Schema

What is the expected form of the data? What is the file format? Is it structured, semistructured, or unstructured? What data types are expected? How does the data fit into a larger hierarchy? Is it connected to other data through shared keys or other relationships?

Note that schema need not be *relational*. Rather, data becomes more useful when we have as much information about its structure and organization. For images stored in a data lake, this schema information might explain the image format, resolution, and the way the images fit into a larger hierarchy.

Schema can function as a sort of Rosetta stone, instructions that tell us how to read the data. Two major schema patterns exist: schema on write and schema on read. *Schema on write* is essentially the traditional data warehouse pattern: a table has an integrated schema; any writes to the table must conform. To support schema on write, a data lake must integrate a schema metastore.

With *schema on read*, the schema is dynamically created when data is written, and a reader must determine the schema when reading the data. Ideally, schema on read is implemented using file formats that implement built-in schema information, such as Parquet or JSON. CSV files are notorious for schema inconsistency and are not recommended in this setting.

The principal advantage of schema on write is that it enforces data standards, making data easier to consume and utilize in the future. Schema on read emphasizes flexibility, allowing virtually any data to be written. This comes at the cost of greater difficulty consuming data in the future.

Separation of Compute from Storage

A key idea we revisit throughout this book is the separation of compute from storage. This has emerged as a standard data access and query pattern in today's cloud era. Data lakes, as we discussed, store data in object stores and spin up temporary compute capacity to read and process it. Most fully

managed OLAP products now rely on object storage behind the scenes. To understand the motivations for separating compute and storage, we should first look at the colocation of compute and storage.

Colocation of compute and storage

Colocation of compute and storage has long been a standard method to improve database performance. For transactional databases, data colocation allows fast, low-latency disk reads and high bandwidth. Even when we virtualize storage (e.g., using Amazon EBS), data is located relatively close to the host machine.

The same basic idea applies for analytics query systems running across a cluster of machines. For example, with HDFS and MapReduce, the standard approach is to locate data blocks that need to be scanned in the cluster, and then push individual *map* jobs out to these blocks. The data scan and processing for the map step are strictly local. The *reduce* step involves shuffling data across the cluster, but keeping map steps local effectively preserves more bandwidth for shuffling, delivering better overall performance; map steps that filter heavily also dramatically reduce the amount of data to be shuffled.

Separation of compute and storage

If colocation of compute and storage delivers high performance, why the shift toward separation of compute and storage? Several motivations exist.

Ephemerality and scalability

In the cloud, we've seen a dramatic shift toward ephemerality. In general, it's cheaper to buy and host a server than to rent it from a cloud provider, *provided that you're running it 24 hours a day nonstop for years on end*. In practice, workloads vary dramatically, and significant efficiencies are realized with a pay-as-you-go model if servers can scale up and down. This is true for web servers in online retail, and it is also true for big data batch jobs that may run only periodically.

Ephemeral compute resources allow engineers to spin up massive clusters to complete jobs on time, and then delete clusters when these jobs are done. The performance benefits of temporarily operating at ultra-high scale can outweigh the bandwidth limitations of object storage.

Data durability and availability

Cloud object stores significantly mitigate the risk of data loss and generally provide extremely high uptime (availability). For example, S3 stores data across multiple zones; if a natural disaster destroys a zone, data is still available from the remaining zones. Having multiple zones available also reduces the odds of a data outage. If resources in one zone go down, engineers can spin up the same resources in a different zone.

The potential for a misconfiguration that destroys data in object storage is still somewhat scary, but simple-to-deploy mitigations are available. Copying data to multiple cloud regions reduces this risk since configuration changes are generally deployed to only one region at a time. Replicating data to multiple storage providers can further reduce the risk.

Hybrid separation and colocation

The practical realities of separating compute from storage are more complicated than we've implied. In reality, we constantly hybridize colocation and separation to realize the benefits of both approaches. This hybridization is typically done in two ways: multitier caching and hybrid object storage.

With *multitier caching*, we utilize object storage for long-term data retention and access but spin up local storage to be used during queries and various stages of data pipelines. Both Google and Amazon offer versions of hybrid object storage (object storage that is tightly integrated with compute).

Let's look at examples of how some popular processing engines hybridize separation and colocation of storage and compute.

Example: AWS EMR with S3 and HDFS

Big data services like Amazon EMR spin up temporary HDFS clusters to process data. Engineers have the option of referencing both S3 and HDFS as a filesystem. A common pattern is to stand up HDFS on SSD drives, pull from S3, and save data from intermediate processing steps on local HDFS. Doing so can realize significant performance gains over processing directly from S3. Full results are written back to S3 once the cluster completes its steps, and the cluster and HDFS are deleted. Other consumers read the output data directly from S3.

Example: Apache Spark

In practice, Spark generally runs jobs on HDFS or some other ephemeral distributed filesystem to support performant storage of data between processing steps. In addition, Spark relies heavily on in-memory storage of data to improve processing. The problem with owning the infrastructure for running Spark is that dynamic RAM (DRAM) is extremely expensive; by separating compute and storage in the cloud, we can rent large quantities of memory and then release that memory when the job completes.

Example: Apache Druid

Apache Druid relies heavily on SSDs to realize high performance. Since SSDs are significantly more expensive than magnetic disks, Druid keeps only one copy of data in its cluster, reducing “live” storage costs by a factor of three.

