Chapter 8. Queries, Modeling, and Transformation

Up to this point, the stages of the data engineering lifecycle have primarily been about passing data from one place to another or storing it. In this chapter, you'll learn how to make data useful. By understanding queries, modeling, and transformations (see Figure 8-1), you'll have the tools to turn raw data ingredients into something consumable by downstream stakeholders.

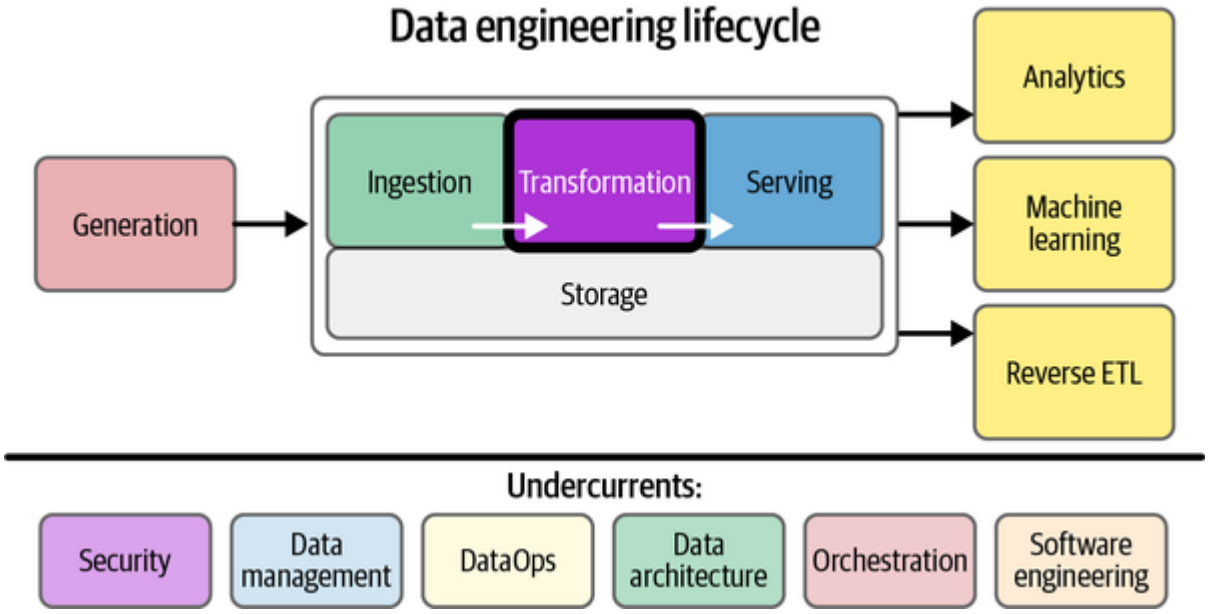

Figure 8-1. Transformations allow us to create value from data

We'll first discuss queries and the significant patterns underlying them. Second, we will look at the major data modeling patterns you can use to introduce business logic into your data. Then, we'll cover transformations, which take the logic of your data models and the results of queries and make them useful for more straightforward downstream consumption. Finally, we'll cover whom you'll work with and the undercurrents as they relate to this chapter.

A variety of techniques can be used to query, model, and transform data in SQL and NoSQL databases. This section focuses on queries made to an OLAP system, such as a data warehouse or data lake. Although many languages exist for querying, for the sake of convenience and familiarity, throughout most of this chapter, we'll focus heavily on SQL, the most popular and universal query language. Most of the concepts for OLAP databases and SQL will translate to other types of databases and query languages. This chapter assumes you have an understanding of the SQL language and related concepts like primary and foreign keys. If these ideas are unfamiliar to you, countless resources are available to help you get started.

A note on the terms used in this chapter. For convenience, we'll use the term *database* as a shorthand for a query engine and the storage it's querying; this could be a cloud data warehouse or Apache Spark querying data stored in S3. We assume the database has a storage engine that organizes the data under the hood. This extends to file-based queries (loading a CSV file into a Python notebook) and queries against file formats such as Parquet.

Also, note that this chapter focuses mainly on the query, modeling patterns, and transformations related to structured and semistructured data, which data engineers use often. Many of the practices discussed can also be applied to working with unstructured data such as images, video, and raw text.

Before we get into modeling and transforming data, let's look at queries—what they are, how they work, considerations for improving query performance, and queries on streaming data.

Queries

Queries are a fundamental part of data engineering, data science, and analysis. Before you learn about the underlying patterns and technologies for transformations, you need to understand what queries are, how they work on various data, and techniques for improving query performance.

This section primarily concerns itself with queries on tabular and semistructured data. As a data engineer, you'll most frequently query and transform these data types. Before we get into more complicated topics about queries, data modeling, and transformations, let's start by answering a pretty simple question: what is a query?

What Is a Query?

We often run into people who know how to write SQL but are unfamiliar with how a query works under the hood. Some of this introductory material on queries will be familiar to experienced data engineers; feel free to skip ahead if this applies to you.

A *query* allows you to retrieve and act on data. Recall our conversation in Chapter 5 about CRUD. When a query retrieves data, it is issuing a request to read a pattern of records. This is the *R* (read) in CRUD. You might issue a query that gets all records from a table `foo`, such as `SELECT * FROM foo`. Or, you might apply a predicate (logical condition) to filter your data by retrieving only records where the `id` is `1`, using the SQL query `SELECT * FROM foo WHERE id=1`.

Many databases allow you to create, update, and delete data. These are the *CUD* in CRUD; your query will either create, mutate, or destroy existing records. Let's review some other common acronyms you'll run into when working with query languages.

Data definition language

At a high level, you first need to create the database objects before adding data. You'll use *data definition language* (DDL) commands to perform operations on database objects, such as the database itself, schemas, tables, or users; DDL defines the state of objects in your database.

Data engineers use common SQL DDL expressions: `CREATE`, `DROP`, and `UPDATE`. For example, you can create a database by using the DDL expression `CREATE DATABASE bar`. After that, you can also create new tables (`CREATE table bar_table`) or delete a table (`DROP table bar_table`).

Data manipulation language

After using DDL to define database objects, you need to add and alter data within these objects, which is the primary purpose of *data manipulation language* (DML). Some common DML commands you'll use as a data engineer are as follows:

```
SELECT
INSERT
UPDATE
DELETE
COPY
MERGE
```

For example, you can `INSERT` new records into a database table, `UPDATE` existing ones, and `SELECT` specific records.

Data control language

You most likely want to limit access to database objects and finely control *who* has access to *what*. *Data control language* (DCL) allows you to control access to the database objects or the data by using SQL commands such as `GRANT`, `DENY`, and `REVOKE`.

Let's walk through a brief example using DCL commands. A new data scientist named Sarah joins your company, and she needs read-only access to a database called *data_science_db*. You give Sarah access to this database by using the following DCL command:

```
GRANT SELECT ON data_science_db TO user_name Sarah;
```

It's a hot job market, and Sarah has worked at the company for only a few months before getting poached by a big tech company. So long, Sarah! Being a security-minded data engineer, you remove Sarah's ability to read from the database:

```
REVOKE SELECT ON data_science_db TO user_name Sarah;
```

Access-control requests and issues are common, and understanding DCL will help you resolve problems if you or a team member can't access the data they need, as well as prevent access to data they don't need.

Transaction control language

As its name suggests, *transaction control language* (TCL) supports commands that control the details of transactions. With TCL, we can define commit checkpoints, conditions when actions will be rolled back, and more. Two common TCL commands include `COMMIT` and `ROLLBACK`.

The Life of a Query

How does a query work, and what happens when a query is executed? Let's cover the high-level basics of query execution (Figure 8-2), using an example of a typical SQL query executing in a database.

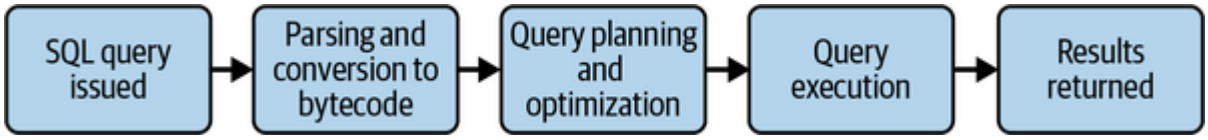

Figure 8-2. The life of a SQL query in a database

While running a query might seem simple—write code, run it, and get results—a lot is going on under the hood. When you execute a SQL query, here's a summary of what happens:

1. The database engine compiles the SQL, parsing the code to check for proper semantics and ensuring that the database objects referenced exist and that the current user has the appropriate access to these objects.

2. The SQL code is converted into bytecode. This bytecode expresses the steps that must be executed on the database engine in an efficient, machine-readable format.

3. The database's query optimizer analyzes the bytecode to determine how to execute the query, reordering and refactoring steps to use available resources as efficiently as possible.

4. The query is executed, and results are produced.

The Query Optimizer

Queries can have wildly different execution times, depending on how they're executed. A query optimizer's job is to optimize query performance and minimize costs by breaking the query into appropriate steps in an efficient order. The optimizer will assess joins, indexes, data scan size, and other factors. The query optimizer attempts to execute the query in the least expensive manner.

Query optimizers are fundamental to how your query will perform. Every database is different and executes queries in ways that are obviously and subtly different from each other. You won't directly work with a query optimizer, but understanding some of its functionality will help you write more performant queries. You'll need to know how to analyze a query's performance, using things like an explain plan or query analysis, described in the following section.

Improving Query Performance

In data engineering, you'll inevitably encounter poorly performing queries. Knowing how to identify and fix these queries is invaluable. Don't fight your database. Learn to work with its strengths and augment its weaknesses. This section shows various ways to improve your query performance.

Optimize your join strategy and schema

A single dataset (such as a table or file) is rarely useful on its own; we create value by combining it with other datasets. *Joins* are one of the most common means of combining datasets and creating new ones. We assume that you're familiar with the significant types of joins (e.g., inner, outer, left, cross) and the types of join relationships (e.g., one to one, one to many, many to one, and many to many).

Joins are critical in data engineering and are well supported and performant in many databases. Even columnar databases, which in the past had a reputation for slow join performance, now generally offer excellent performance.

A common technique for improving query performance is to *prejoin* data. If you find that analytics queries are joining the same data repeatedly, it often makes sense to join the data in advance and have queries read from the prejoined version of the data so that you're not repeating computationally intensive work. This may mean changing the schema and relaxing normalization conditions to widen tables and utilize newer data structures (such as arrays or structs) for replacing frequently joined entity relationships. Another strategy is maintaining a more normalized schema but prejoining tables for the most common analytics and data science use cases. We can simply create prejoined tables and train users to utilize these or join inside materialized views (see "Materialized Views, Federation, and Query Virtualization").

Next, consider the details and complexity of your join conditions. Complex join logic may consume significant computational resources. We can improve performance for complex joins in a few ways.

Many row-oriented databases allow you to index a result computed from a row. For instance, PostgreSQL allows you to create an index on a string field converted to lowercase; when the optimizer encounters a query where the `lower()` function appears inside a predicate, it can apply the index. You can also create a new derived column for joining, though you will need to train users to join on this column.

Row Explosion

An obscure but frustrating problem is row explosion. This occurs when we have a large number of many-to-many matches, either because of repetition in join keys or as a consequence of join logic. Suppose the join key in table A has the value `this` repeated five times, and the join key in table B contains this same value repeated 10 times. This leads to a cross-join of these rows: every `this` row from table A paired with every `this` row from table B. This creates 5 × 10 = 50 rows in the output.

Now suppose that many other repeats are in the join key. Row explosion often generates enough rows to consume a massive quantity of database resources or even cause a query to fail.

It is also essential to know how your query optimizer handles joins. Some databases can reorder joins and predicates, while others cannot. A row explosion in an early query stage may cause the query to fail, even though a later predicate should correctly remove many of the repeats in the output. Predicate reordering can significantly reduce the computational resources required by a query.

Finally, use common table expressions (CTEs) instead of nested subqueries or temporary tables. CTEs allow users to compose complex queries together in a readable fashion, helping you understand the flow of your query. The importance of readability for complex queries cannot be understated.

In many cases, CTEs will also deliver better performance than a script that creates intermediate tables; if you have to create intermediate tables, consider creating temporary tables. If you'd like to learn more about CTEs, a quick web search will yield plenty of helpful information.

Use the explain plan and understand your query's performance

As you learned in the preceding section, the database's query optimizer influences the execution of a query. The query optimizer's explain plan will show you how the query optimizer determined its optimum lowest-cost query, the database objects used (tables, indexes, cache, etc.), and various resource consumption and performance statistics in each query stage. Some databases provide a visual representation of query stages. In contrast, others make the explain plan available via SQL with the `EXPLAIN` command, which displays the sequence of steps the database will take to execute the query.

In addition to using `EXPLAIN` to understand *how* your query will run, you should monitor your query's performance, viewing metrics on database resource consumption. The following are some areas to monitor:

- Usage of key resources such as disk, memory, and network.
- Data loading time versus processing time.
- Query execution time, number of records, the size of the data scanned, and the quantity of data shuffled.
- Competing queries that might cause resource contention in your database.
- Number of concurrent connections used versus connections available. Oversubscribed concurrent connections can have negative effects on your users who may not be able to connect to the database.

Avoid full table scans

All queries scan data, but not all scans are created equal. As a rule of thumb, you should query only the data you need. When you run `SELECT *` with no predicates, you're scanning the entire table and retrieving every row and column. This is very inefficient performance-wise and expensive, especially if you're using a pay-as-you-go database that charges you either for bytes scanned or compute resources utilized while a query is running.

Whenever possible, use *pruning* to reduce the quantity of data scanned in a query. Columnar and row-oriented databases require different pruning strategies. In a column-oriented database, you should select only the columns you need. Most column-oriented OLAP databases also provide additional tools for optimizing your tables for better query performance. For instance, if you have a very large table (several terabytes in size or greater), Snowflake and BigQuery give you the option to define a cluster key on a table, which orders the table's data in a way that allows queries to more efficiently access portions of very large datasets. BigQuery also allows you to partition a table into smaller

segments, allowing you to query only specific partitions instead of the entire table. (Be aware that inappropriate clustering and key distribution strategies can degrade performance.)

In row-oriented databases, pruning usually centers around table indexes, which you learned in Chapter 6. The general strategy is to create table indexes that will improve performance for your most performance-sensitive queries while not overloading the table with so many indexes such that you degrade performance.

Know how your database handles commits

A database *commit* is a change within a database, such as creating, updating, or deleting a record, table, or other database objects. Many databases support *transactions*—i.e., a notion of committing several operations simultaneously in a way that maintains a consistent state. Please note that the term *transaction* is somewhat overloaded; see Chapter 5. The purpose of a transaction is to keep a consistent state of a database both while it's active and in the event of a failure. Transactions also handle isolation when multiple concurrent events might be reading, writing, and deleting from the same database objects. Without transactions, users would get potentially conflicting information when querying a database.

You should be intimately familiar with how your database handles commits and transactions, and determine the expected consistency of query results. Does your database handle writes and updates in an ACID-compliant manner? Without ACID compliance, your query might return unexpected results. This could result from a dirty read, which happens when a row is read and an uncommitted transaction has altered the row. Are dirty reads an expected behavior of your database? If so, how do you handle this? Also, be aware that during update and delete transactions, some databases create new files to represent the new state of the database and retain the old files for failure checkpoint references. In these databases, running a large number of small commits can lead to clutter and consume significant storage space that might need to be vacuumed periodically.

Let's briefly consider three databases to understand the impact of commits (note these examples are current as of the time of this writing). First, suppose we're looking at a PostgreSQL RDBMS and applying ACID transactions. Each transaction consists of a package of operations that will either fail or succeed as a group. We can also run analytics queries across many rows; these queries will present a consistent picture of the database at a point in time.

The disadvantage of the PostgreSQL approach is that it requires *row locking* (blocking reads and writes to certain rows), which can degrade performance in various ways. PostgreSQL is not optimized for large scans or the massive amounts of data appropriate for large-scale analytics applications.

Next, consider Google BigQuery. It utilizes a point-in-time full table commit model. When a read query is issued, BigQuery will read from the latest committed snapshot of the table. Whether the query runs for one second or two hours, it will read only from that snapshot and will not see any subsequent changes. BigQuery does not lock the table while I read from it. Instead, subsequent write operations will create new commits and new snapshots while the query continues to run on the snapshot where it started.

To prevent the inconsistent state, BigQuery allows only one write operation at a time. In this sense, BigQuery provides no write concurrency whatsoever. (In the sense that it can write massive amounts of data in parallel *inside a single write query*, it is highly concurrent.) If more than one client attempts to write simultaneously, write queries are queued in order of arrival. BigQuery's commit model is similar to the commit models used by Snowflake, Spark, and others.

Last, let's consider MongoDB. We refer to MongoDB as a *variable-consistency database*. Engineers have various configurable consistency options, both for the database and at the level of individual queries. MongoDB is celebrated for its extraordinary scalability and write concurrency but is somewhat notorious for issues that arise when engineers abuse it.[1]

For instance, in certain modes, MongoDB supports ultra-high write performance. However, this comes at a cost: the database will unceremoniously and silently discard writes if it gets overwhelmed with traffic. This is perfectly suitable for applications that can stand to lose some data—for example, IoT applications where we simply want many measurements but don't care about capturing all measurements. It is not a great fit for applications that need to capture exact data and statistics.

None of this is to say these are bad databases. They're all fantastic databases when they are chosen for appropriate applications and configured correctly. The same goes for virtually any database technology.

Companies don't hire engineers simply to hack on code in isolation. To be worthy of their title, engineers should develop a deep understanding of the problems they're tasked with solving and the technology tools. This applies to commit and consistency models and every other aspect of technology performance. Appropriate technology choices and configuration can ultimately differentiate extraordinary success and massive failure. Refer to Chapter 6 for a deeper discussion of consistency.

Vacuum dead records

As we just discussed, transactions incur the overhead of creating new records during certain operations, such as updates, deletes, and index operations, while retaining the old records as pointers to the last state of the database. As these old records accumulate in the database filesystem, they eventually no longer need to be referenced. You should remove these dead records in a process called *vacuuming*.

You can vacuum a single table, multiple tables, or all tables in a database. No matter how you choose to vacuum, deleting dead database records is important for a few reasons. First, it frees up space for new records, leading to less table bloat and faster queries. Second, new and relevant records mean query plans are more accurate; outdated records can lead the query optimizer to generate suboptimal and inaccurate plans. Finally, vacuuming cleans up poor indexes, allowing for better index performance.

Vacuum operations are handled differently depending on the type of database. For example, in databases backed by object storage (BigQuery, Snowflake, Databricks), the only downside of old data retention is that it uses storage space, potentially costing money depending on the storage pricing model for the database. In Snowflake, users cannot directly vacuum. Instead, they control a "time-travel" interval that determines how long table snapshots are retained before they are auto vacuumed. BigQuery utilizes a fixed seven-day history window. Databricks generally retains data indefinitely until it is manually vacuumed; vacuuming is important to control direct S3 storage costs.

