Chapter 7. Ingestion

You've learned about the various source systems you'll likely encounter as a data engineer and about ways to store data. Let's now turn our attention to the patterns and choices that apply to ingesting data from various source systems. In this chapter, we discuss data ingestion (see Figure 7-1), the key engineering considerations for the ingestion phase, the major patterns for batch and streaming ingestion, technologies you'll encounter, whom you'll work with as you develop your data ingestion pipeline, and how the undercurrents feature in the ingestion phase.

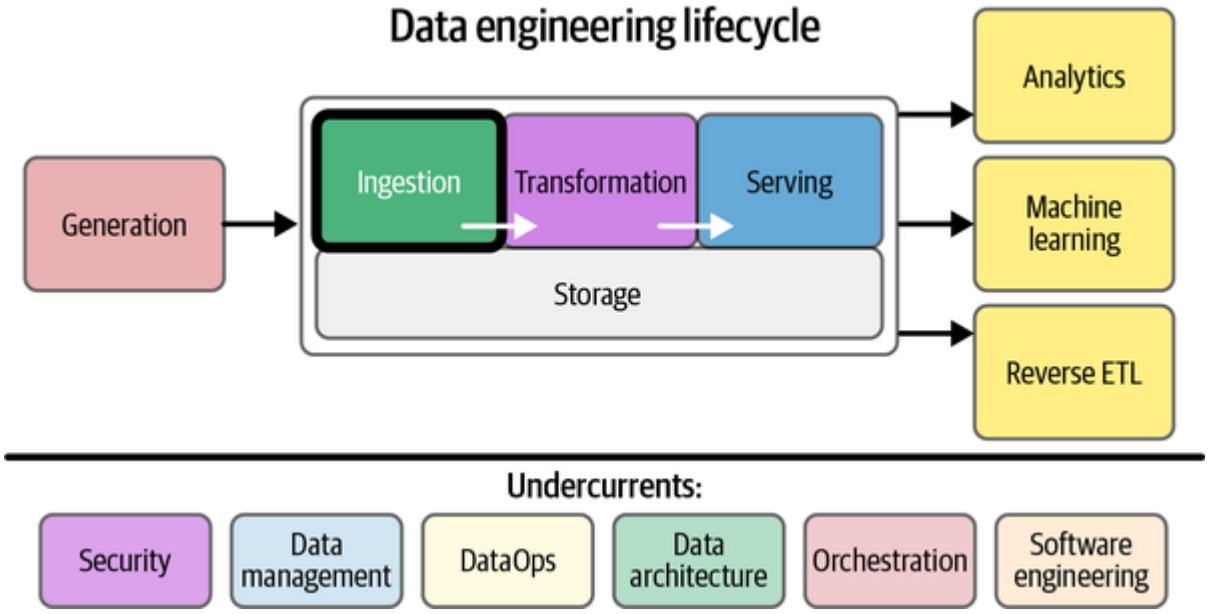

Figure 7-1. To begin processing data, we must ingest it

What Is Data Ingestion?

Data ingestion is the process of moving data from one place to another. Data ingestion implies data movement from source systems into storage in the data engineering lifecycle, with ingestion as an intermediate step (Figure 7-2).

Figure 7-2. Data from system 1 is ingested into system 2

It's worth quickly contrasting data ingestion with data integration. Whereas *data ingestion* is data movement from point A to B, *data integration* combines data from disparate sources into a new dataset. For example, you can use data integration to combine data from a CRM system, advertising analytics data, and web analytics to create a user profile, which is saved to your data warehouse. Furthermore, using reverse ETL, you can send this newly created user profile *back* to your CRM so salespeople can use the data for prioritizing leads. We describe data integration more fully in Chapter 8, where we discuss data transformations; reverse ETL is covered in Chapter 9.

We also point out that data ingestion is different from *internal ingestion* within a system. Data stored in a database is copied from one table to another, or data in a stream is temporarily cached. We consider this another part of the general data transformation process covered in Chapter 8.

Data Pipelines Defined

Data pipelines begin in source systems, but ingestion is the stage where data engineers begin actively designing data pipeline activities. In the data engineering space, a good deal of ceremony occurs around data movement and processing patterns, with established patterns such as ETL, newer patterns such as ELT, and new names for long-established practices (reverse ETL) and data sharing.

All of these concepts are encompassed in the idea of a *data pipeline*. It is essential to understand the details of these various patterns and know that a modern data pipeline includes all of them. As the world moves away from a traditional monolithic approach with rigid constraints on data movement, and toward an open ecosystem of cloud services that are assembled like LEGO bricks to realize products, data engineers prioritize using the right tools to accomplish the desired outcome over adhering to a narrow philosophy of data movement.

In general, here's our definition of a data pipeline:

> A data pipeline is the combination of architecture, systems, and processes that move data through the stages of the data engineering lifecycle.

Our definition is deliberately fluid—and intentionally vague—to allow data engineers to plug in whatever they need to accomplish the task at hand. A data pipeline could be a traditional ETL system, where data is ingested from an on-premises transactional system, passed through a monolithic processor, and written into a data warehouse. Or it could be a cloud-based data pipeline that pulls data from 100 sources, combines it into 20 wide tables, trains five other ML models, deploys them into production, and monitors ongoing performance. A data pipeline should be flexible enough to fit any needs along the data engineering lifecycle.

Let's keep this notion of data pipelines in mind as we proceed through this chapter.

Key Engineering Considerations for the Ingestion Phase

When preparing to architect or build an ingestion system, here are some primary considerations and questions to ask yourself related to data ingestion:

- What's the use case for the data I'm ingesting?
- Can I reuse this data and avoid ingesting multiple versions of the same dataset?
- Where is the data going? What's the destination?
- How often should the data be updated from the source?
- What is the expected data volume?
- What format is the data in? Can downstream storage and transformation accept this format?
- Is the source data in good shape for immediate downstream use? That is, is the data of good quality? What post-processing is required to serve it? What are data-quality risks (e.g., could bot traffic to a website contaminate the data)?
- Does the data require in-flight processing for downstream ingestion if the data is from a streaming source?

These questions undercut batch and streaming ingestion and apply to the underlying architecture you'll create, build, and maintain. Regardless of how often the data is ingested, you'll want to consider these factors when designing your ingestion architecture:

- Bounded versus unbounded
- Frequency
- Synchronous versus asynchronous
- Serialization and deserialization
- Throughput and scalability
- Reliability and durability
- Payload
- Push versus pull versus poll patterns

Let's look at each of these.

Bounded Versus Unbounded Data

As you might recall from Chapter 3, data comes in two forms: bounded and unbounded (Figure 7-3). *Unbounded data* is data as it exists in reality, as events happen, either sporadically or continuously, ongoing and flowing. *Bounded data* is a convenient way of bucketing data across some sort of boundary, such as time.