Of course, maintaining data durability is still critical, so Druid uses an object store as its durability layer. When data is ingested, it’s processed, serialized into compressed columns, and written to cluster SSDs and object storage. In the event of node failure or cluster data corruption, data can be automatically recovered to new nodes. In addition, the cluster can be shut down and then fully recovered from SSD storage.

Example: Hybrid Object Storage

Google’s Colossus file storage system supports fine-grained control of data block location, although this functionality is not exposed directly to the public. BigQuery uses this feature to colocate customer tables in a single

location, allowing ultra-high bandwidth for queries in that location.[4] We refer to this as *hybrid object storage* because it combines the clean abstractions of object storage with some advantages of colocating compute and storage. Amazon also offers some notion of hybrid object storage through S3 Select, a feature that allows users to filter S3 data directly in S3 clusters before data is returned across the network.

We speculate that public clouds will adopt hybrid object storage more widely to improve the performance of their offerings and make more efficient use of available network resources. Some may be already doing so without disclosing this publicly.

The concept of hybrid object storage underscores that there can still be advantages to having low-level access to hardware rather than relying on someone else's public cloud. Public cloud services do not expose low-level details of hardware and systems (e.g., data block locations for Colossus), but these details can be extremely useful in performance optimization and enhancement. See our discussion of cloud economics in Chapter 4.

While we're now seeing a mass migration of data to public clouds, we believe that many hyper-scale data service vendors that currently run on public clouds provided by other vendors may build their data centers in the future, albeit with deep network integration into public clouds.

Zero copy cloning

Cloud-based systems based around object storage support *zero copy cloning*. This typically means that a new virtual copy of an object is created (e.g., a new table) without necessarily physically copying the underlying data. Typically, new pointers are created to the raw data files, and future changes to these tables will not be recorded in the old table. For those familiar with the inner workings of object-oriented languages such as Python, this type of "shallow" copying is familiar from other contexts.

Zero copy cloning is a compelling feature, but engineers must understand its strengths and limitations. For example, cloning an object in a data lake

environment and then deleting the files in the original object might also wipe out the new object.

For fully managed object-store-based systems (e.g., Snowflake and BigQuery), engineers need to be extremely familiar with the exact limits of shallow copying. Engineers have more access to underlying object storage in data lake systems such as Databricks—a blessing and a curse. Data engineers should exercise great caution before deleting any raw files in the underlying object store. Databricks and other data lake management technologies sometimes also support a notion of *deep copying*, whereby all underlying data objects are copied. This is a more expensive process, but also more robust in the event that files are unintentionally lost or deleted.

Data Storage Lifecycle and Data Retention

Storing data isn't as simple as just saving it to object storage or disk and forgetting about it. You need to think about the data storage lifecycle and data retention. When you think about access frequency and use cases, ask, "How important is the data to downstream users, and how often do they need to access it?" This is the data storage lifecycle. Another question you should ask is, "How long should I keep this data?" Do you need to retain data indefinitely, or are you fine discarding it past a certain time frame? This is data retention. Let's dive into each of these.

Hot, warm, and cold data

Did you know that data has a temperature? Depending on how frequently data is accessed, we can roughly bucket the way it is stored into three categories of persistence: hot, warm, and cold. Query access patterns differ for each dataset (Figure 6-9). Typically, newer data is queried more often than older data. Let's look at hot, cold, and warm data in that order.

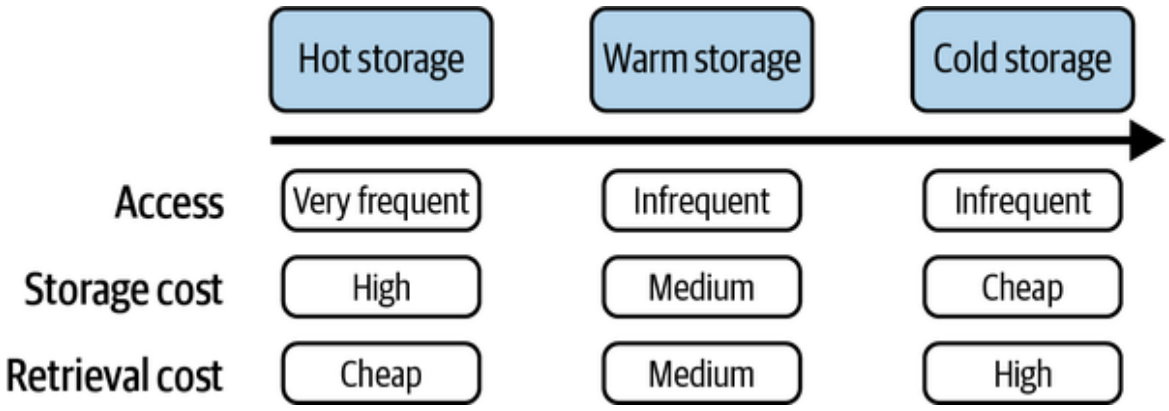

Figure 6-9. Hot, warm, and cold data costs associated with access frequency

Hot data

Hot data has instant or frequent access requirements. The underlying storage for hot data is suited for fast access and reads, such as SSD or memory. Because of the type of hardware involved with hot data, storing hot data is often the most expensive form of storage. Example use cases for hot data include retrieving product recommendations and product page results. The cost of storing hot data is the highest of these three storage tiers, but retrieval is often inexpensive.

