialects over Time

convergence in modern SQL, Comparison of SQL Dialects over Time

date/time functions, Comparison of SQL Dialects over Time

GenAI platforms, Generative AI in SQL

GoogleSQL pipe syntax, GoogleSQL Pipe Syntax-When not to use pipe syntax

identifier case sensitivity, Comparison of SQL Dialects over Time

newer cloud platforms, Comparison of SQL Dialects over Time

string functions, Comparison of SQL Dialects over Time

vendor-specific extensions, Comparison of SQL Dialects over Time

window functions, Advanced Window Functions for Analytic Queries

evolution of SQL

core philosophy maintained, Evolution of SQL

historical overview, Historical Overview: From Relational to Modern SQL-The Graph Revolution with SQL:2023

future of, The Future of SQL-Declarative Governance and Policies

as a modeling language, SQL as a modeling language

best practices, SQL as a modeling language

modern SQL, SQL in Data Science

current SQL:2023, Evolution of SQL, The Graph Revolution with SQL:2023, Navigating Graph Structures in SQL

limitations, SQL in Data Science

strengths, SQL in Data Science

multipurpose language, Transition to Multipurpose SQL-Embedded Analytics with DuckDB

analytical and OLAP functions, Extended Analytical and OLAP Functions

data transformations, SQL for Data Transformations

embedded analytics with DuckDB, Embedded Analytics with DuckDB

graph queries, Navigating Graph Structures in SQL

ML and LLM integration, Integrating ML and LLMs in SQL Platforms-Integrating ML and LLMs in SQL Platforms

semi-structured data handling, Semi-Structured Data Handling in SQL

temporal data management, Temporal Data Management

unstructured data handling, Dealing with Unstructured Data in SQL

Turing-complete language, Recursive CTEs for Hierarchical and Iterative Queries

examples, Recursive CTEs for Hierarchical and Iterative Queries

SQL injection protection from prepared statements, Not using prepared statements (when executing many similar queries)

SQL-86, Standard SQL-86: Laying the Foundations

SQL-89, Standard SQL-89: Adding Integrity and Security

SQL-92, Standard SQL-92: Adding Modeling Capabilities

SQL-native GenAI definition, Generative AI in SQL

SQL3 (SQL:1999), Standard SQL:1999—Handling Industry Needs

SQL:1999 (SQL3), Standard SQL:1999—Handling Industry Needs

CTEs introduced, Common Table Expressions and Subquery Factoring

SQL:2003, From the SQL:2003 Standard to SQL:2008

window functions standardized, Extended Analytical and OLAP Functions

SQL:2006, From the SQL:2003 Standard to SQL:2008

SQL:2008, From the SQL:2003 Standard to SQL:2008

SQL:2011, From the SQL:2011 Standard to SQL:2016

system-versioned temporal tables, Temporal Data Management

SQL:2016, From the SQL:2011 Standard to SQL:2016

SQL:2023

current SQL, Evolution of SQL

graph processing, The Graph Revolution with SQL:2023, Navigating Graph Structures in SQL

star schema, Dimensional Modeling: Star Schema for Analytics-Dimensional Modeling: Star Schema for Analytics

choosing star or snowflake, Snowflake Schemas: Adding Normalization to Dimensions

cornerstone of data warehousing, Dimensional Modeling: Star Schema for Analytics, Snowflake Schemas: Adding Normalization to Dimensions

snowflake schemas versus, Snowflake Schemas: Adding Normalization to Dimensions

summary of traditional models, Summary of Traditional Models

statistical analysis, Advanced Statistical Analysis-Kruskal—Wallis test

about SQL capabilities, Advanced Statistical Analysis

confidence intervals, Confidence Intervals: Estimating the Mean-Confidence Intervals: Estimating the Mean

data processing and analysis, Data Processing and Analysis-Outlier Detection

correlation and association, Correlation and Association

dataset statistical description, Dataset Statistical Description-Dataset Statistical Description

outlier detection, Outlier Detection

regression from aggregates, Regression from Aggregates

multiple groups testing

Kruskal—Wallis test, Kruskal—Wallis test

one-way analysis of variance, One-way ANOVA: A handmade analysis-One-way ANOVA: A handmade analysis

non-parametric statistical tests, Mann–Whitney U

normality testing, Normality Testing

p-values, Mann–Whitney U

calculating, One-way ANOVA: A handmade analysis

two-sample tests

Mann–Whitney U test, Mann–Whitney U-Multiple Groups Testing

two-sample t-test, Two-sample t-test-Two-sample t-test

storage and processing in layered architecture, Storage and processing

data lakehouse storage, Pipeline Overview and Design Principles, Storage Layer-Auditing and recovery: Time travel with SQL