Amazon Redshift handles its cluster disks in many configurations,[2] and vacuuming can impact performance and available storage. `VACUUM` runs automatically behind the scenes, but users may sometimes want to run it manually for tuning purposes.

Vacuuming becomes even more critical for relational databases such as PostgreSQL and MySQL. Large numbers of transactional operations can cause a rapid accumulation of dead records, and engineers working in these systems need to familiarize themselves with the details and impact of vacuuming.

Leverage cached query results

Let's say you have an intensive query that you often run on a database that charges you for the amount of data you query. Each time a query is run, this costs you money. Instead of rerunning the same query on the database repeatedly and incurring massive charges, wouldn't it be nice if the results of the query were stored and available for instant retrieval? Thankfully, many cloud OLAP databases cache query results.

When a query is initially run, it will retrieve data from various sources, filter and join it, and output a result. This initial query—a cold query—is similar to the notion of cold data we explored in Chapter 6.

For argument's sake, let's say this query took 40 seconds to run. Assuming your database caches query results, rerunning the same query might return results in 1 second or less. The results were cached, and the query didn't need to run cold. Whenever possible, leverage query cache results to reduce pressure on your database and provide a better user experience for frequently run queries. Note also that *materialized views* provide another form of query caching (see "Materialized Views, Federation, and Query Virtualization").

Queries on Streaming Data

Streaming data is constantly in flight. As you might imagine, querying streaming data is different from batch data. To fully take advantage of a data stream, we must adapt query patterns that reflect its real-time nature. For example, systems such as Kafka and Pulsar make it easier to query streaming data sources. Let's look at some common ways to do this.

Basic query patterns on streams

Recall continuous CDC, discussed in Chapter 7. CDC, in this form, essentially sets up an analytics database as a fast follower to a production database. One of the longest-standing streaming query patterns simply entails querying the analytics database, retrieving statistical results and aggregations with a slight lag behind the production database.

The fast-follower approach

How is this a streaming query pattern? Couldn't we accomplish the same thing simply by running our queries on the production database? In principle, yes; in practice, no. Production databases generally aren't equipped to handle production workloads and simultaneously run large analytics scans across significant quantities of data. Running such queries can slow the production application or even cause it to crash.[3] The basic CDC query pattern allows us to serve real-time analytics with a minimal impact on the production system.

The fast-follower pattern can utilize a conventional transactional database as the follower, but there are significant advantages to using a proper OLAP-oriented system (Figure 8-3). Both Druid and BigQuery combine a streaming buffer with long-term columnar storage in a setup somewhat similar to the Lambda architecture (see Chapter 3). This works extremely well for computing trailing statistics on vast historical data with near real-time updates.

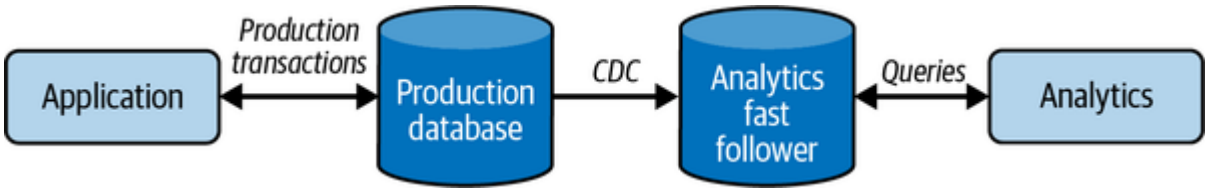

Figure 8-3. CDC with a fast-follower analytics database

The fast-follower CDC approach has critical limitations. It doesn't fundamentally rethink batch query patterns. You're still running `SELECT` queries against the current table state, and missing the opportunity to dynamically trigger events off changes in the stream.

The Kappa architecture

Next, recall the Kappa architecture we discussed in Chapter 3. The principal idea of this architecture is to handle all data like events and store these events as a stream rather than a table (Figure 8-4). When production application databases are the source, Kappa architecture stores events from CDC. Event streams can also flow directly from an application backend, from a swarm of IoT devices, or any system that generates events and can push them over a network. Instead of simply treating a streaming

storage system as a buffer, Kappa architecture retains events in storage during a more extended retention period, and data can be directly queried from this storage. The retention period can be pretty long (months or years). Note that this is much longer than the retention period used in purely real-time oriented systems, usually a week at most.

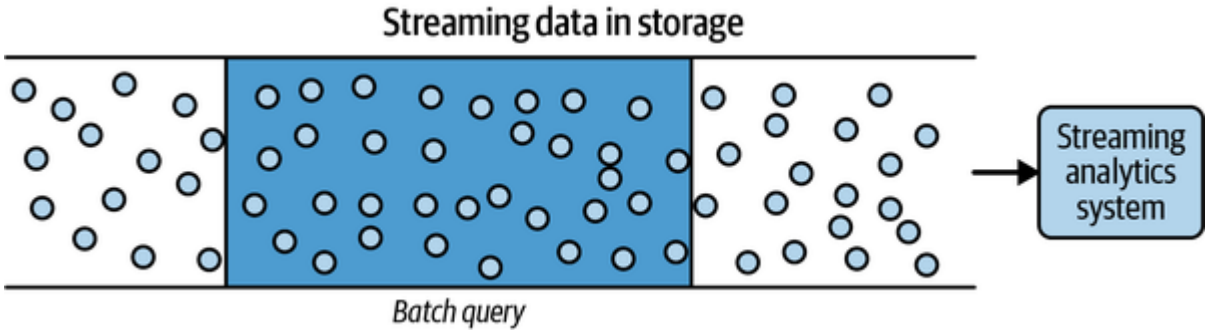

Figure 8-4. The Kappa architecture is built around streaming storage and ingest systems

The “big idea” in Kappa architecture is to treat streaming storage as a real-time transport layer and a database for retrieving and querying historical data. This happens either through the direct query capabilities of the streaming storage system or with the help of external tools. For example, Kafka KSQL supports aggregation, statistical calculations, and even sessionization. If query requirements are more complex or data needs to be combined with other data sources, an external tool such as Spark reads a time range of data from Kafka and computes the query results. The streaming storage system can also feed other applications or a stream processor such as Flink or Beam.

Windows, triggers, emitted statistics, and late-arriving data

One fundamental limitation of traditional batch queries is that this paradigm generally treats the query engine as an external observer. An actor external to the data causes the query to run—perhaps an hourly cron job or a product manager opening a dashboard.

Most widely used streaming systems, on the other hand, support the notion of computations triggered directly from the data itself. They might emit mean and median statistics every time a certain number of records are collected in the buffer or output a summary when a user session closes.

Windows are an essential feature in streaming queries and processing. Windows are small batches that are processed based on dynamic triggers. Windows are generated dynamically over time in some ways. Let’s look at some common types of windows: session, fixed-time, and sliding. We’ll also look at watermarks.

Session window

A *session window* groups events that occur close together, and filters out periods of inactivity when no events occur. We might say that a user session is any time interval with no inactivity gap of five minutes or more. Our batch system collects data by a user ID key, orders events, determines the gaps and session boundaries, and calculates statistics for each session. Data engineers often sessionize data retrospectively by applying time conditions to user activity on web and desktop apps.

In a streaming session, this process can happen dynamically. Note that session windows are per key; in the preceding example, each user gets their own set of windows. The system accumulates data per user. If a five-minute gap with no activity occurs, the system closes the window, sends its calculations, and flushes the data. If new events arrive for the user, the system starts a new session window.

Session windows may also make a provision for late-arriving data. Allowing data to arrive up to five minutes late to account for network conditions and system latency, the system will open the window if a late-arriving event indicates activity less than five minutes after the last event. We will have more to

say about late-arriving data throughout this chapter. Figure 8-5 shows three session windows, each separated by five minutes of inactivity.

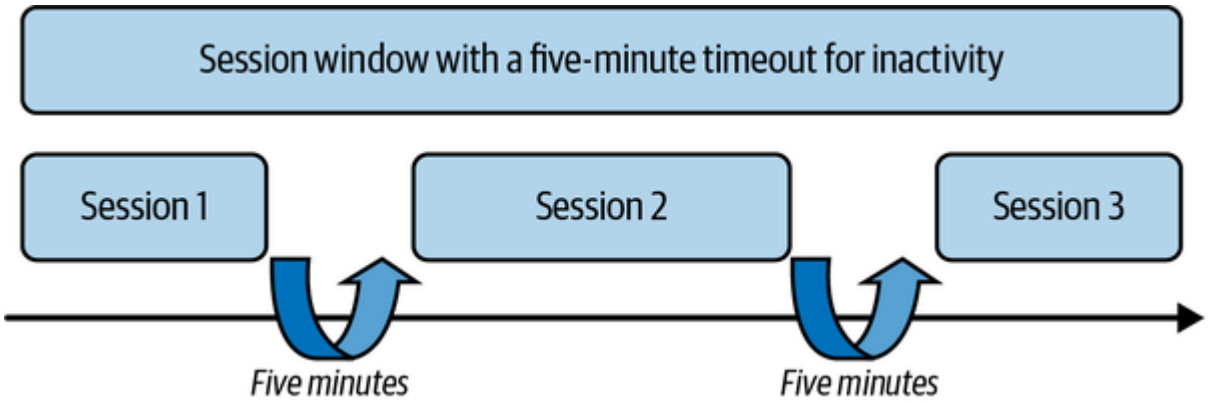

Figure 8-5. Session window with a five-minute timeout for inactivity

Making sessionization dynamic and near real-time fundamentally changes its utility. With retrospective sessionization, we could automate specific actions a day or an hour after a user session closed (e.g., a follow-up email with a coupon for a product viewed by the user). With dynamic sessionization, the user could get an alert in a mobile app that is immediately useful based on their activity in the last 15 minutes.

Fixed-time windows

A *fixed-time* (aka *tumbling*) window features fixed time periods that run on a fixed schedule and processes all data since the previous window is closed. For example, we might close a window every 20 seconds and process all data arriving from the previous window to give a mean and median statistic (Figure 8-6). Statistics would be emitted as soon as they could be calculated after the window closed.

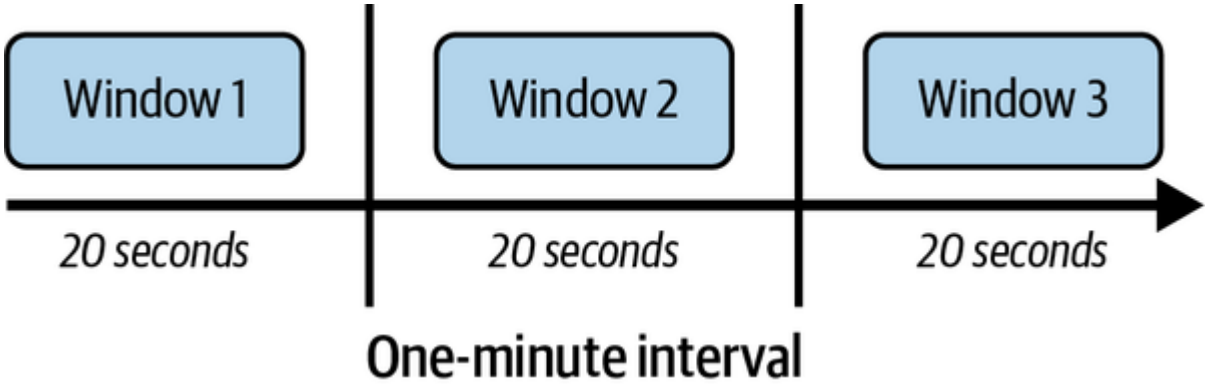

Figure 8-6. Tumbling/fixed window

This is similar to traditional batch ETL processing, where we might run a data update job every day or every hour. The streaming system allows us to generate windows more frequently and deliver results with lower latency. As we'll repeatedly emphasize, batch is a special case of streaming.

Sliding windows

Events in a sliding window are bucketed into windows of fixed time length, where separate windows might overlap. For example, we could generate a new 60-second window every 30 seconds (Figure 8-7). Just as we did before, we can emit mean and median statistics.

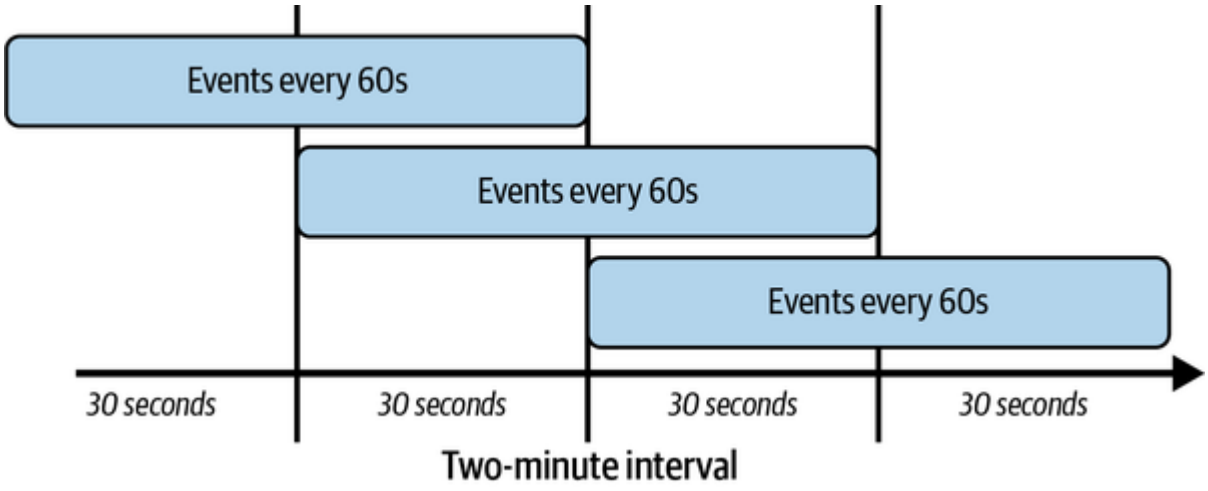

Figure 8-7. Sliding windows

The sliding can vary. For example, we might think of the window as truly sliding continuously but emitting statistics only when certain conditions (triggers) are met. Suppose we used a 30-second continuously sliding window but calculated a statistic only when a user clicked a particular banner. This would lead to an extremely high rate of output when many users click the banner, and no calculations during a lull.

Watermarks

We've covered various types of windows and their uses. As discussed in Chapter 7, data is sometimes ingested out of the order from which it originated. A *watermark* (Figure 8-8) is a threshold used by a window to determine whether data in a window is within the established time interval or whether it's considered late. If data arrives that is new to the window but older than the timestamp of the watermark, it is considered to be late-arriving data.

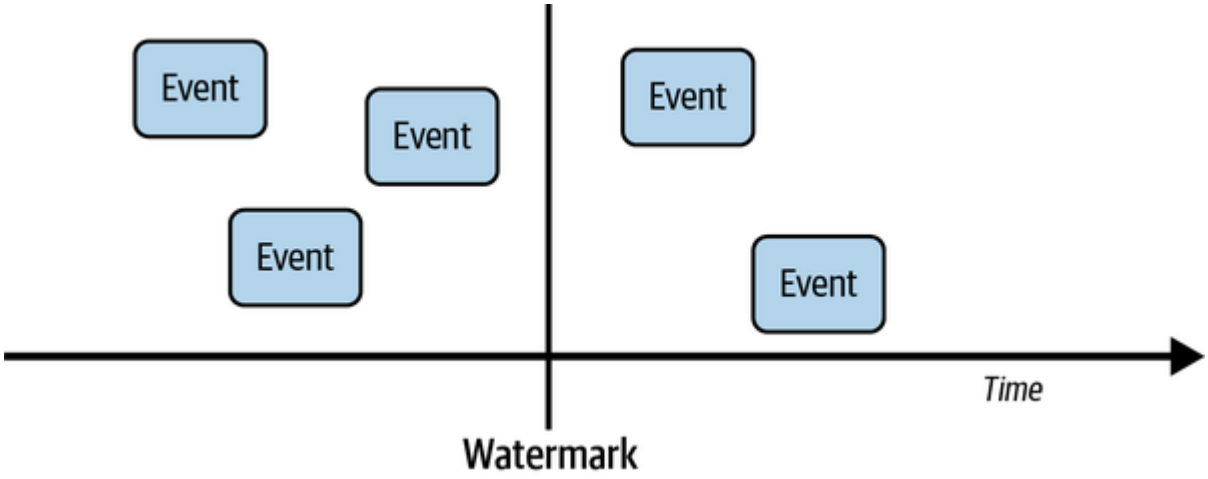

Figure 8-8. Watermark acting as a threshold for late-arriving data

Combining streams with other data

As we've mentioned before, we often derive value from data by combining it with other data. Streaming data is no different. For instance, multiple streams can be combined, or a stream can be combined with batch historical data.

Conventional table joins

Some tables may be fed by streams (Figure 8-9). The most basic approach to this problem is simply joining these two tables in a database. A stream can feed one or both of these tables.

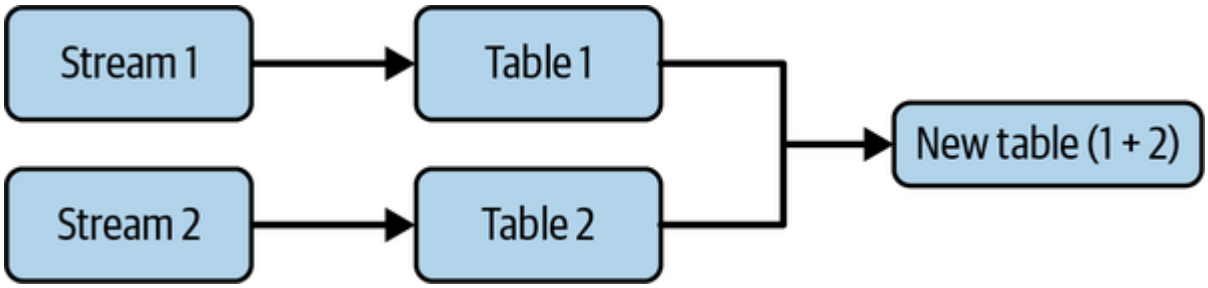

Figure 8-9. Joining two tables fed by streams

Enrichment

Enrichment means that we join a stream to other data (Figure 8-10). Typically, this is done to provide enhanced data into another stream. For example, suppose that an online retailer receives an event stream from a partner business containing product and user IDs. The retailer wishes to enhance these events with product details and demographic information on the users. The retailer feeds these events to a serverless function that looks up the product and user in an in-memory database (say, a cache), adds the required information to the event, and outputs the enhanced events to another stream.