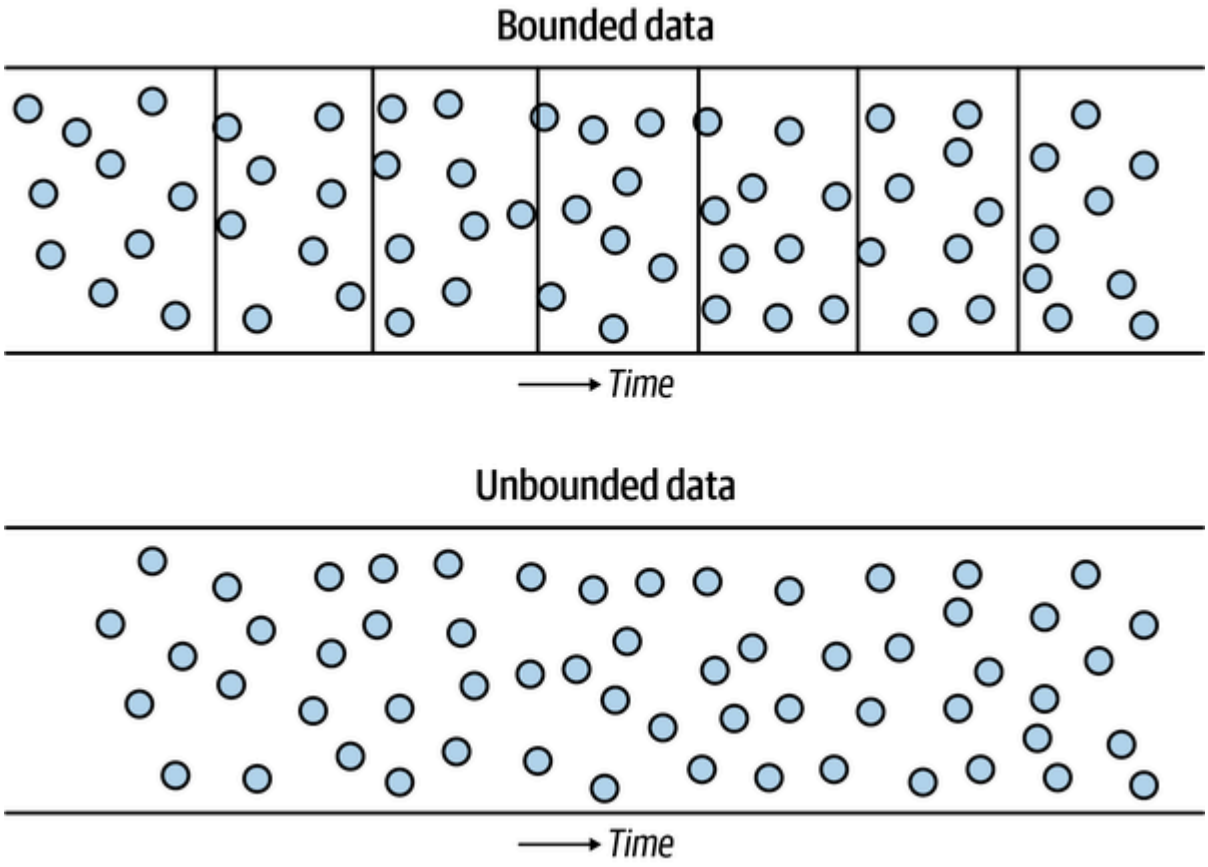

Figure 7-3. Bounded versus unbounded data

Let us adopt this mantra: *All data is unbounded until it's bounded.* Like many mantras, this one is not precisely accurate 100% of the time. The grocery list that I scribbled this afternoon is bounded data. I wrote it as a stream of consciousness (unbounded data) onto a piece of scrap paper, where the thoughts now exist as a list of things (bounded data) I need to buy at the grocery store. However, the idea is correct for practical purposes for the vast majority of data you'll handle in a business context. For

example, an online retailer will process customer transactions 24 hours a day until the business fails, the economy grinds to a halt, or the sun explodes.

Business processes have long imposed artificial bounds on data by cutting discrete batches. Keep in mind the true unboundedness of your data; streaming ingestion systems are simply a tool for preserving the unbounded nature of data so that subsequent steps in the lifecycle can also process it continuously.

Frequency

One of the critical decisions that data engineers must make in designing data-ingestion processes is the data-ingestion frequency. Ingestion processes can be batch, micro-batch, or real-time.

Ingestion frequencies vary dramatically from slow to fast (Figure 7-4). On the slow end, a business might ship its tax data to an accounting firm once a year. On the faster side, a CDC system could retrieve new log updates from a source database once a minute. Even faster, a system might continuously ingest events from IoT sensors and process these within seconds. Data-ingestion frequencies are often mixed in a company, depending on the use case and technologies.

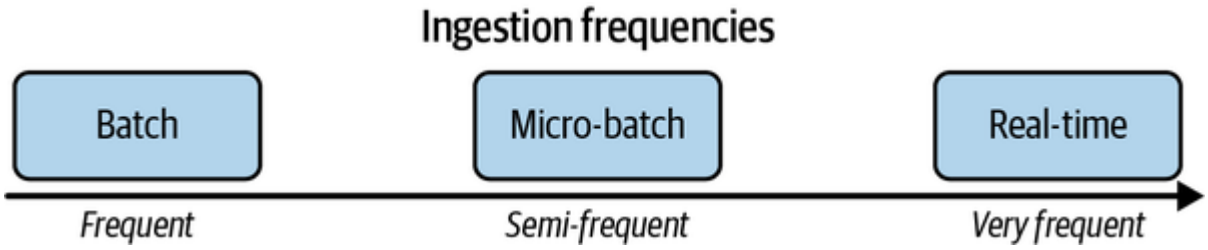

Figure 7-4. The spectrum batch to real-time ingestion frequencies

We note that “real-time” ingestion patterns are becoming increasingly common. We put “real-time” in quotation marks because no ingestion system is genuinely real-time. Any database, queue or pipeline has inherent latency in delivering data to a target system. It is more accurate to speak of *near real-time*, but we often use *real-time* for brevity. The near real-time pattern generally does away with an explicit update frequency; events are processed in the pipeline either one by one as they arrive or in micro-batches (i.e., batches over concise time intervals). For this book, we will use *real-time* and *streaming* interchangeably.

Even with a streaming data-ingestion process, batch processing downstream is relatively standard. At the time of this writing, ML models are typically trained on a batch basis, although continuous online training is becoming more prevalent. Rarely do data engineers have the option to build a purely near real-time pipeline with no batch components. Instead, they choose where batch boundaries will occur —i.e., the data engineering lifecycle data will be broken into batches. Once data reaches a batch process, the batch frequency becomes a bottleneck for all downstream processing.

In addition, streaming systems are the best fit for many data source types. In IoT applications, the typical pattern is for each sensor to write events or measurements to streaming systems as they happen. While this data can be written directly into a database, a streaming ingestion platform such as Amazon Kinesis or Apache Kafka is a better fit for the application. Software applications can adopt similar patterns by writing events to a message queue as they happen rather than waiting for an extraction process to pull events and state information from a backend database. This pattern works exceptionally well for event-driven architectures already exchanging messages through queues. And again, streaming architectures generally coexist with batch processing.

