

Chapter 5. Data Generation in Source Systems

Welcome to the first stage of the data engineering lifecycle: data generation in source systems. As we described earlier, the job of a data engineer is to take data from source systems, do something with it, and make it helpful in serving downstream use cases. But before you get raw data, you must understand where the data exists, how it is generated, and its characteristics and quirks.

This chapter covers some popular operational source system patterns and the significant types of source systems. Many source systems exist for data generation, and we're not exhaustively covering them all. We'll consider the data these systems generate and things you should consider when working with source systems. We also discuss how the undercurrents of data engineering apply to this first phase of the data engineering lifecycle (Figure 5-1).

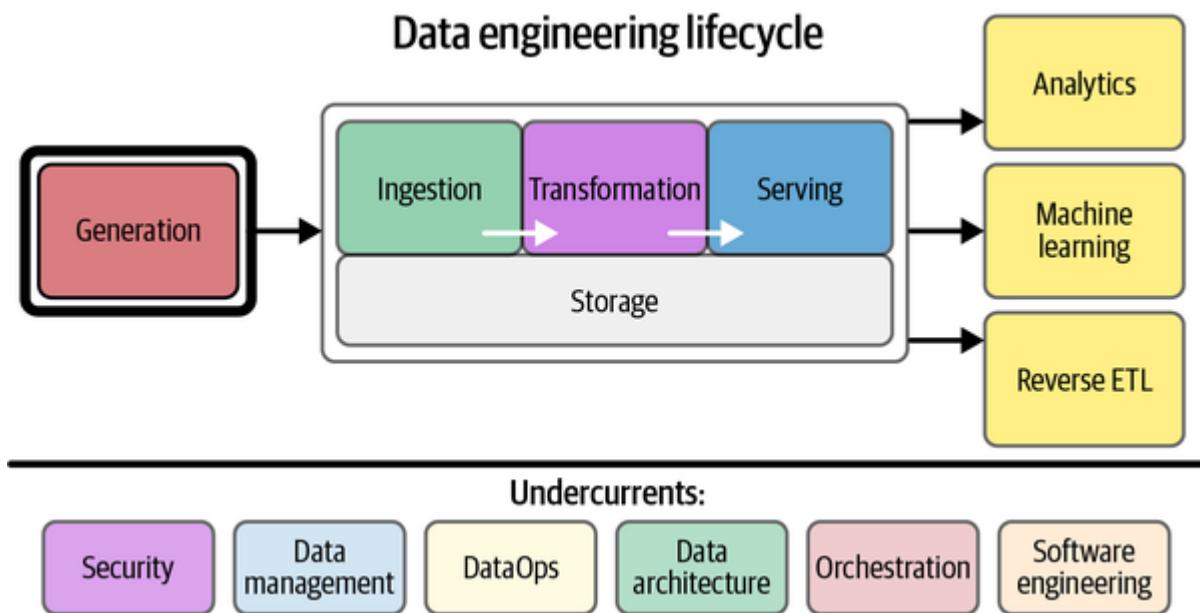


Figure 5-1. Source systems generate the data for the rest of the data engineering lifecycle

As data proliferates, especially with the rise of data sharing (discussed next), we expect that a data engineer's role will shift heavily toward understanding the interplay between data sources and destinations. The basic plumbing tasks of data engineering—moving data from A to B—will simplify dramatically. On the other hand, it will remain critical to understand the nature of data as it's created in source systems.

Sources of Data: How Is Data Created?

As you learn about the various underlying operational patterns of the systems that generate data, it's essential to understand how data is created. Data is an unorganized, context-less collection of facts and figures. It can be created in many ways, both analog and digital.

Analog data creation occurs in the real world, such as vocal speech, sign language, writing on paper, or playing an instrument. This analog data is often transient; how often have you had a verbal conversation whose contents are lost to the ether after the conversation ends?

Digital data is either created by converting analog data to digital form or is the native product of a digital system. An example of analog to digital is a mobile texting app that converts analog speech into digital text. An example of digital data creation is a credit card transaction on an ecommerce platform. A customer places an order, the transaction is charged to their credit card, and the information for the transaction is saved to various databases.

We'll utilize a few common examples in this chapter, such as data created when interacting with a website or mobile application. But in truth, data is everywhere in the world around us. We capture data from IoT devices, credit card terminals, telescope sensors, stock trades, and more.

Get familiar with your source system and how it generates data. Put in the effort to read the source system documentation and understand its patterns and quirks. If your source system is an RDBMS, learn how it operates (writes, commits, queries, etc.); learn the ins and outs of the source system that might affect your ability to ingest from it.

Source Systems: Main Ideas

Source systems produce data in various ways. This section discusses the main ideas you'll frequently encounter as you work with source systems.

Files and Unstructured Data

A *file* is a sequence of bytes, typically stored on a disk. Applications often write data to files. Files may store local parameters, events, logs, images, and audio.

In addition, files are a universal medium of data exchange. As much as data engineers wish that they could get data programmatically, much of the world still sends and receives files. For example, if you're getting data from a government agency, there's an excellent chance you'll download the data as an Excel or CSV file or receive the file in an email.

The main types of source file formats you'll run into as a data engineer—files that originate either manually or as an output from a source system process—are Excel, CSV, TXT, JSON, and XML. These files have their quirks and can be structured (Excel, CSV), semistructured (JSON, XML, CSV), or unstructured (TXT, CSV). Although you'll use certain formats heavily as a data engineer (such as Parquet, ORC, and Avro), we'll cover these later and put the spotlight here on source system files. [Chapter 6](#) covers the technical details of files.

APIs

Application programming interfaces (APIs) are a standard way of exchanging data between systems. In theory, APIs simplify the data ingestion task for data engineers. In practice, many APIs still expose a good deal of data complexity for engineers to manage. Even with the rise of various services and frameworks, and services for automating API data ingestion, data engineers must often invest a good deal of energy into maintaining custom API connections. We discuss APIs in greater detail later in this chapter.

Application Databases (OLTP Systems)

An *application database* stores the state of an application. A standard example is a database that stores account balances for bank accounts. As customer transactions and payments happen, the application updates bank account balances.

Typically, an application database is an *online transaction processing* (OLTP) system—a database that reads and writes individual data records at a high rate. OLTP systems are often referred to as *transactional databases*, but this does not necessarily imply that the system in question supports *atomic transactions*.

More generally, OLTP databases support low latency and high concurrency. An RDBMS database can select or update a row in less than a millisecond (not accounting for network latency) and handle thousands of reads and writes per second. A document database cluster can manage even higher

document commit rates at the expense of potential inconsistency. Some graph databases can also handle transactional use cases.

Fundamentally, OLTP databases work well as application backends when thousands or even millions of users might be interacting with the application simultaneously, updating and writing data concurrently. OLTP systems are less suited to use cases driven by analytics at scale, where a single query must scan a vast amount of data.

ACID

Support for atomic transactions is one of a critical set of database characteristics known together as ACID (as you may recall from [Chapter 3](#), this stands for *atomicity, consistency, isolation, durability*). *Consistency* means that any database read will return the last written version of the retrieved item. *Isolation* entails that if two updates are in flight concurrently for the same thing, the end database state will be consistent with the sequential execution of these updates in the order they were submitted. *Durability* indicates that committed data will never be lost, even in the event of power loss.

Note that ACID characteristics are not required to support application backends, and relaxing these constraints can be a considerable boon to performance and scale. However, ACID characteristics guarantee that the database will maintain a consistent picture of the world, dramatically simplifying the app developer's task.

All engineers (data or otherwise) must understand operating with and without ACID. For instance, to improve performance, some distributed databases use relaxed consistency constraints, such as *eventual consistency*, to improve performance. Understanding the consistency model you're working with helps you prevent disasters.

Atomic transactions

An *atomic transaction* is a set of several changes that are committed as a unit. In the example in [Figure 5-2](#), a traditional banking application running on an RDBMS executes a SQL statement that checks two account balances, one in Account A (the source) and another in Account B (the destination). Money is then moved from Account A to Account B if sufficient funds are in Account A. The entire transaction should run with updates to both account balances or fail without updating either account balance. That is, the whole operation should happen as a *transaction*.

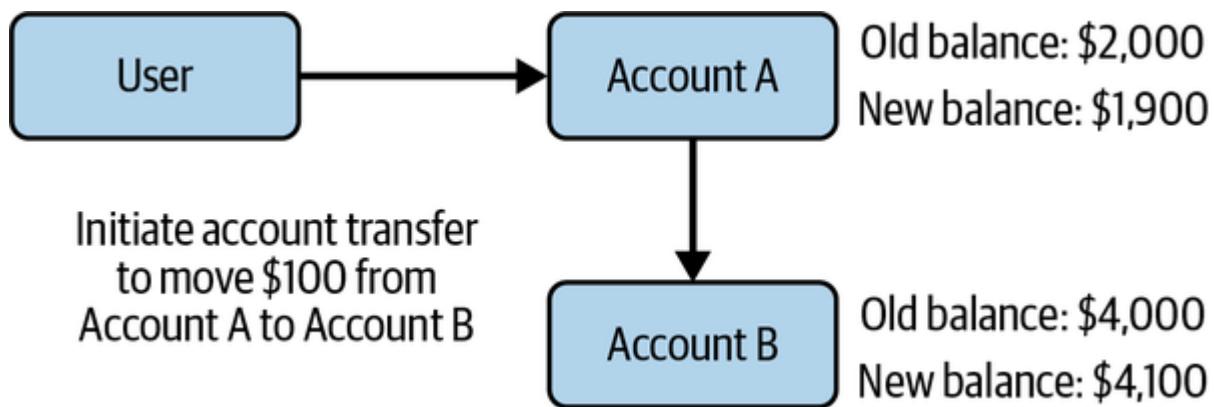


Figure 5-2. Example of an atomic transaction: a bank account transfer using OLTP

OLTP and analytics

Often, small companies run analytics directly on an OLTP. This pattern works in the short term but is ultimately not scalable. At some point, running analytical queries on OLTP runs into performance issues due to structural limitations of OLTP or resource contention with competing transactional workloads. Data engineers must understand the inner workings of OLTP and application backends to

set up appropriate integrations with analytics systems without degrading production application performance.