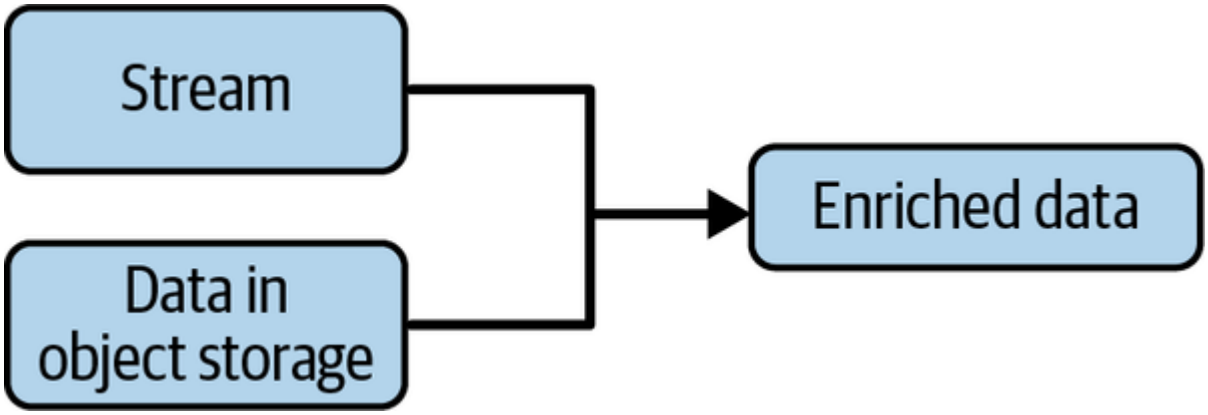

Figure 8-10. In this example, a stream is enriched with data residing in object storage, resulting in a new enriched dataset

In practice, the enrichment source could originate almost anywhere—a table in a cloud data warehouse or RDBMS, or a file in object storage. It's simply a question of reading from the source and storing the requisite enrichment data in an appropriate place for retrieval by the stream.

Stream-to-stream joining

Increasingly, streaming systems support direct stream-to-stream joining. Suppose that an online retailer wishes to join its web event data with streaming data from an ad platform. The company can feed both streams into Spark, but a variety of complications arise. For instance, the streams may have significantly different latencies for arrival at the point where the join is handled in the streaming system. The ad platform may provide its data with a five-minute delay. In addition, certain events may be significantly delayed—for example, a session close event for a user, or an event that happens on the phone offline and shows up in the stream only after the user is back in mobile network range.

As such, typical streaming join architectures rely on streaming buffers. The buffer retention interval is configurable; a longer retention interval requires more storage and other resources. Events get joined with data in the buffer and are eventually evicted after the retention interval has passed (Figure 8-11).[4]

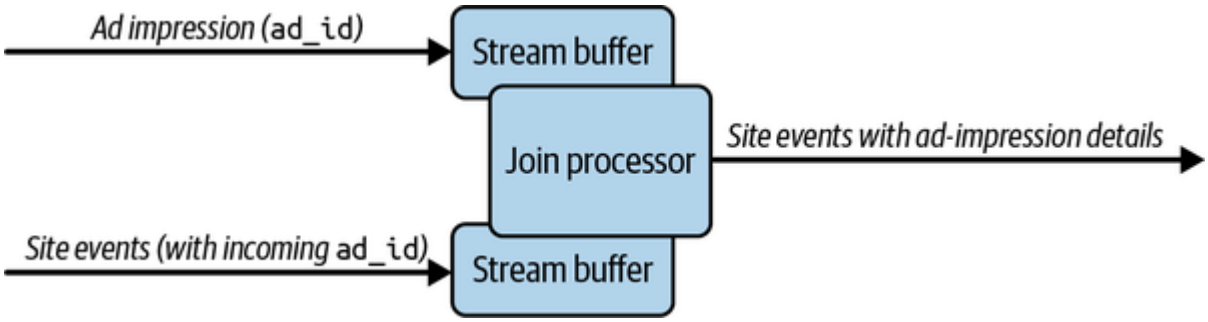

Figure 8-11. An architecture to join streams buffers each stream and joins events if related events are found during the buffer retention interval

Now that we've covered how queries work for batch and streaming data, let's discuss making your data useful by modeling it.

Data Modeling

Data modeling is something that we see overlooked disturbingly often. We often see data teams jump into building data systems without a game plan to organize their data in a way that's useful for the business. This is a mistake. Well-constructed data architectures must reflect the goals and business logic of the organization that relies on this data. Data modeling involves deliberately choosing a coherent structure for data and is a critical step to make data useful for the business.

Data modeling has been a practice for decades in one form or another. For example, various types of normalization techniques (discussed in "Normalization") have been used to model data since the early days of RDBMSs; data warehousing modeling techniques have been around since at least the early 1990s and arguably longer. As pendulums in technology often go, data modeling became somewhat unfashionable in the early to mid-2010s. The rise of data lake 1.0, NoSQL, and big data systems allowed engineers to bypass traditional data modeling, sometimes for legitimate performance gains. Other times, the lack of rigorous data modeling created data swamps, along with lots of redundant, mismatched, or simply wrong data.

Nowadays, the pendulum seems to be swinging back toward data modeling. The growing popularity of data management (in particular, data governance and data quality) is pushing the need for coherent business logic. The meteoric rise of data's prominence in companies creates a growing recognition that modeling is critical for realizing value at the higher levels of the Data Science Hierarchy of Needs pyramid. That said, we believe that new paradigms are required to truly embrace the needs of streaming data and ML. In this section, we survey current mainstream data modeling techniques and briefly muse on the future of data modeling.

What Is a Data Model?

A *data model* represents the way data relates to the real world. It reflects how the data must be structured and standardized to best reflect your organization's processes, definitions, workflows, and logic. A good data model captures how communication and work naturally flow within your organization. In contrast, a poor data model (or nonexistent one) is haphazard, confusing, and incoherent.

Some data professionals view data modeling as tedious and reserved for "big enterprises." Like most good hygiene practices—such as flossing your teeth and getting a good night's sleep—data modeling is acknowledged as a good thing to do but is often ignored in practice. Ideally, every organization should model its data if only to ensure that business logic and rules are translated at the data layer.

When modeling data, it's critical to focus on translating the model to business outcomes. A good data model should correlate with impactful business decisions. For example, a *customer* might mean

different things to different departments in a company. Is someone who's bought from you over the last 30 days a customer? What if they haven't bought from you in the previous six months or a year? Carefully defining and modeling this customer data can have a massive impact on downstream reports on customer behavior or the creation of customer churn models whereby the time since the last purchase is a critical variable.

Tip

A good data model contains consistent definitions. In practice, definitions are often messy throughout a company. Can you think of concepts or terms in your company that might mean different things to different people?

Our discussion focuses mainly on batch data modeling since that's where most data modeling techniques arose. We will also look at some approaches to modeling streaming data and general considerations for modeling.

Conceptual, Logical, and Physical Data Models

When modeling data, the idea is to move from abstract modeling concepts to concrete implementation. Along this continuum (Figure 8-12), three main data models are conceptual, logical, and physical. These models form the basis for the various modeling techniques we describe in this chapter:

Conceptual

> Contains business logic and rules and describes the system's data, such as schemas, tables, and fields (names and types). When creating a conceptual model, it's often helpful to visualize it in an entity-relationship (ER) diagram, which is a standard tool for visualizing the relationships among various entities in your data (orders, customers, products, etc.). For example, an ER diagram might encode the connections among customer ID, customer name, customer address, and customer orders. Visualizing entity relationships is highly recommended for designing a coherent conceptual data model.

Logical

> Details how the conceptual model will be implemented in practice by adding significantly more detail. For example, we would add information on the types of customer ID, customer names, and custom addresses. In addition, we would map out primary and foreign keys.

Physical

> Defines how the logical model will be implemented in a database system. We would add specific databases, schemas, and tables to our logical model, including configuration details.

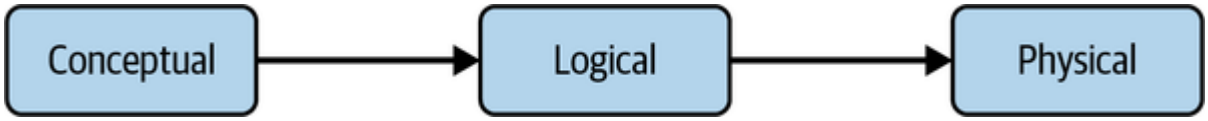

Figure 8-12. The continuum of data models: conceptual, logical, and physical

Successful data modeling involves business stakeholders at the inception of the process. Engineers need to obtain definitions and business goals for the data. Modeling data should be a full-contact sport whose goal is to provide the business with quality data for actionable insights and intelligent automation. This is a practice that everyone must continuously participate in.

Another important consideration for data modeling is the *grain* of the data, which is the resolution at which data is stored and queried. The grain is typically at the level of a primary key in a table, such as

customer ID, order ID, and product ID; it's often accompanied by a date or timestamp for increased fidelity.

For example, suppose that a company has just begun to deploy BI reporting. The company is small enough that the same person is filling the role of data engineer and analyst. A request comes in for a report that summarizes daily customer orders. Specifically, the report should list all customers who ordered, the number of orders they placed that day, and the total amount they spent.

This report is inherently coarse-grained. It contains no details on spending per order or the items in each order. It is tempting for the data engineer/analyst to ingest data from the production orders database and boil it down to a reporting table with only the basic aggregated data required for the report. However, this would entail starting over when a request comes in for a report with finer-grained data aggregation.

Since the data engineer is actually quite experienced, they elect to create tables with detailed data on customer orders, including each order, item, item cost, item IDs, etc. Essentially, their tables contain all details on customer orders. The data's grain is at the customer-order level. This customer-order data can be analyzed as is, or aggregated for summary statistics on customer order activity.

In general, you should strive to model your data at the lowest level of grain possible. From here, it's easy to aggregate this highly granular dataset. The reverse isn't true, and it's generally impossible to restore details that have been aggregated away.

Normalization

Normalization is a database data modeling practice that enforces strict control over the relationships of tables and columns within a database. The goal of normalization is to remove the redundancy of data within a database and ensure referential integrity. Basically, it's *don't repeat yourself* (DRY) applied to data in a database.[5]

Normalization is typically applied to relational databases containing tables with rows and columns (we use the terms *column* and *field* interchangeably in this section). It was first introduced by relational database pioneer Edgar Codd in the early 1970s.

Codd outlined four main objectives of normalization:[6]

- To free the collection of relations from undesirable insertion, update, and deletion dependencies
- To reduce the need for restructuring the collection of relations, as new types of data are introduced, and thus increase the lifespan of application programs
- To make the relational model more informative to users
- To make the collection of relations neutral to the query statistics, where these statistics are liable to change as time goes by

Codd introduced the idea of *normal forms*. The normal forms are sequential, with each form incorporating the conditions of prior forms. We describe Codd's first three normal forms here:

Denormalized

No normalization. Nested and redundant data is allowed.

First normal form (1NF)

Each column is unique and has a single value. The table has a unique primary key.

Second normal form (2NF)

The requirements of 1NF, plus partial dependencies are removed.

Third normal form (3NF)

The requirements of 2NF, plus each table contains only relevant fields related to its primary key and has no transitive dependencies.

It's worth spending a moment to unpack a couple of terms we just threw at you. A *unique primary key* is a single field or set of multiple fields that uniquely determines rows in the table. Each key value occurs at most once; otherwise, a value would map to multiple rows in the table. Thus, every other value in a row is dependent on (can be determined from) the key. A *partial dependency* occurs when a subset of fields in a composite key can be used to determine a nonkey column of the table. A *transitive dependency* occurs when a nonkey field depends on another nonkey field.

Let's look at stages of normalization—from denormalized to 3NF—using an ecommerce example of customer orders (Table 8-1). We'll provide concrete explanations of each of the concepts introduced in the previous paragraph.

Table 8-1. `OrderDetail`

OrderID	OrderItems	CustomerID	CustomerName	OrderDate

100	[{
	"sku": 1,
	"price": 50,
	"quantity": 1,
	"name:": "Thingamajig"
	}, {
	"sku": 2,
	"price": 25,
	"quantity": 2,
	"name:": "Whatchamacallit"
	}]
		5	Joe Reis	2022-03-01

First, this denormalized `OrderDetail` table contains five fields. The primary key is `OrderID`. Notice that the `OrderItems` field contains a nested object with two SKUs along with their price, quantity, and name.

To convert this data to 1NF, let's move `OrderItems` into four fields (Table 8-2). Now we have an `OrderDetail` table in which fields do not contain repeats or nested data.

Table 8-2. `OrderDetail` without repeats or nested data

OrderID	Sku	Price	Quantity	ProductName	CustomerID	CustomerName	OrderDate
100	1	50	1	Thingamajig	5	Joe Reis	2022-03-01
100	2	25	2	Whatchamacallit	5	Joe Reis	2022-03-01

The problem is that now we don't have a unique primary key. That is, 100 occurs in the `OrderID` column in two different rows. To get a better grasp of the situation, let's look at a larger sample from our table (Table 8-3).

Table 8-3. `OrderDetail` with a larger sample

OrderID	Sku	Price	Quantity	ProductName	CustomerID	CustomerName	OrderDate
100	1	50	1	Thingamajig	5	Joe Reis	2022-03-01
100	2	25	2	Whatchamacallit	5	Joe Reis	2022-03-01
101	3	75	1	Whozeewhatzit	7	Matt Housley	2022-03-01
102	1	50	1	Thingamajig	7	Matt Housley	2022-03-01

To create a unique primary (composite) key, let's number the lines in each order by adding a column called `LineItemNumber` (Table 8-4).

Table 8-4. `OrderDetail` with `LineItemNumber` column

OrderID	LineItemNumber	Sku	Price	Quantity	ProductName	CustomerID	CustomerName	OrderDate

OrderID	LineItemNumber	Sku	Price	Quantity	ProductName	CustomerID	CustomerName	OrderDate
100	1	1	50	1	Thingamajig	5	Joe Reis	2022-03-01
100	2	2	25	2	Whatchamacallit	5	Joe Reis	2022-03-01
101	1	3	75	1	Whozeewhatzit	7	Matt Housley	2022-03-01
102	1	1	50	1	Thingamajig	7	Matt Housley	2022-03-01

The composite key (`OrderID`, `LineItemNumber`) is now a unique primary key.

To reach 2NF, we need to ensure that no partial dependencies exist. A *partial dependency* is a nonkey column that is fully determined by a subset of the columns in the unique primary (composite) key; partial dependencies can occur only when the primary key is composite. In our case, the last three columns are determined by order number. To fix this problem, let's split `OrderDetail` into two tables: `Orders` and `OrderLineItem` (Tables 8-5 and 8-6).

Table 8-5. `Orders`

OrderID	CustomerID	CustomerName	OrderDate
100	5	Joe Reis	2022-03-01
101	7	Matt Housley	2022-03-01
102	7	Matt Housley	2022-03-01

Table 8-6. `OrderLineItem`

OrderID	LineItemNumber	Sku	Price	Quantity	ProductName
100	1	1	50	1	Thingamajig
100	2	2	25	2	Whatchamacallit
101	1	3	75	1	Whozeewhatzit
102	1	1	50	1	Thingamajig

The composite key (`OrderID`, `LineItemNumber`) is a unique primary key for `OrderLineItem`, while `OrderID` is a primary key for `Orders`.

Notice that `Sku` determines `ProductName` in `OrderLineItem`. That is, `Sku` depends on the composite key, and `ProductName` depends on `Sku`. This is a transitive dependency. Let's break `OrderLineItem` into `OrderLineItem` and `Skus` (Tables 8-7 and 8-8).

Table 8-7. `OrderLineItem`

OrderID	LineItemNumber	Sku	Price	Quantity
100	1	1	50	1
100	2	2	25	2
101	1	3	75	1
102	1	1	50	1

Table 8-8. `Skus`

Sku	ProductName
1	Thingamajig
2	Whatchamacallit
3	Whozeewhatzit

Now, both `OrderLineItem` and `Skus` are in 3NF. Notice that `Orders` does not satisfy 3NF. What transitive dependencies are present? How would you fix this?

Additional normal forms exist (up to 6NF in the Boyce-Codd system), but these are much less common than the first three. A database is usually considered normalized if it's in third normal form, and that's the convention we use in this book.

The degree of normalization that you should apply to your data depends on your use case. No one-size-fits-all solution exists, especially in databases where some denormalization presents performance advantages. Although denormalization may seem like an antipattern, it's common in many OLAP systems that store semistructured data. Study normalization conventions and database best practices to choose an appropriate strategy.

Techniques for Modeling Batch Analytical Data

When describing data modeling for data lakes or data warehouses, you should assume that the raw data takes many forms (e.g., structured and semistructured), but the output is a structured data model of rows and columns. However, several approaches to data modeling can be used in these environments. The big approaches you'll likely encounter are Kimball, Inmon, and Data Vault.

In practice, some of these techniques can be combined. For example, we see some data teams start with Data Vault and then add a Kimball star schema alongside it. We'll also look at wide and denormalized data models and other batch data-modeling techniques you should have in your arsenal. As we discuss each of these techniques, we will use the example of modeling transactions occurring in an ecommerce order system.

Note

Our coverage of the first three approaches—Inmon, Kimball, and Data Vault—is cursory and hardly does justice to their respective complexity and nuance. At the end of each section, we list the canonical books from their creators. For a data engineer, these books are must-reads, and we highly encourage you to read them, if only to understand how and why data modeling is central to batch analytical data.

Inmon

The father of the data warehouse, Bill Inmon, created his approach to data modeling in 1989. Before the data warehouse, the analysis would often occur directly on the source system itself, with the obvious consequence of bogging down production transactional databases with long-running queries. The goal of the data warehouse was to separate the source system from the analytical system.

Inmon defines a data warehouse the following way:[7]

> A data warehouse is a subject-oriented, integrated, nonvolatile, and time-variant collection of data in support of management's decisions. The data warehouse contains granular corporate data. Data in the data warehouse is able to be used for many different purposes, including sitting and waiting for future requirements which are unknown today.

The four critical parts of a data warehouse can be described as follows:

Subject-oriented

> The data warehouse focuses on a specific subject area, such as sales or marketing.

Integrated

> Data from disparate sources is consolidated and normalized.

Nonvolatile

> Data remains unchanged after data is stored in a data warehouse.

Time-variant

> Varying time ranges can be queried.

Let's look at each of these parts to understand its influence on an Inmon data model. First, the logical model must focus on a specific area. For instance, if the *subject orientation* is "sales," then the logical model contains all details related to sales—business keys, relationships, attributes, etc. Next, these details are *integrated* into a consolidated and highly normalized data model. Finally, the data is stored unchanged in a *nonvolatile* and *time-variant* way, meaning you can (theoretically) query the original data for as long as storage history allows. The Inmon data warehouse must strictly adhere to all four of

these critical parts *in support of management's decisions*. This is a subtle point, but it positions the data warehouse for analytics, not OLTP.

Here is another key characteristic of Inmon's data warehouse:[8]

> The second salient characteristic of the data warehouse is that it is integrated. Of all the aspects of a data warehouse, integration is the most important. Data is fed from multiple, disparate sources into the data warehouse. As the data is fed, it is converted, reformatted, resequenced, summarized, etc. The result is that data—once it resides in the data warehouse—has a single physical corporate image.