Synchronous Versus Asynchronous Ingestion

With *synchronous ingestion*, the source, ingestion, and destination have complex dependencies and are tightly coupled. As you can see in Figure 7-5, each stage of the data engineering lifecycle has

processes A, B, and C directly dependent upon one another. If process A fails, processes B and C cannot start; if process B fails, process C doesn't start. This type of synchronous workflow is common in older ETL systems, where data extracted from a source system must then be transformed before being loaded into a data warehouse. Processes downstream of ingestion can't start until all data in the batch has been ingested. If the ingestion or transformation process fails, the entire process must be rerun.

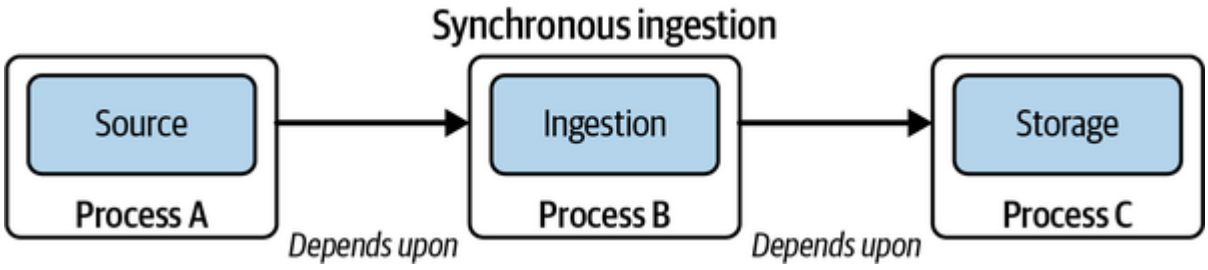

Figure 7-5. A synchronous ingestion process runs as discrete batch steps

Here's a mini case study of how *not* to design your data pipelines. At one company, the transformation process itself was a series of dozens of tightly coupled synchronous workflows, with the entire process taking over 24 hours to finish. If any step of that transformation pipeline failed, the whole transformation process had to be restarted from the beginning! In this instance, we saw process after process fail, and because of nonexistent or cryptic error messages, fixing the pipeline was a game of whack-a-mole that took over a week to diagnose and cure. Meanwhile, the business didn't have updated reports during that time. People weren't happy.

With *asynchronous ingestion*, dependencies can now operate at the level of individual events, much as they would in a software backend built from microservices (Figure 7-6). Individual events become available in storage as soon as they are ingested individually. Take the example of a web application on AWS that emits events into Amazon Kinesis Data Streams (here acting as a buffer). The stream is read by Apache Beam, which parses and enriches events, and then forwards them to a second Kinesis stream; Kinesis Data Firehose rolls up events and writes objects to Amazon S3.

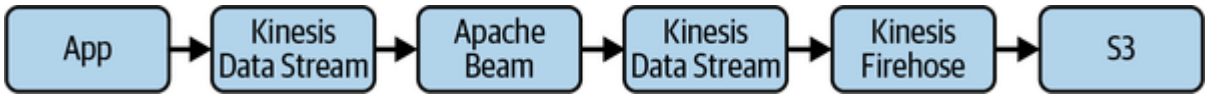

Figure 7-6. Asynchronous processing of an event stream in AWS

The big idea is that rather than relying on asynchronous processing, where a batch process runs for each stage as the input batch closes and certain time conditions are met, each stage of the asynchronous pipeline can process data items as they become available in parallel across the Beam cluster. The processing rate depends on available resources. The Kinesis Data Stream acts as the shock absorber, moderating the load so that event rate spikes will not overwhelm downstream processing. Events will move through the pipeline quickly when the event rate is low, and any backlog has cleared. Note that we could modify the scenario and use a Kinesis Data Stream for storage, eventually extracting events to S3 before they expire out of the stream.

Serialization and Deserialization

Moving data from source to destination involves serialization and deserialization. As a reminder, *serialization* means encoding the data from a source and preparing data structures for transmission and intermediate storage stages.

When ingesting data, ensure that your destination can deserialize the data it receives. We've seen data ingested from a source but then sitting inert and unusable in the destination because the data cannot be properly deserialized. See the more extensive discussion of serialization in Appendix A.

Throughput and Scalability

In theory, your ingestion should never be a bottleneck. In practice, ingestion bottlenecks are pretty standard. Data throughput and system scalability become critical as your data volumes grow and requirements change. Design your systems to scale and shrink to flexibly match the desired data throughput.

Where you're ingesting data from matters a lot. If you're receiving data as it's generated, will the upstream system have any issues that might impact your downstream ingestion pipelines? For example, suppose a source database goes down. When it comes back online and attempts to backfill the lapsed data loads, will your ingestion be able to keep up with this sudden influx of backlogged data?

Another thing to consider is your ability to handle bursty data ingestion. Data generation rarely happens at a constant rate and often ebbs and flows. Built-in buffering is required to collect events during rate spikes to prevent data from getting lost. Buffering bridges the time while the system scales and allows storage systems to accommodate bursts even in a dynamically scalable system.

Whenever possible, use managed services that handle the throughput scaling for you. While you can manually accomplish these tasks by adding more servers, shards, or workers, often this isn't value-added work, and there's a good chance you'll miss something. Much of this heavy lifting is now automated. Don't reinvent the data ingestion wheel if you don't have to.

Reliability and Durability

Reliability and durability are vital in the ingestion stages of data pipelines. *Reliability* entails high uptime and proper failover for ingestion systems. *Durability* entails making sure that data isn't lost or corrupted.

Some data sources (e.g., IoT devices and caches) may not retain data if it is not correctly ingested. Once lost, it is gone for good. In this sense, the *reliability* of ingestion systems leads directly to the *durability* of generated data. If data is ingested, downstream processes can theoretically run late if they break temporarily.

Our advice is to evaluate the risks and build an appropriate level of redundancy and self-healing based on the impact and cost of losing data. Reliability and durability have both direct and indirect costs. For example, will your ingestion process continue if an AWS zone goes down? How about a whole region? How about the power grid or the internet? Of course, nothing is free. How much will this cost you? You might be able to build a highly redundant system and have a team on call 24 hours a day to handle outages. This also means your cloud and labor costs become prohibitive (direct costs), and the ongoing work takes a significant toll on your team (indirect costs). There's no single correct answer, and you need to evaluate the costs and benefits of your reliability and durability decisions.

Don't assume that you can build a system that will reliably and durably ingest data in every possible scenario. Even the nearly infinite budget of the US federal government can't guarantee this. In many extreme scenarios, ingesting data actually won't matter. There will be little to ingest if the internet goes down, even if you build multiple air-gapped data centers in underground bunkers with independent power. Continually evaluate the trade-offs and costs of reliability and durability.

Payload

A *payload* is the dataset you're ingesting and has characteristics such as kind, shape, size, schema and data types, and metadata. Let's look at some of these characteristics to understand why this matters.

Kind

The *kind* of data you handle directly impacts how it's dealt with downstream in the data engineering lifecycle. Kind consists of type and format. Data has a type—tabular, image, video, text, etc. The type directly influences the data format or the way it is expressed in bytes, names, and file extensions. For example, a tabular kind of data may be in formats such as CSV or Parquet, with each of these formats having different byte patterns for serialization and deserialization. Another kind of data is an image, which has a format of JPG or PNG and is inherently unstructured.