Query results cache is another example of hot data. When a query is run, some query engines will persist the query results in the cache. For a limited time, when the same query is run, instead of rerunning the same query against storage, the query results cache serves the cached results. This allows for much faster query response times versus redundantly issuing the same query repeatedly. In upcoming chapters, we cover query results caches in more detail.

Warm data

Warm data is accessed semi-regularly, say, once per month. No hard and fast rules indicate how often warm data is accessed, but it's less than hot data and more than cold data. The major cloud providers offer object storage tiers that accommodate warm data. For example,S3 offers an Infrequently Accessed Tier, and Google Cloud has a similar storage tier called Nearline. Vendors give their models of recommended access frequency, and engineers can also do their cost modeling and monitoring.

Storage of warm data is cheaper than hot data, with slightly more expensive retrieval costs.

Cold data

On the other extreme, *cold data* is infrequently accessed data. The hardware used to archive cold data is typically cheap and durable, such as HDD, tape storage, and cloud-based archival systems. Cold data is mainly meant for long-term archival, when there's little to no intention to access the data. Though storing cold data is cheap, retrieving cold data is often expensive.

Storage tier considerations

When considering the storage tier for your data, consider the costs of each tier. If you store all of your data in hot storage, all of the data can be accessed quickly. But this comes at a tremendous price! Conversely, if you store all data in cold storage to save on costs, you'll certainly lower your storage costs, but at the expense of prolonged retrieval times and high retrieval costs if you need to access data. The storage price goes down from faster/higher performing storage to lower storage.

Cold storage is popular for archiving data. Historically, cold storage involved physical backups and often mailing this data to a third party that would archive it in a literal vault. Cold storage is increasingly popular in the cloud. Every cloud vendor offers a cold data solution, and you should weigh the cost of pushing data into cold storage versus the cost and time to retrieve the data.

Data engineers need to account for spillover from hot to warm/cold storage. Memory is expensive and finite. For example, if hot data is stored in memory, it can be spilled to disk when there's too much new data to store and not enough memory. Some databases may move infrequently accessed data to warm or cold tiers, offloading the data to either HDD or object storage. The latter is increasingly more common because of the cost-effectiveness of object storage. If you're in the cloud and using managed services, disk spillover will happen automatically.

If you're using cloud-based object storage, create automated lifecycle policies for your data. This will drastically reduce your storage costs. For example, if your data needs to be accessed only once a month, move the data to an infrequent access storage tier. If your data is 180 days old and not accessed for current queries, move it to an archival storage tier. In both cases, you can automate the migration of data away from regular object storage, and you'll save money. That said, consider the retrieval costs—both in time and money—using infrequent or archival style storage tiers. Access and retrieval times and costs may vary depending on the cloud provider. Some cloud providers make it simple and cheap to migrate data into archive storage, but it is costly and slow to retrieve your data.

Data retention

Back in the early days of "big data," there was a tendency to err on the side of accumulating every piece of data possible, regardless of its usefulness. The expectation was, "we might need this data in the future." This data hoarding inevitably became unwieldy and dirty, giving rise to data swamps, and regulatory crackdowns on data retention, among other consequences and nightmares. Nowadays, data engineers need to consider data retention: what data do you *need* to keep, and how *long* should you keep it? Here are some things to think about with data retention.

Value

Data is an asset, so you should know the value of the data you're storing. Of course, value is subjective and depends on what it's worth to your immediate use case and your broader organization. Is this data impossible to re-create, or can it easily be re-created by querying upstream systems? What's the impact to downstream users if this data is available versus if it is not?

Time

The value to downstream users also depends upon the age of the data. New data is typically more valuable and frequently accessed than older data. Technical limitations may determine how long you can store data in certain

storage tiers. For example, if you store hot data in cache or memory, you'll likely need to set a time to live (TTL), so you can expire data after a certain point or persist it to warm or cold storage. Otherwise, your hot storage will become full, and queries against the hot data will suffer from performance lags.

Compliance

Certain regulations (e.g., HIPAA and Payment Card Industry, or PCI) might require you to keep data for a certain time. In these situations, the data simply needs to be accessible upon request, even if the likelihood of an access request is low. Other regulations might require you to hold data for only a limited period of time, and you'll need to have the ability to delete specific information on time and within compliance guidelines. You'll need a storage and archival data process—along with the ability to search the data—that fits the retention requirements of the particular regulation with which you need to comply. Of course, you'll want to balance compliance against cost.

Cost

Data is an asset that (hopefully) has an ROI. On the cost side of ROI, an obvious storage expense is associated with data. Consider the timeline in which you need to retain data. Given our discussion about hot, warm, and cold data, implement automatic data lifecycle management practices and move the data to cold storage if you don't need the data past the required retention date. Or delete data if it's truly not needed.

Single-Tenant Versus Multitenant Storage

In Chapter 3, we covered the trade-offs between single-tenant and multitenant architecture. To recap, with *single-tenant* architecture, each group of tenants (e.g., individual users, groups of users, accounts, or customers) gets its own dedicated set of resources such as networking, compute, and storage. A *multitenant* architecture inverts this and shares these resources among groups of users. Both architectures are widely used.

This section looks at the implications of single-tenant and multitenant storage.

Adopting single-tenant storage means that every tenant gets their dedicated storage. In the example in Figure 6-10, each tenant gets a database. No data is shared among these databases, and storage is totally isolated. An example of using single-tenant storage is that each customer's data must be stored in isolation and cannot be blended with any other customer's data. In this case, each customer gets their own database.

