APIs

The bulk of software engineering is just plumbing.

—Karl Hughes

As we mentioned in Chapter 5, APIs are a data source that continues to grow in importance and popularity. A typical organization may have hundreds of external data sources such as SaaS platforms or partner companies. The hard reality is that no proper standard exists for data exchange over APIs. Data engineers can spend a significant amount of time reading documentation, communicating with external data owners, and writing and maintaining API connection code.

Three trends are slowly changing this situation. First, many vendors provide API client libraries for various programming languages that remove much of the complexity of API access.

Second, numerous data connector platforms are available now as SaaS, open source, or managed open source. These platforms provide turnkey data connectivity to many data sources; they offer frameworks for writing custom connectors for unsupported data sources. See "Managed Data Connectors".

The third trend is the emergence of data sharing (discussed in Chapter 5)—i.e., the ability to exchange data through a standard platform such as BigQuery, Snowflake, Redshift, or S3. Once data lands on one of these platforms, it is straightforward to store it, process it, or move it somewhere else. Data sharing has had a large and rapid impact in the data engineering space.

Don't reinvent the wheel when data sharing is not an option and direct API access is necessary. While a managed service might look like an expensive option, consider the value of your time and the opportunity cost of building API connectors when you could be spending your time on higher-value work.

In addition, many managed services now support building custom API connectors. This may provide API technical specifications in a standard

format or writing connector code that runs in a serverless function framework (e.g., AWS Lambda) while letting the managed service handle the details of scheduling and synchronization. Again, these services can be a huge time-saver for engineers, both for development and ongoing maintenance.

Reserve your custom connection work for APIs that aren't well supported by existing frameworks; you will find that there are still plenty of these to work on. Handling custom API connections has two main aspects: software development and ops. Follow software development best practices; you should use version control, continuous delivery, and automated testing. In addition to following DevOps best practices, consider an orchestration framework, which can dramatically streamline the operational burden of data ingestion.

Message Queues and Event-Streaming Platforms

Message queues and event-streaming platforms are widespread ways to ingest real-time data from web and mobile applications, IoT sensors, and smart devices. As real-time data becomes more ubiquitous, you'll often find yourself either introducing or retrofitting ways to handle real-time data in your ingestion workflows. As such, it's essential to know how to ingest real-time data. Popular real-time data ingestion includes message queues or event-streaming platforms, which we covered in Chapter 5. Though these are both source systems, they also act as ways to ingest data. In both cases, you consume events from the publisher you subscribe to.

Recall the differences between messages and streams. A *message* is handled at the individual event level and is meant to be transient. Once a message is consumed, it is acknowledged and removed from the queue. On the other hand, a *stream* ingests events into an ordered log. The log persists for as long as you wish, allowing events to be queried over various ranges, aggregated, and combined with other streams to create new transformations published to downstream consumers. In Figure 7-14, we have two producers (producers 1 and 2) sending events to two consumers (consumers

1 and 2). These events are combined into a new dataset and sent to a producer for downstream consumption.

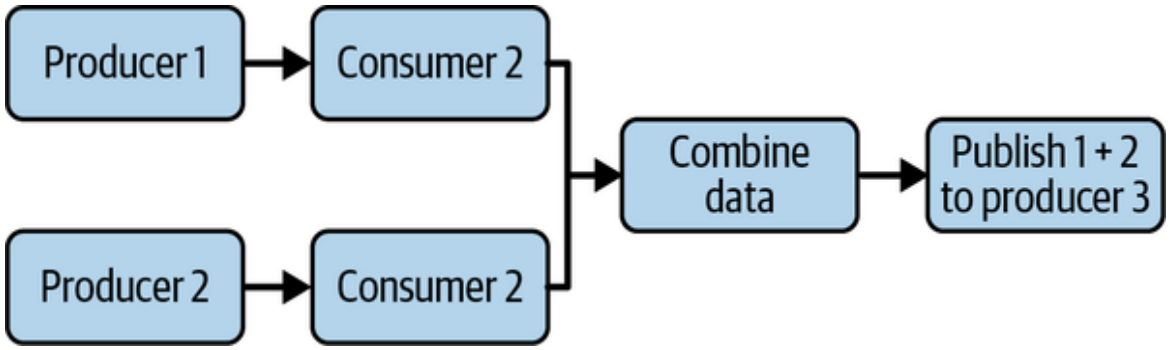

Figure 7-14. Two datasets are produced and consumed (producers 1 and 2), and then combined, with the combined data published to a new producer (producer 3)

The last point is an essential difference between batch and streaming ingestion. Whereas batch usually involves static workflows (ingest data, store it, transform it, and serve it), messages and streams are fluid. Ingestion can be nonlinear, with data being published, consumed, republished, and re-consumed. When designing your real-time ingestion workflows, keep in mind how data will flow.

Another consideration is the throughput of your real-time data pipelines. Messages and events should flow with as little latency as possible, meaning you should provision adequate partition (or shard) bandwidth and throughput. Provide sufficient memory, disk, and CPU resources for event processing, and if you're managing your real-time pipelines, incorporate autoscaling to handle spikes and save money as load decreases. For these reasons, managing your streaming platform can entail significant overhead. Consider managed services for your real-time ingestion pipelines, and focus your attention on ways to get value from your real-time data.