Shape

Every payload has a *shape* that describes its dimensions. Data shape is critical across the data engineering lifecycle. For instance, an image's pixel and red, green, blue (RGB) dimensions are necessary for training deep learning models. As another example, if you're trying to import a CSV file into a database table, and your CSV has more columns than the database table, you'll likely get an error during the import process. Here are some examples of the shapes of various kinds of data:

Tabular

: The number of rows and columns in the dataset, commonly expressed as *M* rows and *N* columns

Semistructured JSON

: The key-value pairs and nesting depth occur with subelements

Unstructured text

: Number of words, characters, or bytes in the text body

Images

: The width, height, and RGB color depth (e.g., 8 bits per pixel)

Uncompressed audio

: Number of channels (e.g., two for stereo), sample depth (e.g., 16 bits per sample), sample rate (e.g., 48 kHz), and length (e.g., 10,003 seconds)

Size

The *size* of the data describes the number of bytes of a payload. A payload may range in size from single bytes to terabytes and larger. To reduce the size of a payload, it may be compressed into various formats such as ZIP and TAR (see the discussion of compression in Appendix A).

A massive payload can also be split into chunks, which effectively reduces the size of the payload into smaller subsections. When loading a huge file into a cloud object storage or data warehouse, this is a common practice as the small individual files are easier to transmit over a network (especially if they're compressed). The smaller chunked files are sent to their destination and then reassembled after all data has arrived.

Schema and data types

Many data payloads have a schema, such as tabular and semistructured data. As mentioned earlier in this book, a schema describes the fields and types of data within those fields. Other data, such as unstructured text, images, and audio, will not have an explicit schema or data types. However, they might come with technical file descriptions on shape, data and file format, encoding, size, etc.

Although you can connect to databases in various ways (such as file export, CDC, JDBC/ODBC), the connection is easy. The great engineering challenge is understanding the underlying schema. Applications organize data in various ways, and engineers need to be intimately familiar with the organization of the data and relevant update patterns to make sense of it. The problem has been somewhat exacerbated by the popularity of object-relational mapping (ORM), which automatically generates schemas based on object structure in languages such as Java or Python. Natural structures in an object-oriented language often map to something messy in an operational database. Data engineers may need to familiarize themselves with the class structure of application code.

Schema is not only for databases. As we've discussed, APIs present their schema complications. Many vendor APIs have friendly reporting methods that prepare data for analytics. In other cases, engineers are not so lucky. The API is a thin wrapper around underlying systems, requiring engineers to understand application internals to use the data.

Much of the work associated with ingesting from source schemas happens in the data engineering lifecycle transformation stage, which we discuss in Chapter 8. We've placed this discussion here because data engineers need to begin studying source schemas as soon they plan to ingest data from a new source.

Communication is critical for understanding source data, and engineers also have the opportunity to reverse the flow of communication and help software engineers improve data where it is produced. Later in this chapter, we'll return to this topic in "Whom You'll Work With".

Detecting and handling upstream and downstream schema changes

Changes in schema frequently occur in source systems and are often well out of data engineers' control. Examples of schema changes include the following:

- Adding a new column
- Changing a column type
- Creating a new table
- Renaming a column

It's becoming increasingly common for ingestion tools to automate the detection of schema changes and even auto-update target tables. Ultimately, this is something of a mixed blessing. Schema changes can still break pipelines downstream of staging and ingestion.

Engineers must still implement strategies to respond to changes automatically and alert on changes that cannot be accommodated automatically. Automation is excellent, but the analysts and data scientists who rely on this data should be informed of the schema changes that violate existing assumptions. Even if automation can accommodate a change, the new schema may adversely affect the performance of reports and models. Communication between those making schema changes and those impacted by these changes is as important as reliable automation that checks for schema changes.

Schema registries

In streaming data, every message has a schema, and these schemas may evolve between producers and consumers. A *schema registry* is a metadata repository used to maintain schema and data type integrity in the face of constantly changing schemas. Schema registries can also track schema versions and history. It describes the data model for messages, allowing consistent serialization and deserialization between producers and consumers. Schema registries are used in most major data tools and clouds.

Metadata

In addition to the apparent characteristics we've just covered, a payload often contains metadata, which we first discussed in Chapter 2. Metadata is data about data. Metadata can be as critical as the data itself. One of the significant limitations of the early approach to the data lake—or data swamp, which could become a data superfund site—was a complete lack of attention to metadata. Without a detailed description of the data, it may be of little value. We've already discussed some types of metadata (e.g., schema) and will address them many times throughout this chapter.

Push Versus Pull Versus Poll Patterns

We mentioned push versus pull when we introduced the data engineering lifecycle in Chapter 2. A *push* strategy (Figure 7-7) involves a source system sending data to a target, while a *pull* strategy (Figure 7-8) entails a target reading data directly from a source. As we mentioned in that discussion, the lines between these strategies are blurry.

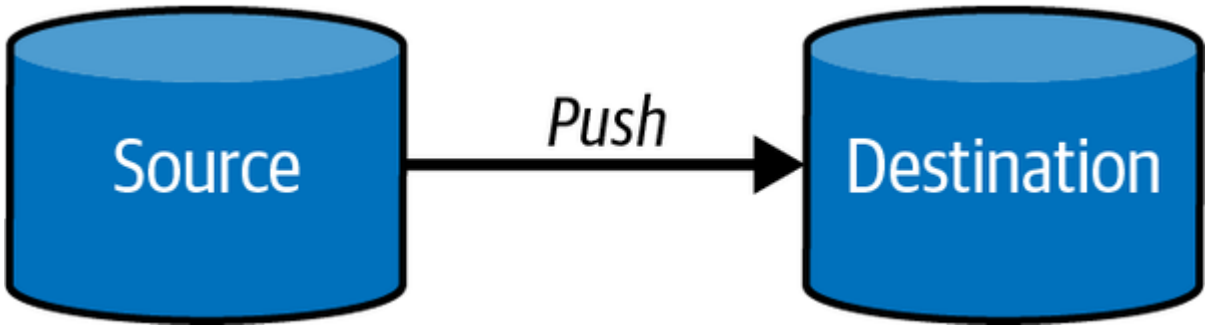

Figure 7-7. Pushing data from source to destination

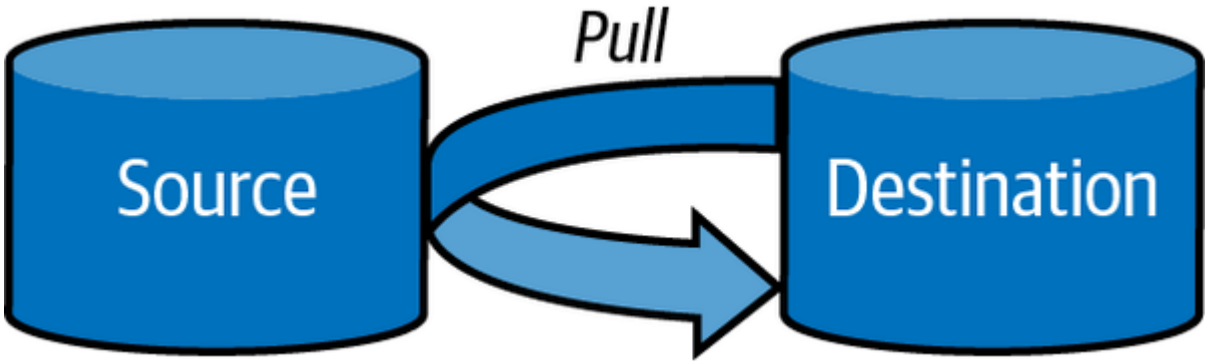

Figure 7-8. A destination pulling data from a source

Another pattern related to pulling is *polling* for data (Figure 7-9). Polling involves periodically checking a data source for any changes. When changes are detected, the destination pulls the data as it would in a regular pull situation.