Figure 6-10. In single-tenant storage, each tenant gets their own database

Separate data storage implies separate and independent schemas, bucket structures, and everything related to storage. This means you have the liberty of designing each tenant's storage environment to be uniform or let them evolve however they may. Schema variation across customers can be an advantage and a complication; as always, consider the trade-offs. If each tenant's schema isn't uniform across all tenants, this has major consequences if you need to query multiple tenants' tables to create a unified view of all tenant data.

Multitenant storage allows for the storage of multiple tenants within a single database. For example, instead of the single-tenant scenario where customers get their own database, multiple customers may reside in the same database schemas or tables in a multitenant database. Storing multitenant data means each tenant's data is stored in the same place (Figure 6-11).

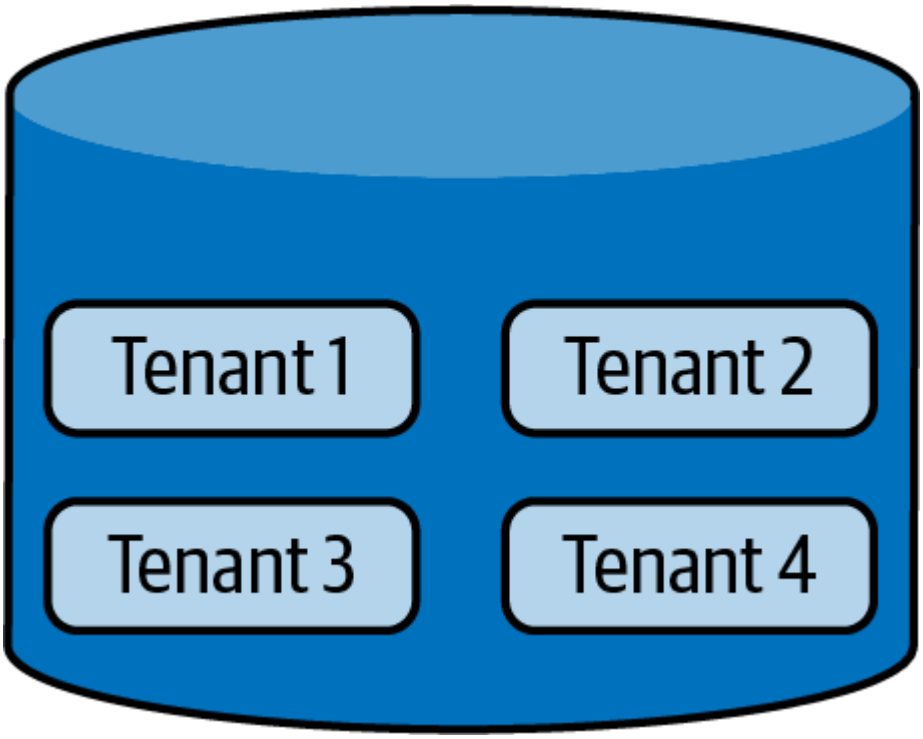

Figure 6-11. In this multitenant storage, four tenants occupy the same database

You need to be aware of querying both single and multitenant storage, which we cover in more detail in Chapter 8.

Whom You'll Work With

Storage is at the heart of data engineering infrastructure. You'll interact with the people who own your IT infrastructure—typically, DevOps, security, and cloud architects. Defining domains of responsibility between data engineering and other teams is critical. Do data engineers have the authority to deploy their infrastructure in an AWS account, or must another team handle these changes? Work with other teams to define streamlined processes so that teams can work together efficiently and quickly.

The division of responsibilities for data storage will depend significantly on the maturity of the organization involved. The data engineer will likely manage the storage systems and workflow if the company is early in its data maturity. If the company is later in its data maturity, the data engineer will probably manage a section of the storage system. This data engineer will

also likely interact with engineers on either side of storage—ingestion and transformation.

The data engineer needs to ensure that the storage systems used by downstream users are securely available, contain high-quality data, have ample storage capacity, and perform when queries and transformations are run.

Undercurrents

The undercurrents for storage are significant because storage is a critical hub for all stages of the data engineering lifecycle. Unlike other undercurrents for which data might be in motion (ingestion) or queried and transformed, the undercurrents for storage differ because storage is so ubiquitous.

Security

While engineers often view security as an impediment to their work, they should embrace the idea that security is a key enabler. Robust security with fine-grained data access control allows data to be shared and consumed more widely within a business. The value of data goes up significantly when this is possible.

As always, exercise the principle of least privilege. Don’t give full database access to anyone unless required. This means most data engineers don’t need full database access in practice. Also, pay attention to the column, row, and cell-level access controls in your database. Give users only the information they need and no more.

Data Management

Data management is critical as we read and write data with storage systems.

Data catalogs and metadata management

Data is enhanced by robust metadata. Cataloging enables data scientists, analysts, and ML engineers by enabling data discovery. Data lineage accelerates the time to track down data problems and allows consumers to locate upstream raw sources. As you build out your storage systems, invest in your metadata. Integration of a data dictionary with these other tools allows users to share and record institutional knowledge robustly.

Metadata management also significantly enhances data governance. Beyond simply enabling passive data cataloging and lineage, consider implementing analytics over these systems to get a clear, active picture of what's happening with your data.