As companies offer more analytics capabilities in SaaS applications, the need for hybrid capabilities—quick updates with combined analytics capabilities—has created new challenges for data engineers. We'll use the term *data application* to refer to applications that hybridize transactional and analytics workloads.

Online Analytical Processing System

In contrast to an OLTP system, an *online analytical processing* (OLAP) system is built to run large analytics queries and is typically inefficient at handling lookups of individual records. For example, modern column databases are optimized to scan large volumes of data, dispensing with indexes to improve scalability and scan performance. Any query typically involves scanning a minimal data block, often 100 MB or more in size. Trying to look up thousands of individual items per second in such a system will bring it to its knees unless it is combined with a caching layer designed for this use case.

Note that we're using the term *OLAP* to refer to any database system that supports high-scale interactive analytics queries; we are not limiting ourselves to systems that support OLAP cubes (multidimensional arrays of data). The *online* part of OLAP implies that the system constantly listens for incoming queries, making OLAP systems suitable for interactive analytics.

Although this chapter covers source systems, OLAPs are typically storage and query systems for analytics. Why are we talking about them in our chapter on source systems? In practical use cases, engineers often need to read data from an OLAP system. For example, a data warehouse might serve data used to train an ML model. Or, an OLAP system might serve a reverse ETL workflow, where derived data in an analytics system is sent back to a source system, such as a CRM, SaaS platform, or transactional application.

Change Data Capture

Change data capture (CDC) is a method for extracting each change event (insert, update, delete) that occurs in a database. CDC is frequently leveraged to replicate between databases in near real time or create an event stream for downstream processing.

CDC is handled differently depending on the database technology. Relational databases often generate an event log stored directly on the database server that can be processed to create a stream. (See [“Database Logs”](#).) Many cloud NoSQL databases can send a log or event stream to a target storage location.

Logs

A *log* captures information about events that occur in systems. For example, a log may capture traffic and usage patterns on a web server. Your desktop computer's operating system (Windows, macOS, Linux) logs events as the system boots and when applications start or crash, for example.

Logs are a rich data source, potentially valuable for downstream data analysis, ML, and automation. Here are a few familiar sources of logs:

- Operating systems
- Applications
- Servers
- Containers

- Networks
- IoT devices

All logs track events and event metadata. At a minimum, a log should capture who, what, and when:

Who

The human, system, or service account associated with the event (e.g., a web browser user agent or a user ID)

What happened

The event and related metadata

When

The timestamp of the event

Log encoding

Logs are encoded in a few ways:

Binary-encoded logs

These encode data in a custom compact format for space efficiency and fast I/O. Database logs, discussed in [“Database Logs”](#), are a standard example.

Semistructured logs

These are encoded as text in an object serialization format (JSON, more often than not). Semistructured logs are machine-readable and portable. However, they are much less efficient than binary logs. And though they are nominally machine-readable, extracting value from them often requires significant custom code.

Plain-text (unstructured) logs

These essentially store the console output from software. As such, no general-purpose standards exist. These logs can provide helpful information for data scientists and ML engineers, though extracting useful information from the raw text data might be complicated.

Log resolution

Logs are created at various resolutions and log levels. The *log resolution* refers to the amount of event data captured in a log. For example, database logs capture enough information from database events to allow reconstructing the database state at any point in time.

On the other hand, capturing all data changes in logs for a big data system often isn't practical. Instead, these logs may note only that a particular type of commit event has occurred. The *log level* refers to the conditions required to record a log entry, specifically concerning errors and debugging. Software is often configurable to log every event or to log only errors, for example.

Log latency: Batch or real time

Batch logs are often written continuously to a file. Individual log entries can be written to a messaging system such as Kafka or Pulsar for real-time applications.

Database Logs

Database logs are essential enough that they deserve more detailed coverage. Write-ahead logs—typically, binary files stored in a specific database-native format—play a crucial role in database guarantees and recoverability. The database server receives write and update requests to a database table (see [Figure 5-3](#)), storing each operation in the log before acknowledging the request. The acknowledgment comes with a log-associated guarantee: even if the server fails, it can recover its state on reboot by completing the unfinished work from the logs.

Database logs are extremely useful in data engineering, especially for CDC to generate event streams from database changes.

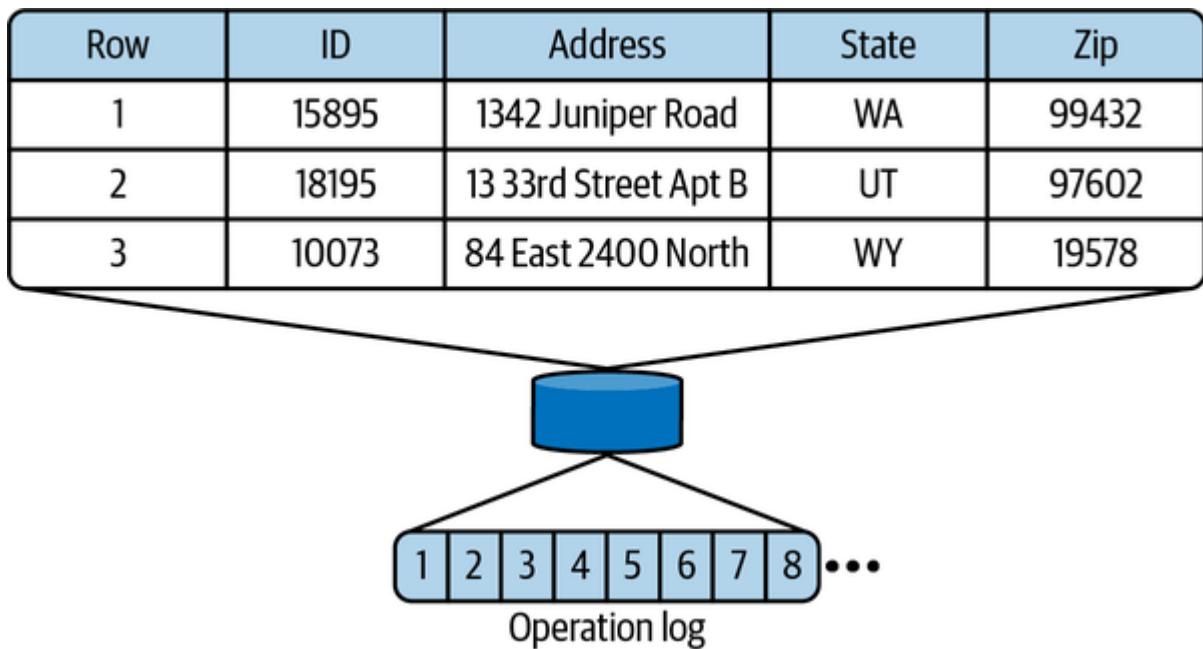


Figure 5-3. Database logs record operations on a table

CRUD

CRUD, which stands for *create, read, update, and delete*, is a transactional pattern commonly used in programming and represents the four basic operations of persistent storage. CRUD is the most common pattern for storing application state in a database. A basic tenet of CRUD is that data must be created before being used. After the data has been created, the data can be read and updated. Finally, the data may need to be destroyed. CRUD guarantees these four operations will occur on data, regardless of its storage.

CRUD is a widely used pattern in software applications, and you'll commonly find CRUD used in APIs and databases. For example, a web application will make heavy use of CRUD for RESTful HTTP requests and storing and retrieving data from a database.

As with any database, we can use snapshot-based extraction to get data from a database where our application applies CRUD operations. On the other hand, event extraction with CDC gives us a complete history of operations and potentially allows for near real-time analytics.

Insert-Only

The *insert-only pattern* retains history directly in a table containing data. Rather than updating records, new records get inserted with a timestamp indicating when they were created ([Table 5-1](#)). For example, suppose you have a table of customer addresses. Following a CRUD pattern, you would simply update the record if the customer changed their address. With the insert-only pattern, a new

address record is inserted with the same customer ID. To read the current customer address by customer ID, you would look up the latest record under that ID.

Table 5-1. An insert-only pattern produces multiple versions of a record

Record ID	Value	Timestamp
1	40	2021-09-19T00:10:23+00:00
1	51	2021-09-30T00:12:00+00:00

In a sense, the insert-only pattern maintains a database log directly in the table itself, making it especially useful if the application needs access to history. For example, the insert-only pattern would work well for a banking application designed to present customer address history.