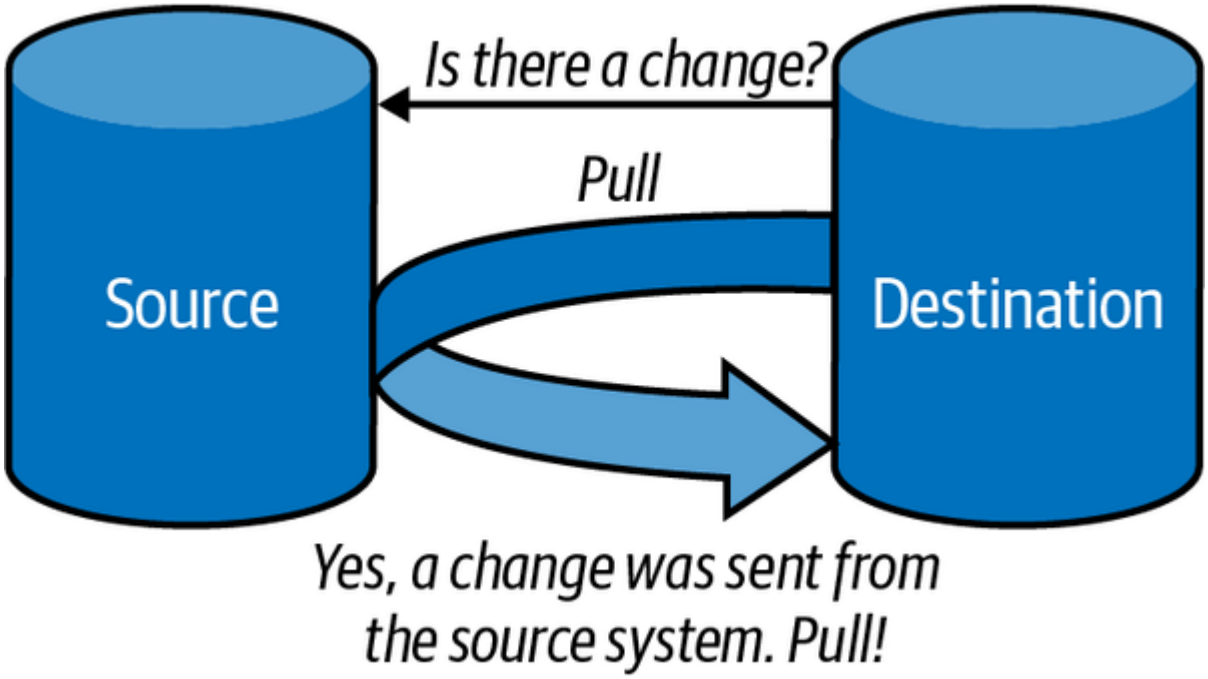

Figure 7-9. Polling for changes in a source system

Batch Ingestion Considerations

Batch ingestion, which involves processing data in bulk, is often a convenient way to ingest data. This means that data is ingested by taking a subset of data from a source system, based either on a time interval or the size of accumulated data (Figure 7-10).

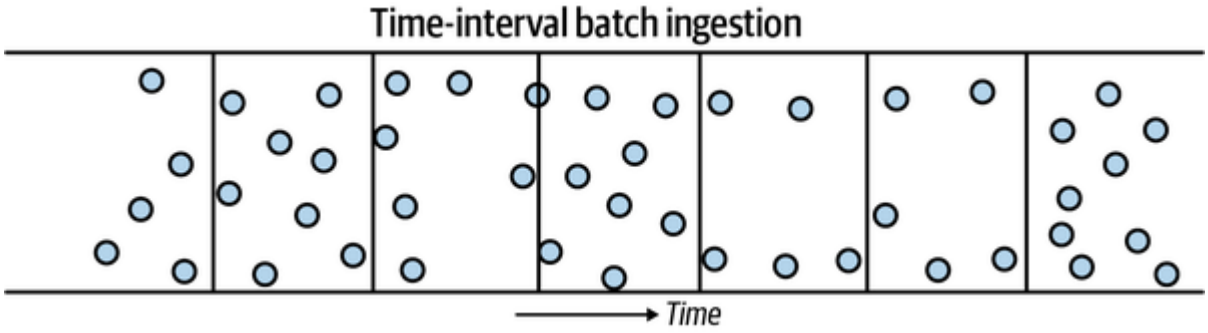

Figure 7-10. Time-interval batch ingestion

Time-interval batch ingestion is widespread in traditional business ETL for data warehousing. This pattern is often used to process data once a day, overnight during off-hours, to provide daily reporting, but other frequencies can also be used.

Size-based batch ingestion (Figure 7-11) is quite common when data is moved from a streaming-based system into object storage; ultimately, you must cut the data into discrete blocks for future processing in a data lake. Some size-based ingestion systems can break data into objects based on various criteria, such as the size in bytes of the total number of events.

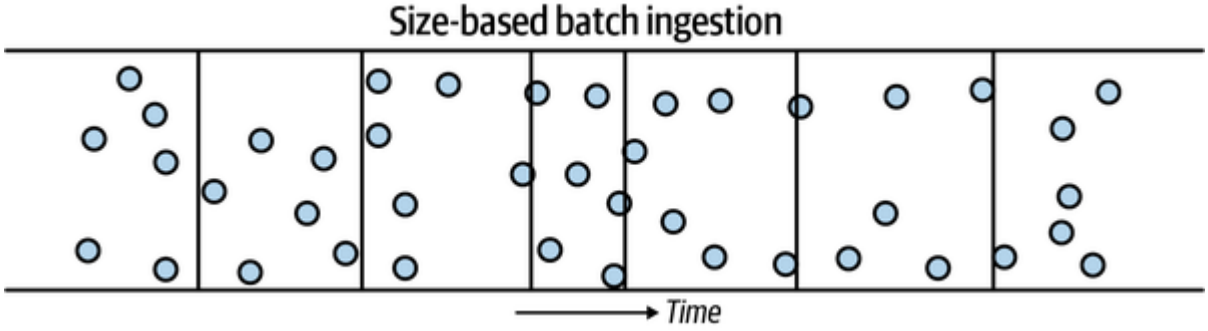

Figure 7-11. Size-based batch ingestion

Some commonly used batch ingestion patterns, which we discuss in this section, include the following:

- Snapshot or differential extraction
- File-based export and ingestion
- ETL versus ELT
- Inserts, updates, and batch size
- Data migration

Snapshot or Differential Extraction

Data engineers must choose whether to capture full snapshots of a source system or differential (sometimes called *incremental*) updates. With *full snapshots*, engineers grab the entire current state of the source system on each update read. With the *differential update* pattern, engineers can pull only the updates and changes since the last read from the source system. While differential updates are ideal for minimizing network traffic and target storage usage, full snapshot reads remain extremely common because of their simplicity.