Data versioning in object storage

Major cloud object storage systems enable data versioning. Data versioning can help with error recovery when processes fail, and data becomes corrupted. Versioning is also beneficial for tracking the history of datasets used to build models. Just as code version control allows developers to track down commits that cause bugs, data version control can aid ML engineers in tracking changes that lead to model performance degradation.

Privacy

GDPR and other privacy regulations have significantly impacted storage system design. Any data with privacy implications has a lifecycle that data engineers must manage. Data engineers must be prepared to respond to data deletion requests and selectively remove data as required. In addition, engineers can accommodate privacy and security through anonymization and masking.

DataOps

DataOps is not orthogonal to data management, and a significant area of overlap exists. DataOps concerns itself with traditional operational monitoring of storage systems and monitoring the data itself, inseparable from metadata and quality.

Systems monitoring

Data engineers must monitor storage in a variety of ways. This includes monitoring infrastructure storage components, where they exist, but also monitoring object storage and other "serverless" systems. Data engineers should take the lead on FinOps (cost management), security monitoring, and access monitoring.

Observing and monitoring data

While metadata systems as we've described are critical, good engineering must consider the entropic nature of data by actively seeking to understand its characteristics and watching for major changes. Engineers can monitor data statistics, apply anomaly detection methods or simple rules, and actively test and validate for logical inconsistencies.

Data Architecture

Chapter 3 covers the basics of data architecture, as storage is the critical underbelly of the data engineering lifecycle.

Consider the following data architecture tips. Design for required reliability and durability. Understand the upstream source systems and how that data, once ingested, will be stored and accessed. Understand the types of data models and queries that will occur downstream.

If data is expected to grow, can you negotiate storage with your cloud provider? Take an active approach to FinOps, and treat it as a central part of architecture conversations. Don't prematurely optimize, but prepare for scale if business opportunities exist in operating on large data volumes.

Lean toward fully managed systems, and understand provider SLAs. Fully managed systems are generally far more robust and scalable than systems you have to babysit.

Orchestration

Orchestration is highly entangled with storage. Storage allows data to flow through pipelines, and orchestration is the pump. Orchestration also helps engineers cope with the complexity of data systems, potentially combining many storage systems and query engines.

Software Engineering

We can think about software engineering in the context of storage in two ways. First, the code you write should perform well with your storage system. Make sure the code you write stores the data correctly and doesn't accidentally cause data, memory leaks, or performance issues. Second, define your storage infrastructure as code and use ephemeral compute resources when it's time to process your data. Because storage is increasingly distinct from compute, you can automatically spin resources up and down while keeping your data in object storage. This keeps your infrastructure clean and avoids coupling your storage and query layers.

Conclusion

Storage is everywhere and underlays many stages of the data engineering lifecycle. In this chapter, you learned about the raw ingredients, types, abstractions, and big ideas around storage systems. Gain deep knowledge of the inner workings and limitations of the storage systems you'll use. Know the types of data, activities, and workloads appropriate for your storage.

Additional Resources

- "Column-oriented DBMS" Wikipedia page
- "Rowise vs. Columnar Database? Theory and in Practice" by Mangat Rai Modi
- "The Design and Implementation of Modern Column-Oriented Database Systems" by Daniel Abadi et al.

- “Diving Into Delta Lake: Schema Enforcement and Evolution” by Burak Yavuz et al.
- “Snowflake Solution Anti-Patterns: The Probable Data Scientist” by John Aven
- “The What, When, Why, and How of Incremental Loads” by Tim Mitchell
- IDC’s “Data Creation and Replication Will Grow at a Faster Rate than Installed Storage Capacity, According to the IDC Global DataSphere and StorageSphere Forecasts” press release
- “What Is a Vector Database?” by Bryan Turriff
- “Hot Data vs. Cold Data: Why It Matters” by Afzaal Ahmad Zeeshan

---

1 Andy Klein, “Hard Disk Drive (HDD) vs. Solid-State Drive (SSD): What’s the Diff?,” Backblaze blog, October 5, 2021, *https://oreil.ly/XBps8*.

2 Benoit Dageville, “The Snowflake Elastic Data Warehouse,” *SIGMOD ’16: Proceedings of the 2016 International Conference on Management of Data* (June 2016): 215–226, *https://oreil.ly/Tc1su*.

3 James Dixon, “Data Lakes Revisited,” *James Dixon’s Blog*, September 25, 2014, *https://oreil.ly/FH25v*.

4 Valliappa Lakshmanan and Jordan Tigani, *GoogleBig Query: The Definitive Guide* (Seastopol, CA: O’Reilly, 2019), page 15, *https://oreil.ly/5aXXu*

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.