A separate analytics insert-only pattern is often used with regular CRUD application tables. In the insert-only ETL pattern, data pipelines insert a new record in the target analytics table anytime an update occurs in the CRUD table.

Insert-only has a couple of disadvantages. First, tables can grow quite large, especially if data frequently changes, since each change is inserted into the table. Sometimes records are purged based on a record sunset date or a maximum number of record versions to keep table size reasonable. The second disadvantage is that record lookups incur extra overhead because looking up the current state involves running `MAX(created_timestamp)`. If hundreds or thousands of records are under a single ID, this lookup operation is expensive to run.

Messages and Streams

Related to event-driven architecture, two terms that you'll often see used interchangeably are *message queue* and *streaming platform*, but a subtle but essential difference exists between the two. Defining and contrasting these terms is worthwhile since they encompass many big ideas related to source systems and practices and technologies spanning the entire data engineering lifecycle.

A *message* is raw data communicated across two or more systems ([Figure 5-4](#)). For example, we have System 1 and System 2, where System 1 sends a message to System 2. These systems could be different microservices, a server sending a message to a serverless function, etc. A message is typically sent through a *message queue* from a publisher to a consumer, and once the message is delivered, it is removed from the queue.



Figure 5-4. A message passed between two systems

Messages are discrete and singular signals in an event-driven system. For example, an IoT device might send a message with the latest temperature reading to a message queue. This message is then ingested by a service that determines whether the furnace should be turned on or off. This service sends a message to a furnace controller that takes the appropriate action. Once the message is received, and the action is taken, the message is removed from the message queue.

By contrast, a *stream* is an append-only log of event records. (Streams are ingested and stored in *event-streaming platforms*, which we discuss at greater length in [“Message Queues and Event-Streaming Platforms”](#).) As events occur, they are accumulated in an ordered sequence ([Figure 5-5](#)); a timestamp or an ID might order events. (Note that events aren't always delivered in exact order because of the subtleties of distributed systems.)

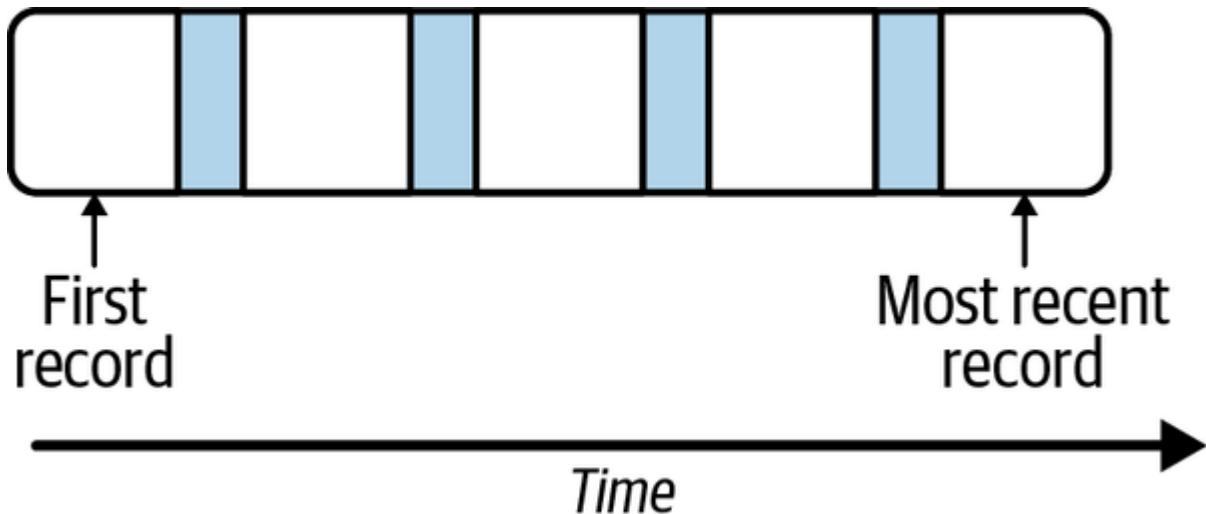


Figure 5-5. A stream, which is an ordered append-only log of records

You'll use streams when you care about what happened over many events. Because of the append-only nature of streams, records in a stream are persisted over a long retention window—often weeks or months—allowing for complex operations on records such as aggregations on multiple records or the ability to rewind to a point in time within the stream.

It's worth noting that systems that process streams can process messages, and streaming platforms are frequently used for message passing. We often accumulate messages in streams when we want to perform message analytics. In our IoT example, the temperature readings that trigger the furnace to turn on or off might also be later analyzed to determine temperature trends and statistics.

Types of Time

While time is an essential consideration for all data ingestion, it becomes that much more critical and subtle in the context of streaming, where we view data as continuous and expect to consume it shortly after it is produced. Let's look at the key types of time you'll run into when ingesting data: the time that the event is generated, when it's ingested and processed, and how long processing took ([Figure 5-6](#)).

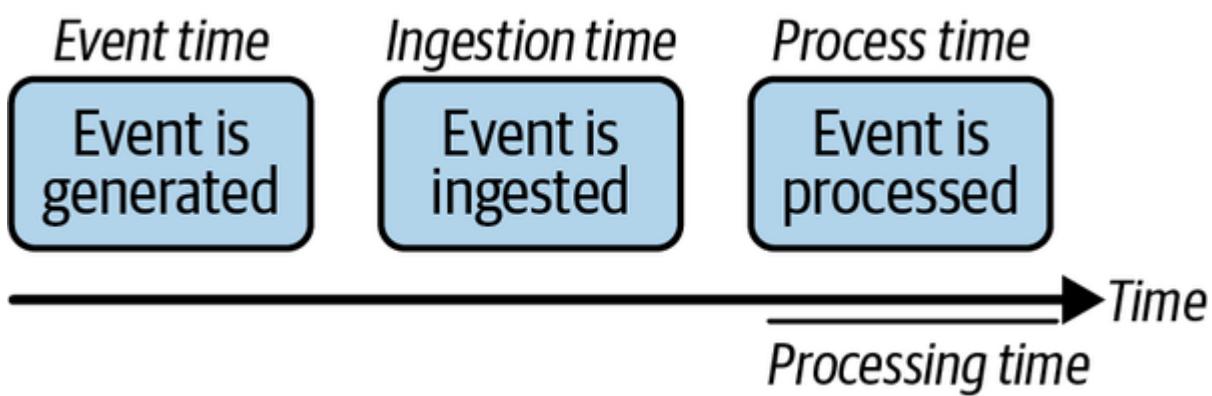


Figure 5-6. Event, ingestion, process, and processing time

Event time indicates when an event is generated in a source system, including the timestamp of the original event itself. An undetermined time lag will occur upon event creation, before the event is ingested and processed downstream. Always include timestamps for each phase through which an event travels. Log events as they occur and at each stage of time—when they're created, ingested, and processed. Use these timestamp logs to accurately track the movement of your data through your data pipelines.

After data is created, it is ingested somewhere. *Ingestion time* indicates when an event is ingested from source systems into a message queue, cache, memory, object storage, a database, or any place else that data is stored (see [Chapter 6](#)). After ingestion, data may be processed immediately; or within minutes, hours, or days; or simply persist in storage indefinitely.

Process time occurs after ingestion time, when the data is processed (typically, a transformation). *Processing time* is how long the data took to process, measured in seconds, minutes, hours, etc.

You'll want to record these various times, preferably in an automated way. Set up monitoring along your data workflows to capture when events occur, when they're ingested and processed, and how long it took to process events.

Source System Practical Details

This section discusses the practical details of interacting with modern source systems. We'll dig into the details of commonly encountered databases, APIs, and other aspects. This information will have a shorter shelf life than the main ideas discussed previously; popular API frameworks, databases, and other details will continue to change rapidly.

Nevertheless, these details are critical knowledge for working data engineers. We suggest that you study this information as baseline knowledge but read extensively to stay abreast of ongoing developments.

Databases

In this section, we'll look at common source system database technologies that you'll encounter as a data engineer and high-level considerations for working with these systems. There are as many types of databases as there are use cases for data.

Major considerations for understanding database technologies

Here, we introduce major ideas that occur across a variety of database technologies, including those that back software applications and those that support analytics use cases:

Database management system

A database system used to store and serve data. Abbreviated as DBMS, it consists of a storage engine, query optimizer, disaster recovery, and other key components for managing the database system.

Lookups

How does the database find and retrieve data? Indexes can help speed up lookups, but not all databases have indexes. Know whether your database uses indexes; if so, what are the best patterns for designing and maintaining them? Understand how to leverage for efficient extraction. It also helps to have a basic knowledge of the major types of indexes, including B-tree and log-structured merge-trees (LSM).

Query optimizer

Does the database utilize an optimizer? What are its characteristics?

Scaling and distribution

Does the database scale with demand? What scaling strategy does it deploy? Does it scale horizontally (more database nodes) or vertically (more resources on a single machine)?

Modeling patterns

What modeling patterns work best with the database (e.g., data normalization or wide tables)? (See [Chapter 8](#) for our discussion of data modeling.)