File-Based Export and Ingestion

Data is quite often moved between databases and systems using files. Data is serialized into files in an exchangeable format, and these files are provided to an ingestion system. We consider file-based export to be a *push-based* ingestion pattern. This is because data export and preparation work is done on the source system side.

File-based ingestion has several potential advantages over a direct database connection approach. It is often undesirable to allow direct access to backend systems for security reasons. With file-based ingestion, export processes are run on the data-source side, giving source system engineers complete control over what data gets exported and how the data is preprocessed. Once files are done, they can be provided to the target system in various ways. Common file-exchange methods are object storage, secure file transfer protocol (SFTP), electronic data interchange (EDI), or secure copy (SCP).

ETL Versus ELT

Chapter 3 introduced ETL and ELT, both extremely common ingestion, storage, and transformation patterns you'll encounter in batch workloads. The following are brief definitions of the extract and load parts of ETL and ELT:

Extract

This means getting data from a source system. While *extract* seems to imply *pulling* data, it can also be push based. Extraction may also require reading metadata and schema changes.

Load

Once data is extracted, it can either be transformed (ETL) before loading it into a storage destination or simply loaded into storage for future transformation. When loading data, you should be mindful of the type of system you're loading, the schema of the data, and the performance impact of loading.

We cover ETL and ELT in greater detail in Chapter 8.

Inserts, Updates, and Batch Size

Batch-oriented systems often perform poorly when users attempt to perform many small-batch operations rather than a smaller number of large operations. For example, while it is common to insert one row at a time in a transactional database, this is a bad pattern for many columnar databases, as it forces the creation of many small, suboptimal files and forces the system to run a high number of *create object* operations. Running many small in-place update operations is an even bigger problem because it causes the database to scan each existing column file to run the update.

Understand the appropriate update patterns for the database or data store you're working with. Also, understand that certain technologies are purpose-built for high insert rates. For example, Apache Druid and Apache Pinot can handle high insert rates. SingleStore can manage hybrid workloads that combine OLAP and OLTP characteristics. BigQuery performs poorly on a high rate of vanilla SQL single-row inserts but extremely well if data is fed in through its stream buffer. Know the limits and characteristics of your tools.

Data Migration

Migrating data to a new database or environment is not usually trivial, and data needs to be moved in bulk. Sometimes this means moving data sizes that are hundreds of terabytes or much larger, often involving the migration of specific tables and moving entire databases and systems.

Data migrations probably aren't a regular occurrence as a data engineer, but you should be familiar with them. As is often the case for data ingestion, schema management is a crucial consideration. Suppose you're migrating data from one database system to a different one (say, SQL Server to Snowflake). No matter how closely the two databases resemble each other, subtle differences almost always exist in the way they handle schema. Fortunately, it is generally easy to test ingestion of a sample of data and find schema issues before undertaking a complete table migration.

Most data systems perform best when data is moved in bulk rather than as individual rows or events. File or object storage is often an excellent intermediate stage for transferring data. Also, one of the biggest challenges of database migration is not the movement of the data itself but the movement of data pipeline connections from the old system to the new one.

Be aware that many tools are available to automate various types of data migrations. Especially for large and complex migrations, we suggest looking at these options before doing this manually or writing your own migration solution.

Message and Stream Ingestion Considerations

Ingesting event data is common. This section covers issues you should consider when ingesting events, drawing on topics covered in Chapters 5 and 6.

Schema Evolution

Schema evolution is common when handling event data; fields may be added or removed, or value types might change (say, a string to an integer). Schema evolution can have unintended impacts on your data pipelines and destinations. For example, an IoT device gets a firmware update that adds a new field to the event it transmits, or a third-party API introduces changes to its event payload or countless other scenarios. All of these potentially impact your downstream capabilities.

To alleviate issues related to schema evolution, here are a few suggestions. First, if your event-processing framework has a schema registry (discussed earlier in this chapter), use it to version your schema changes. Next, a dead-letter queue (described in "Error Handling and Dead-Letter Queues") can help you investigate issues with events that are not properly handled. Finally, the low-fidelity route (and the most effective) is regularly communicating with upstream stakeholders about potential schema changes and proactively addressing schema changes with the teams introducing these changes instead of reacting to the receiving end of breaking changes.

Late-Arriving Data

Though you probably prefer all event data to arrive on time, event data might arrive late. A group of events might occur around the same time frame (similar event times), but some might arrive later than others (late ingestion times) because of various circumstances.

For example, an IoT device might be late sending a message because of internet latency issues. This is common when ingesting data. You should be aware of late-arriving data and the impact on downstream systems and uses. Suppose you assume that ingestion or process time is the same as the event time. You may get some strange results if your reports or analysis depend on an accurate portrayal of when events occur. To handle late-arriving data, you need to set a cutoff time for when late-arriving data will no longer be processed.

Ordering and Multiple Delivery

Streaming platforms are generally built out of distributed systems, which can cause some complications. Specifically, messages may be delivered out of order and more than once (at-least-once delivery). See the event-streaming platforms discussion in Chapter 5 for more details.

Replay

Replay allows readers to request a range of messages from the history, allowing you to rewind your event history to a particular point in time. Replay is a key capability in many streaming ingestion platforms and is particularly useful when you need to reingest and reprocess data for a specific time range. For example, RabbitMQ typically deletes messages after all subscribers consume them. Kafka, Kinesis, and Pub/Sub all support event retention and replay.

Time to Live

How long will you preserve your event record? A key parameter is *maximum message retention time*, also known as the *time to live* (TTL). TTL is usually a configuration you'll set for how long you want events to live before they are acknowledged and ingested. Any unacknowledged event that's not ingested after its TTL expires automatically disappears. This is helpful to reduce backpressure and unnecessary event volume in your event-ingestion pipeline.

Find the right balance of TTL impact on our data pipeline. An extremely short TTL (milliseconds or seconds) might cause most messages to disappear before processing. A very long TTL (several weeks or months) will create a backlog of many unprocessed messages, resulting in long wait times.

Let's look at how some popular platforms handle TTL at the time of this writing. Google Cloud Pub/Sub supports retention periods of up to 7 days. Amazon Kinesis Data Streams retention can be turned up to 365 days. Kafka can be configured for indefinite retention, limited by available disk space. (Kafka also supports the option to write older messages to cloud object storage, unlocking virtually unlimited storage space and retention.)

Message Size

Message size is an easily overlooked issue: you must ensure that the streaming framework in question can handle the maximum expected message size. Amazon Kinesis supports a maximum message size of 1 MB. Kafka defaults to this maximum size but can be configured for a maximum of 20 MB or more. (Configurability may vary on managed service platforms.)