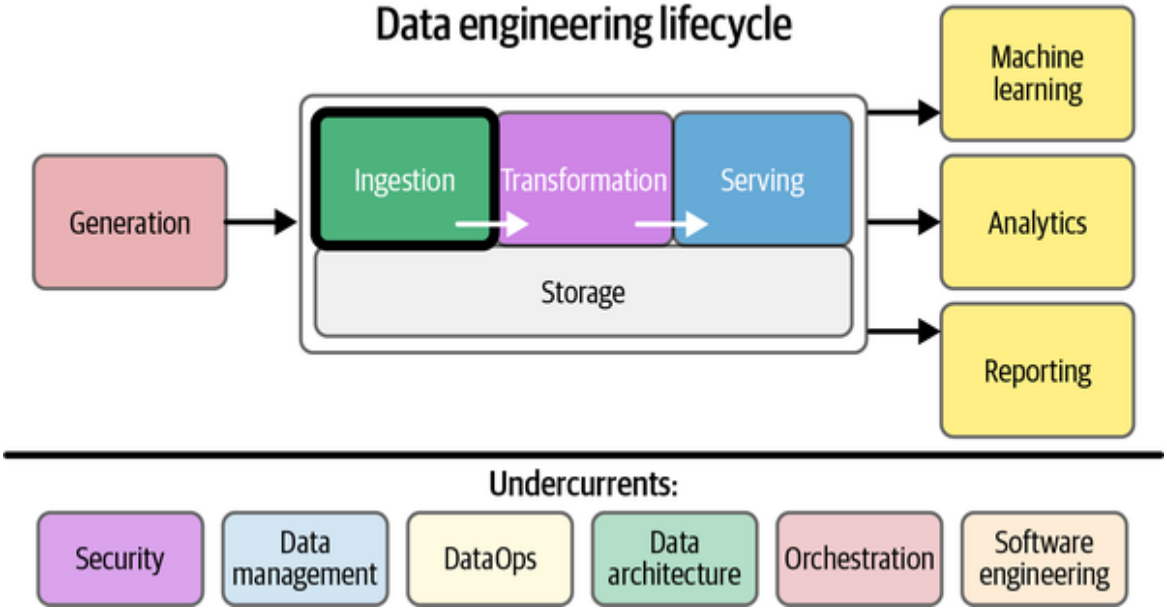

Figure 7-1. To begin processing data, we must ingest it

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 elastic scalability
- Reliability and durability
- Payload
- Push versus pull versus poll patterns

Let's look at each of these.

Bounded Versus Unbounded

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.

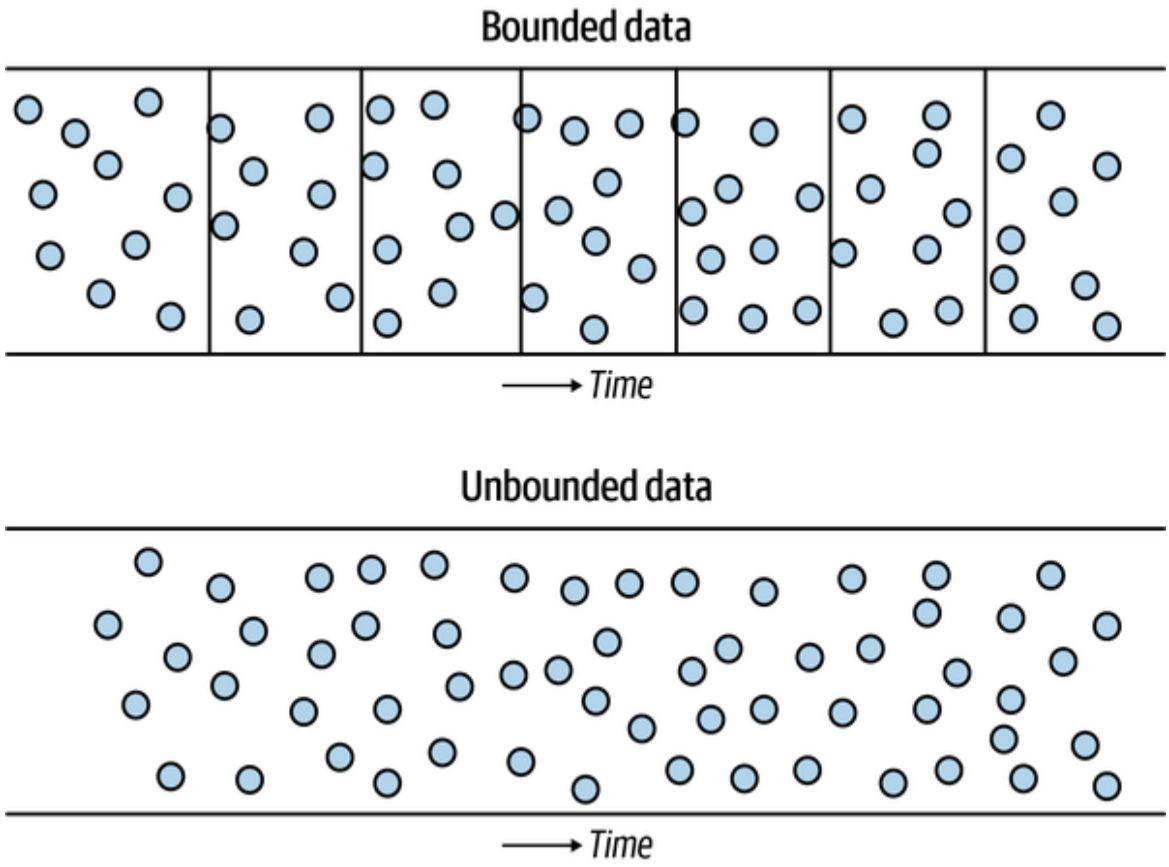

Figure 7-3. Bounded versus unbounded data

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.