Managed Data Connectors

These days, if you're considering writing a data ingestion connector to a database or API, ask yourself: has this already been created? Furthermore, is there a service that will manage the nitty-gritty details of this connection for me? "APIs" mentions the popularity of managed data connector

platforms and frameworks. These tools aim to provide a standard set of connectors available out of the box to spare data engineers building complicated plumbing to connect to a particular source. Instead of creating and managing a data connector, you outsource this service to a third party.

Generally, options in the space allow users to set a target and source, ingest in various ways (e.g., CDC, replication, truncate and reload), set permissions and credentials, configure an update frequency, and begin syncing data. The vendor or cloud behind the scenes fully manages and monitors data syncs. If data synchronization fails, you'll receive an alert with logged information on the cause of the error.

We suggest using managed connector platforms instead of creating and managing your connectors. Vendors and OSS projects each typically have hundreds of prebuilt connector options and can easily create custom connectors. The creation and management of data connectors is largely undifferentiated heavy lifting these days and should be outsourced whenever possible.

Moving Data with Object Storage

Object storage is a multitenant system in public clouds, and it supports storing massive amounts of data. This makes object storage ideal for moving data in and out of data lakes, between teams, and transferring data between organizations. You can even provide short-term access to an object with a signed URL, giving a user temporary permission.

In our view, object storage is the most optimal and secure way to handle file exchange. Public cloud storage implements the latest security standards, has a robust track record of scalability and reliability, accepts files of arbitrary types and sizes, and provides high-performance data movement. We discussed object storage much more extensively in Chapter 6.

EDI

Another practical reality for data engineers is *electronic data interchange* (EDI). The term is vague enough to refer to any data movement method. It

usually refers to somewhat archaic means of file exchange, such as by email or flash drive. Data engineers will find that some data sources do not support more modern means of data transport, often because of archaic IT systems or human process limitations.

Engineers can at least enhance EDI through automation. For example, they can set up a cloud-based email server that saves files onto company object storage as soon as they are received. This can trigger orchestration processes to ingest and process data. This is much more robust than an employee downloading the attached file and manually uploading it to an internal system, which we still frequently see.

Databases and File Export

Engineers should be aware of how the source database systems handle file export. Export involves large data scans that significantly load the database for many transactional systems. Source system engineers must assess when these scans can be run without affecting application performance and might opt for a strategy to mitigate the load. Export queries can be broken into smaller exports by querying over key ranges or one partition at a time. Alternatively, a read replica can reduce load. Read replicas are especially appropriate if exports happen many times a day and coincide with a high source system load.

Major cloud data warehouses are highly optimized for direct file export. For example, Snowflake, BigQuery, Redshift, and others support direct export to object storage in various formats.

Practical Issues with Common File Formats

Engineers should also be aware of the file formats to export. CSV is still ubiquitous and highly error prone at the time of this writing. Namely, CSV's default delimiter is also one of the most familiar characters in the English language—the comma! But it gets worse.

CSV is by no means a uniform format. Engineers must stipulate the delimiter, quote characters, and escaping to appropriately handle the export

of string data. CSV also doesn't natively encode schema information or directly support nested structures. CSV file encoding and schema information must be configured in the target system to ensure appropriate ingestion. Autodetection is a convenience feature provided in many cloud environments but is inappropriate for production ingestion. As a best practice, engineers should record CSV encoding and schema details in file metadata.

More robust and expressive export formats include Parquet, Avro, Arrow, and ORC or JSON. These formats natively encode schema information and handle arbitrary string data with no particular intervention. Many of them also handle nested data structures natively so that JSON fields are stored using internal nested structures rather than simple strings. For columnar databases, columnar formats (Parquet, Arrow, ORC) allow more efficient data export because columns can be directly transcoded between formats. These formats are also generally more optimized for query engines. The Arrow file format is designed to map data directly into processing engine memory, providing high performance in data lake environments.

The disadvantage of these newer formats is that many of them are not natively supported by source systems. Data engineers are often forced to work with CSV data and then build robust exception handling and error detection to ensure data quality on ingestion. See Appendix A for a more extensive discussion of file formats.

Shell

The *shell* is an interface by which you may execute commands to ingest data. The shell can be used to script workflows for virtually any software tool, and shell scripting is still used extensively in ingestion processes. A shell script might read data from a database, reserialize it into a different file format, upload it to object storage, and trigger an ingestion process in a target database. While storing data on a single instance or server is not highly scalable, many of our data sources are not particularly large, and such approaches work just fine.

In addition, cloud vendors generally provide robust CLI-based tools. It is possible to run complex ingestion processes simply by issuing commands to the AWS CLI. As ingestion processes grow more complicated and the SLA grows more stringent, engineers should consider moving to a proper orchestration system.

SSH

SSH is not an ingestion strategy but a protocol used with other ingestion strategies. We use SSH in a few ways. First, SSH can be used for file transfer with SCP, as mentioned earlier. Second, SSH tunnels are used to allow secure, isolated connections to databases.

Application databases should never be directly exposed on the internet. Instead, engineers can set up a bastion host—i.e., an intermediate host instance that can connect to the database in question. This host machine is exposed on the internet, although locked down for minimal access from only specified IP addresses to specified ports. To connect to the database, a remote machine first opens an SSH tunnel connection to the bastion host, and then connects from the host machine to the database.

SFTP and SCP

Accessing and sending data both from secure FTP (SFTP) and secure copy (SCP) are techniques you should be familiar with, even if data engineers do not typically use these regularly (IT or security/secOps will handle this).