Error Handling and Dead-Letter Queues

Sometimes events aren't successfully ingested. Perhaps an event is sent to a nonexistent topic or message queue, the message size may be too large, or the event has expired past its TTL. Events that cannot be ingested need to be rerouted and stored in a separate location called a *dead-letter queue*.

A dead-letter queue segregates problematic events from events that can be accepted by the consumer (Figure 7-12). If events are not rerouted to a dead-letter queue, these erroneous events risk blocking other messages from being ingested. Data engineers can use a dead-letter queue to diagnose why event ingestions errors occur and solve data pipeline problems, and might be able to reprocess some messages in the queue after fixing the underlying cause of errors.

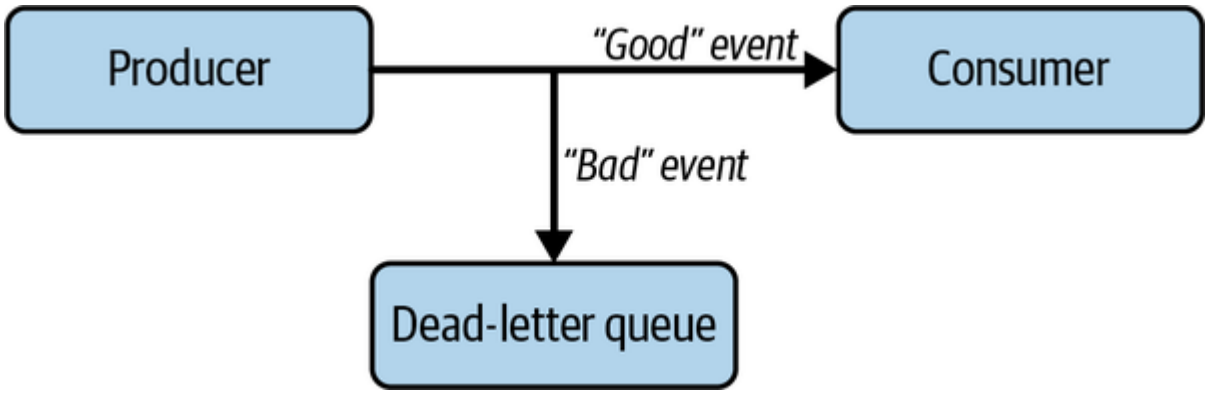

Figure 7-12. "Good" events are passed to the consumer, whereas "bad" events are stored in a dead-letter queue

Consumer Pull and Push

A consumer subscribing to a topic can get events in two ways: push and pull. Let's look at the ways some streaming technologies pull and push data. Kafka and Kinesis support only pull subscriptions. Subscribers read messages from a topic and confirm when they have been processed. In addition to pull subscriptions, Pub/Sub and RabbitMQ support push subscriptions, allowing these services to write messages to a listener.

Pull subscriptions are the default choice for most data engineering applications, but you may want to consider push capabilities for specialized applications. Note that pull-only message ingestion systems can still push if you add an extra layer to handle this.

Location

It is often desirable to integrate streaming across several locations for enhanced redundancy and to consume data close to where it is generated. As a general rule, the closer your ingestion is to where

data originates, the better your bandwidth and latency. However, you need to balance this against the costs of moving data between regions to run analytics on a combined dataset. As always, data egress costs can spiral quickly. Do a careful evaluation of the trade-offs as you build out your architecture.

Ways to Ingest Data

Now that we've described some of the significant patterns underlying batch and streaming ingestion, let's focus on ways you can ingest data. Although we will cite some common ways, keep in mind that the universe of data ingestion practices and technologies is vast and growing daily.

Direct Database Connection

Data can be pulled from databases for ingestion by querying and reading over a network connection. Most commonly, this connection is made using ODBC or JDBC.

ODBC uses a driver hosted by a client accessing the database to translate commands issued to the standard ODBC API into commands issued to the database. The database returns query results over the wire, where the driver receives them and translates them back into a standard form to be read by the client. For ingestion, the application utilizing the ODBC driver is an ingestion tool. The ingestion tool may pull data through many small queries or a single large query.

JDBC is conceptually remarkably similar to ODBC. A Java driver connects to a remote database and serves as a translation layer between the standard JDBC API and the native network interface of the target database. It might seem strange to have a database API dedicated to a single programming language, but there are strong motivations for this. The Java Virtual Machine (JVM) is standard, portable across hardware architectures and operating systems, and provides the performance of compiled code through a just-in-time (JIT) compiler. The JVM is an extremely popular compiling VM for running code in a portable manner.

JDBC provides extraordinary database driver portability. ODBC drivers are shipped as OS and architecture native binaries; database vendors must maintain versions for each architecture/OS version that they wish to support. On the other hand, vendors can ship a single JDBC driver that is compatible with any JVM language (e.g., Java, Scala, Clojure, or Kotlin) and JVM data framework (i.e., Spark.) JDBC has become so popular that it is also used as an interface for non-JVM languages such as Python; the Python ecosystem provides translation tools that allow Python code to talk to a JDBC driver running on a local JVM.

JDBC and ODBC are used extensively for data ingestion from relational databases, returning to the general concept of direct database connections. Various enhancements are used to accelerate data ingestion. Many data frameworks can parallelize several simultaneous connections and partition queries to pull data in parallel. On the other hand, nothing is free; using parallel connections also increases the load on the source database.

JDBC and ODBC were long the gold standards for data ingestion from databases, but these connection standards are beginning to show their age for many data engineering applications. These connection standards struggle with nested data, and they send data as rows. This means that native nested data must be reencoded as string data to be sent over the wire, and columns from columnar databases must be reserialized as rows.

As discussed in "File-Based Export and Ingestion", many databases now support native file export that bypasses JDBC/ODBC and exports data directly in formats such as Parquet, ORC, and Avro. Alternatively, many cloud data warehouses provide direct REST APIs.

JDBC connections should generally be integrated with other ingestion technologies. For example, we commonly use a reader process to connect to a database with JDBC, write the extracted data into

multiple objects, and then orchestrate ingestion into a downstream system (see Figure 7-13). The reader process can run in a wholly ephemeral cloud instance or in an orchestration system.

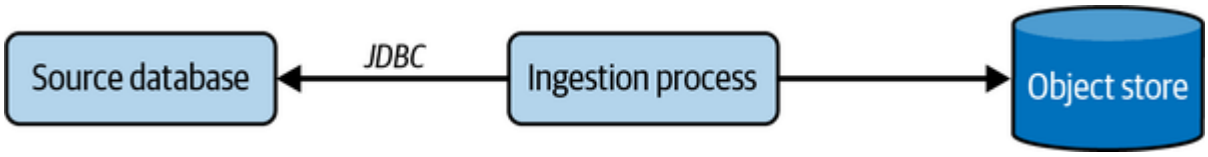

Figure 7-13. An ingestion process reads from a source database using JDBC, and then writes objects into object storage. A target database (not shown) can be triggered to ingest the data with an API call from an orchestration system.

Change Data Capture

Change data capture (CDC), introduced in Chapter 2, is the process of ingesting changes from a source database system. For example, we might have a source PostgreSQL system that supports an application and periodically or continuously ingests table changes for analytics.