With Inmon's data warehouse, data is integrated from across the organization in a granular, highly normalized ER model, with a relentless emphasis on ETL. Because of the subject-oriented nature of the data warehouse, the Inmon data warehouse consists of key source databases and information systems used in an organization. Data from key business source systems is ingested and integrated into a highly normalized (3NF) data warehouse that often closely resembles the normalization structure of the source system itself; data is brought in incrementally, starting with the highest-priority business areas. The strict normalization requirement ensures as little data duplication as possible, which leads to fewer downstream analytical errors because data won't diverge or suffer from redundancies. The data warehouse represents a "single source of truth," which supports the overall business's information requirements. The data is presented for downstream reports and analysis via business and department-specific data marts, which may also be denormalized.

Let's look at how an Inmon data warehouse is used for ecommerce (Figure 8-13). The business source systems are orders, inventory, and marketing. The data from these source systems are ETLed to the data warehouse and stored in 3NF. Ideally, the data warehouse holistically encompasses the business's information. To serve data for department-specific information requests, ETL processes take data from the data warehouse, transform the data, and place it in downstream data marts to be viewed in reports.

A popular option for modeling data in a data mart is a star schema (discussed in the following section on Kimball), though any data model that provides easily accessible information is also suitable. In the preceding example, sales, marketing, and purchasing have their own star schema, fed upstream from the granular data in the data warehouse. This allows each department to have its own data structure that's unique and optimized to its specific needs.

Inmon continues to innovate in the data warehouse space, currently focusing on textual ETL in the data warehouse. He's also a prolific writer and thinker, writing over 60 books and countless articles. For further reading about Inmon's data warehouse, please refer to his books listed in "Additional Resources".

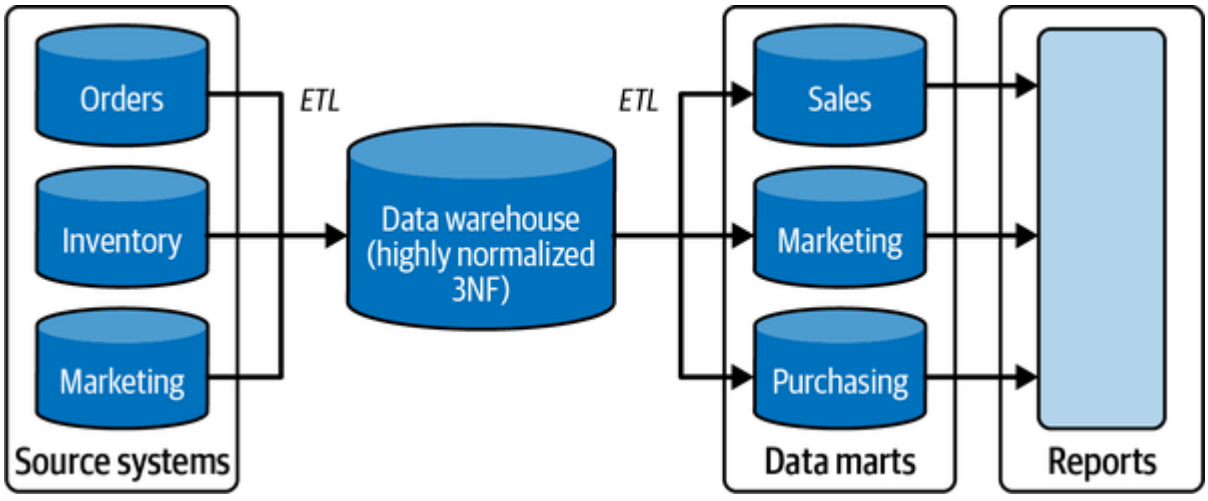

Figure 8-13. An ecommerce data warehouse

Kimball

If there are spectrums to data modeling, Kimball is very much on the opposite end of Inmon. Created by Ralph Kimball in the early 1990s, this approach to data modeling focuses less on normalization, and in some cases accepting denormalization. As Inmon says about the difference between the data warehouse and data mart, "A data mart is never a substitute for a data warehouse."[9]

Whereas Inmon integrates data from across the business in the data warehouse, and serves department-specific analytics via data marts, the Kimball model is bottom-up, encouraging you to model and serve department or business analytics in the data warehouse itself (Inmon argues this approach skews the definition of a data warehouse). The Kimball approach effectively makes the data mart the data warehouse itself. This may enable faster iteration and modeling than Inmon, with the trade-off of potential looser data integration, data redundancy, and duplication.

In Kimball's approach, data is modeled with two general types of tables: facts and dimensions. You can think of a *fact table* as a table of numbers, and *dimension tables* as qualitative data referencing a fact. Dimension tables surround a single fact table in a relationship called a *star schema* (Figure 8-14).[10] Let's look at facts, dimensions, and star schemas.

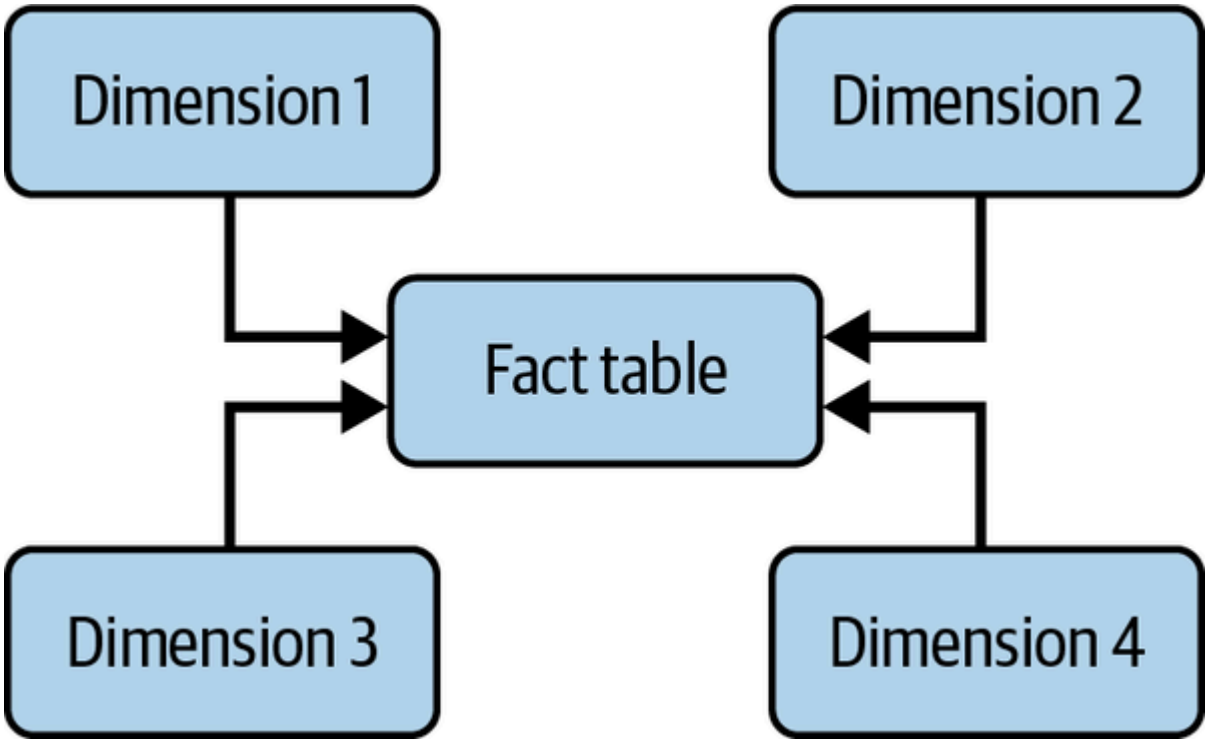

Figure 8-14. A Kimball star schema, with facts and dimensions

Fact tables

The first type of table in a star schema is the fact table, which contains *factual*, quantitative, and event-related data. The data in a fact table is immutable because facts relate to events. Therefore, fact tables don't change and are append-only. Fact tables are typically narrow and long, meaning they have not a lot of columns but a lot of rows that represent events. Fact tables should be at the lowest grain possible.

Queries against a star schema start with the fact table. Each row of a fact table should represent the grain of the data. Avoid aggregating or deriving data within a fact table. If you need to perform aggregations or derivations, do so in a downstream query, data mart table, or view. Finally, fact tables don't reference other fact tables; they reference only dimensions.

Let's look at an example of an elementary fact table (Table 8-9). A common question in your company might be, "Show me gross sales, by each customer order, by date." Again, facts should be at the lowest

grain possible—in this case, the orderID of the sale, customer, date, and gross sale amount. Notice that the data types in the fact table are all numbers (integers and floats); there are no strings. Also, in this example, `CustomerKey 7` has two orders on the same day, reflecting the grain of the table. Instead, the fact table has keys that reference dimension tables containing their respective attributes, such as the customer and date information. The gross sales amount represents the total sale for the sales *event*.

Table 8-9. A fact table

OrderID	CustomerKey	DateKey	GrossSalesAmt
100	5	20220301	100.00
101	7	20220301	75.00
102	7	20220301	50.00

Dimension tables

The second primary type of table in a Kimball data model is called a *dimension*. Dimension tables provide the reference data, attributes, and relational context for the events stored in fact tables. Dimension tables are smaller than fact tables and take an opposite shape, typically wide and short. When joined to a fact table, dimensions can describe the events' what, where, and when. Dimensions are denormalized, with the possibility of duplicate data. This is OK in the Kimball data model. Let's look at the two dimensions referenced in the earlier fact table example.

In a Kimball data model, dates are typically stored in a date dimension, allowing you to reference the date key (`DateKey`) between the fact and date dimension table. With the date dimension table, you can easily answer questions like, "What are my total sales in the first quarter of 2022?" or "How many more customers shop on Tuesday than Wednesday?" Notice we have five fields in addition to the date key (Table 8-10). The beauty of a date dimension is that you can add as many new fields as makes sense to analyze your data.

Table 8-10. A date dimension table

DateKey	Date-ISO	Year	Quarter	Month	Day-of-week
20220301	2022-03-01	2022	1	3	Tuesday
20220302	2022-03-02	2022	1	3	Wednesday
20220303	2022-03-03	2022	1	3	Thursday

Table 8-11 also references another dimension—the customer dimension—by the `CustomerKey` field. The customer dimension contains several fields that describe the customer: first and last name, zip code, and a couple of peculiar-looking date fields. Let's look at these date fields, as they illustrate another concept in the Kimball data model: a Type 2 slowly changing dimension, which we'll describe in greater detail next.

Table 8-11. A Type 2 customer dimension table

CustomerKey	FirstName	LastName	ZipCode	EFF_StartDate	EFF_EndDate
5	Joe	Reis	84108	2019-01-04	9999-01-01
7	Matt	Housley	84101	2020-05-04	2021-09-19
7	Matt	Housley	84123	2021-09-19	9999-01-01
11	Lana	Belle	90210	2022-02-04	9999-01-01

For example, take a look at `CustomerKey 5`, with the `EFF_StartDate` (`EFF_StartDate` means *effective start date*) of `2019-01-04` and an `EFF_EndDate` of `9999-01-01`. This means Joe Reis's customer record was created in the customer dimension table on 2019-01-04 and has an end date of 9999-01-01. Interesting. What does this end date mean? It means the customer record is active and isn't changed.

Now let's look at Matt Housley's customer record (`CustomerKey = 7`). Notice the two entries for Housley's start date: `2020-05-04` and `2021-09-19`. It looks like Housley changed his zip code on 2021-09-19, resulting in a change to his customer record. When the data is queried for the most recent customer records, you will query where the end date is equal to 9999-01-01.

A slowly changing dimension (SCD) is necessary to track changes in dimensions. The preceding example is a Type 2 SCD: a new record is inserted when an existing record changes. Though SCDs can go up to seven levels, let's look at the three most common ones:

Type 1

Overwrite existing dimension records. This is super simple and means you have no access to the deleted historical dimension records.

Type 2

Keep a full history of dimension records. When a record changes, that specific record is flagged as changed, and a new dimension record is created that reflects the current status of the attributes. In our example, Housley moved to a new zip code, which triggered his initial record to reflect an effective end date, and a new record was created to show his new zip code.

Type 3

A Type 3 SCD is similar to a Type 2 SCD, but instead of creating a new row, a change in a Type 3 SCD creates a new field. Using the preceding example, let's see what this looks like as a Type 3 SCD in the following tables.

In Table 8-12, Housley lives in the 84101 zip code. When Housley moves to a new zip code, the Type 3 SCD creates two new fields, one for his new zip code and the date of the change (Table 8-13). The original zip code field is also renamed to reflect that this is the older record.

Table 8-12. Type 3 slowly changing dimension

CustomerKey	FirstName	LastName	ZipCode
7	Matt	Housley	84101

Table 8-13. Type 3 customer dimension table

CustomerKey	FirstName	LastName	Original ZipCode	Current ZipCode	CurrentDate
7	Matt	Housley	84101	84123	2021-09-19

Of the types of SCDs described, Type 1 is the default behavior of most data warehouses, and Type 2 is the one we most commonly see used in practice. There's a lot to know about dimensions, and we suggest using this section as a starting point to get familiar with how dimensions work and how they're used.

Star schema

Now that you have a basic understanding of facts and dimensions, it's time to integrate them into a star schema. The *star schema* represents the data model of the business. Unlike highly normalized approaches to data modeling, the star schema is a fact table surrounded by the necessary dimensions. This results in fewer joins than other data models, which speeds up query performance. Another advantage of a star schema is it's arguably easier for business users to understand and use.

Note that the star schema shouldn't reflect a particular report, though you can model a report in a downstream data mart or directly in your BI tool. The star schema should capture the facts and attributes of your *business logic* and be flexible enough to answer the respective critical questions.

Because a star schema has one fact table, sometimes you'll have multiple star schemas that address different facts of the business. You should strive to reduce the number of dimensions whenever possible since this reference data can potentially be reused among different fact tables. A dimension that is reused across multiple star schemas, thus sharing the same fields, is called a *conformed dimension*. A conformed dimension allows you to combine multiple fact tables across multiple star schemas. Remember, redundant data is OK with the Kimball method, but avoid replicating the same dimension tables to avoid drifting business definitions and data integrity.

The Kimball data model and star schema have a lot of nuance. You should be aware that this mode is appropriate only for batch data and not for streaming data. Because the Kimball data model is popular, there's a good chance you'll run into it.

Data Vault

Whereas Kimball and Inmon focus on the structure of business logic in the data warehouse, the *Data Vault* offers a different approach to data modeling.[11] Created in the 1990s by Dan Linstedt, the Data Vault methodology separates the structural aspects of a source system's data from its attributes. Instead of representing business logic in facts, dimensions, or highly normalized tables, a Data Vault simply loads data from source systems directly into a handful of purpose-built tables in an insert-only manner. Unlike the other data modeling approaches you've learned about, there's no notion of good, bad, or conformed data in a Data Vault.

Data moves fast these days, and data models need to be agile, flexible, and scalable; the Data Vault methodology aims to meet this need. The goal of this methodology is to keep the data as closely aligned to the business as possible, even while the business's data evolves.

A Data Vault model consists of three main types of tables: hubs, links, and satellites (Figure 8-15). In short, a *hub* stores business keys, a *link* maintains relationships among business keys, and a *satellite* represents a business key's attributes and context. A user will query a hub, which will link to a satellite table containing the query's relevant attributes. Let's explore hubs, links, and satellites in more detail.

Figure 8-15. Data Vault tables: hubs, links, and satellites connected together

Hubs

Queries often involve searching by a business key, such as a customer ID or an order ID from our ecommerce example. A hub is the central entity of a Data Vault that retains a record of all unique business keys loaded into the Data Vault.

A hub always contains the following standard fields:

Hash key

: The primary key used to join data between systems. This is a calculated hash field (MD5 or similar).

Load date

: The date the data was loaded into the hub.

Record source

: The source from which the unique record was obtained.

Business key(s)

: The key used to identify a unique record.

It's important to note that a hub is insert-only, and data is not altered in a hub. Once data is loaded into a hub, it's permanent.

When designing a hub, identifying the business key is critical. Ask yourself: What is the *identifiable business element*?[12] Put another way, how do users commonly look for data? Ideally, this is discovered as you build the conceptual data model of your organization and before you start building your Data Vault.

Using our ecommerce scenario, let's look at an example of a hub for products. First, let's look at the physical design of a product hub (Table 8-14).

Table 8-14. A physical design for a product hub

HubProduct
ProductHashKey
LoadDate
RecordSource
ProductID

In practice, the product hub looks like this when populated with data (Table 8-15). In this example, three different products are loaded into a hub from an ERP system on two separate dates.

Table 8-15. A product hub populated with data

ProductHashKey	**LoadDate**	**RecordSource**	**ProductID**
4041fd80ab...	2020-01-02	ERP	1
de8435530d...	2021-03-09	ERP	2
cf27369bd8...	2021-03-09	ERP	3

While we're at it, let's create another hub for orders (Table 8-16) using the same schema as `HubProduct`, and populate it with some sample order data.

Table 8-16. An order hub populated with data

OrderHashKey	**LoadDate**	**RecordSource**	**OrderID**
f899139df5...	2022-03-01	Website	100
38b3eff8ba...	2022-03-01	Website	101
ec8956637a...	2022-03-01	Website	102

Links

A *link table* tracks the relationships of business keys between hubs. Link tables connect hubs, ideally at the lowest possible grain. Because link tables connect data from various hubs, they are many to many. The Data Vault model's relationships are straightforward and handled through changes to the links. This provides excellent flexibility in the inevitable event that the underlying data changes. You simply create a new link that ties business concepts (or hubs) to represent the new relationship. That's it! Now let's look at ways to view data contextually using satellites.

Back to our ecommerce example, we'd like to associate orders with products. Let's see what a link table might look like for orders and products (Table 8-17).

Table 8-17. A link table for products and orders

LinkOrderProduct
OrderProductHashKey
LoadDate
RecordSource
ProductHashKey
OrderHashKey

When the `LinkOrderProduct` table is populated, here's what it looks like (Table 8-18). Note that we're using the order's record source in this example.

Table 8-18. A link table connecting orders and products

OrderProductHashKey	LoadDate	RecordSource	ProductHashKey	OrderHashKey
ff64ec193d...	2022-03-01	Website	4041fd80ab...	f899139df5...
ff64ec193d...	2022-03-01	Website	de8435530d...	f899139df5...
e232628c25...	2022-03-01	Website	cf27369bd8...	38b3eff8ba...
26166a5871...	2022-03-01	Website	4041fd80ab...	ec8956637a...

Satellites

We've described relationships between hubs and links that involve keys, load dates, and record sources. How do you get a sense of what these relationships mean? *Satellites* are descriptive attributes that give meaning and context to hubs. Satellites can connect to either hubs or links. The only required fields in a satellite are a primary key consisting of the business key of the parent hub and a load date. Beyond that, a satellite can contain however many attributes that make sense.