Engineers rightfully cringe at the mention of SFTP (occasionally, we even hear instances of FTP being used in production). Regardless, SFTP is still a practical reality for many businesses. They work with partner businesses that consume or provide data using SFTP, and are unwilling to rely on other standards. To avoid data leaks, security analysis is critical in these situations.

SCP is a file-exchange protocol that runs over an SSH connection related to SSH. SCP can be a secure file-transfer option if it is configured correctly.

Again, adding additional network access control (defense in depth) to enhance SCP security is highly recommended.

Webhooks

Webhooks, as we discussed in Chapter 5, are often referred to as *reverse APIs*. For a typical REST data API, the data provider gives engineers API specifications that they use to write their data ingestion code. The code makes requests and receives data in responses.

With a webhook (Figure 7-15), the data provider defines an API request specification, but the data provider *makes API calls* rather than receiving them; it's the data consumer's responsibility to provide an API endpoint for the provider to call. The consumer is responsible for ingesting each request and handling data aggregation, storage, and processing.

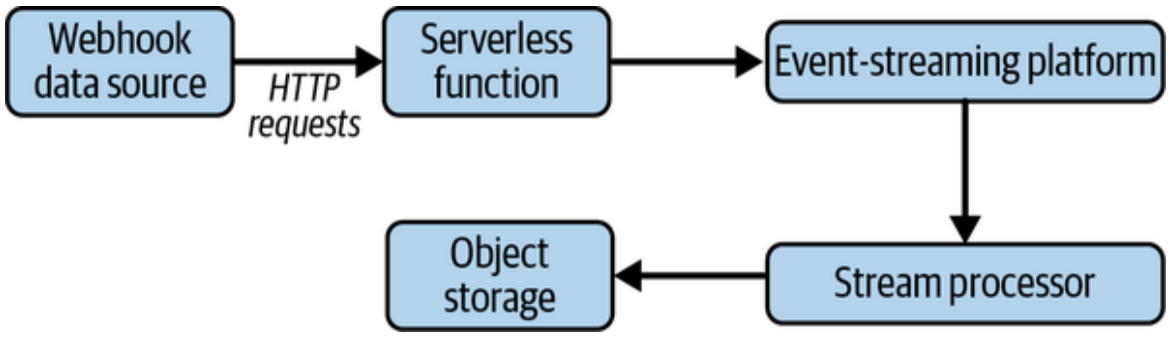

Figure 7-15. A basic webhook ingestion architecture built from cloud services

Webhook-based data ingestion architectures can be brittle, difficult to maintain, and inefficient. Using appropriate off-the-shelf tools, data engineers can build more robust webhook architectures with lower maintenance and infrastructure costs. For example, a webhook pattern in AWS might use a serverless function framework (Lambda) to receive incoming events, a managed event-streaming platform to store and buffer messages (Kinesis), a stream-processing framework to handle real-time analytics (Flink), and an object store for long-term storage (S3).

You'll notice that this architecture does much more than simply ingest the data. This underscores ingestion's entanglement with the other stages of the

data engineering lifecycle; it is often impossible to define your ingestion architecture without making decisions about storage and processing.

Web Interface

Web interfaces for data access remain a practical reality for data engineers. We frequently run into situations where not all data and functionality in a SaaS platform is exposed through automated interfaces such as APIs and file drops. Instead, someone must manually access a web interface, generate a report, and download a file to a local machine. This has obvious drawbacks, such as people forgetting to run the report or having their laptop die. Where possible, choose tools and workflows that allow for automated access to data.

Web Scraping

Web scraping automatically extracts data from web pages, often by combing the web page's various HTML elements. You might scrape ecommerce sites to extract product pricing information or scrape multiple news sites for your news aggregator. Web scraping is widespread, and you may encounter it as a data engineer. It's also a murky area where ethical and legal lines are blurry.

Here is some top-level advice to be aware of before undertaking any web-scraping project. First, ask yourself if you should be web scraping or if data is available from a third party. If your decision is to web scrape, be a good citizen. Don't inadvertently create a denial-of-service (DoS) attack, and don't get your IP address blocked. Understand how much traffic you generate and pace your web-crawling activities appropriately. Just because you can spin up thousands of simultaneous Lambda functions to scrape doesn't mean you should; excessive web scraping could lead to the disabling of your AWS account.

Second, be aware of the legal implications of your activities. Again, generating DoS attacks can entail legal consequences. Actions that violate terms of service may cause headaches for your employer or you personally.

Third, web pages constantly change their HTML element structure, making it tricky to keep your web scraper updated. Ask yourself, is the headache of maintaining these systems worth the effort?

Web scraping has interesting implications for the data engineering lifecycle processing stage; engineers should think about various factors at the beginning of a web-scraping project. What do you intend to do with the data? Are you just pulling required fields from the scraped HTML by using Python code and then writing these values to a database? Do you intend to maintain the complete HTML code of the scraped websites and process this data using a framework like Spark? These decisions may lead to very different architectures downstream of ingestion.

Transfer Appliances for Data Migration

For massive data (100 TB or more), transferring data directly over the internet may be a slow and costly process. At this scale, the fastest, most efficient way to move data is not over the wire but by truck. Cloud vendors offer the ability to send your data via a physical "box of hard drives." Simply order a storage device, called a *transfer appliance*, load your data from your servers, and then send it back to the cloud vendor, which will upload your data.