CRUD

How is data queried, created, updated, and deleted in the database? Every type of database handles CRUD operations differently.

Consistency

Is the database fully consistent, or does it support a relaxed consistency model (e.g., eventual consistency)? Does the database support optional consistency modes for reads and writes (e.g., strongly consistent reads)?

We divide databases into relational and nonrelational categories. In truth, the nonrelational category is far more diverse, but relational databases still occupy significant space in application backends.

Relational databases

A *relational database management system* (RDBMS) is one of the most common application backends. Relational databases were developed at IBM in the 1970s and popularized by Oracle in the 1980s. The growth of the internet saw the rise of the LAMP stack (Linux, Apache web server, MySQL, PHP) and an explosion of vendor and open source RDBMS options. Even with the rise of NoSQL databases (described in the following section), relational databases have remained extremely popular.

Data is stored in a table of *relations* (rows), and each relation contains multiple *fields* (columns); see [Figure 5-7](#). Note that we use the terms *column* and *field* interchangeably throughout this book. Each relation in the table has the same *schema* (a sequence of columns with assigned static types such as string, integer, or float). Rows are typically stored as a contiguous sequence of bytes on disk.

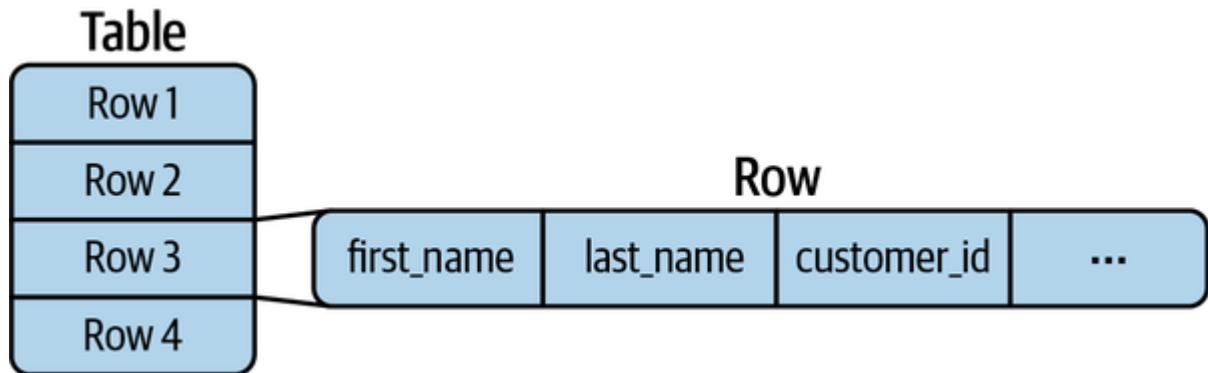


Figure 5-7. RDBMS stores and retrieves data at a row level

Tables are typically indexed by a *primary key*, a unique field for each row in the table. The indexing strategy for the primary key is closely connected with the layout of the table on disk.

Tables can also have various *foreign keys*—fields with values connected with the values of primary keys in other tables, facilitating joins, and allowing for complex schemas that spread data across multiple tables. In particular, it is possible to design a *normalized schema*. Normalization is a strategy for ensuring that data in records is not duplicated in multiple places, thus avoiding the need to update states in multiple locations at once and preventing inconsistencies (see [Chapter 8](#)).

RDBMS systems are typically ACID compliant. Combining a normalized schema, ACID compliance, and support for high transaction rates makes relational database systems ideal for storing rapidly

changing application states. The challenge for data engineers is to determine how to capture state information over time.

A full discussion of the theory, history, and technology of RDBMS is beyond the scope of this book. We encourage you to study RDBMS systems, relational algebra, and strategies for normalization because they're widespread, and you'll encounter them frequently. See "[Additional Resources](#)" for suggested books.

Nonrelational databases: NoSQL

While relational databases are terrific for many use cases, they're not a one-size-fits-all solution. We often see that people start with a relational database under the impression it's a universal appliance and shoehorn in a ton of use cases and workloads. As data and query requirements morph, the relational database collapses under its weight. At that point, you'll want to use a database that's appropriate for the specific workload under pressure. Enter nonrelational or NoSQL databases. *NoSQL*, which stands for *not only SQL*, refers to a whole class of databases that abandon the relational paradigm.

On the one hand, dropping relational constraints can improve performance, scalability, and schema flexibility. But as always in architecture, trade-offs exist. NoSQL databases also typically abandon various RDBMS characteristics, such as strong consistency, joins, or a fixed schema.

A big theme of this book is that data innovation is constant. Let's take a quick look at the history of NoSQL, as it's helpful to gain a perspective on why and how data innovations impact your work as a data engineer. In the early 2000s, tech companies such as Google and Amazon began to outgrow their relational databases and pioneered new distributed, nonrelational databases to scale their web platforms.

While the term *NoSQL* first appeared in 1998, the modern version was coined by Eric Evans in the 2000s.¹ He tells the story in a [2009 blog post](#):

I've spent the last couple of days at [nosqleast](#) and one of the hot topics here is the name "nosql." Understandably, there are a lot of people who worry that the name is Bad, that it sends an inappropriate or inaccurate message. While I make no claims to the idea, I do have to accept some blame for what it is now being called. How's that? Johan Oskarsson was organizing the first meetup and asked the question "What's a good name?" on IRC; it was one of three or four suggestions that I spouted off in the span of like 45 seconds, without thinking.

My regret, however, isn't about what the name says; it's about what it doesn't. When Johan originally had the idea for the first meetup, he seemed to be thinking Big Data and linearly scalable distributed systems, but the name is so vague that it opened the door to talk submissions for literally anything that stored data, and wasn't an RDBMS.

NoSQL remains vague in 2022, but it's been widely adopted to describe a universe of "new school" databases, alternatives to relational databases.

There are numerous flavors of NoSQL database designed for almost any imaginable use case. Because there are far too many NoSQL databases to cover exhaustively in this section, we consider the following database types: key-value, document, wide-column, graph, search, and time series. These databases are all wildly popular and enjoy widespread adoption. A data engineer should understand these types of databases, including usage considerations, the structure of the data they store, and how to leverage each in the data engineering lifecycle.

Key-value stores

A *key-value database* is a nonrelational database that retrieves records using a key that uniquely identifies each record. This is similar to hash map or dictionary data structures presented in many

programming languages but potentially more scalable. Key-value stores encompass several NoSQL database types—for example, document stores and wide column databases (discussed next).

Different types of key-value databases offer a variety of performance characteristics to serve various application needs. For example, in-memory key-value databases are popular for caching session data for web and mobile applications, where ultra-fast lookup and high concurrency are required. Storage in these systems is typically temporary; if the database shuts down, the data disappears. Such caches can reduce pressure on the main application database and serve speedy responses.

Of course, key-value stores can also serve applications requiring high-durability persistence. An ecommerce application may need to save and update massive amounts of event state changes for a user and their orders. A user logs into the ecommerce application, clicks around various screens, adds items to a shopping cart, and then checks out. Each event must be durably stored for retrieval. Key-value stores often persist data to disk and across multiple nodes to support such use cases.

Document stores

As mentioned previously, a *document store* is a specialized key-value store. In this context, a *document* is a nested object; we can usually think of each document as a JSON object for practical purposes. Documents are stored in collections and retrieved by key. A *collection* is roughly equivalent to a table in a relational database (see [Table 5-2](#)).

Table 5-2. Comparison of
RDBMS and document
terminology

RDBMS Document database

Table Collection

Row Document, items, entity

One key difference between relational databases and document stores is that the latter does not support joins. This means that data cannot be easily *normalized*, i.e., split across multiple tables. (Applications can still join manually. Code can look up a document, extract a property, and then retrieve another document.) Ideally, all related data can be stored in the same document.

In many cases, the same data must be stored in multiple documents spread across numerous collections; software engineers must be careful to update a property everywhere it is stored. (Many document stores support a notion of transactions to facilitate this.)

Document databases generally embrace all the flexibility of JSON and don't enforce schema or types; this is a blessing and a curse. On the one hand, this allows the schema to be highly flexible and expressive. The schema can also evolve as an application grows. On the flip side, we've seen document databases become absolute nightmares to manage and query. If developers are not careful in managing schema evolution, data may become inconsistent and bloated over time. Schema evolution can also break downstream ingestion and cause headaches for data engineers if it's not communicated in a timely fashion (before deployment).

The following is an example of data that is stored in a collection called `users`. The collection key is the `id`. We also have a `name` (along with `first` and `last` as child elements) and an array of the user's favorite bands within each document:

```
{
  "users": [
    {
      "id": 1234,
      "name": {
        "first": "Joe",
        "last": "Reis"
      },
      "favorite_bands": [
        ...
      ]
    }
  ]
}
```

```

    "AC/DC",
    "Slayer",
    "WuTang Clan",
    "Action Bronson"
]
},
{
"id":1235,
"name":{
    "first":"Matt",
    "last":"Housley"
},
"favorite_bands":[
    "Dave Matthews Band",
    "Creed",
    "Nickelback"
]
}
]
}

```

To query the data in this example, you can retrieve records by key. Note that most document databases also support the creation of indexes and lookup tables to allow retrieval of documents by specific properties. This is often invaluable in application development when you need to search for documents in various ways. For example, you could set an index on `name`.