Let's look at an example of a satellite for the `Product` hub (Table 8-19). In this example, the `SatelliteProduct` table contains additional information about the product, such as product name and price.

Table 8-19.

SatelliteProduct
SatelliteProduct
ProductHashKey
LoadDate
RecordSource
ProductName
Price

And here's the `SatelliteProduct` table with some sample data (Table 8-20).

Table 8-20. A product satellite table with sample data

ProductHashKey	LoadDate	RecordSource	ProductName	Price
4041fd80ab...	2020-01-02	ERP	Thingamajig	50
de8435530d...	2021-03-09	ERP	Whatchamacallit	25
cf27369bd8...	2021-03-09	ERP	Whozeewhatzit	75

Let's tie this all together and join the hub, product, and link tables into a Data Vault (Figure 8-16).

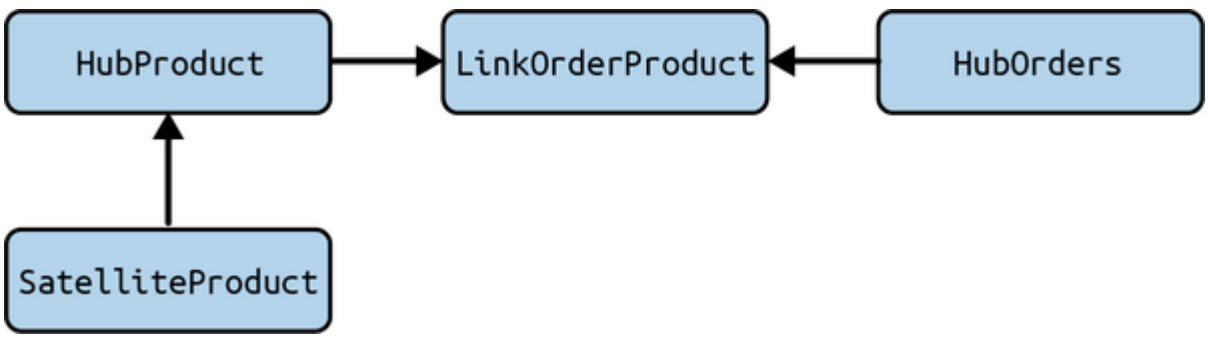

Figure 8-16. The Data Vault for orders and products

Other types of Data Vault tables exist, including point-in-time (PIT) and bridge tables. We don't cover these here, but mention them because the Data Vault is quite comprehensive. Our goal is to simply give you an overview of the Data Vault's power.

Unlike other data modeling techniques we've discussed, in a Data Vault, the business logic is created and interpreted when the data from these tables is queried. Please be aware that the Data Vault model can be used with other data modeling techniques. It's not unusual for a Data Vault to be the landing zone for analytical data, after which it's separately modeled in a data warehouse, commonly using a star schema. The Data Vault model also can be adapted for NoSQL and streaming data sources. The Data Vault is a huge topic, and this section is simply meant to make you aware of its existence.

Wide denormalized tables

The strict modeling approaches we've described, especially Kimball and Inmon, were developed when data warehouses were expensive, on premises, and heavily resource-constrained with tightly coupled compute and storage. While batch data modeling has traditionally been associated with these strict approaches, more relaxed approaches are becoming more common.

There are reasons for this. First, the popularity of the cloud means that storage is dirt cheap. It's cheaper to store data than agonize over the optimum way to represent the data in storage. Second, the popularity of nested data (JSON and similar) means schemas are flexible in source and analytical systems.

You have the option to rigidly model your data as we've described, or you can choose to throw all of your data into a single wide table. A *wide table* is just what it sounds like: a highly denormalized and very wide collection of many fields, typically created in a columnar database. A field may be a single value or contain nested data. The data is organized along with one or multiple keys; these keys are closely tied to the *grain* of the data.

A wide table can potentially have thousands of columns, whereas fewer than 100 are typical in relational databases. Wide tables are usually sparse; the vast majority of entries in a given field may be null. This is extremely expensive in a traditional relational database because the database allocates a fixed amount of space for each field entry; nulls take up virtually no space in a columnar database. A wide schema in a relational database dramatically slows reading because each row must allocate all the space specified by the wide schema, and the database must read the contents of each row in its entirety. On the other hand, a columnar database reads only columns selected in a query, and reading nulls is essentially free.

Wide tables generally arise through schema evolution; engineers gradually add fields over time. Schema evolution in a relational database is a slow and resource-heavy process. In a columnar database, adding a field is initially just a change to metadata. As data is written into the new field, new files are added to the column.

Analytics queries on wide tables often run faster than equivalent queries on highly normalized data requiring many joins. Removing joins can have a huge impact on scan performance. The wide table simply contains all of the data you would have joined in a more rigorous modeling approach. Facts and dimensions are represented in the same table. The lack of data model rigor also means not a lot of thought is involved. Load your data into a wide table and start querying it. Especially with schemas in source systems becoming more adaptive and flexible, this data usually results from high-volume transactions, meaning there's a lot of data. Storing this as nested data in your analytical storage has a lot of benefits.

Throwing all of your data into a single table might seem like heresy for a hardcore data modeler, and we've seen plenty of criticism. What are some of these criticisms? The biggest criticism is as you blend your data, you lose the business logic in your analytics. Another downside is the performance of updates to things like an element in an array, which can be very painful.

Let's look at an example of a wide table (Table 8-21), using the original denormalized table from our earlier normalization example. This table can have many more columns—hundreds or more!—and we include only a handful of columns for brevity and ease of understanding. As you can see, this table combines various data types, represented along a grain of orders for a customer on a date.

We suggest using a wide table when you don't care about data modeling, or when you have a lot of data that needs more flexibility than traditional data-modeling rigor provides. Wide tables also lend themselves to streaming data, which we'll discuss next. As data moves toward fast-moving schemas and streaming-first, we expect to see a new wave of data modeling, perhaps something along the lines of "relaxed normalization."

Table 8-21. An example of denormalized data

OrderID	OrderItems	CustomerID	CustomerName	OrderDate	Site	SiteRegion
100	[{ "sku": 1, "price": 50, "quantity": 1, "name:": "Thingamajig" }, { "sku": 2, "price": 25, "quantity": 2, "name:": "Whatchamacallit" }]	5	Joe Reis	2022-03-01	abc.com	US

What If You Don't Model Your Data?

You also have the option of *not* modeling your data. In this case, just query data sources directly. This pattern is often used, especially when companies are just getting started and want to get quick insights or share analytics with their users. While it allows you to get answers to various questions, you should consider the following:

- If I don't model my data, how do I know the results of my queries are consistent?
- Do I have proper definitions of business logic in the source system, and will my query produce truthful answers?
- What query load am I putting on my source systems, and how does this impact users of these systems?

At some point, you'll probably gravitate toward a stricter batch data model paradigm and a dedicated data architecture that doesn't rely on the source systems for the heavy lifting.

Modeling Streaming Data

Whereas many data-modeling techniques are well established for batch, this is not the case for streaming data. Because of the unbounded and continuous nature of streaming data, translating batch techniques like Kimball to a streaming paradigm is tricky, if not impossible. For example, given a stream of data, how would you continuously update a Type-2 slowly changing dimension without bringing your data warehouse to its knees?

The world is evolving from batch to streaming and from on premises to the cloud. The constraints of the older batch methods no longer apply. That said, big questions remain about how to model data to balance the need for business logic against fluid schema changes, fast-moving data, and self-service. What is the streaming equivalent of the preceding batch data model approaches? There isn't (yet) a consensus approach on streaming data modeling. We spoke with many experts in streaming data systems, many of whom told us that traditional batch-oriented data modeling doesn't apply to streaming. A few suggested the Data Vault as an option for streaming data modeling.

As you may recall, two main types of streams exist: event streams and CDC. Most of the time, the shape of the data in these streams is semistructured, such as JSON. The challenge with modeling streaming data is that the payload's schema might change on a whim. For example, suppose you have

an IoT device that recently upgraded its firmware and introduced a new field. In that case, it's possible that your downstream destination data warehouse or processing pipeline isn't aware of this change and breaks. That's not great. As another example, a CDC system might recast a field as a different type—say, a string instead of an International Organization for Standardization (ISO) datetime format. Again, how does the destination handle this seemingly random change?

The streaming data experts we've talked with overwhelmingly suggest you anticipate changes in the source data and keep a flexible schema. This means there's no rigid data model in the analytical database. Instead, assume the source systems are providing the correct data with the right business definition and logic, as it exists today. And because storage is cheap, store the recent streaming and saved historical data in a way they can be queried together. Optimize for comprehensive analytics against a dataset with a flexible schema. Furthermore, instead of reacting to reports, why not create automation that responds to anomalies and changes in the streaming data instead?

The world of data modeling is changing, and we believe a sea change will soon occur in data model paradigms. These new approaches will likely incorporate metrics and semantic layers, data pipelines, and traditional analytics workflows in a streaming layer that sits directly on top of the source system. Since data is being generated in real time, the notion of artificially separating source and analytics systems into two distinct buckets may not make as much sense as when data moved more slowly and predictably. Time will tell…

We have more to say on the future of streaming data in Chapter 11.

Transformations

> The net result of transforming data is the ability to unify and integrate data. Once data is transformed, the data can be viewed as a single entity. But without transforming data, you cannot have a unified view of data across the organization.
>
> Bill Inmon[13]

Now that we've covered queries and data modeling, you might be wondering, if I can model data, query it, and get results, why do I need to think about transformations? Transformations manipulate, enhance, and save data for downstream use, increasing its value in a scalable, reliable, and cost-effective manner.

Imagine running a query every time you want to view results from a particular dataset. You'd run the same query dozens or hundreds of times a day. Imagine that this query involves parsing, cleansing, joining, unioning, and aggregating across 20 datasets. To further exacerbate the pain, the query takes 30 minutes to run, consumes significant resources, and incurs substantial cloud charges over several repetitions. You and your stakeholders would probably go insane. Thankfully, you can *save the results of your query* instead, or at least run the most compute-intensive portions only once, so subsequent queries are simplified.

A transformation differs from a query. A *query* retrieves the data from various sources based on filtering and join logic. A *transformation* persists the results for consumption by additional transformations or queries. These results may be stored ephemerally or permanently.

Besides persistence, a second aspect that differentiates transformations from queries is complexity. You'll likely build complex pipelines that combine data from multiple sources and reuse intermediate results for multiple final outputs. These complex pipelines might normalize, model, aggregate, or featurize data. While you can build complex dataflows in single queries using common table expressions, scripts, or DAGs, this quickly becomes unwieldy, inconsistent, and intractable. Enter transformations.

Transformations critically rely on one of the major undercurrents in this book: orchestration. Orchestration combines many discrete operations, such as intermediate transformations, that store data

temporarily or permanently for consumption by downstream transformations or serving. Increasingly, transformation pipelines span not only multiple tables and datasets but also multiple systems.

Batch Transformations

Batch transformations run on discrete chunks of data, in contrast to streaming transformations, where data is processed continuously as it arrives. Batch transformations can run on a fixed schedule (e.g., daily, hourly, or every 15 minutes) to support ongoing reporting, analytics, and ML models. In this section, you'll learn various batch transformation patterns and technologies.

Distributed joins

The basic idea behind distributed joins is that we need to break a *logical join* (the join defined by the query logic) into much smaller *node joins* that run on individual servers in the cluster. The basic distributed join patterns apply whether one is in MapReduce (discussed in "MapReduce"), BigQuery, Snowflake, or Spark, though the details of intermediate storage between processing steps vary (on disk or in memory). In the best-case scenario, the data on one side of the join is small enough to fit on a single node (*broadcast join*). Often, a more resource-intensive *shuffle hash join* is required.

Broadcast join

A *broadcast join* is generally asymmetric, with one large table distributed across nodes and one small table that can easily fit on a single node (Figure 8-17). The query engine "broadcasts" the small table (table A) out to all nodes, where it gets joined to the parts of the large table (table B). Broadcast joins are far less compute intensive than shuffle hash joins.

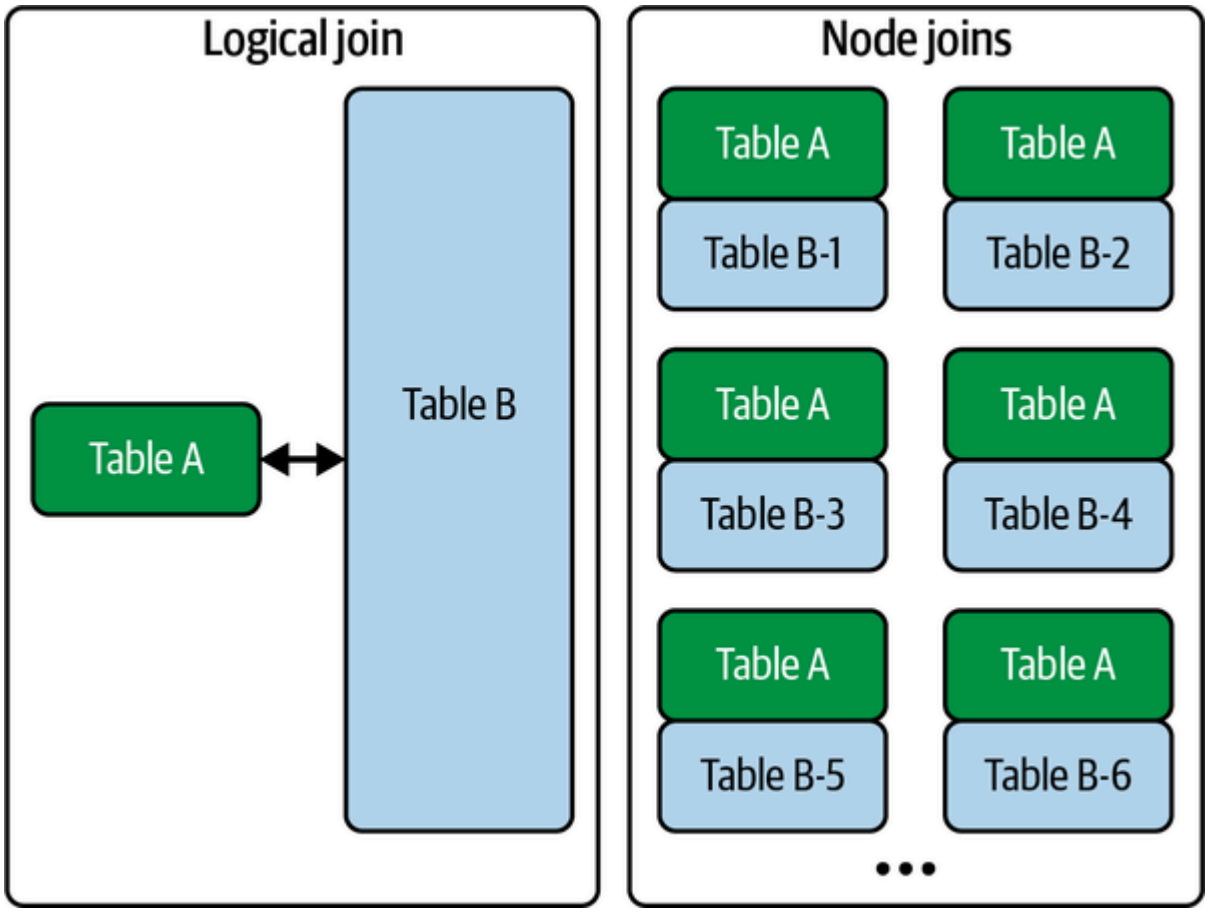

Figure 8-17. In a broadcast join, the query engine sends table A out to all nodes in the cluster to be joined with the various parts of table B

In practice, table A is often a down-filtered larger table that the query engine collects and broadcasts. One of the top priorities in query optimizers is join reordering. With the early application of filters, and movement of small tables to the left (for left joins), it is often possible to dramatically reduce the amount of data that is processed in each join. Prefiltering data to create broadcast joins where possible can dramatically improve performance and reduce resource consumption.

Shuffle hash join

If neither table is small enough to fit on a single node, the query engine will use a *shuffle hash join*. In Figure 8-18, the same nodes are represented above and below the dotted line. The area above the dotted line represents the initial partitioning of tables A and B across the nodes. In general, this partitioning will have no relation to the join key. A hashing scheme is used to repartition data by join key.

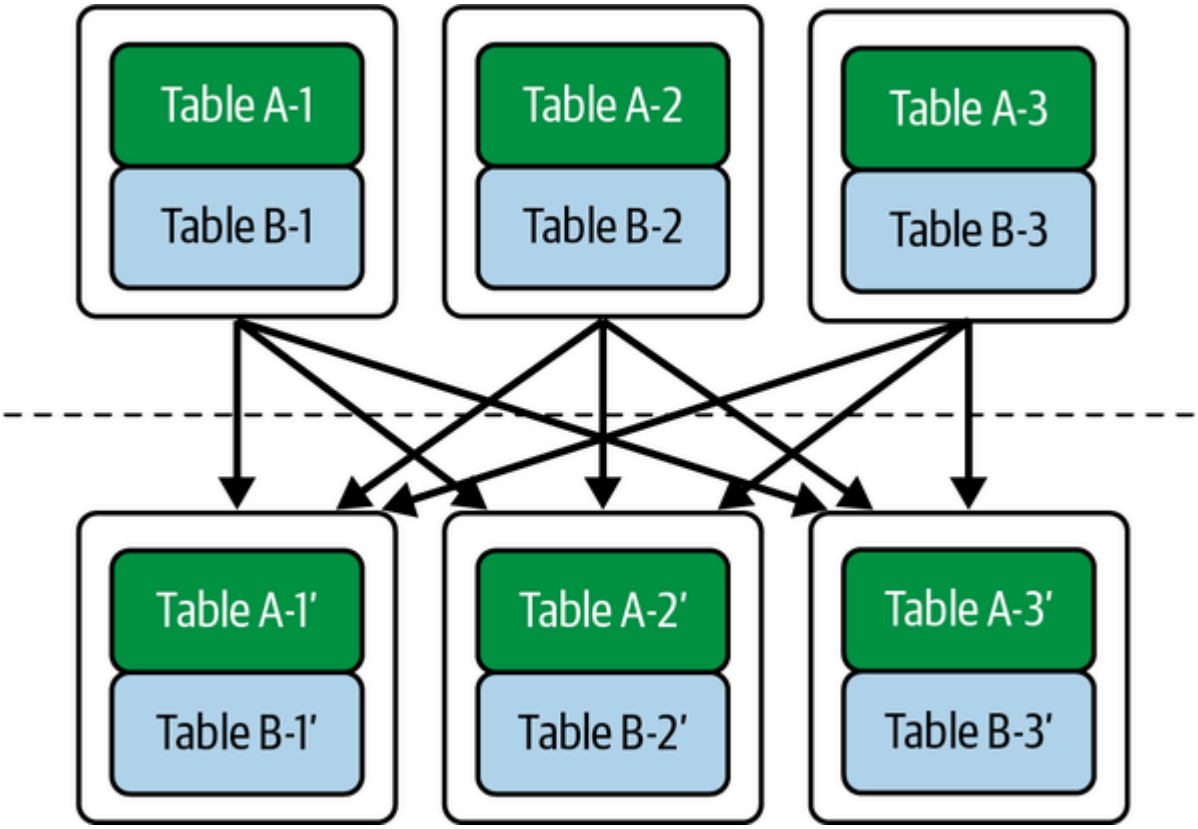

Figure 8-18. Shuffle hash join

In this example, the hashing scheme will partition the join key into three parts, with each part assigned to a node. The data is then reshuffled to the appropriate node, and the new partitions for tables A and B on each node are joined. Shuffle hash joins are generally more resource intensive than broadcast joins.