The suggestion is to consider using a transfer appliance if your data size hovers around 100 TB. On the extreme end, AWS even offers Snowmobile, a transfer appliance sent to you in a semitrailer! Snowmobile is intended to lift and shift an entire data center, in which data sizes are in the petabytes or greater.

Transfer appliances are handy for creating hybrid-cloud or multicloud setups. For example, Amazon's data transfer appliance (AWS Snowball) supports import and export. To migrate into a second cloud, users can export their data into a Snowball device, and then import it into a second transfer appliance to move data into GCP or Azure. This might sound awkward, but even when it's feasible to push data over the internet between

clouds, data egress fees make this a costly proposition. Physical transfer appliances are a cheaper alternative when the data volumes are significant.

Remember that transfer appliances and data migration services are one-time data ingestion events and are not suggested for ongoing workloads. Suppose you have workloads requiring constant data movement in either a hybrid or multicloud scenario. In that case, your data sizes are presumably batching or streaming much smaller data sizes on an ongoing basis.

Data Sharing

Data sharing is growing as a popular option for consuming data (see Chapters 5 and 6.) Data providers will offer datasets to third-party subscribers, either for free or at a cost. These datasets are often shared in a read-only fashion, meaning you can integrate these datasets with your own data (and other third-party datasets), but you do not own the shared dataset. In the strict sense, this isn't ingestion, where you get physical possession of the dataset. If the data provider decides to remove your access to a dataset, you'll no longer have access to it.

Many cloud platforms offer data sharing, allowing you to share your data and consume data from various providers. Some of these platforms also provide data marketplaces where companies and organizations can offer their data for sale.

Whom You'll Work With

Data ingestion sits at several organizational boundaries. In developing and managing data ingestion pipelines, data engineers will work with both people and systems sitting upstream (data producers) and downstream (data consumers).

Upstream Stakeholders

A significant disconnect often exists between those responsible for *generating data*—typically, software engineers—and the data engineers

who will prepare this data for analytics and data science. Software engineers and data engineers usually sit in separate organizational silos; if they think about data engineers, they typically see them simply as downstream consumers of the data exhaust from their application, not as stakeholders.

We see this current state of affairs as a problem and a significant opportunity. Data engineers can improve the quality of their data by inviting software engineers to be stakeholders in data engineering outcomes. The vast majority of software engineers are well aware of the value of analytics and data science but don't necessarily have aligned incentives to contribute to data engineering efforts directly.

Simply improving communication is a significant first step. Often software engineers have already identified potentially valuable data for downstream consumption. Opening a communication channel encourages software engineers to get data into shape for consumers and communicate about data changes to prevent pipeline regressions.

Beyond communication, data engineers can highlight the contributions of software engineers to team members, executives, and especially product managers. Involving product managers in the outcome and treating downstream data processed as part of a product encourages them to allocate scarce software development to collaboration with data engineers. Ideally, software engineers can work partially as extensions of the data engineering team; this allows them to collaborate on various projects, such as creating an event-driven architecture to enable real-time analytics.

Downstream Stakeholders

Who is the ultimate customer for data ingestion? Data engineers focus on data practitioners and technology leaders such as data scientists, analysts, and chief technical officers. They would do well also to remember their broader circle of business stakeholders such as marketing directors, vice presidents over the supply chain, and CEOs.

Too often, we see data engineers pursuing sophisticated projects (e.g., real-time streaming buses or complex data systems) while digital marketing managers next door are left downloading Google Ads reports manually. View data engineering as a business, and recognize who your customers are. Often basic automation of ingestion processes has significant value, especially for departments like marketing that control massive budgets and sit at the heart of revenue for the business. Basic ingestion work may seem tedious, but delivering value to these core parts of the company will open up more budget and more exciting long-term data engineering opportunities.

Data engineers can also invite more executive participation in this collaborative process. For a good reason, data-driven culture is quite fashionable in business leadership circles. Still, it is up to data engineers and other data practitioners to provide executives with guidance on the best structure for a data-driven business. This means communicating the value of lowering barriers between data producers and data engineers while supporting executives in breaking down silos and setting up incentives to lead to a more unified data-driven culture.

Once again, *communication* is the watchword. Honest communication early and often with stakeholders will go a long way to ensure that your data ingestion adds value.

Undercurrents

Virtually all the undercurrents touch the ingestion phase, but we'll emphasize the most salient ones here.

Security

Moving data introduces security vulnerabilities because you have to transfer data between locations. The last thing you want is to capture or compromise the data while moving.

Consider where the data lives and where it is going. Data that needs to move within your VPC should use secure endpoints and never leave the confines of the VPC. Use a VPN or a dedicated private connection if you need to send data between the cloud and an on-premises network. This might cost money, but the security is a good investment. If your data traverses the public internet, ensure that the transmission is encrypted. It is always a good practice to encrypt data over the wire.

Data Management

Naturally, data management begins at data ingestion. This is the starting point for lineage and data cataloging; from this point on, data engineers need to think about master data management, ethics, privacy, and compliance.

Schema changes

Schema changes (such as adding, changing, or removing columns in a database table) remain, from our perspective, an unsettled issue in data management. The traditional approach is a careful command-and-control review process. Working with clients at large enterprises, we have been quoted lead times of six months for the addition of a single field. This is an unacceptable impediment to agility.