Another critical technical detail for data engineers is that document stores are generally not ACID compliant, unlike relational databases. Technical expertise in a particular document store is essential to understanding performance, tuning, configuration, related effects on writes, consistency, durability, etc. For example, many document stores are *eventually consistent*. Allowing data distribution across a cluster is a boon for scaling and performance but can lead to catastrophes when engineers and developers don't understand the implications.²

To run analytics on document stores, engineers generally must run a full scan to extract all data from a collection or employ a CDC strategy to send events to a target stream. The full scan approach can have both performance and cost implications. The scan often slows the database as it runs, and many serverless cloud offerings charge a significant fee for each full scan. In document databases, it's often helpful to create an index to help speed up queries. We discuss indexes and query patterns in [Chapter 8](#).

Wide-column

A *wide-column database* is optimized for storing massive amounts of data with high transaction rates and extremely low latency. These databases can scale to extremely high write rates and vast amounts of data. Specifically, wide-column databases can support petabytes of data, millions of requests per second, and sub-10ms latency. These characteristics have made wide-column databases popular in ecommerce, fintech, ad tech, IoT, and real-time personalization applications. Data engineers must be aware of the operational characteristics of the wide-column databases they work with to set up a suitable configuration, design the schema, and choose an appropriate row key to optimize performance and avoid common operational issues.

These databases support rapid scans of massive amounts of data, but they do not support complex queries. They have only a single index (the row key) for lookups. Data engineers must generally extract data and send it to a secondary analytics system to run complex queries to deal with these limitations. This can be accomplished by running large scans for the extraction or employing CDC to capture an event stream.

Graph databases

Graph databases explicitly store data with a mathematical graph structure (as a set of nodes and edges).³ Neo4j has proven extremely popular, while Amazon, Oracle, and other vendors offer their graph database products. Roughly speaking, graph databases are a good fit when you want to analyze the connectivity between elements.

For example, you could use a document database to store one document for each user describing their properties. You could add an array element for *connections* that contains directly connected users' IDs in a social media context. It's pretty easy to determine the number of direct connections a user has, but suppose you want to know how many users can be reached by traversing two direct connections. You could answer this question by writing complex code, but each query would run slowly and consume significant resources. The document store is simply not optimized for this use case.

Graph databases are designed for precisely this type of query. Their data structures allow for queries based on the connectivity between elements; graph databases are indicated when we care about understanding complex traversals between elements. In the parlance of graphs, we store *nodes* (users in the preceding example) and *edges* (connections between users). Graph databases support rich data models for both nodes and edges. Depending on the underlying graph database engine, graph databases utilize specialized query languages such as SPARQL, Resource Description Framework (RDF), Graph Query Language (GQL), and Cypher.

As an example of a graph, consider a network of four users. User 1 follows User 2, who follows User 3 and User 4; User 3 also follows User 4 ([Figure 5-8](#)).

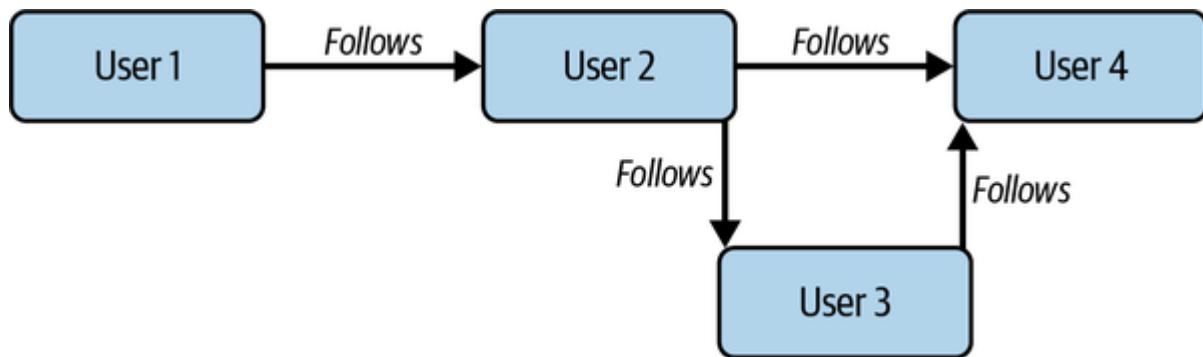


Figure 5-8. A social network graph

We anticipate that graph database applications will grow dramatically outside of tech companies; market analyses also predict rapid growth.⁴ Of course, graph databases are beneficial from an operational perspective and support the kinds of complex social relationships critical to modern applications. Graph structures are also fascinating from the perspective of data science and ML, potentially revealing deep insights into human interactions and behavior.

This introduces unique challenges for data engineers who may be more accustomed to dealing with structured relations, documents, or unstructured data. Engineers must choose whether to do the following:

- Map source system graph data into one of their existing preferred paradigms
- Analyze graph data within the source system itself
- Adopt graph-specific analytics tools

Graph data can be reencoded into rows in a relational database, which may be a suitable solution depending on the analytics use case. Transactional graph databases are also designed for analytics, although large queries may overload production systems. Contemporary cloud-based graph databases support read-heavy graph analytics on massive quantities of data.

Search

A *search database* is a nonrelational database used to search your data's complex and straightforward semantic and structural characteristics. Two prominent use cases exist for a search database: text search and log analysis. Let's cover each of these separately.

Text search involves searching a body of text for keywords or phrases, matching on exact, fuzzy, or semantically similar matches. *Log analysis* is typically used for anomaly detection, real-time monitoring, security analytics, and operational analytics. Queries can be optimized and sped up with the use of indexes.

Depending on the type of company you work at, you may use search databases either regularly or not at all. Regardless, it's good to be aware they exist in case you come across them in the wild. Search databases are popular for fast search and retrieval and can be found in various applications; an ecommerce site may power its product search using a search database. As a data engineer, you might be expected to bring data from a search database (such as Elasticsearch, Apache Solr or Lucene, or Algolia) into downstream KPI reports or something similar.

Time series

A *time series* is a series of values organized by time. For example, stock prices might move as trades are executed throughout the day, or a weather sensor will take atmospheric temperatures every minute. Any events that are recorded over time—either regularly or sporadically—are time-series data. A *time-series database* is optimized for retrieving and statistical processing of time-series data.

While time-series data such as orders, shipments, logs, and so forth have been stored in relational databases for ages, these data sizes and volumes were often tiny. As data grew faster and bigger, new special-purpose databases were needed. Time-series databases address the needs of growing, high-velocity data volumes from IoT, event and application logs, ad tech, and fintech, among many other use cases. Often these workloads are write-heavy. As a result, time-series databases often utilize memory buffering to support fast writes and reads.

We should distinguish between measurement and event-based data, common in time-series databases. *Measurement data* is generated regularly, such as temperature or air-quality sensors. *Event-based data* is irregular and created every time an event occurs—for instance, when a motion sensor detects movement.

The schema for a time series typically contains a timestamp and a small set of fields. Because the data is time-dependent, the data is ordered by the timestamp. This makes time-series databases suitable for operational analytics but not great for BI use cases. Joins are not common, though some quasi time-series databases such as Apache Druid support joins. Many time-series databases are available, both as open source and paid options.

APIs

APIs are now a standard and pervasive way of exchanging data in the cloud, for SaaS platforms, and between internal company systems. Many types of API interfaces exist across the web, but we are principally interested in those built around HTTP, the most popular type on the web and in the cloud.

REST

We'll first talk about REST, currently the dominant API paradigm. As noted in [Chapter 4](#), REST stands for *representational state transfer*. This set of practices and philosophies for building HTTP web APIs was laid out by Roy Fielding in 2000 in a PhD dissertation. REST is built around HTTP verbs, such as GET and PUT; in practice, modern REST uses only a handful of the verb mappings outlined in the original dissertation.

One of the principal ideas of REST is that interactions are stateless. Unlike in a Linux terminal session, there is no notion of a session with associated state variables such as a working directory;

each REST call is independent. REST calls can change the system's state, but these changes are global, applying to the full system rather than a current session.

Critics point out that REST is in no way a full specification.⁵ REST stipulates basic properties of interactions, but developers utilizing an API must gain a significant amount of domain knowledge to build applications or pull data effectively.

We see great variation in levels of API abstraction. In some cases, APIs are merely a thin wrapper over internals that provides the minimum functionality required to protect the system from user requests. In other examples, a REST data API is a masterpiece of engineering that prepares data for analytics applications and supports advanced reporting.

A couple of developments have simplified setting up data-ingestion pipelines from REST APIs. First, data providers frequently supply client libraries in various languages, especially in Python. Client libraries remove much of the boilerplate labor of building API interaction code. Client libraries handle critical details such as authentication and map fundamental methods into accessible classes.

Second, various services and open source libraries have emerged to interact with APIs and manage data synchronization. Many SaaS and open source vendors provide off-the-shelf connectors for common APIs. Platforms also simplify the process of building custom connectors as required.