advantages of lakehouses, The Data Lakehouse

decoupling storage and compute, The Data Lakehouse

Iceberg schema evolution, Data Storage and Management with Apache Iceberg

open table formats, Open table formats

self-managing with Iceberg orchestration, Data Storage and Management with Apache Iceberg

medallion architecture, Storage and processing

streaming ingestion, Data ingestion

data file compaction, Data Storage and Management with Apache Iceberg

ecommerce SQL data pipeline, Streaming Ingestion with Spark Structured Streaming and a Flavor of Apache Flink SQL-Apache Flink SQL

Apache Flink SQL instead of Spark, Apache Flink SQL

GCS to Iceberg and Spark (Bronze), GCS to Iceberg and Spark structured streaming (Bronze layer)-GCS to Iceberg and Spark structured streaming (Bronze layer)

implementation approach, Data Ingestion in a SQL-Centric Architecture

Lambda style ingestion pattern, Streaming Ingestion with Spark Structured Streaming and a Flavor of Apache Flink SQL

Pub/Sub to Google Cloud Storage, Pub/Sub to GCS (landing zone)

why Spark Structured Streaming, Why Spark in a SQL-centric architecture?

streaming transformations, Transformation Patterns: ETL, ELT, and Streaming

streaming versus batch data pipelines, Batch Versus Streaming: Choosing the Right SQL Pipeline Approach-Batch Versus Streaming: Choosing the Right SQL Pipeline Approach

hybrid approach, Batch Versus Streaming: Choosing the Right SQL Pipeline Approach

Kappa architecture, Batch Versus Streaming: Choosing the Right SQL Pipeline Approach

Lambda architecture, Batch Versus Streaming: Choosing the Right SQL Pipeline Approach

real-time and near real-time, Batch Versus Streaming: Choosing the Right SQL Pipeline Approach

sliding windows, Batch Versus Streaming: Choosing the Right SQL Pipeline Approach

string functions in SQL dialects, Comparison of SQL Dialects over Time

Structured English Query Language (SEQUEL), Historical Overview: From Relational to Modern SQL

Sudoku solver in SQL online, Recursive CTEs for Hierarchical and Iterative Queries

sums of squares, One-way ANOVA: A handmade analysis

supervised learning models, Supervised learning models-Supervised learning models

training a Random Forest classifier, Supervised learning models

applying the model, Supervised learning models

system-versioned temporal tables, Temporal Data Management

T

table partitioning in PostgreSQL, Indexing and Efficient Data Access

target encoding with K-fold out-of-fold mean, Target encoding (leakage-aware) with K-fold out-of-fold mean-Target encoding (leakage-aware) with

K-fold out-of-fold mean

about K-fold out-of-fold mean, Target encoding (leakage-aware) with K-fold out-of-fold mean

about target encoding, Target encoding (leakage-aware) with K-fold out-of-fold mean

temperature values in text generation, Structured outputs

temporal data management

system-versioned temporal tables, Temporal Data Management

time travel to query historical data via Iceberg, Auditing and recovery: Time travel with SQL-Auditing and recovery: Time travel with SQL

Term Frequency–Inverse Document Frequency (TF-IDF), Tokenization

Terraform

BigQuery configuration script, Infrastructure as Code (IaC) in scaling and for data platforms

Infrastructure as Code, Infrastructure as Code (IaC) in scaling and for data platforms

Snowflake configuration script, Infrastructure as Code (IaC) in scaling and for data platforms

testing strategies, Implementing testing strategies

chart comparing testing strategies, Implementing testing strategies

Consume layer semantic contracts, Semantic Contracts are Testable Contracts

dbt best practices, Transformation and Data Modeling with SQL

tests and documentation, Tests and documentation-Tests and documentation

governance by dbt, Testing and quality rules

layered testing, Implementing testing strategies

sandbox environment created via IaC, Infrastructure as Code (IaC) in scaling and for data platforms

skipping tests as anti-pattern, Skipping Tests for SQL Models

text feature engineering, Text feature engineering-Character n-grams

about encoding free-form text, Text feature engineering

character n-grams, Character n-grams

tokenization, Tokenization

text generation

free-form text, Generating free-form text-Generating free-form text

structured outputs, Structured outputs-Structured outputs

tips for producing, Structured outputs

temperature values, Structured outputs

text-to-SQL agent, Building a SQL-generating agent-Building a SQL-generating agent

text-to-SQL, Text-to-SQL

agent executing SQL from natural language, Building a SQL-generating agent-Building a SQL-generating agent