On the opposite end of the spectrum, any schema change in the source triggers target tables to be re-created with the new schema. This solves schema problems at the ingestion stage but can still break downstream pipelines and destination storage systems.

One possible solution, which we, the authors, have meditated on for a while, is an approach pioneered by Git version control. When Linus Torvalds was developing Git, many of his choices were inspired by the limitations of Concurrent Versions System (CVS). CVS is completely centralized; it supports only one current official version of the code, stored on a central project server. To make Git a truly distributed system, Torvalds

used the notion of a tree; each developer could maintain their processed branch of the code and then merge to or from other branches.

A few years ago, such an approach to data was unthinkable. On-premises MPP systems are typically operated at close to maximum storage capacity. However, storage is cheap in big data and cloud data warehouse environments. One may quite easily maintain multiple versions of a table with different schemas and even different upstream transformations. Teams can support various "development" versions of a table by using orchestration tools such as Airflow; schema changes, upstream transformation, and code changes can appear in development tables before official changes to the *main* table.

Data ethics, privacy, and compliance

Clients often ask for our advice on encrypting sensitive data in databases, which generally leads us to ask a fundamental question: do you need the sensitive data you're trying to encrypt? As it turns out, this question often gets overlooked when creating requirements and solving problems.

Data engineers should always train themselves to ask this question when setting up ingestion pipelines. They will inevitably encounter sensitive data; the natural tendency is to ingest it and forward it to the next step in the pipeline. But if this data is not needed, why collect it at all? Why not simply drop sensitive fields before data is stored? Data cannot leak if it is never collected.

Where it is truly necessary to keep track of sensitive identities, it is common practice to apply tokenization to anonymize identities in model training and analytics. But engineers should look at where this tokenization is used. If possible, hash data at ingestion time.

Data engineers cannot avoid working with highly sensitive data in some cases. Some analytics systems must present identifiable, sensitive information. Engineers must act under the highest ethical standards whenever they handle sensitive data. In addition, they can put in place a variety of practices to reduce the direct handling of sensitive data. Aim as

much as possible for *touchless production* where sensitive data is involved. This means that engineers develop and test code on simulated or cleansed data in development and staging environments but automated code deployments to production.

Touchless production is an ideal that engineers should strive for, but situations inevitably arise that cannot be fully solved in development and staging environments. Some bugs may not be reproducible without looking at the live data that is triggering a regression. For these cases, put a broken-glass process in place: require at least two people to approve access to sensitive data in the production environment. This access should be tightly scoped to a particular issue and come with an expiration date.

Our last bit of advice on sensitive data: be wary of naive technological solutions to human problems. Both encryption and tokenization are often treated like privacy magic bullets. Most cloud-based storage systems and nearly all databases encrypt data at rest and in motion by default. Generally, we don't see encryption problems but data access problems. Is the solution to apply an extra layer of encryption to a single field or to control access to that field? After all, one must still tightly manage access to the encryption key. Legitimate use cases exist for single-field encryption, but watch out for ritualistic encryption.

On the tokenization front, use common sense and assess data access scenarios. If someone had the email of one of your customers, could they easily hash the email and find the customer in your data? Thoughtlessly hashing data without salting and other strategies may not protect privacy as well as you think.

DataOps

Reliable data pipelines are the cornerstone of the data engineering lifecycle. When they fail, all downstream dependencies come to a screeching halt. Data warehouses and data lakes aren't replenished with fresh data, and data scientists and analysts can't effectively do their jobs; the business is forced to fly blind.

Ensuring that your data pipelines are properly monitored is a crucial step toward reliability and effective incident response. If there's one stage in the data engineering lifecycle where monitoring is critical, it's in the ingestion stage. Weak or nonexistent monitoring means the pipelines may or may not be working. Referring back to our earlier discussion on time, be sure to track the various aspects of time—event creation, ingestion, process, and processing times. Your data pipelines should predictably process data in batches or streams. We've seen countless examples of reports and ML models generated from stale data. In one extreme case, an ingestion pipeline failure wasn't detected for six months. (One might question the concrete utility of the data in this instance, but that's another matter.) This was very much avoidable through proper monitoring.

What should you monitor? Uptime, latency, and data volumes processed are good places to start. If an ingestion job fails, how will you respond? In general, build monitoring into your pipelines from the beginning rather than waiting for deployment.

Monitoring is key, and knowledge of the behavior of the upstream systems you depend on and how they generate data. You should be aware of the number of events generated per time interval you're concerned with (events/minute, events/second, and so on) and the average size of each event. Your data pipeline should handle both the frequency and size of the events you're ingesting.

This also applies to third-party services. In the case of these services, what you've gained in terms of lean operational efficiencies (reduced headcount) is replaced by systems you depend on being outside of your control. If you're using a third-party service (cloud, data integration service, etc.), how will you be alerted if there's an outage? What's your response plan if a service you depend on suddenly goes offline?

Sadly, no universal response plan exists for third-party failures. If you can fail over to other servers, preferably in another zone or region, definitely set this up.

If your data ingestion processes are built internally, do you have the proper testing and deployment automation to ensure that the code functions in production? And if the code is buggy or fails, can you roll it back to a working version?