There are numerous data APIs without client libraries or out-of-the-box connector support. As we emphasize throughout the book, engineers would do well to reduce undifferentiated heavy lifting by using off-the-shelf tools. However, low-level *plumbing* tasks still consume many resources. At virtually any large company, data engineers will need to deal with the problem of writing and maintaining custom code to pull data from APIs, which requires understanding the structure of the data as provided, developing appropriate data-extraction code, and determining a suitable data synchronization strategy.

GraphQL

GraphQL was created at Facebook as a query language for application data and an alternative to generic REST APIs. Whereas REST APIs generally restrict your queries to a specific data model, GraphQL opens up the possibility of retrieving multiple data models in a single request. This allows for more flexible and expressive queries than with REST. GraphQL is built around JSON and returns data in a shape resembling the JSON query.

There's something of a holy war between REST and GraphQL, with some engineering teams partisans of one or the other and some using both. In reality, engineers will encounter both as they interact with source systems.

Webhooks

Webhooks are a simple event-based data-transmission pattern. The data source can be an application backend, a web page, or a mobile app. When specified events happen in the source system, this triggers a call to an HTTP endpoint hosted by the data consumer. Notice that the connection goes from the source system to the data sink, the opposite of typical APIs. For this reason, webhooks are often called *reverse APIs*.

The endpoint can do various things with the POST event data, potentially triggering a downstream process or storing the data for future use. For analytics purposes, we're interested in collecting these events. Engineers commonly use message queues to ingest data at high velocity and volume. We will talk about message queues and event streams later in this chapter.

RPC and gRPC

A *remote procedure call* (RPC) is commonly used in distributed computing. It allows you to run a procedure on a remote system.

gRPC is a remote procedure call library developed internally at Google in 2015 and later released as an open standard. Its use at Google alone would be enough to merit inclusion in our discussion. Many Google services, such as Google Ads and GCP, offer gRPC APIs. gRPC is built around the Protocol Buffers open data serialization standard, also developed by Google.

gRPC emphasizes the efficient bidirectional exchange of data over HTTP/2. *Efficiency* refers to aspects such as CPU utilization, power consumption, battery life, and bandwidth. Like GraphQL, gRPC imposes much more specific technical standards than REST, thus allowing the use of common client libraries and allowing engineers to develop a skill set that will apply to any gRPC interaction code.

Data Sharing

The core concept of cloud data sharing is that a multitenant system supports security policies for sharing data among tenants. Concretely, any public cloud object storage system with a fine-grained permission system can be a platform for data sharing. Popular cloud data-warehouse platforms also support data-sharing capabilities. Of course, data can also be shared through download or exchange over email, but a multitenant system makes the process much easier.

Many modern sharing platforms (especially cloud data warehouses) support row, column, and sensitive data filtering. Data sharing also streamlines the notion of the *data marketplace*, available on several popular clouds and data platforms. Data marketplaces provide a centralized location for data commerce, where data providers can advertise their offerings and sell them without worrying about the details of managing network access to data systems.

Data sharing can also streamline data pipelines within an organization. Data sharing allows units of an organization to manage their data and selectively share it with other units while still allowing individual units to manage their compute and query costs separately, facilitating data decentralization. This facilitates decentralized data management patterns such as data mesh.⁶

Data sharing and data mesh align closely with our philosophy of common architecture components. Choose common components (see [Chapter 3](#)) that allow the simple and efficient interchange of data and expertise rather than embracing the most exciting and sophisticated technology.

Third-Party Data Sources

The consumerization of technology means every company is essentially now a technology company. The consequence is that these companies—and increasingly government agencies—want to make their data available to their customers and users, either as part of their service or as a separate subscription. For example, the US Bureau of Labor Statistics publishes various statistics about the US labor market. The National Aeronautics and Space Administration (NASA) publishes various data from its research initiatives. Facebook shares data with businesses that advertise on its platform.

Why would companies want to make their data available? Data is sticky, and a flywheel is created by allowing users to integrate and extend their application into a user's application. Greater user adoption and usage means more data, which means users can integrate more data into their applications and data systems. The side effect is there are now almost infinite sources of third-party data.

Direct third-party data access is commonly done via APIs, through data sharing on a cloud platform, or through data download. APIs often provide deep integration capabilities, allowing customers to pull and push data. For example, many CRMs offer APIs that their users can integrate into their systems and applications. We see a common workflow to get data from a CRM, blend the CRM data through

the customer scoring model, and then use reverse ETL to send that data back into CRM for salespeople to contact better-qualified leads.

Message Queues and Event-Streaming Platforms

Event-driven architectures are pervasive in software applications and are poised to grow their popularity even further. First, message queues and event-streaming platforms—critical layers in event-driven architectures—are easier to set up and manage in a cloud environment. Second, the rise of data apps—applications that directly integrate real-time analytics—are growing from strength to strength. Event-driven architectures are ideal in this setting because events can both trigger work in the application and feed near real-time analytics.

Please note that streaming data (in this case, messages and streams) cuts across many data engineering lifecycle stages. Unlike an RDBMS, which is often directly attached to an application, the lines of streaming data are sometimes less clear-cut. These systems are used as source systems, but they will often cut across the data engineering lifecycle because of their transient nature. For example, you can use an event-streaming platform for message passing in an event-driven application, a source system. The same event-streaming platform can be used in the ingestion and transformation stage to process data for real-time analytics.

As source systems, message queues and event-streaming platforms are used in numerous ways, from routing messages between microservices ingesting millions of events per second of event data from web, mobile, and IoT applications. Let's look at message queues and event-streaming platforms a bit more closely.

Message queues

A *message queue* is a mechanism to asynchronously send data (usually as small individual messages, in the kilobytes) between discrete systems using a publish and subscribe model. Data is published to a message queue and is delivered to one or more subscribers ([Figure 5-9](#)). The subscriber acknowledges receipt of the message, removing it from the queue.



Figure 5-9. A simple message queue

Message queues allow applications and systems to be decoupled from each other and are widely used in microservices architectures. The message queue buffers messages to handle transient load spikes and makes messages durable through a distributed architecture with replication.

Message queues are a critical ingredient for decoupled microservices and event-driven architectures. Some things to keep in mind with message queues are frequency of delivery, message ordering, and scalability.

Message ordering and delivery

The order in which messages are created, sent, and received can significantly impact downstream subscribers. In general, order in distributed message queues is a tricky problem. Message queues often apply a fuzzy notion of order and first in, first out (FIFO). Strict FIFO means that if message A is ingested before message B, message A will always be delivered before message B. In practice,

messages might be published and received out of order, especially in highly distributed message systems.

For example, Amazon SQS [standard queues](#) make the best effort to preserve message order. SQS also offers [FIFO queues](#), which offer much stronger guarantees at the cost of extra overhead.

In general, don't assume that your messages will be delivered in order unless your message queue technology guarantees it. You typically need to design for out-of-order message delivery.

Delivery frequency

Messages can be sent exactly once or at least once. If a message is sent *exactly once*, then after the subscriber acknowledges the message, the message disappears and won't be delivered again.⁷ Messages sent *at least once* can be consumed by multiple subscribers or by the same subscriber more than once. This is great when duplications or redundancy don't matter.

Ideally, systems should be *idempotent*. In an idempotent system, the outcome of processing a message once is identical to the outcome of processing it multiple times. This helps to account for a variety of subtle scenarios. For example, even if our system can guarantee exactly-once delivery, a consumer might fully process a message but fail right before acknowledging processing. The message will effectively be processed twice, but an idempotent system handles this scenario gracefully.

Scalability

The most popular message queues utilized in event-driven applications are horizontally scalable, running across multiple servers. This allows these queues to scale up and down dynamically, buffer messages when systems fall behind, and durably store messages for resilience against failure. However, this can create a variety of complications, as mentioned previously (multiple deliveries and fuzzy ordering).

Event-streaming platforms

In some ways, an *event-streaming platform* is a continuation of a message queue in that messages are passed from producers to consumers. As discussed previously in this chapter, the big difference between messages and streams is that a message queue is primarily used to route messages with certain delivery guarantees. In contrast, an event-streaming platform is used to ingest and process data in an ordered log of records. In an event-streaming platform, data is retained for a while, and it is possible to replay messages from a past point in time.

Let's describe an event related to an event-streaming platform. As mentioned in [Chapter 3](#), an event is "something that happened, typically a change in the *state* of something." An event has the following features: a key, a value, and a timestamp. Multiple key-value timestamps might be contained in a single event. For example, an event for an ecommerce order might look like this:

```
{
  "Key": "Order # 12345",
  "Value": "SKU 123, purchase price of $100",
  "Timestamp": "2023-01-02 06:01:00"
}
```

Let's look at some of the critical characteristics of an event-streaming platform that you should be aware of as a data engineer.

Topics

In an event-streaming platform, a producer streams events to a topic, a collection of related events. A topic might contain fraud alerts, customer orders, or temperature readings from IoT devices, for

example. A topic can have zero, one, or multiple producers and customers on most event-streaming platforms.

Using the preceding event example, a topic might be `web_orders`. Also, let's send this topic to a couple of consumers, such as `fulfillment` and `marketing`. This is an excellent example of blurred lines between analytics and an event-driven system. The `fulfillment` subscriber will use events to trigger a fulfillment process, while `marketing` runs real-time analytics or trains and runs ML models to tune marketing campaigns ([Figure 5-10](#)).