ETL, ELT, and data pipelines

As we discussed in Chapter 3, a widespread transformation pattern dating to the early days of relational databases is a batch ETL. Traditional ETL relies on an external transformation system to pull, transform, and clean data while preparing it for a target schema, such as a data mart or a Kimball star schema. The transformed data would then be loaded into a target system, such as a data warehouse, where business analytics could be performed.

The ETL pattern itself was driven by the limitations of both source and target systems. The extract phase tended to be a major bottleneck, with the constraints of the source RDBMS limiting the rate at which data could be pulled. And, the transformation was handled in a dedicated system because the target system was extremely resource-constrained in both storage and CPU capacity.

A now-popular evolution of ETL is ELT. As data warehouse systems have grown in performance and storage capacity, it has become common to simply extract raw data from a source system, import it into a data warehouse with minimal transformation, and then clean and transform it directly in the warehouse system. (See our discussion of data warehouses in Chapter 3 for a more detailed discussion of the difference between ETL and ELT.)

A second, slightly different notion of ELT was popularized with the emergence of data lakes. In this version, the data is not transformed at the time it's loaded. Indeed, massive quantities of data may be loaded with no preparation and no plan whatsoever. The assumption is that the transformation step will happen at some undetermined future time. Ingesting data without a plan is a great recipe for a data swamp. As Inmon says:[14]

> I've always been a fan of ETL because of the fact that ETL forces you to transform data before you put it into a form where you can work with it. But some organizations want to simply take the data, put it into a database, then do the transformation.... I've seen too many cases where the organization says, oh we'll just put the data in and transform it later. And guess what? Six months later, that data [has] never been touched.

We have also seen that the line between ETL and ELT can become somewhat blurry in a data lakehouse environment. With object storage as a base layer, it's no longer clear what's in the database and out of the database. The ambiguity is further exacerbated with the emergence of data federation, virtualization, and live tables. (We discuss these topics later in this section.)

Increasingly, we feel that the terms *ETL* and *ELT* should be applied only at the micro level (within individual transformation pipelines) rather than at the macro level (to describe a transformation pattern for a whole organization). Organizations no longer need to standardize on ETL or ELT but can instead focus on applying the proper technique on a case-by-case basis as they build data pipelines.

SQL and code-based transformation tools

At this juncture, the distinction between SQL-based and non-SQL-based transformation systems feels somewhat synthetic. Since the introduction of Hive on the Hadoop platform, SQL has become a first-class citizen in the big data ecosystem. For example, Spark SQL was an early feature of Apache Spark. Streaming-first frameworks such as Kafka, Flink, and Beam also support SQL, with varying features and functionality.

It is more appropriate to think about SQL-only tools versus those that support more powerful, general-purpose programming paradigms. SQL-only transformation tools span a wide variety of proprietary and open source options.

SQL is declarative...but it can still build complex data workflows

We often hear SQL dismissed because it is "not procedural." This is technically correct. SQL is a declarative language: instead of coding a data processing procedure, SQL writers stipulate the characteristics of their final data in set-theoretic language; the SQL compiler and optimizer determine the steps required to put data in this state.

People sometimes imply that because SQL is not procedural, it cannot build out complex pipelines. This is false. SQL can effectively be used to build complex DAGs using common table expressions, SQL scripts, or an orchestration tool.

To be clear, SQL has limits, but we often see engineers doing things in Python and Spark that could be more easily and efficiently done in SQL. For a better idea of the trade-offs we're talking about, let's look at a couple of examples of Spark and SQL.

Example: When to avoid SQL for batch transformations in Spark

When you're determining whether to use native Spark or PySpark code instead of Spark SQL or another SQL engine, ask yourself the following questions:

1. How difficult is it to code the transformation in SQL?
2. How readable and maintainable will the resulting SQL code be?
3. Should some of the transformation code be pushed into a custom library for future reuse across the organization?

Regarding question 1, many transformations coded in Spark could be realized in fairly simple SQL statements. On the other hand, if the transformation is not realizable in SQL, or if it would be extremely awkward to implement, native Spark is a better option. For example, we might be able to implement word stemming in SQL by placing word suffixes in a table, joining with that table, using a parsing function to find suffixes in words, and then reducing the word to its stem by using a substring function. However, this sounds like an extremely complex process with numerous edge cases to consider. A more powerful procedural programming language is a better fit here.

Question 2 is closely related. The word-stemming query will be neither readable nor maintainable.

Regarding question 3, one of the major limitations of SQL is that it doesn't include a natural notion of libraries or reusable code. One exception is that some SQL engines allow you to maintain user-defined functions (UDFs) as objects inside a database.[15] However, these aren't committed to a Git repository without an external CI/CD system to manage deployment. Furthermore, SQL doesn't have a good notion of reusability for more complex query components. Of course, reusable libraries are easy to create in Spark and PySpark.

We will add that it is possible to recycle SQL in two ways. First, we can easily reuse the *results* of a SQL query by committing to a table or creating a view. This process is often best handled in an orchestration tool such as Airflow so that downstream queries can start once the source query has finished. Second, Data Build Tool (dbt) facilitates the reuse of SQL statements and offers a templating language that makes customization easier.

Example: Optimizing Spark and other processing frameworks

Spark acolytes often complain that SQL doesn't give them control over data processing. The SQL engine takes your statements, optimizes them, and compiles them into its processing steps. (In practice, optimization may happen before or after compilation, or both.)

This is a fair complaint, but a corollary exists. With Spark and other code-heavy processing frameworks, the code writer becomes responsible for much of the optimization that is handled automatically in a SQL-based engine. The Spark API is powerful and complex, meaning it is not so easy to identify candidates for reordering, combination, or decomposition. When embracing Spark, data engineering teams need to actively engage with the problems of Spark optimization, especially for expensive, long-running jobs. This means building optimization expertise on the team and teaching individual engineers how to optimize.

A few top-level things to keep in mind when coding in native Spark:

1. Filter early and often.
2. Rely heavily on the core Spark API, and learn to understand the Spark native way of doing things. Try to rely on well-maintained public libraries if the native Spark API doesn't support your use case. Good Spark code is substantially declarative.
3. Be careful with UDFs.
4. Consider intermixing SQL.

Recommendation 1 applies to SQL optimization as well, with the difference being that Spark may not be able to reorder something that SQL would handle for you automatically. Spark is a big data processing framework, but the less data you have to process, the less resource-heavy and more performant your code will be.

If you find yourself writing extremely complex custom code, pause and determine whether there's a more native way of doing whatever you're trying to accomplish. Learn to understand idiomatic Spark by reading public examples and working through tutorials. Is there something in the Spark API that can accomplish what you're trying to do? Is there a well-maintained and optimized public library that can help?

The third recommendation is crucial for PySpark. In general, PySpark is an API wrapper for Scala Spark. Your code pushes work into native Scala code running in the JVM by calling the API. Running Python UDFs forces data to be passed to Python, where processing is less efficient. If you find yourself using Python UDFs, look for a more Spark-native way to accomplish what you're doing. Go back to the recommendation: is there a way to accomplish your task by using the core API or a well-maintained library? If you must use UDFs, consider rewriting them in Scala or Java to improve performance.

As for recommendation 4, using SQL allows us to take advantage of the Spark Catalyst optimizer, which may be able to squeeze out more performance than we can with native Spark code. SQL is often easier to write and maintain for simple operations. Combining native Spark and SQL lets us realize the best of both worlds—powerful, general-purpose functionality combined with simplicity where applicable.

Much of the optimization advice in this section is fairly generic and would apply just as well to Apache Beam, for example. The main point is that programmable data processing APIs require a bit more optimization finesse than SQL, which is perhaps less powerful and easier to use.

Update patterns

Since transformations persist data, we will often update persisted data in place. Updating data is a major pain point for data engineering teams, especially as they transition between data engineering technologies. We're discussing DML in SQL, which we introduced earlier in the chapter.

We've mentioned several times throughout the book that the original data lake concept didn't really account for updating data. This now seems nonsensical for several reasons. Updating data has long been a key part of handling data transformation results, even though the big data community dismissed it. It is silly to rerun significant amounts of work because we have no update capabilities. Thus, the data lakehouse concept now builds in updates. Also, GDPR and other data deletion standards now *require* organizations to delete data in a targeted fashion, even in raw datasets.

Let's consider several basic update patterns.

Truncate and reload

Truncate is an update pattern that doesn't update anything. It simply wipes the old data. In a truncate-and-reload update pattern, a table is cleared of data, and transformations are rerun and loaded into this table, effectively generating a new table version.

Insert only

Insert only inserts new records without changing or deleting old records. Insert-only patterns can be used to maintain a current view of data—for example, if new versions of records are inserted without deleting old records. A query or view can present the current data state by finding the newest record by primary key. Note that columnar databases don't typically enforce primary keys. The primary key would be a construct used by engineers to maintain a notion of the current state of the table. The

downside to this approach is that it can be extremely computationally expensive to find the latest record at query time. Alternatively, we can use a materialized view (covered later in the chapter), an insert-only table that maintains all records, and a truncate-and-reload target table that holds the current state for serving data.

Warning

When inserting data into a column-oriented OLAP database, the common problem is that engineers transitioning from row-oriented systems attempt to use single-row inserts. This antipattern puts a massive load on the system. It also causes data to be written in many separate files; this is extremely inefficient for subsequent reads, and the data must be reclustered later. Instead, we recommend loading data in a periodic micro-batch or batch fashion.

We'll mention an exception to the advice not to insert frequently: the enhanced Lambda architecture used by BigQuery and Apache Druid, which hybridizes a streaming buffer with columnar storage. Deletes and in-place updates can still be expensive, as we'll discuss next.

Delete

Deletion is critical when a source system deletes data and satisfies recent regulatory changes. In columnar systems and data lakes, deletes are more expensive than inserts.

When deleting data, consider whether you need to do a hard or soft delete. A *hard delete* permanently removes a record from a database, while a *soft delete* marks the record as "deleted." Hard deletes are useful when you need to remove data for performance reasons (say, a table is too big), or if there's a legal or compliance reason to do so. Soft deletes might be used when you don't want to delete a record permanently but also want to filter it out of query results.

A third approach to deletes is closely related to soft deletes: *insert deletion* inserts a new record with a `deleted` flag without modifying the previous version of the record. This allows us to follow an insert-only pattern but still account for deletions. Just note that our query to get the latest table state gets a little more complicated. We must now deduplicate, find the latest version of each record by key, and not show any record whose latest version shows `deleted`.

Upsert/merge

Of these update patterns, the upsert and merge patterns are the ones that consistently cause the most trouble for data engineering teams, especially for people transitioning from row-based data warehouses to column-based cloud systems.

Upserting takes a set of source records and looks for matches against a target table by using a primary key or another logical condition. (Again, it's the responsibility of the data engineering team to manage this primary key by running appropriate queries. Most columnar systems will not enforce uniqueness.) When a key match occurs, the target record gets updated (replaced by the new record). When no match exists, the database inserts the new record. The merge pattern adds to this the ability to delete records.

So, what's the problem? The upsert/merge pattern was originally designed for row-based databases. In row-based databases, updates are a natural process: the database looks up the record in question and changes it in place.

On the other hand, file-based systems don't actually support in-place file updates. All of these systems utilize copy on write (COW). If one record in a file is changed or deleted, the whole file must be rewritten with the new changes.

This is part of the reason that early adopters of big data and data lakes rejected updates: managing files and updates seemed too complicated. So they simply used an insert-only pattern and assumed that data

consumers would determine the current state of the data at query time or in downstream transformations. In reality, columnar databases such as Vertica have long supported in-place updates by hiding the complexity of COW from users. They scan files, change the relevant records, write new files, and change file pointers for the table. The major columnar cloud data warehouses support updates and merges, although engineers should investigate update support if they consider adopting an exotic technology.

There are a few key things to understand here. Even though distributed columnar data systems support native update commands, merges come at a cost: the performance impact of updating or deleting a single record can be quite high. On the other hand, merges can be extremely performant for large update sets and may even outperform transactional databases.

In addition, it is important to understand that COW seldom entails rewriting the whole table. Depending on the database system in question, COW can operate at various resolutions (partition, cluster, block). To realize performant updates, focus on developing an appropriate partitioning and clustering strategy based on your needs and the innards of the database in question.

As with inserts, be careful with your update or merge frequency. We've seen many engineering teams transition between database systems and try to run near real-time merges from CDC just as they did on their old system. It simply doesn't work. No matter how good your CDC system is, this approach will bring most columnar data warehouses to their knees. We've seen systems fall weeks behind on updates, where an approach that simply merged every hour would make much more sense.

We can use various approaches to bring columnar databases closer to real time. For example, BigQuery allows us to stream insert new records into a table, and then supports specialized materialized views that present an efficient, near real-time deduplicated table view. Druid uses two-tier storage and SSDs to support ultrafast real-time queries.

Schema updates

Data has entropy and may change without your control or consent. External data sources may change their schema, or application development teams may add new fields to the schema. One advantage of columnar systems over row-based systems is that while updating the data is more difficult, updating the schema is easier. Columns can typically be added, deleted, and renamed.

In spite of these technological improvements, practical organizational schema management is more challenging. Will some schema updates be automated? (This is the approach that Fivetran uses when replicating from sources.) As convenient as this sounds, there's a risk that downstream transformations will break.

Is there a straightforward schema update request process? Suppose a data science team wants to add a column from a source that wasn't previously ingested. What will the review process look like? Will downstream processes break? (Are there queries that run `SELECT *` rather than using explicit column selection? This is generally bad practice in columnar databases.) How long will it take to implement the change? Is it possible to create a table fork—i.e., a new table version specific to this project?

A new interesting option has emerged for semistructured data. Borrowing an idea from document stores, many cloud data warehouses now support data types that encode arbitrary JSON data. One approach stores raw JSON in a field while storing frequently accessed data in adjacent flattened fields. This takes up additional storage space but allows for the convenience of flattened data, with the flexibility of semistructured data for advanced users. Frequently accessed data in the JSON field can be added directly into the schema over time.

This approach works extremely well when data engineers must ingest data from an application document store with a frequently changing schema. Semistructured data available as a first-class citizen in data warehouses is extremely flexible and opens new opportunities for data analysts and data scientists since data is no longer constrained to rows and columns.

Data wrangling

Data wrangling takes messy, malformed data and turns it into useful, clean data. This is generally a batch transformation process.

Data wrangling has long been a major source of pain and job security for data engineers. For example, suppose that developers receive EDI data (see Chapter 7) from a partner business regarding transactions and invoices, potentially a mix of structured data and text. The typical process of wrangling this data involves first trying to ingest it. Often, the data is so malformed that a good deal of text preprocessing is involved. Developers may choose to ingest the data as a single text field table—an entire row ingested as a single field. Developers then begin writing queries to parse and break apart the data. Over time, they discover data anomalies and edge cases. Eventually, they will get the data into rough shape. Only then can the process of downstream transformation begin.

Data wrangling tools aim to simplify significant parts of this process. These tools often put off data engineers because they claim to be no code, which sounds unsophisticated. We prefer to think of data wrangling tools as integrated development environments (IDEs) for malformed data. In practice, data engineers spend way too much time parsing nasty data; automation tools allow data engineers to spend time on more interesting tasks. Wrangling tools may also allow engineers to hand some parsing and ingestion work off to analysts.

Graphical data-wrangling tools typically present a sample of data in a visual interface, with inferred types, statistics including distributions, anomalous data, outliers, and nulls. Users can then add processing steps to fix data issues. A step might provide instructions for dealing with mistyped data, splitting a text field into multiple parts, or joining with a lookup table.

Users can run the steps on a full dataset when the full job is ready. The job typically gets pushed to a scalable data processing system such as Spark for large datasets. After the job runs, it will return errors and unhandled exceptions. The user can further refine the recipe to deal with these outliers.

We highly recommend that both aspiring and seasoned engineers experiment with wrangling tools; major cloud providers sell their version of data-wrangling tools, and many third-party options are available. Data engineers may find that these tools significantly streamline certain parts of their jobs. Organizationally, data engineering teams may want to consider training specialists in data wrangling if they frequently ingest from new, messy data sources.

Example: Data transformation in Spark

Let’s look at a practical, concrete example of data transformation. Suppose we build a pipeline that ingests data from three API sources in JSON format. This initial ingestion step is handled in Airflow. Each data source gets its prefix (filepath) in an S3 bucket.

Airflow then triggers a Spark job by calling an API. This Spark job ingests each of the three sources into a dataframe, converting the data into a relational format, with nesting in certain columns. The Spark job combines the three sources into a single table and then filters the results with a SQL statement. The results are finally written out to a Parquet-formatted Delta Lake table stored in S3.

In practice, Spark creates a DAG of steps based on the code that we write for ingesting, joining, and writing out the data. The basic ingestion of data happens in cluster memory, although one of the data sources is large enough that it must spill to disk during the ingestion process. (This data gets written to cluster storage; it will be reloaded into memory for subsequent processing steps.)

The join requires a shuffle operation. A key is used to redistribute data across the cluster; once again, a spill to disk occurs as the data is written to each node. The SQL transformation filters through the rows in memory and discards the unused rows. Finally, Spark converts the data into Parquet format, compresses it, and writes it back to S3. Airflow periodically calls back to Spark to see if the job is completed. Once it confirms that the job has finished, it marks the full Airflow DAG as completed. (Note that we have two DAG constructs here, an Airflow DAG and a DAG specific to the Spark job.)

Business logic and derived data

One of the most common use cases for transformation is to render business logic. We've placed this discussion under batch transformations because this is where this type of transformation happens most frequently, but note that it could also happen in a streaming pipeline.

Suppose that a company uses multiple specialized internal profit calculations. One version might look at profits before marketing costs, and another might look at a profit after subtracting marketing costs. Even though this appears to be a straightforward accounting exercise, each of these metrics is highly complex to render.

Profit before marketing costs might need to account for fraudulent orders; determining a reasonable profit estimate for the previous business day entails estimating what percentage of revenue and profit will ultimately be lost to orders canceled in the coming days as the fraud team investigates suspicious orders. Is there a special flag in the database that indicates an order with a high probability of fraud, or one that has been automatically canceled? Does the business assume that a certain percentage of orders will be canceled because of fraud even before the fraud-risk evaluation process has been completed for specific orders?