Data-quality tests

We often refer to data as a silent killer. If quality, valid data is the foundation of success in today's businesses, using bad data to make decisions is much worse than having no data. Bad data has caused untold damage to businesses; these data disasters are sometimes called *datastrophes*.[1]

Data is entropic; it often changes in unexpected ways without warning. One of the inherent differences between DevOps and DataOps is that we expect software regressions only when we deploy changes, while data often presents regressions independently because of events outside our control.

DevOps engineers are typically able to detect problems by using binary conditions. Has the request failure rate breached a certain threshold? How about response latency? In the data space, regressions often manifest as subtle statistical distortions. Is a change in search-term statistics a result of customer behavior? Of a spike in bot traffic that has escaped the net? Of a site test tool deployed in some other part of the company?

Like system failures in DevOps, some data regressions are immediately visible. For example, in the early 2000s, Google provided search terms to websites when users arrived from search. In 2011, Google began withholding this information in some cases to protect user privacy better. Analysts quickly saw "not provided" bubbling to the tops of their reports.[2]

The truly dangerous data regressions are silent and can come from inside or outside a business. Application developers may change the meaning of database fields without adequately communicating with data teams. Changes to data from third-party sources may go unnoticed. In the best-case scenario, reports break in obvious ways. Often business metrics are distorted unbeknownst to decision makers.

Whenever possible, work with software engineers to fix data-quality issues at the source. It's surprising how many data-quality issues can be handled by respecting basic best practices in software engineering, such as logs to capture the history of data changes, checks (nulls, etc.), and exception handling (try, catch, etc.).

Traditional data testing tools are generally built on simple binary logic. Are nulls appearing in a non-nullable field? Are new, unexpected items showing up in a categorical column? Statistical data testing is a new realm, but one that is likely to grow dramatically in the next five years.

Orchestration

Ingestion generally sits at the beginning of a large and complex data graph; since ingestion is the first stage of the data engineering lifecycle, ingested data will flow into many more data processing steps, and data from many sources will commingle in complex ways. As we've emphasized throughout this book, orchestration is a crucial process for coordinating these steps.

Organizations in an early stage of data maturity may choose to deploy ingestion processes as simple scheduled cron jobs. However, it is crucial to recognize that this approach is brittle and can slow the velocity of data engineering deployment and development.

As data pipeline complexity grows, true orchestration is necessary. By true orchestration, we mean a system capable of scheduling complete task graphs rather than individual tasks. An orchestration can start each ingestion task at the appropriate scheduled time. Downstream processing and transform steps begin as ingestion tasks are completed. Further downstream, processing steps lead to additional processing steps.

Software Engineering

The ingestion stage of the data engineering lifecycle is engineering intensive. This stage sits at the edge of the data engineering domain and often interfaces with external systems, where software and data engineers have to build a variety of custom plumbing.

Behind the scenes, ingestion is incredibly complicated, often with teams operating open source frameworks like Kafka or Pulsar, or some of the biggest tech companies running their own forked or homegrown ingestion solutions. As discussed in this chapter, managed data connectors have simplified the ingestion process, such as Fivetran, Matillion, and Airbyte. Data engineers should take advantage of the best available tools—primarily, managed tools and services that do a lot of the heavy lifting for you—and develop high software development competency in areas where it matters. It pays to use proper version control and code review processes and implement appropriate tests even for any ingestion-related code.

When writing software, your code needs to be decoupled. Avoid writing monolithic systems with tight dependencies on the source or destination systems.

Conclusion

In your work as a data engineer, ingestion will likely consume a significant part of your energy and effort. At the heart, ingestion is plumbing, connecting pipes to other pipes, ensuring that data flows consistently and securely to its destination. At times, the minutiae of ingestion may feel tedious, but the exciting data applications (e.g., analytics and ML) cannot happen without it.

As we've emphasized, we're also in the midst of a sea change, moving from batch toward streaming data pipelines. This is an opportunity for data engineers to discover interesting applications for streaming data, communicate these to the business, and deploy exciting new technologies.

Additional Resources

- Google Cloud's "Streaming Pipelines" web page
- Microsoft's "Snapshot Window (Azure Stream Analytics)" documentation

- Airbyte's "Connections and Sync Modes" web page

---

1 Andy Petrella, "Datastrophes," *Medium*, March 1, 2021, *https://oreil.ly/h6FRW*.

2 Danny Sullivan, "Dark Google: One Year Since Search Terms Went 'Not Provided,'" *MarTech*, October 19, 2012, https://oreil.ly/Fp8ta

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.

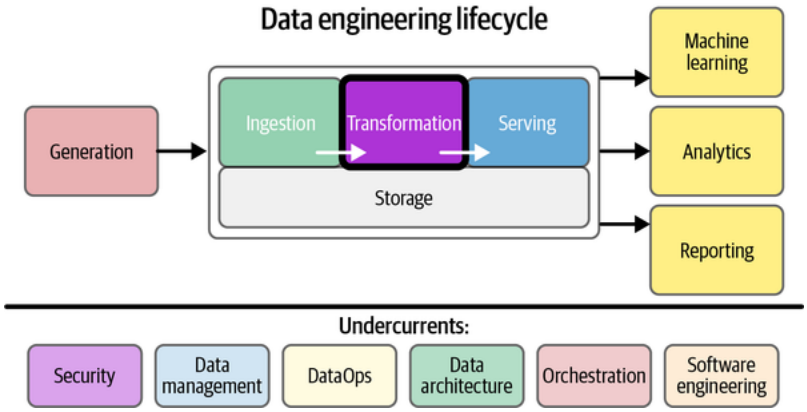

Figure 8-1. Transformations allow us to create value from data

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 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.