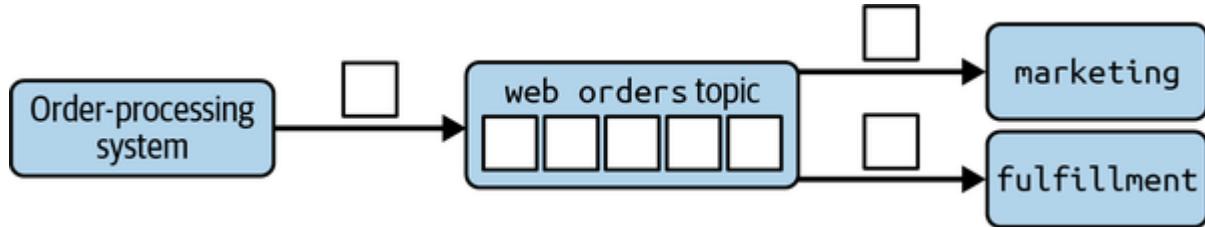


Figure 5-10. An order-processing system generates events (small squares) and publishes them to the `web_orders` topic. Two subscribers—`marketing` and `fulfillment`—pull events from the topic.

Stream partitions

Stream partitions are subdivisions of a stream into multiple streams. A good analogy is a multilane freeway. Having multiple lanes allows for parallelism and higher throughput. Messages are distributed across partitions by *partition key*. Messages with the same partition key will always end up in the same partition.

In [Figure 5-11](#), for example, each message has a numeric ID—shown inside the circle representing the message—that we use as a partition key. To determine the partition, we divide by 3 and take the remainder. Going from bottom to top, the partitions have remainder 0, 1, and 2, respectively.

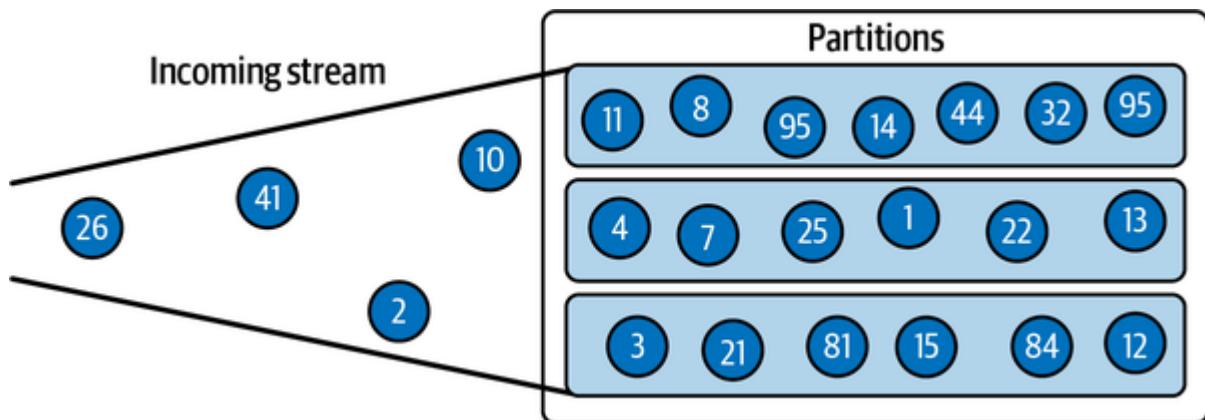


Figure 5-11. An incoming message stream broken into three partitions

Set a partition key so that messages that should be processed together have the same partition key. For example, it is common in IoT settings to want to send all messages from a particular device to the same processing server. We can achieve this by using a device ID as the partition key, and then setting up one server to consume from each partition.

A key concern with stream partitioning is ensuring that your partition key does not generate *hotspotting*—a disproportionate number of messages delivered to one partition. For example, if each IoT device were known to be located in a particular US state, we might use the state as the partition key. Given a device distribution proportional to state population, the partitions containing California, Texas, Florida, and New York might be overwhelmed, with other partitions relatively underutilized. Ensure that your partition key will distribute messages evenly across partitions.

Fault tolerance and resilience

Event-streaming platforms are typically distributed systems, with streams stored on various nodes. If a node goes down, another node replaces it, and the stream is still accessible. This means records aren't lost; you may choose to delete records, but that's another story. This fault tolerance and resilience make streaming platforms a good choice when you need a system that can reliably produce, store, and ingest event data.

Whom You'll Work With

When accessing source systems, it's essential to understand the people with whom you'll work. In our experience, good diplomacy and relationships with the stakeholders of source systems are an underrated and crucial part of successful data engineering.

Who are these stakeholders? Typically, you'll deal with two categories of stakeholders: systems and data stakeholders ([Figure 5-12](#)). A *systems stakeholder* builds and maintains the source systems; these might be software engineers, application developers, and third parties. Data stakeholders own and control access to the data you want, generally handled by IT, a data governance group, or third parties. The systems and data stakeholders are often different people or teams; sometimes, they are the same.

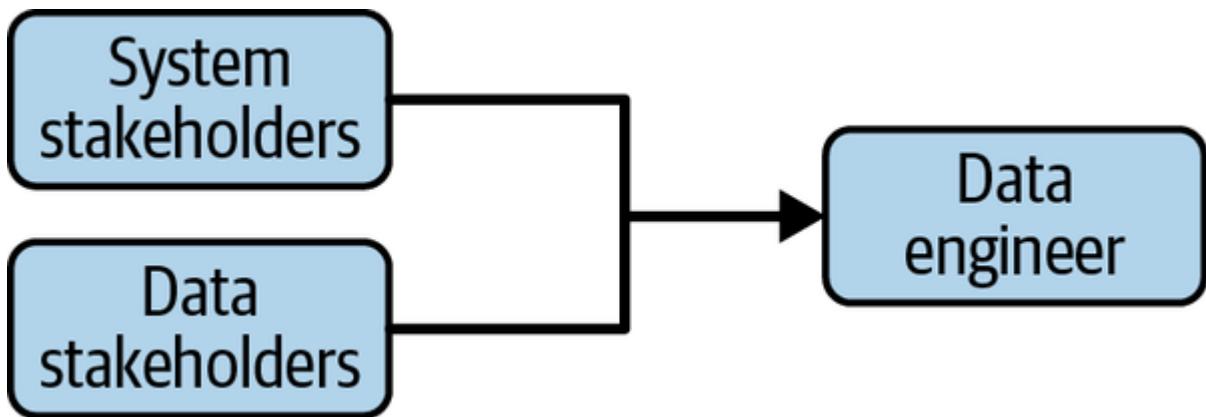


Figure 5-12. The data engineer's upstream stakeholders

You're often at the mercy of the stakeholder's ability to follow correct software engineering, database management, and development practices. Ideally, the stakeholders are doing DevOps and working in an agile manner. We suggest creating a feedback loop between data engineers and stakeholders of the source systems to create awareness of how data is consumed and used. This is among the single most overlooked areas where data engineers can get a lot of value. When something happens to the upstream source data—and something will happen, whether it's a schema or data change, a failed server or database, or other important events—you want to make sure that you're made aware of the impact these issues will have on your data engineering systems.

It might help to have a data contract in place with your upstream source system owners. What is a data contract? James Denmore offers this definition.⁸

A data contract is a written agreement between the owner of a source system and the team ingesting data from that system for use in a data pipeline. The contract should state what data is being extracted, via what method (full, incremental), how often, as well as who (person, team) are the contacts for both the source system and the ingestion. Data contracts should be stored in a well-known and easy-to-find location such as a GitHub repo or internal documentation site. If possible, format data contracts in a standardized form so they can be integrated into the development process or queried programmatically.

In addition, consider establishing an SLA with upstream providers. An SLA provides expectations of what you can expect from the source systems you rely upon. An example of an SLA might be “data from source systems will be reliably available and of high quality.” A service-level objective (SLO) measures performance against what you've agreed to in the SLA. For example, given your example

SLA, an SLO might be “source systems will have 99% uptime.” If a data contract or SLA/SLO seems too formal, at least verbally set expectations for source system guarantees for uptime, data quality, and anything else of importance to you. Upstream owners of source systems need to understand your requirements so they can provide you with the data you need.

Undercurrents and Their Impact on Source Systems

Unlike other parts of the data engineering lifecycle, source systems are generally outside the control of the data engineer. There’s an implicit assumption (some might call it *hope*) that the stakeholders and owners of the source systems—and the data they produce—are following best practices concerning data management, DataOps (and DevOps), DODD (mentioned in [Chapter 2](#)) data architecture, orchestration, and software engineering. The data engineer should get as much upstream support as possible to ensure that the undercurrents are applied when data is generated in source systems. Doing so will make the rest of the steps in the data engineering lifecycle proceed a lot more smoothly.

How do the undercurrents impact source systems? Let’s have a look.

Security

Security is critical, and the last thing you want is to accidentally create a point of vulnerability in a source system. Here are some areas to consider:

- Is the source system architected so data is secure and encrypted, both with data at rest and while data is transmitted?
- Do you have to access the source system over the public internet, or are you using a virtual private network (VPN)?
- Keep passwords, tokens, and credentials to the source system securely locked away. For example, if you’re using Secure Shell (SSH) keys, use a key manager to protect your keys; the same rule applies to passwords—use a password manager or a single sign-on (SSO) provider.
- Do you trust the source system? Always be sure to trust but verify that the source system is legitimate. You don’t want to be on the receiving end of data from a malicious actor.