For profits after marketing costs, we must account for all the complexities of the previous metric, plus the marketing costs attributed to the specific order. Does the company have a naive attribution model —e.g., marketing costs attributed to items weighted by price? Marketing costs might also be attributed per department, or item category, or—in the most sophisticated organizations—per individual item based on user ad clicks.

The business logic transformation that generates this nuanced version of profit must integrate all the subtleties of attribution—i.e., a model that links orders to specific ads and advertising costs. Is attribution data stored in the guts of ETL scripts, or is it pulled from a table that is automatically generated from ad platform data?

This type of reporting data is a quintessential example of *derived data*—data computed from other data stored in a data system. Derived data critics will point out that it is challenging for the ETL to maintain consistency in the derived metrics.[16] For example, if the company updates its attribution model, this change may need to be merged into many ETL scripts for reporting. (ETL scripts are notorious for breaking the DRY principle.) Updating these ETL scripts is a manual and labor-intensive process, involving domain expertise in processing logic and previous changes. Updated scripts must also be validated for consistency and accuracy.

From our perspective, these are legitimate criticisms but not necessarily very constructive because the alternative to derived data in this instance is equally distasteful. Analysts will need to run their reporting queries if profit data is not stored in the data warehouse, including profit logic. Updating complex ETL scripts to represent changes to business logic accurately is an overwhelming, labor-intensive task, but getting analysts to update their reporting queries consistently is well-nigh impossible.

One interesting alternative is to push business logic into a *metrics layer*,[17] but still leverage the data warehouse or other tool to do the computational heavy lifting. A metrics layer encodes business logic and allows analysts and dashboard users to build complex analytics from a library of defined metrics. The metrics layer generates queries from the metrics and sends these to the database. We discuss semantic and metrics layers in more detail in Chapter 9.

MapReduce

No discussion of batch transformation can be complete without touching on MapReduce. This isn't because MapReduce is widely used by data engineers these days. MapReduce was the defining batch data transformation pattern of the big data era, it still influences many distributed systems data engineers use today, and it's useful for data engineers to understand at a basic level. MapReduce was

introduced by Google in a follow-up to its paper on GFS. It was initially the de facto processing pattern of Hadoop, the open source analogue technology of GFS that we introduced in Chapter 6.

A simple MapReduce job consists of a collection of map tasks that read individual data blocks scattered across the nodes, followed by a shuffle that redistributes result data across the cluster and a reduce step that aggregates data on each node. For example, suppose that we wanted to run the following SQL query:

```
SELECT COUNT(*), user_id
FROM user_events
GROUP BY user_id;
```

The table data is spread across nodes in data blocks; the MapReduce job generates one map task per block. Each map task essentially runs the query on a single block—i.e., it generates a count for each user ID that appears in the block. While a block might contain hundreds of megabytes, the full table could be petabytes in size. However, the map portion of the job is a nearly perfect example of embarrassing parallelism; the data scan rate across the full cluster essentially scales linearly with the number of nodes.

We then need to aggregate (reduce) to gather results from the full cluster. We're not gathering results to a single node; rather, we redistribute results by key so that each key ends up on one and only one node. This is the shuffle step, which is often executed using a hashing algorithm on keys. Once the map results have been shuffled, we sum the results for each key. The key/count pairs can be written to the local disk on the node where they are computed. We collect the results stored across nodes to view the full query results.

Real-world MapReduce jobs can be far more complex than what we describe here. A complex query that filters with a `WHERE` clause joins three tables and applies a window function that would consist of many map and reduce stages.

After MapReduce

Google's original MapReduce model is extremely powerful but is now viewed as excessively rigid. It utilizes numerous short-lived ephemeral tasks that read from and write to disk. In particular, no intermediate state is preserved in memory; all data is transferred between tasks by storing it to disk or pushing it over the network. This simplifies state and workflow management and minimizes memory consumption, but it can also drive high-disk bandwidth utilization and increase processing time.

The MapReduce paradigm was constructed around the idea that magnetic disk capacity and bandwidth were so cheap that it made sense to simply throw a massive amount of disk at data to realize ultra-fast queries. This worked to an extent; MapReduce repeatedly set data processing records during the early days of Hadoop.

However, we have lived in a post-MapReduce world for quite some time. Post-MapReduce processing does not truly discard MapReduce; it still includes the elements of map, shuffle, and reduce, but it relaxes the constraints of MapReduce to allow for in-memory caching.[18] Recall that RAM is much faster than SSD and HDDs in transfer speed and seek time. Persisting even a tiny amount of judiciously chosen data in memory can dramatically speed up specific data processing tasks and utterly crush the performance of MapReduce.

For example, Spark, BigQuery, and various other data processing frameworks were designed around in-memory processing. These frameworks treat data as a distributed set that resides in memory. If data overflows available memory, this causes a *spill to disk*. The disk is treated as a second-class data-storage layer for processing, though it is still highly valuable.

The cloud is one of the drivers for the broader adoption of memory caching; it is much more effective to lease memory during a specific processing job than to own it 24 hours a day. Advancements in leveraging memory for transformations will continue to yield gains for the foreseeable future.

Materialized Views, Federation, and Query Virtualization

In this section, we look at several techniques that virtualize query results by presenting them as table-like objects. These techniques can become part of a transformation pipeline or sit right before end-user data consumption.

Views

First, let's review views to set the stage for materialized views. A *view* is a database object that we can select from just like any other table. In practice, a view is just a query that references other tables. When we select from a view, that database creates a new query that combines the view subquery with our query. The query optimizer then optimizes and runs the full query.

Views play a variety of roles in a database. First, views can serve a security role. For example, views can select only specific columns and filter rows, thus providing restricted data access. Various views can be created for job roles depending on user data access.

Second, a view might be used to provide a current deduplicated picture of data. If we're using an insert-only pattern, a view may be used to return a deduplicated version of a table showing only the latest version of each record.

Third, views can be used to present common data access patterns. Suppose that marketing analysts must frequently run a query that joins five tables. We could create a view that joins together these five tables into a wide table. Analysts can then write queries that filter and aggregate on top of this view.

Materialized views

We mentioned materialized views in our earlier discussion of query caching. A potential disadvantage of (nonmaterialized) views is that they don't do any precomputation. In the example of a view that joins five tables, this join must run every time a marketing analyst runs a query on this view, and the join could be extremely expensive.

A materialized view does some or all of the view computation in advance. In our example, a materialized view might save the five table join results every time a change occurs in the source tables. Then, when a user references the view, they're querying from the prejoined data. A materialized view is a de facto transformation step, but the database manages execution for convenience.

Materialized views may also serve a significant query optimization role depending on the database, even for queries that don't directly reference them. Many query optimizers can identify queries that "look like" a materialized view. An analyst may run a query that uses a filter that appears in a materialized view. The optimizer will rewrite the query to select from the precomputed results.

Composable materialized views

In general, materialized views do not allow for composition—that is, a materialized view cannot select from another materialized view. However, we've recently seen the emergence of tools that support this capability. For example, Databricks has introduced the notion of *live tables*. Each table is updated as data arrives from sources. Data flows down to subsequent tables asynchronously.

Federated queries

Federated queries are a database feature that allows an OLAP database to select from an external data source, such as object storage or RDBMS. For example, let's say you need to combine data across object storage and various tables in MySQL and PostgreSQL databases. Your data warehouse can issue a federated query to these sources and return the combined results (Figure 8-19).

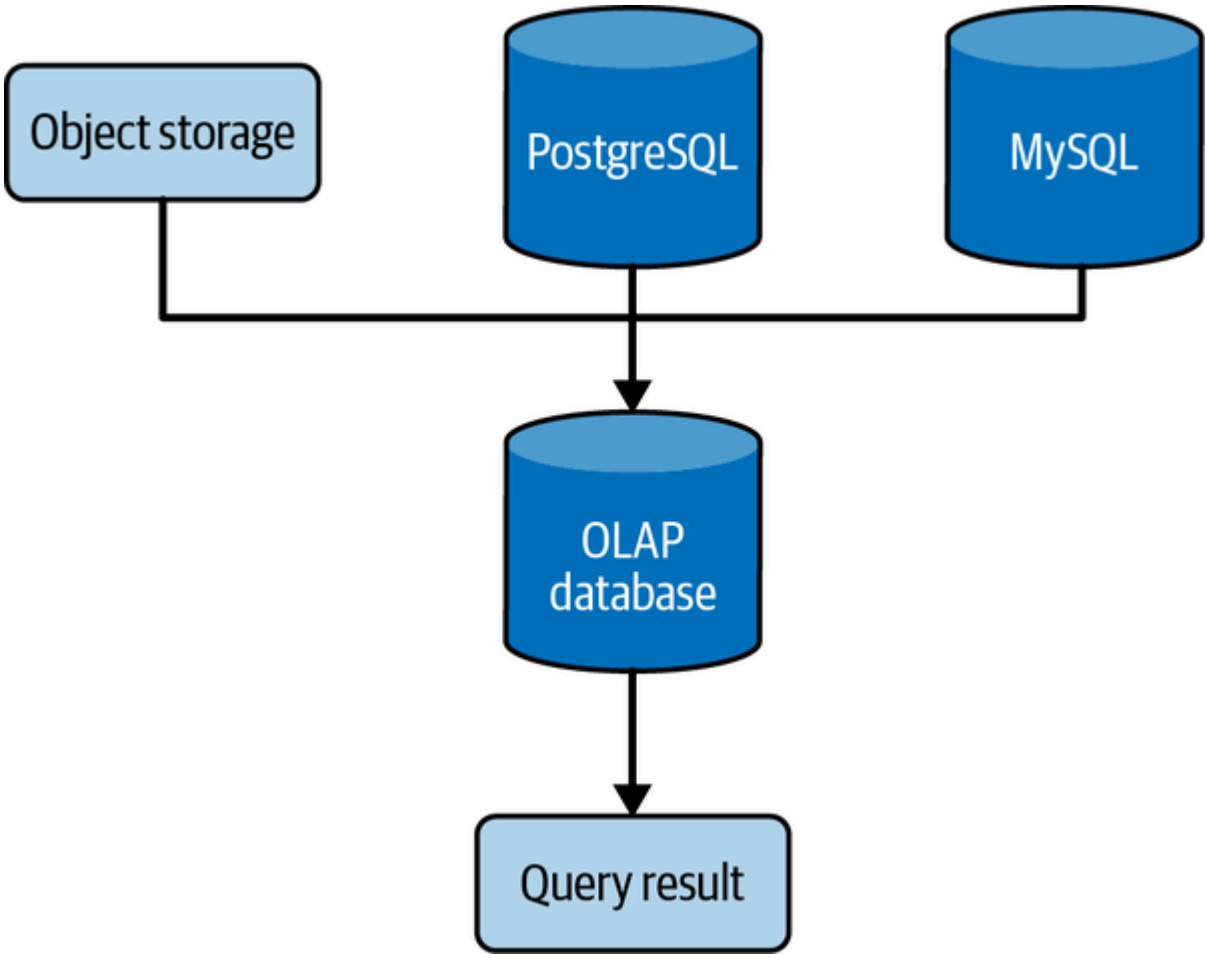

Figure 8-19. An OLAP database issues a federated query that gets data from object storage, MySQL, and PostgreSQL and returns a query result with the combined data

As another example, Snowflake supports the notion of external tables defined on S3 buckets. An external data location and a file format are defined when creating the table, but data is not yet ingested into the table. When the external table is queried, Snowflake reads from S3 and processes the data based on the parameters set at the time of the table's creation. We can even join S3 data to internal database tables. This makes Snowflake and similar databases more compatible with a data lake environment.

Some OLAP systems can convert federated queries into materialized views. This gives us much of the performance of a native table without the need to manually ingest data every time the external source changes. The materialized view gets updated whenever the external data changes.

Data virtualization

Data virtualization is closely related to federated queries, but this typically entails a data processing and query system that doesn't store data internally. Right now, Trino (e.g., Starburst) and Presto are examples par excellence. Any query/processing engine that supports external tables can serve as a data virtualization engine. The most significant considerations with data virtualization are supported external sources and performance.

A closely related concept is the notion of *query pushdown*. Suppose I wanted to query data from Snowflake, join data from a MySQL database, and filter the results. Query pushdown aims to move as much work as possible to the source databases. The engine might look for ways to push filtering predicates into the queries on the source systems. This serves two purposes: first, it offloads computation from the virtualization layer, taking advantage of the query performance of the source. Second, it potentially reduces the quantity of data that must push across the network, a critical bottleneck for virtualization performance.

Data virtualization is a good solution for organizations with data stored across various data sources. However, data virtualization should not be used haphazardly. For example, virtualizing a production MySQL database doesn't solve the core problem of analytics queries adversely impacting the production system—because Trino does not store data internally, it will pull from MySQL every time it runs a query.

Alternatively, data virtualization can be used as a component of data ingestion and processing pipelines. For instance, Trino might be used to select from MySQL once a day at midnight when the load on the production system is low. Results could be saved into S3 for consumption by downstream transformations and daily queries, protecting MySQL from direct analytics queries.

Data virtualization can be viewed as a tool that expands the data lake to many more sources by abstracting away barriers used to silo data between organizational units. An organization can store frequently accessed, transformed data in S3 and virtualize access between various parts of the company. This fits closely with the notion of a *data mesh* (discussed in Chapter 3), wherein small teams are responsible for preparing their data for analytics and sharing it with the rest of the company; virtualization can serve as a critical access layer for practical sharing.

Streaming Transformations and Processing

We've already discussed stream processing in the context of queries. The difference between streaming transformations and streaming queries is subtle and warrants more explanation.

Basics

Streaming queries run dynamically to present a current view of data, as discussed previously. *Streaming transformations* aim to prepare data for downstream consumption.

For instance, a data engineering team may have an incoming stream carrying events from an IoT source. These IoT events carry a device ID and event data. We wish to dynamically enrich these events with other device metadata, which is stored in a separate database. The stream-processing engine queries a separate database containing this metadata by device ID, generates new events with the added data, and passes it on to another stream. Live queries and triggered metrics run on this enriched stream (see Figure 8-20).

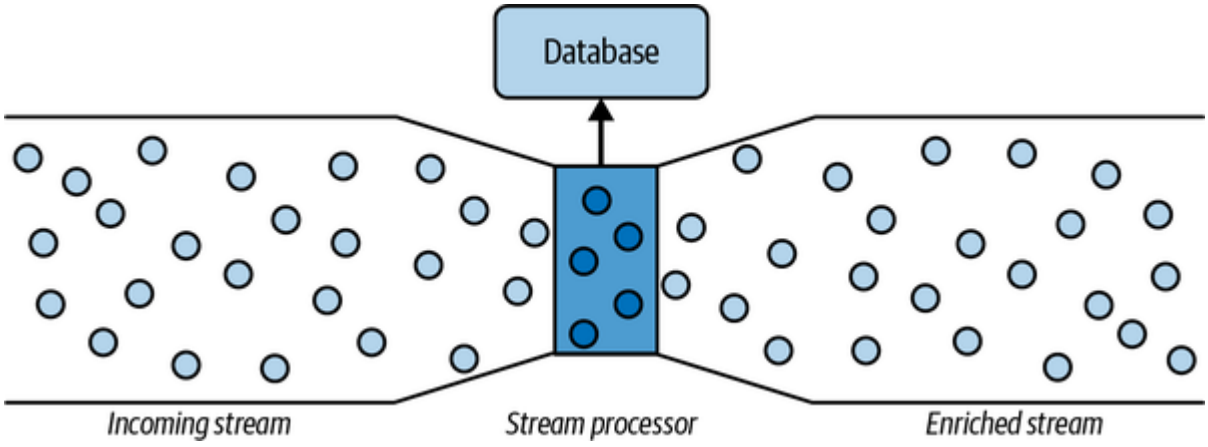

Figure 8-20. An incoming stream is carried by a streaming event platform and passed into a stream processor

Transformations and queries are a continuum

The line between transformations and queries is also blurry in batch processing, but the differences become even more subtle in the domain of streaming. For example, if we dynamically compute roll-up statistics on windows, and then send the output to a target stream, is this a transformation or a query?

Maybe we will eventually adopt new terminology for stream processing that better represents real-world use cases. For now, we will do our best with the terminology we have.

Streaming DAGs

One interesting notion closely related to stream enrichment and joins is the *streaming DAG*.[19] We first talked about this idea in our discussion of orchestration in Chapter 2. Orchestration is inherently a batch concept, but what if we wanted to enrich, merge, and split multiple streams in real time?

Let's take a simple example where streaming DAG would be useful. Suppose that we want to combine website clickstream data with IoT data. This will allow us to get a unified view of user activity by combining IoT events with clicks. Furthermore, each data stream needs to be preprocessed into a standard format (see Figure 8-21).

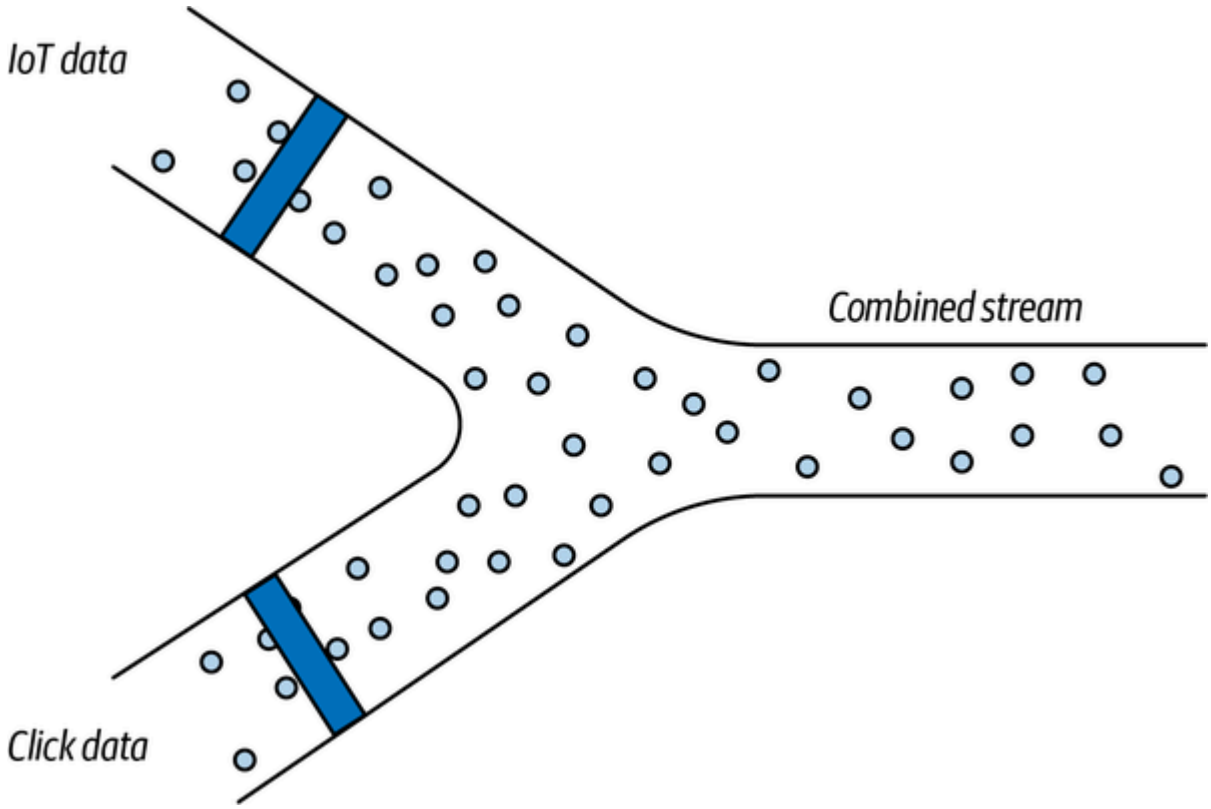

Figure 8-21. A simple streaming DAG

This has long been possible by combining a streaming store (e.g., Kafka) with a stream processor (e.g., Flink). Creating the DAG amounted to building a complex Rube Goldberg machine, with numerous topics and processing jobs connected.