TCL

TCL stands for *transaction control language*. As the name suggests, 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.

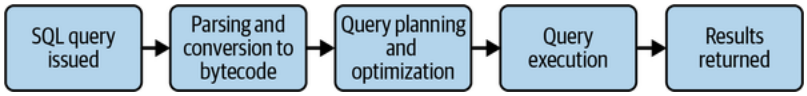

Figure 8-2. The life of a SQL query in a database

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 *pre-join* 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 pre-joined 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 pre-joining tables for the most common analytics and data science use cases. We can simply create pre-joined 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.[1] 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 $5 \times 10 = 50$ 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.[2]

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,[3] 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 help reduce pressure on your database while providing 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.[4] 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.

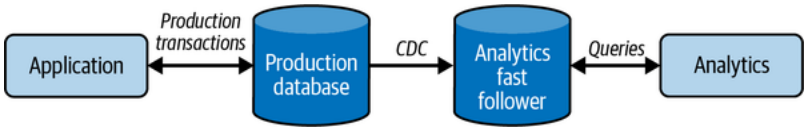

Figure 8-3. CDC with a fast-follower analytics database

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.

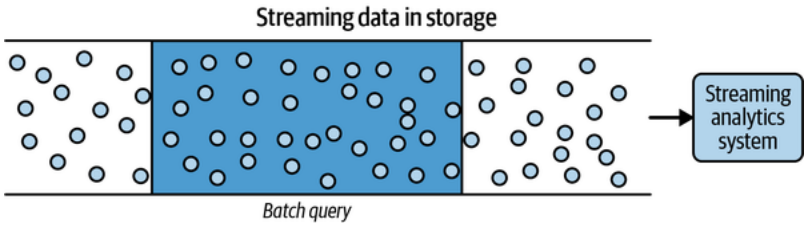

Figure 8-4. The Kappa architecture is built around streaming storage and ingest systems

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 use, 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.

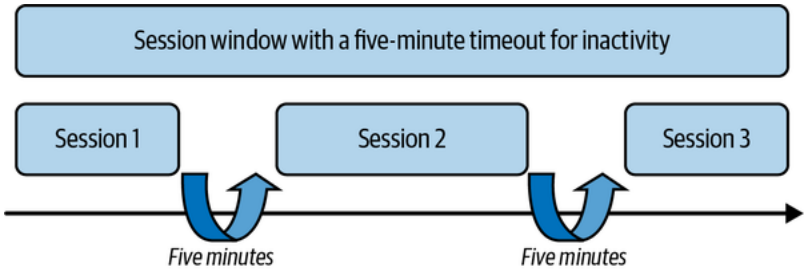

Figure 8-5. Session window with a five-minute timeout for inactivity

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.

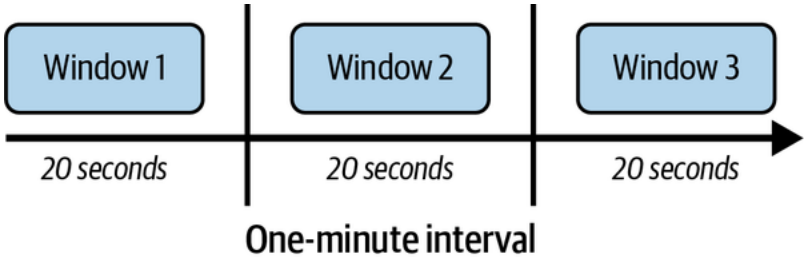

Figure 8-6. Tumbling/fixed window

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. 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. Sliding windows are also known as *hopping windows*.

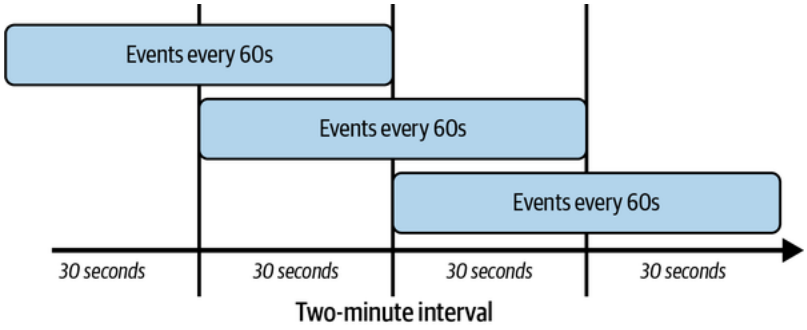

Figure 8-7. Sliding windows

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.

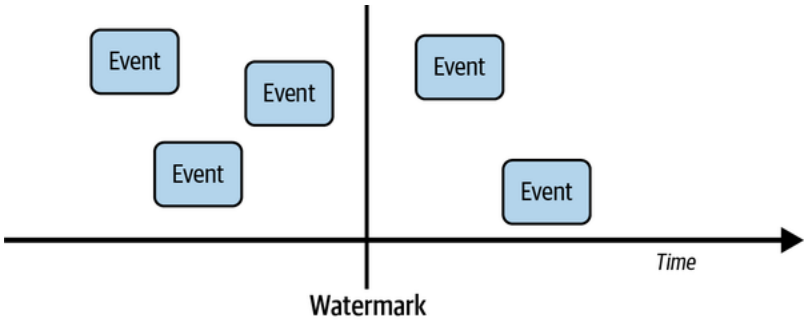

Figure 8-8. Watermark acting as a threshold for late-arriving data

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.