Data Management

Data management of source systems is challenging for data engineers. In most cases, you will have only peripheral control—if any control at all—over source systems and the data they produce. To the extent possible, you should understand the way data is managed in source systems since this will directly influence how you ingest, store, and transform the data.

Here are some areas to consider:

Data governance

Are upstream data and systems governed in a reliable, easy-to-understand fashion? Who manages the data?

Data quality

How do you ensure data quality and integrity in upstream systems? Work with source system teams to set expectations on data and communication.

Schema

Expect that upstream schemas will change. Where possible, collaborate with source system teams to be notified of looming schema changes.

Master data management

Is the creation of upstream records controlled by a master data management practice or system?

Privacy and ethics

Do you have access to raw data, or will the data be obfuscated? What are the implications of the source data? How long is it retained? Does it shift locations based on retention policies?

Regulatory

Based upon regulations, are you supposed to access the data?

DataOps

Operational excellence—DevOps, DataOps, MLOps, *X*Ops—should extend up and down the entire stack and support the data engineering and lifecycle. While this is ideal, it's often not fully realized.

Because you're working with stakeholders who control both the source systems and the data they produce, you need to ensure that you can observe and monitor the uptime and usage of the source systems and respond when incidents occur. For example, when the application database you depend on for CDC exceeds its I/O capacity and needs to be rescaled, how will that affect your ability to receive data from this system? Will you be able to access the data, or will it be unavailable until the database is rescaled? How will this affect reports? In another example, if the software engineering team is continuously deploying, a code change may cause unanticipated failures in the application itself. How will the failure impact your ability to access the databases powering the application? Will the data be up-to-date?

Set up a clear communication chain between data engineering and the teams supporting the source systems. Ideally, these stakeholder teams have incorporated DevOps into their workflow and culture. This will go a long way to accomplishing the goals of DataOps (a sibling of DevOps), to address and reduce errors quickly. As we mentioned earlier, data engineers need to weave themselves into the DevOps practices of stakeholders, and vice versa. Successful DataOps works when all people are on board and focus on making systems holistically work.

A few DataOps considerations are as follows:

Automation

There's the automation impacting the source system, such as code updates and new features. Then there's the DataOps automation that you've set up for your data workflows. Does an issue in the source system's automation impact your data workflow automation? If so, consider decoupling these systems so they can perform automation independently.

Observability

How will you know when there's an issue with a source system, such as an outage or a data-quality issue? Set up monitoring for source system uptime (or use the monitoring created by the team that owns the source system). Set up checks to ensure that data from the source system conforms with expectations for downstream usage. For example, is the data of good quality? Is the schema conformant? Are customer records consistent? Is data hashed as stipulated by the internal policy?

Incident response

What's your plan if something bad happens? For example, how will your data pipeline behave if a source system goes offline? What's your plan to backfill the “lost” data once the source system is back online?

Data Architecture

Similar to data management, unless you’re involved in the design and maintenance of the source system architecture, you’ll have little impact on the upstream source system architecture. You should also understand how the upstream architecture is designed and its strengths and weaknesses. Talk often with the teams responsible for the source systems to understand the factors discussed in this section and ensure that their systems can meet your expectations. Knowing where the architecture performs well and where it doesn’t will impact how you design your data pipeline.

Here are some things to consider regarding source system architectures:

Reliability

All systems suffer from entropy at some point, and outputs will drift from what’s expected. Bugs are introduced, and random glitches happen. Does the system produce predictable outputs? How often can we expect the system to fail? What’s the mean time to repair to get the system back to sufficient reliability?

Durability

Everything fails. A server might die, a cloud’s zone or region could go offline, or other issues may arise. You need to account for how an inevitable failure or outage will affect your managed data systems. How does the source system handle data loss from hardware failures or network outages? What’s the plan for handling outages for an extended period and limiting the blast radius of an outage?

Availability

What guarantees that the source system is up, running, and available when it’s supposed to be?

People

Who’s in charge of the source system’s design, and how will you know if breaking changes are made in the architecture? A data engineer needs to work with the teams who maintain the source systems and ensure that these systems are architected reliably. Create an SLA with the source system team to set expectations about potential system failure.

Orchestration

When orchestrating within your data engineering workflow, you’ll primarily be concerned with making sure your orchestration can access the source system, which requires the correct network access, authentication, and authorization.

Here are some things to think about concerning orchestration for source systems:

Cadence and frequency

Is the data available on a fixed schedule, or can you access new data whenever you want?

Common frameworks

Do the software and data engineers use the same container manager, such as Kubernetes? Would it make sense to integrate application and data workloads into the same Kubernetes cluster? If you’re using an orchestration framework like Airflow, does it make sense to integrate it with the

upstream application team? There's no correct answer here, but you need to balance the benefits of integration with the risks of tight coupling.

Software Engineering

As the data landscape shifts to tools that simplify and automate access to source systems, you'll likely need to write code. Here are a few considerations when writing code to access a source system:

Networking

Make sure your code will be able to access the network where the source system resides. Also, always think about secure networking. Are you accessing an HTTPS URL over the public internet, SSH, or a VPN?

Authentication and authorization

Do you have the proper credentials (tokens, username/passwords) to access the source system? Where will you store these credentials so they don't appear in your code or version control? Do you have the correct IAM roles to perform the coded tasks?

Access patterns

How are you accessing the data? Are you using an API, and how are you handling REST/GraphQL requests, response data volumes, and pagination? If you're accessing data via a database driver, is the driver compatible with the database you're accessing? For either access pattern, how are things like retries and timeouts handled?

Orchestration

Does your code integrate with an orchestration framework, and can it be executed as an orchestrated workflow?

Parallelization

How are you managing and scaling parallel access to source systems?

Deployment

How are you handling the deployment of source code changes?

Conclusion

Source systems and their data are vital in the data engineering lifecycle. Data engineers tend to treat source systems as “someone else's problem”—do this at your peril! Data engineers who abuse source systems may need to look for another job when production goes down.

If there's a stick, there's also a carrot. Better collaboration with source system teams can lead to higher-quality data, more successful outcomes, and better data products. Create a bidirectional flow of communications with your counterparts on these teams; set up processes to notify of schema and application changes that affect analytics and ML. Communicate your data needs proactively to assist application teams in the data engineering process.

Be aware that the integration between data engineers and source system teams is growing. One example is reverse ETL, which has long lived in the shadows but has recently risen into prominence. We also discussed that the event-streaming platform could serve a role in event-driven architectures and analytics; a source system can also be a data engineering system. Build shared systems where it makes sense to do so.

Look for opportunities to build user-facing data products. Talk to application teams about analytics they would like to present to their users or places where ML could improve the user experience. Make application teams stakeholders in data engineering, and find ways to share your successes.

Now that you understand the types of source systems and the data they generate, we'll next look at ways to store this data.

Additional Resources

- Confluent's [“Schema Evolution and Compatibility” documentation](#)
- [Database Internals](#) by Alex Petrov (O'Reilly)
- *Database System Concepts* by Abraham (Avi) Silberschatz et al. (McGraw Hill)
- [“The Log: What Every Software Engineer Should Know About Real-Time Data’s Unifying Abstraction”](#) by Jay Kreps
- [“Modernizing Business Data Indexing”](#) by Benjamin Douglas and Mohammad Mohtasham
- [“NoSQL: What’s in a Name”](#) by Eric Evans
- [“Test Data Quality at Scale with Deequ”](#) by Dustin Lange et al.
- [“The What, Why, and When of Single-Table Design with DynamoDB”](#) by Alex DeBrie

¹ Keith D. Foote, “A Brief History of Non-Relational Databases,” Dataversity, June 19, 2018, <https://oreil.ly/5Ukg2>.

² Nenad Dalal’s excellent series on the [history of MongoDB](#) recounts some harrowing tales of database abuse and its consequences for fledgling startups.

³ Martin Kleppmann, *Designing Data-Intensive Applications* (Sebastopol, CA: O'Reilly, 2017), 49, <https://oreil.ly/v1NhG>.

⁴ Aashish Mehra, “Graph Database Market Worth \$5.1 Billion by 2026: Exclusive Report by MarketsandMarkets,” Cision PR Newswire, July 30, 2021, <https://oreil.ly/mGVkY>.

⁵ For one example, see Michael S. Mikowski, “RESTful APIs: The Big Lie,” August 10, 2015, <https://oreil.ly/rqja3>.

⁶ Martin Fowler, “How to Move Beyond a Monolithic Data Lake to a Distributed Data Mesh,” MartinFowler.com, May 20, 2019, <https://oreil.ly/TEdJF>.

⁷ Whether *exactly once* is possible is a semantical debate. Technically, exactly once delivery is impossible to guarantee, as illustrated by the [Two Generals Problem](#).

⁸ James Denmore, *Data Pipelines Pocket Reference* (Sebastopol, CA: O'Reilly), <https://oreil.ly/8QdkJ>. Read the book for more information on how a data contract should be written.