Note that our discussion here is by no means exhaustive. We introduce you to common patterns but suggest that you read the documentation on a particular database to handle the details of CDC strategies.

Batch-oriented CDC

If the database table in question has an `updated_at` field containing the last time a record was written or updated, we can query the table to find all updated rows since a specified time. We set the filter timestamp based on when we last captured changed rows from the tables. This process allows us to pull changes and differentially update a target table.

This form of batch-oriented CDC has a key limitation: while we can easily determine which rows have changed since a point in time, we don't necessarily obtain all changes that were applied to these rows. Consider the example of running batch CDC on a bank account table every 24 hours. This operational table shows the current account balance for each account. When money is moved in and out of accounts, the banking application runs a transaction to update the balance.

When we run a query to return all rows in the account table that changed in the last 24 hours, we'll see records for each account that recorded a transaction. Suppose that a certain customer withdrew money five times using a debit card in the last 24 hours. Our query will return only the last account balance recorded in the 24 hour period; other records over the period won't appear. This issue can be mitigated by utilizing an insert-only schema, where each account transaction is recorded as a new record in the table (see "Insert-Only").

Continuous CDC

Continuous CDC captures all table history and can support near real-time data ingestion, either for real-time database replication or to feed real-time streaming analytics. Rather than running periodic queries to get a batch of table changes, continuous CDC treats each write to the database as an event.

We can capture an event stream for continuous CDC in a couple of ways. One of the most common approaches with a transactional database such as PostgreSQL is *log-based CDC*. The database binary log records every change to the database sequentially (see "Database Logs"). A CDC tool can read this log and send the events to a target, such as the Apache Kafka Debezium streaming platform.

Some databases support a simplified, managed CDC paradigm. For instance, many cloud-hosted databases can be configured to directly trigger a serverless function or write to an event stream every time a change happens in the database. This completely frees engineers from worrying about the details of how events are captured in the database and forwarded.

CDC and database replication

CDC can be used to replicate between databases: events are buffered into a stream and *asynchronously* written into a second database. However, many databases natively support a tightly coupled version of replication (synchronous replication) that keeps the replica fully in sync with the primary database. Synchronous replication typically requires that the primary database and the replica are of the same type (e.g., PostgreSQL to PostgreSQL). The advantage of synchronous replication is that the secondary database can offload work from the primary database by acting as a read replica; read queries can be redirected to the replica. The query will return the same results that would be returned from the primary database.

Read replicas are often used in batch data ingestion patterns to allow large scans to run without overloading the primary production database. In addition, an application can be configured to fail over to the replica if the primary database becomes unavailable. No data will be lost in the failover because the replica is entirely in sync with the primary database.

The advantage of asynchronous CDC replication is a loosely coupled architecture pattern. While the replica might be slightly delayed from the primary database, this is often not a problem for analytics applications, and events can now be directed to a variety of targets; we might run CDC replication while simultaneously directing events to object storage and a streaming analytics processor.

CDC considerations

Like anything in technology, CDC is not free. CDC consumes various database resources, such as memory, disk bandwidth, storage, CPU time, and network bandwidth. Engineers should work with production teams and run tests before turning on CDC on production systems to avoid operational problems. Similar considerations apply to synchronous replication.

For batch CDC, be aware that running any large batch query against a transactional production system can cause excessive load. Either run such queries only at off-hours or use a read replica to avoid burdening the primary database.

APIs

> The bulk of software engineering is just plumbing.
>
> Karl Hughes[1]

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.

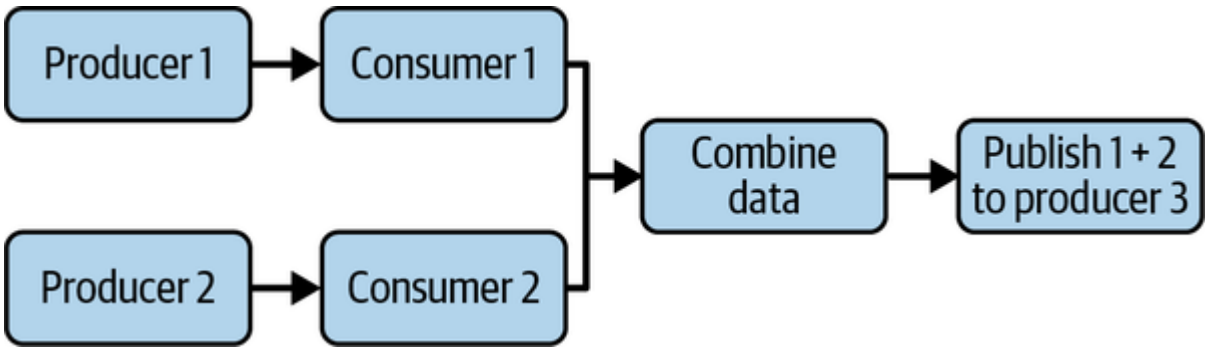

Figure 7-14. Two datasets are produced and consumed (producers 1 and 2) and then combined, with the combined data published to a new producer (producer 3)

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 reconsumed. 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. 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.

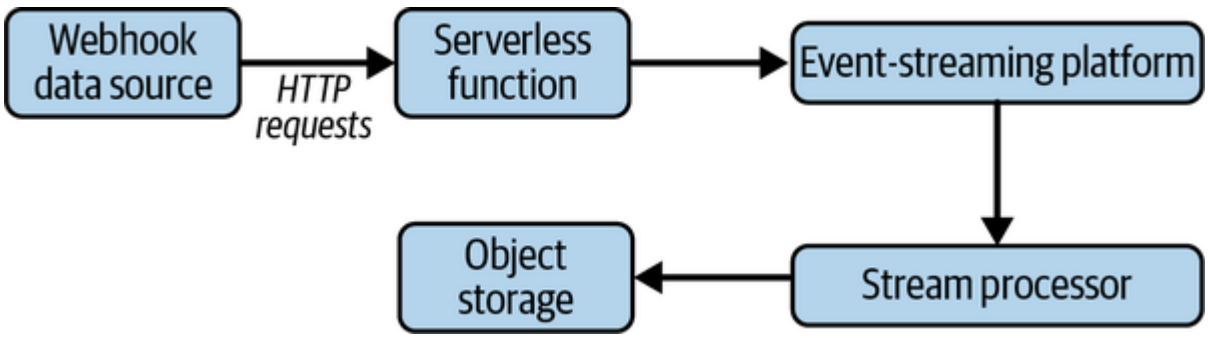

Figure 7-15. A basic webhook ingestion architecture built from cloud services

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 quantities of 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 schema changes, 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, as is 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*.[2]

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.[3]

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

- Airbyte's "Connections and Sync Modes" web page
- Chapter 6, "Batch Is a Special Case of Streaming," in *Introduction to Apache Flink* by Ellen Friedman and Kostas Tzoumas (O'Reilly)
- "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing" by Tyler Akidau et al.
- Google Cloud's "Streaming Pipelines" web page
- Microsoft's "Snapshot Window (Azure Stream Analytics)" documentation

[1] Karl Hughes, "The Bulk of Software Engineering Is Just Plumbing," Karl Hughes website, July 8, 2018, *https://oreil.ly/uIuqJ*.

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

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