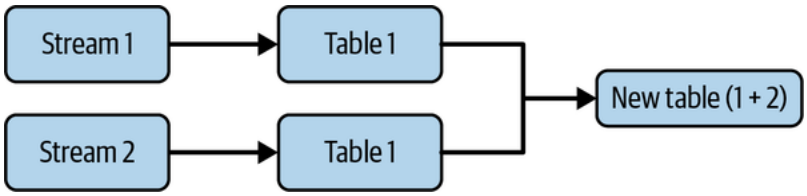

Figure 8-9. Joining two tables fed by streams

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.

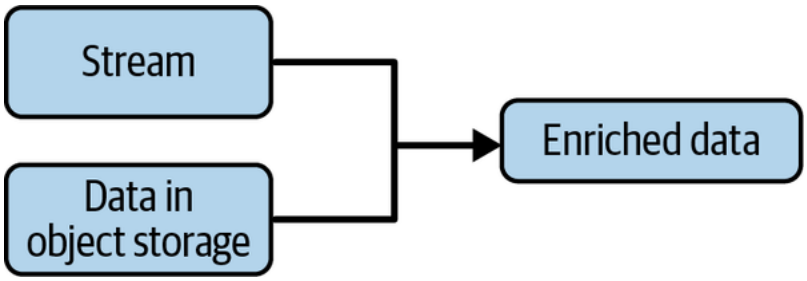

Figure 8-10. In this example, a stream is enriched with data residing in object storage, resulting in a new enriched dataset

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).[5]

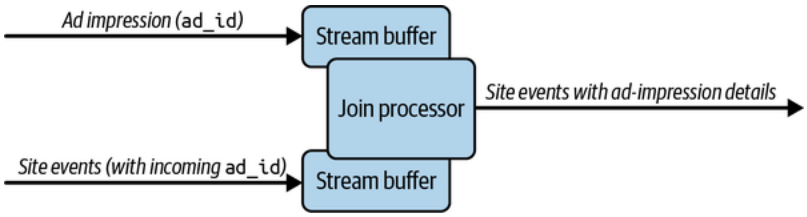

Figure 8-11. An architecture to join streams buffers each stream and joins events if related events are found during the buffer retention interval

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. On the other hand, we believe that new paradigms are required to truly embrace the needs of streaming data and ML. In this section, we survey current 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**
>
> 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.

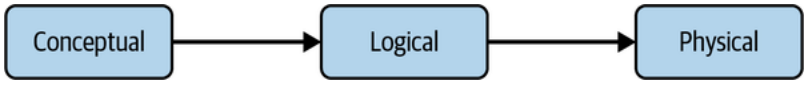

Figure 8-12. The continuum of data models: conceptual, logical, and physical

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 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.[6]

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:[7]

- 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 with out repeats or nested data

OrderID	Sku	Price	Quantity	ProductName	CustomerID	CustomerName	Orc

100	1	50	1	Thingamajig	5	Joe Reis	202
100	2	25	2	Whatchamacallit	5	Joe Reis	202

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 hallargersample

OrderID	Sku	Price	Quantity	ProductName	CustomerID	CustomerName	Orc
100	1	50	1	Thingamajig	5	Joe Reis	202
100	2	25	2	Whatchamacallit	5	Joe Reis	202
101	3	75	1	Whozeewhatzit	7	Matt Housley	202
102	1	50	1	Thingamajig	7	Matt Housley	202

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	Cu
100	1	1	50	1	Thingamajig	5	Joe

100	2	2	25	2	Whatchamacallit	5	Joe
101	1	3	75	1	Whozeewhatzit	7	Mat
102	1	1	50	1	Thingamajig	7	Mat

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 1990. 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:

> *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.*[8]

The four critical parts of a data warehouse mean the following:

Subject-oriented

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

Nonvolatile

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

Integrated

: Data from disparate sources is consolidated and normalized.

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:

> *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.*[9]

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.

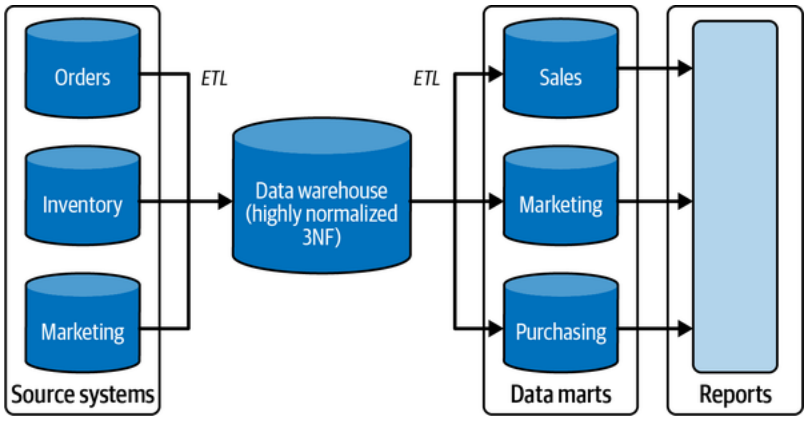

Figure 8-13. An ecommerce data warehouse

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".

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."[10]

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). 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).[11] Let's look at facts, dimensions, and star schemas.