TF-IDF (Term Frequency–Inverse Document Frequency), Tokenization

third normal form (3NF), Third Normal Form—Structured, Relational Modeling-Third Normal Form—Structured, Relational Modeling

all related data in a single location, Third Normal Form—Structured, Relational Modeling

HR system tables in 3NF, Third Normal Form—Structured, Relational Modeling

schema designed in 3NF, Third Normal Form—Structured, Relational Modeling

stable core under analytical models, Third Normal Form—Structured, Relational Modeling

summary of traditional models, Summary of Traditional Models

tight coupling avoided, Best practices for the consumption layer, Tight Coupling Between Pipeline Components

ecommerce data pipeline, Pipeline Overview and Design Principles

time functions

SQL dialect comparison, Comparison of SQL Dialects over Time

window functions, Advanced Window Functions for Analytic Queries, Advanced Window Functions for Analytic Queries

time travel to query historical data via Iceberg, Auditing and recovery: Time travel with SQL-Auditing and recovery: Time travel with SQL

time-series forecasting, Time-series forecasting-Time-series forecasting

about time-series forecasting, Time-series forecasting

tokenization, Tokenization

when to use, Character n-grams

transformation and modeling in layered architecture, Transformation and modeling-SQL as a modeling language

data transformations handled by SQL, SQL for Data Transformations

ecommerce SQL data pipeline, Transformation and Data Modeling with SQL-Tests and documentation

building facts and dimensions with Gold models, Building facts and dimensions in the Gold layer-Building facts and dimensions in the Gold layer

dbt on BigQuery, Transformation and Data Modeling with SQL-Tests and documentation

dbt Silver and Gold model design, Silver and Gold Model Design in dbt: Building a Medallion Architecture-Tests and documentation

staging with Silver models, Staging with Silver models

tests and documentation, Tests and documentation-Tests and documentation

SQL as a modeling language, SQL as a modeling language

best practices, SQL as a modeling language

staged modeling layers, Staged modeling layers

transformation patterns, Transformation patterns

transitive dependency, Third Normal Form—Structured, Relational Modeling

troubleshooting (see debugging)

two-sample t-test, Two-sample t-test-Two-sample t-test

t-statistic, Two-sample t-test

t-value, Two-sample t-test

U

UAT (user acceptance testing), Implementing testing strategies

uncertainty around a sample estimate, Confidence Intervals: Estimating the Mean-Confidence Intervals: Estimating the Mean

unit testing, Implementing testing strategies

unstructured data, Dealing with Unstructured Data in SQL

unsupervised learning models, Unsupervised learning models-Unsupervised learning models

UPSERT operations, Idempotency and rerunnability

user acceptance testing (UAT), Implementing testing strategies

V

validating and evaluating models, Model Evaluation and Validation

vectors

- BigQuery vector search, Integrating ML and LLMs in SQL Platforms
- embeddings, search, and retrieval, Embeddings, Search, and Retrieval-Multimodal embeddings
  - designing an embeddings table, Designing an embeddings table-Designing an embeddings table
  - embeddings as semantic join keys, Embeddings as semantic join keys
  - hybrid search of lexical, vector, reranking, Hybrid search (lexical + vector + reranking)-Hybrid search (lexical + vector + reranking)
  - multimodal embeddings, Multimodal embeddings
  - query-time semantic retrieval, Query-time semantic retrieval with vector search-Query-time semantic retrieval with vector search

Snowflake vector embeddings data type, Integrating ML and LLMs in SQL Platforms

version control

Infrastructure as Code, Infrastructure as Code (IaC) in scaling and for data platforms

prompt version registry table, Run lineage and versioned artifacts

versioned artifacts for reproducibility, Run lineage and versioned artifacts, Practical Deployment Architectures

Vertex AI, Making models available for predictions

permissions documentation online, LLM Functions Inside SQL

remote model connecting to Gemini, LLM Functions Inside SQL

documentation online, LLM Functions Inside SQL

list of models and capabilities online, LLM Functions Inside SQL

Vizier (Google), A compact example

W

W3C (World Wide Web Consortium) Semantic Web stack, RDF and OWL: The Foundations of Semantic Modeling

warehouse-native GenAI definition, Generative AI in SQL

weather dataset statistical description, Dataset Statistical Description-Dataset Statistical Description

dataset setup script online, Dataset Statistical Description

Web Ontology Language (OWL)

semantic modeling, RDF and OWL: The Foundations of Semantic Modeling

transitioning from graph models, Graph Technologies: Neo4j and Cypher

wide tables (see one big table (OBT))