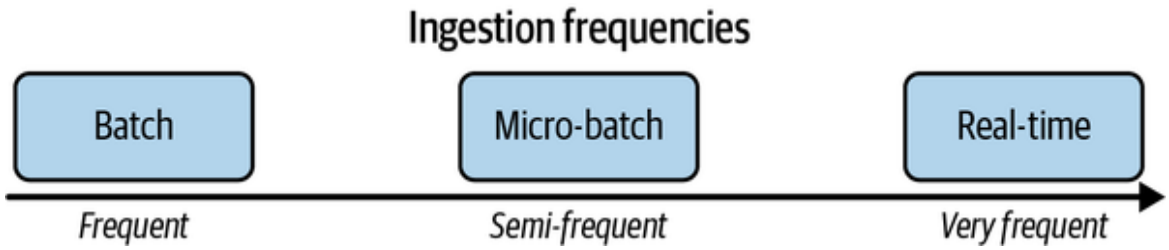

Figure 7-4. The spectrum batch to real-time ingestion frequencies

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.

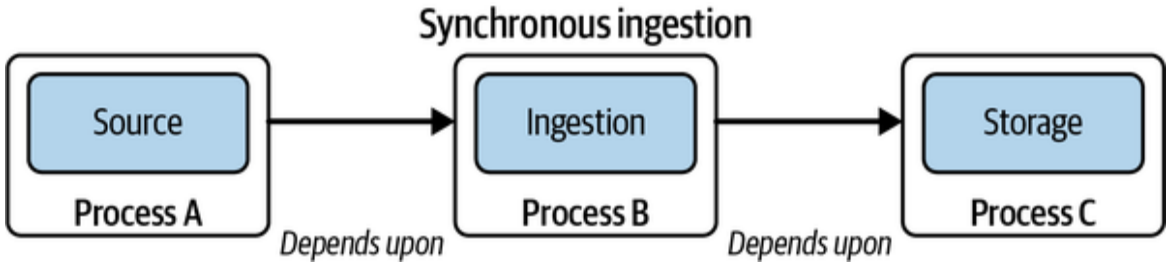

Figure 7-5. A synchronous ingestion process runs as discrete batch steps

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.

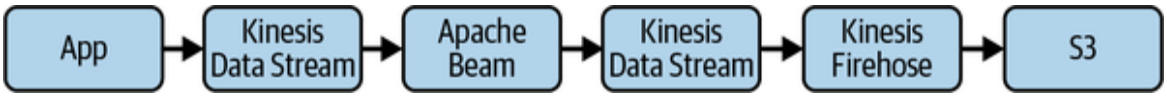

Figure 7-6. Asynchronous processing of an event stream in AWS

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 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 schema changes in upstream and downstream systems

Schema changes 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.

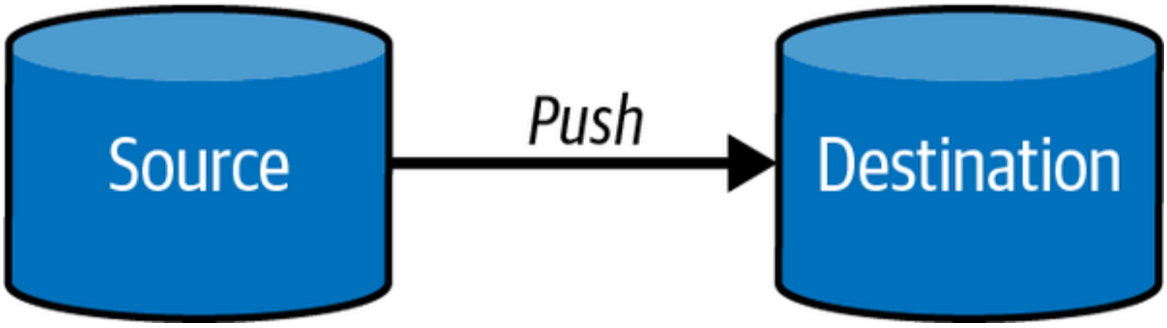

Figure 7-7. Pushing data from source to destination

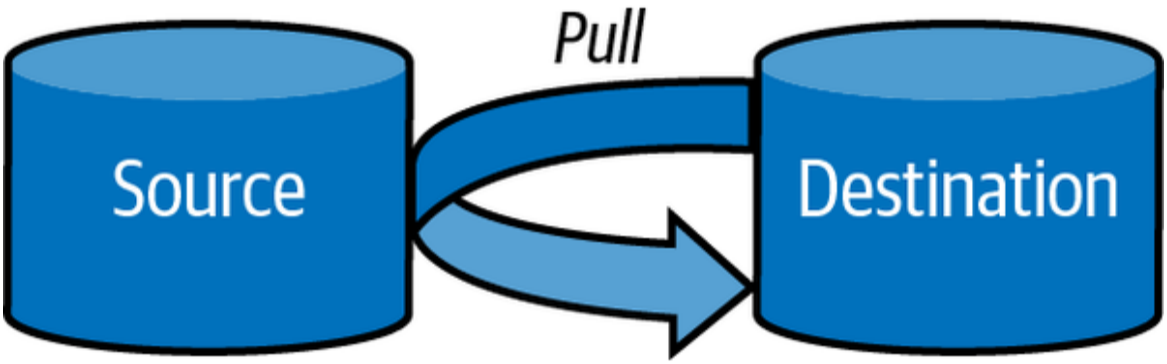

Figure 7-8. A destination pulling data from a source

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.