Pulsar dramatically simplifies this process by treating DAGs as a core streaming abstraction. Rather than managing flows across several systems, engineers can define their streaming DAGs as code inside a single system.

Micro-batch versus true streaming

A long-running battle has been ongoing between micro-batch and true streaming approaches. Fundamentally, it's important to understand your use case, the performance requirements, and the performance capabilities of the framework in question.

Micro-batching is a way to take a batch-oriented framework and apply it in a streaming situation. A micro-batch might run anywhere from every two minutes to every second. Some micro-batch frameworks (e.g., Apache Spark Streaming) are designed for this use case and will perform well with appropriately allocated resources at a high batch frequency. (In truth, DBAs and engineers have long

used micro-batching with more traditional databases; this often led to horrific performance and resource consumption.)

True streaming systems (e.g., Beam and Flink) are designed to process one event at a time. However, this comes with significant overhead. Also, it's important to note that even in these true streaming systems, many processes will still occur in batches. A basic enrichment process that adds data to individual events can deliver one event at a time with low latency. However, a triggered metric on windows may run every few seconds, every few minutes, etc.

When you're using windows and triggers (hence, batch processing), what's the window frequency? What's the acceptable latency? If you are collecting Black Friday sales metrics published every few minutes, micro-batches are probably just fine as long as you set an appropriate micro-batch frequency. On the other hand, if your ops team is computing metrics every second to detect DDoS attacks, true streaming may be in order.

When should you use one over the other? Frankly, there is no universal answer. The term *micro-batch* has often been used to dismiss competing technologies, but it may work just fine for your use case and can be superior in many respects depending on your needs. If your team already has expertise in Spark, you will be able to spin up a Spark (micro-batch) streaming solution extremely fast.

There's no substitute for domain expertise and real-world testing. Talk to experts who can present an even-handed opinion. You can also easily test the alternatives by spinning up tests on cloud infrastructure. Also, watch out for spurious benchmarks provided by vendors. Vendors are notorious for cherry-picking benchmarks and setting up artificial examples that don't match reality (recall our conversation on benchmarks in Chapter 4). Frequently, vendors will show massive advantages in their benchmark results but fail to deliver in the real world for your use case.

Whom You'll Work With

Queries, transformations, and modeling impact all stakeholders up and down the data engineering lifecycle. The data engineer is responsible for several things at this stage in the lifecycle. From a technical angle, the data engineer designs, builds, and maintains the integrity of the systems that query and transform data. The data engineer also implements data models within this system. This is the most "full-contact" stage where your focus is to add as much value as possible, both in terms of functioning systems and reliable and trustworthy data.

Upstream Stakeholders

When it comes to transformations, upstream stakeholders can be broken into two broad categories: those who control the business definitions and those who control the systems generating data.

When interfacing with upstream stakeholders about business definitions and logic, you'll need to know the data sources—what they are, how they're used, and the business logic and definitions involved. You'll work with the engineers in charge of these source systems and the business stakeholders who oversee the complementary products and apps. A data engineer might work alongside "the business" and technical stakeholders on a data model.

The data engineer needs to be involved in designing the data model and later updates because of changes in business logic or new processes. Transformations are easy enough to do; just write a query and plop the results into a table or view. Creating them so they're both performant and valuable to the business is another matter. Always keep the requirements and expectations of the business top of mind when transforming data.

The stakeholders of the upstream systems want to make sure your queries and transformations minimally impact their systems. Ensure bidirectional communication about changes to the data models (column and index changes, for example) in source systems, as these can directly impact queries,

transformations, and analytical data models. Data engineers should know about schema changes, including the addition or deletion of fields, data type changes, and anything else that might materially impact the ability to query and transform data.

Downstream Stakeholders

Transformations are where data starts providing utility to downstream stakeholders. Your downstream stakeholders include many people, including data analysts, data scientists, ML engineers, and "the business." Collaborate with them to ensure the data model and transformations you provide are performant and useful. In terms of performance, queries should execute as quickly as possible in the most cost-effective way. What do we mean by *useful*? Analysts, data scientists, and ML engineers should be able to query a data source with the confidence the data is of the highest quality and completeness and can be integrated into their workflows and data products. The business should be able to trust that transformed data is accurate and actionable.

Undercurrents

The transformation stage is where your data mutates and morphs into something useful for the business. Because there are many moving parts, the undercurrents are especially critical at this stage.

Security

Queries and transformations combine disparate datasets into new datasets. Who has access to this new dataset? If someone does have access to a dataset, continue to control who has access to a dataset's column, row, and cell-level access.

Be aware of attack vectors against your database at query time. Read/write privileges to the database must be tightly monitored and controlled. Query access to the database must be controlled in the same way as you normally control access to your organization's systems and environments.

Keep credentials hidden; avoid copying and pasting passwords, access tokens, or other credentials into code or unencrypted files. It's shockingly common to see code in GitHub repositories with database usernames and passwords pasted directly in the codebase! It goes without saying, don't share passwords with other users. Finally, never allow unsecured or unencrypted data to traverse the public internet.

Data Management

Though data management is essential at the source system stage (and every other stage of the data engineering lifecycle), it's especially critical at the transformation stage. Transformation inherently creates new datasets that need to be managed. As with other stages of the data engineering lifecycle, it's critical to involve all stakeholders in data models and transformations and manage their expectations. Also, make sure everyone agrees on naming conventions that align with the respective business definitions of the data. Proper naming conventions should be reflected in easy-to-understand field names. Users can also check in a data catalog for more clarity on what the field means when it was created, who maintains the dataset, and other relevant information.

Accounting for definitional accuracy is key at the transformation stage. Does the transformation adhere to the expected business logic? Increasingly, the notion of a semantic or metrics layer that sits independent of transformations is becoming popular. Instead of enforcing business logic within the transformation at runtime, why not keep these definitions as a standalone stage before your transformation layer? While it's still early days, expect to see semantic and metrics layers becoming more popular and commonplace in data engineering and data management.

Because transformations involve mutating data, it's critical to ensure that the data you're using is free of defects and represents ground truth. If MDM is an option at your company, pursue its implementation. Conformed dimensions and other transformations rely on MDM to preserve data's original integrity and ground truth. If MDM isn't possible, work with upstream stakeholders who control the data to ensure that any data you're transforming is correct and complies with the agreed-upon business logic.

Data transformations make it potentially difficult to know how a dataset was derived along the same lines. In Chapter 6, we discussed data catalogs. As we transform data, *data lineage* tools become invaluable. Data lineage tools help both data engineers, who must understand previous transformation steps as they create new transformations, and analysts, who need to understand where data came from as they run queries and build reports.

Finally, what impact does regulatory compliance have on your data model and transformations? Are sensitive fields data masked or obfuscated if necessary? Do you have the ability to delete data in response to deletion requests? Does your data lineage tracking allow you to see data derived from deleted data and rerun transformations to remove data downstream of raw sources?

DataOps

With queries and transformations, DataOps has two areas of concern: data and systems. You need to monitor and be alerted for changes or anomalies in these areas. The field of data observability is exploding right now, with a big focus on data reliability. There's even a recent job title called *data reliability engineer*. This section emphasizes data observability and data health, which focuses on the query and transformation stage.

Let's start with the data side of DataOps. When you query data, are the inputs and outputs correct? How do you know? If this query is saved to a table, is the schema correct? How about the shape of the data and related statistics such as min/max values, null counts, and more? You should run data-quality tests on the input datasets and the transformed dataset, which will ensure that the data meets the expectations of upstream and downstream users. If there's a data-quality issue in the transformation, you should have the ability to flag this issue, roll back the changes, and investigate the root cause.

Now let's look at the Ops part of DataOps. How are the systems performing? Monitor metrics such as query queue length, query concurrency, memory usage, storage utilization, network latency, and disk I/O. Use metric data to spot bottlenecks and poor-performing queries that might be candidates for refactoring and tuning. If the query is perfectly fine, you'll have a good idea of where to tune the database itself (for instance, by clustering a table for faster lookup performance). Or, you may need to upgrade the database's compute resources. Today's cloud and SaaS databases give you a ton of flexibility for quickly upgrading (and downgrading) your system. Take a data-driven approach and use your observability metrics to pinpoint whether you have a query or a systems-related issue.

The shift toward SaaS-based analytical databases changes the cost profile of data consumption. In the days of on-premises data warehouses, the system and licenses were purchased up front, with no additional usage cost. Whereas traditional data engineers would focus on performance optimization to squeeze the maximum utility out of their expensive purchases, data engineers working with cloud data warehouses that charge on a consumption basis need to focus on cost management and cost optimization. This is the practice of *FinOps* (see Chapter 4).

Data Architecture

The general rules of good data architecture in Chapter 3 apply to the transformation stage. Build robust systems that can process and transform data without imploding. Your choices for ingestion and storage will directly impact your general architecture's ability to perform reliable queries and transformations. If the ingestion and storage are appropriate to your query and transformation patterns, you should be in a great place. On the other hand, if your queries and transformations don't work well with your upstream systems, you're in for a world of pain.

For example, we often see data teams using the wrong data pipelines and databases for the job. A data team might connect a real-time data pipeline to an RDBMS or Elasticsearch and use this as their data warehouse. These systems are not optimized for high-volume aggregated OLAP queries and will implode under this workload. This data team clearly didn't understand how their architectural choices would impact query performance. Take the time to understand the trade-offs inherent in your architecture choices; be clear about how your data model will work with ingestion and storage systems and how queries will perform.

Orchestration

Data teams often manage their transformation pipelines using simple time-based schedules—e.g., cron jobs. This works reasonably well at first but turns into a nightmare as workflows grow more complicated. Use orchestration to manage complex pipelines using a dependency-based approach. Orchestration is also the glue that allows us to assemble pipelines that span multiple systems.

Software Engineering

When writing transformation code, you can use many languages—such as SQL, Python, and JVM-based languages—platforms ranging from data warehouses to distributed computing clusters, and everything in between. Each language and platform has its strengths and quirks, so you should know the best practices of your tools. For example, you might write data transformations in Python, powered by a distributed system such as Spark or Dask. When running a data transformation, are you using a UDF when a native function might work much better? We've seen cases where poorly written, sluggish UDFs were replaced by a built-in SQL command, with instant and dramatic improvement in performance.

The rise of analytics engineering brings software engineering practices to end users, with the notion of *analytics as code*. Analytics engineering transformation tools like dbt have exploded in popularity, giving analysts and data scientists the ability to write in-database transformations using SQL, without the direct intervention of a DBA or a data engineer. In this case, the data engineer is responsible for setting up the code repository and CI/CD pipeline used by the analysts and data scientists. This is a big change in the role of a data engineer, who would historically build and manage the underlying infrastructure and create the data transformations. As data tools lower the barriers to entry and become more democratized across data teams, it will be interesting to see how the workflows of data teams change.

Using a GUI-based low-code tool, you'll get useful visualizations of the transformation workflow. You still need to understand what's going on under the hood. These GUI-based transformation tools will often generate SQL or some other language behind the scenes. While the point of a low-code tool is to alleviate the need to be involved in low-level details, understanding the code behind the scenes will help with debugging and performance optimization. Blindly assuming that the tool is generating performant code is a mistake.

We suggest that data engineers pay particular attention to software engineering best practices at the query and transformation stage. While it's tempting to simply throw more processing resources at a dataset, knowing how to write clean, performant code is a much better approach.

Conclusion

Transformations sit at the heart of data pipelines. It's critical to keep in mind the purpose of transformations. Ultimately, engineers are not hired to play with the latest technological toys but to serve their customers. Transformations are where data adds value and ROI to the business.

Our opinion is that it is possible to adopt exciting transformation technologies *and* serve stakeholders. Chapter 11 talks about the *live data stack*, essentially reconfiguring the data stack around streaming

data ingestion and bringing transformation workflows closer to the source system applications themselves. Engineering teams that think about real-time data as the technology for the sake of technology will repeat the mistakes of the big data era. But in reality, the majority of organizations that we work with have a business use case that would benefit from streaming data. Identifying these use cases and focusing on the value before choosing technologies and complex systems is key.

As we head into the serving stage of the data engineering lifecycle in Chapter 9, reflect on technology as a tool for realizing organizational goals. If you're a working data engineer, think about how improvements in transformation systems could help you to serve your end customers better. If you're just embarking on a path toward data engineering, think about the kinds of business problems you're interested in solving with technology.

Additional Resources

- "Building a Real-Time Data Vault in Snowflake" by Dmytro Yaroshenko and Kent Graziano
- *Building a Scalable Data Warehouse with Data Vault 2.0* (Morgan Kaufmann) by Daniel Linstedt and Michael Olschimke
- *Building the Data Warehouse* (Wiley), *Corporate Information Factory*, and *The Unified Star Schema* (Technics Publications) by W. H. (Bill) Inmon
- "Caching in Snowflake Data Warehouse" Snowflake Community page
- "Data Warehouse: The Choice of Inmon vs. Kimball" by Ian Abramson
- *The Data Warehouse Toolkit* by Ralph Kimball and Margy Ross (Wiley)
- "Data Vault—An Overview" by John Ryan
- "Data Vault 2.0 Modeling Basics" by Kent Graziano
- "A Detailed Guide on SQL Query Optimization" tutorial by Megha
- "Difference Between Kimball and Inmon" by manmeetjuneja5
- "Eventual vs. Strong Consistency in Distributed Databases" by Saurabh.v
- "The Evolution of the Corporate Information Factory" by Bill Inmon
- Gavroshe USA's "DW 2.0" web page
- Google Cloud's "Using Cached Query Results" documentation
- Holistics' "Cannot Combine Fields Due to Fan-Out Issues?" FAQ page
- "How a SQL Database Engine Works," by Dennis Pham
- "How Should Organizations Structure Their Data?" by Michael Berk
- "Inmon or Kimball: Which Approach Is Suitable for Your Data Warehouse?" by Sansu George
- "Introduction to Data Vault Modeling" document, compiled by Kent Graziano and Dan Linstedt
- "Introduction to Data Warehousing", "Introduction to Dimensional Modelling for Data Warehousing", and "Introduction to Data Vault for Data Warehousing" by Simon Kitching
- Kimball Group's "Four-Step Dimensional Design Process", "Conformed Dimensions", and "Dimensional Modeling Techniques" web pages

- "Kimball vs. Inmon vs. Vault" Reddit thread
- "Modeling of Real-Time Streaming Data?" Stack Exchange thread
- "The New 'Unified Star Schema' Paradigm in Analytics Data Modeling Review" by Andriy Zabavskyy
- Oracle's "Slowly Changing Dimensions" tutorial
- ScienceDirect's "Corporate Information Factory" web page
- "A Simple Explanation of Symmetric Aggregates or 'Why on Earth Does My SQL Look Like That?'" by Lloyd Tabb
- "Streaming Event Modeling" by Paul Stanton
- "Types of Data Warehousing Architecture" by Amritha Fernando
- US patent for "Method and Apparatus for Functional Integration of Metadata"
- Zentut's "Bill Inmon Data Warehouse" web page

[1] See, for example, Emin Gün Sirer, "NoSQL Meets Bitcoin and Brings Down Two Exchanges: The Story of Flexcoin and Poloniex," *Hacking, Distributed*, April 6, 2014, *https://oreil.ly/RM3QX*.

[2] Some Redshift configurations rely on object storage instead.

[3] The authors are aware of an incident involving a new analyst at a large grocery store chain running `SELECT *` on a production database and bringing down a critical inventory database for three days.

[4] Figure 8-11 and the example it depicts are significantly based on "Introducing Stream—Stream Joins in Apache Spark 2.3" by Tathagata Das and Joseph Torres (Databricks *Engineering Blog*, March 13, 2018).

[5] For more details on the DRY principle, see *The Pragmatic Programmer* by David Thomas and Andrew Hunt (Addison-Wesley Professional, 2019).

[6] E. F. Codd, "Further Normalization of the Data Base Relational Model," IBM Research Laboratory (1971), *https://oreil.ly/Muajm*.

[7] H. W. Inmon, *Building the Data Warehouse* (Hoboken: Wiley, 2005).

[8] Inmon, *Building the Data Warehouse*.

[9] Inmon, *Building the Data Warehouse*.

[10] Although dimensions and facts are often associated with Kimball, they were first used at General Mills and Dartmouth University in the 1960s and had early adoption at Nielsen and IRI, among other companies.

[11] The Data Vault has two versions, 1.0 and 2.0. This section focuses on Data Vault 2.0, but we'll call it *Data Vault* for the sake of brevity.

[12] Kent Graziano, "Data Vault 2.0 Modeling Basics," Vertabelo, October 20, 2015, *https://oreil.ly/iuW1U*.

[13] Bill Inmon, “Avoiding the Horrible Task of Integrating Data,” LinkedIn Pulse, March 24, 2022, *https://oreil.ly/yLb71*.

[14] Alex Woodie, “Lakehouses Prevent Data Swamps, Bill Inmon Says,” Datanami, June 1, 2021, *https://oreil.ly/XMwWc*.

[15] We remind you to use UDFs responsibly. SQL UDFs often perform reasonably well. We’ve seen JavaScript UDFs increase query time from a few minutes to several hours.

[16] Michael Blaha, “Be Careful with Derived Data,” Dataversity, December 5, 2016, *https://oreil.ly/garoL*.

[17] Benn Stancil, “The Missing Piece of the Modern Data Stack,” *benn.substack*, April 22, 2021, *https://oreil.ly/GYf3Z*.

[18] “What Is the Difference Between Apache Spark and Hadoop MapReduce?,” Knowledge Powerhouse YouTube video, May 20, 2017, *https://oreil.ly/WN0eX*.

[19] For a detailed application of the concept of a streaming DAG, see “Why We Moved from Apache Kafka to Apache Pulsar” by Simba Khadder, StreamNative blog, April 21, 2020, *https://oreil.ly/Rxfko*.