Wilcoxon rank-sum test, Mann–Whitney U-Multiple Groups Testing

window functions, Advanced Window Functions for Analytic Queries-Advanced Window Functions for Analytic Queries

about window functions, Advanced Window Functions for Analytic Queries

importance of, Advanced Window Functions for Analytic Queries

common categories of, Advanced Window Functions for Analytic Queries

comparing rows without self joins, Advanced Window Functions for Analytic Queries

handled by SQL, Extended Analytical and OLAP Functions

ranking, Advanced Window Functions for Analytic Queries

value window functions, Advanced Window Functions for Analytic Queries

workload isolation

BigQuery and Snowflake in practice, Concurrency and Isolation in Practice

performance tuning, Concurrency and workload isolation-Snowflake concurrency and isolation best practices

BigQuery best practices, BigQuery concurrency and isolation best practices

BigQuery versus Snowflake, Snowflake concurrency and isolation best practices

Snowflake best practices, Snowflake concurrency and isolation best practices

World Wide Web Consortium (W3C) Semantic Web stack, RDF and OWL: The Foundations of Semantic Modeling

X

XML

evolution of SQL, From the SQL:2003 Standard to SQL:2008

semi-structured data handling by SQL, Semi-Structured Data Handling in SQL

Z

zero-ETL, Beyond ETL/ELT and Streaming: Zero-ETL and Data Sharing, Beyond ETL/ELT and Streaming: Zero-ETL and Data Sharing

About the Authors

Rui Machado is a cofounder and CTO of Maio Labs, a company focused on building pragmatic AI solutions that empower companies to own their intelligence and use it for competitive advantage.

His background is in information technologies and data science, and he has over a decade of relevant experience in the architecture and implementation of data warehouses, data lakes, and decision support systems in industries such as retail, ecommerce, supply chain, health care, and social networks.

He was formerly a VP of engineering at H&M Group and previously led engineering and analytics teams at Jumia, Nike, and Meta. He is a coauthor of *Analytics Engineering with SQL and dbt* (O'Reilly).

LinkedIn profile: *https://www.linkedin.com/in/rpmachado*

Hélder Russa is a director of data engineering at H&M Group, leading the sales and global areas. In this role, he is responsible for developing and optimizing the entire data ecosystem, empowering data science, machine learning, and business intelligence teams with scalable, efficient solutions.

He also recently coauthored *Analytics Engineering with SQL and dbt* (O'Reilly), a book aimed at demystifying the role of analytics engineers in modern organizations. Through practical insights, the book explores how SQL and dbt can be leveraged to structure, transform, and operationalize data workflows efficiently.

Passionate about data-driven innovation and knowledge sharing, Hélder continues to contribute to the advancement of data engineering, analytics, and AI-powered decision making.

LinkedIn profile: *https://www.linkedin.com/in/hrussa*

Pedro Esmeriz is the founding AI engineer at Quartz, where he leads the development of the company's core AI system for personalized, autonomous financial decision making. He was formerly head of data science at Maersk and has a background in industrial engineering and management and data analytics. With over nine years of experience designing and deploying production AI systems, his work has spanned

predictive, prescriptive, and optimization systems across fintech, ecommerce, supply chain, and logistics, with a strong focus on turning complex operational and business problems into scalable AI solutions. He is also cofounder of ShopAI and has held leadership roles at Jumia, specializing in artificial intelligence, machine learning, optimization, and MLOps.

LinkedIn profile: *https://www.linkedin.com/in/pedroesmeriz*

Colophon

The animal on the cover of *Advanced SQL* is a red-billed streamertail (*Trochilus polytmus*), a hummingbird also known as a doctor bird. It is the national bird of Jamaica.

The bird's scientific name means "valuable hummingbird," which refers to its emerald plumage. The name "streamertail" refers to the males' long black tail feathers. These feathers also gave rise to the nickname "doctor bird," because they resemble the tailcoats worn by 19th-century physicians. The crown of the head is black, and the beak is a coral red. These hummingbirds are 4–9 inches long and weigh 0.11–0.23 ounces, with the males being considerably larger than the females.

The red-billed streamertail is abundant throughout Jamaica, except for the extreme eastern edge of the island, where the black-billed streamertail lives. It eats mostly nectar and some small insects. Streamertails build nests out of plant fibers that are bound with spiderwebs. They raise three broods of two chicks each per year.

The IUCN status of the red-billed streamertail is Least Concern. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Monica Kamsvaag, based on an antique line engraving from Lydekker's *Royal Natural History*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.