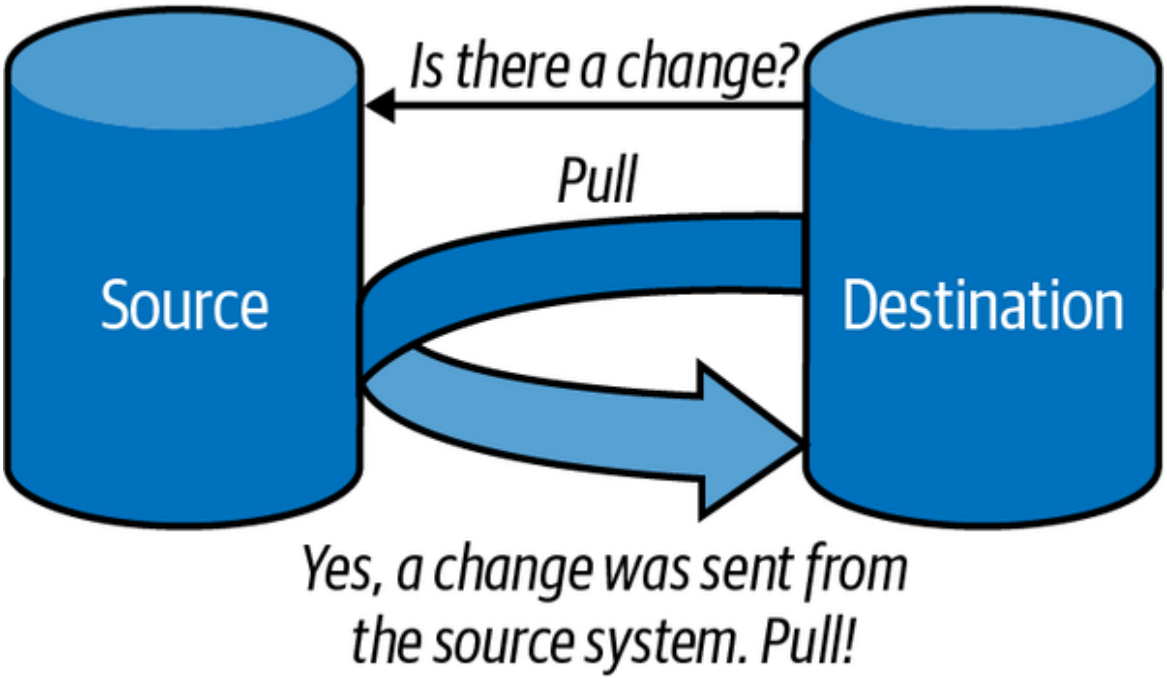

Figure 7-9. Polling for changes in a source system

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).

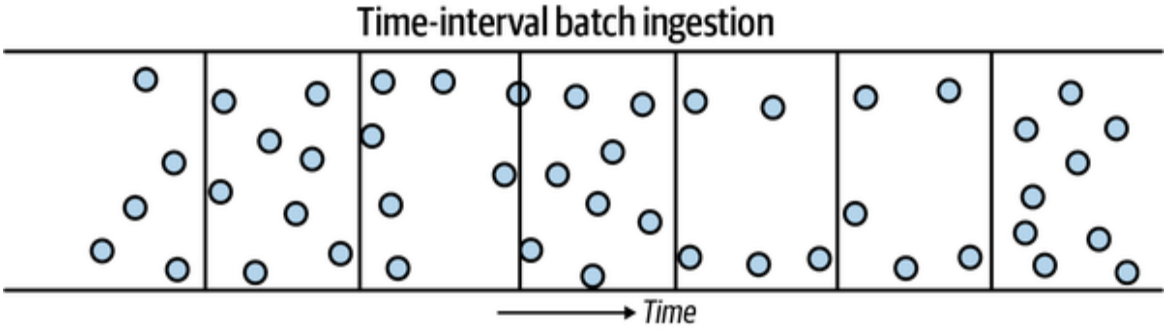

Figure 7-10. Time-interval batch ingestion

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.

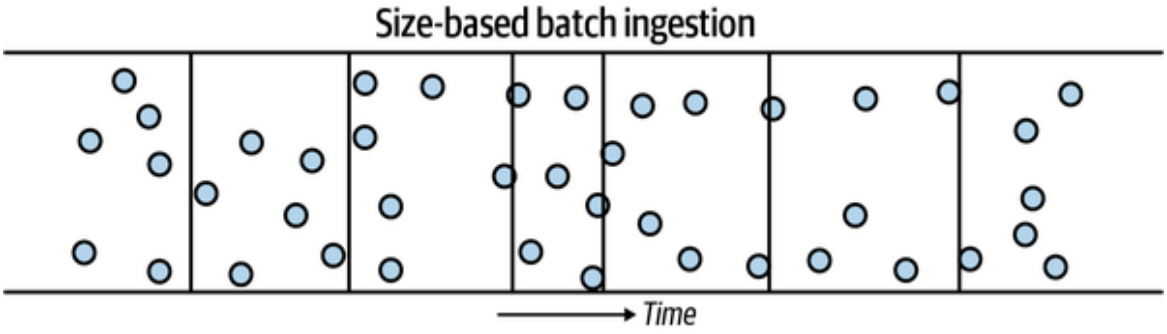

Figure 7-11. Size-based batch ingestion

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. This section covers the extract (*E*) and the load (*L*) parts of ETL and ELT.

Extract

Extract 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 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 re-ingest 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.

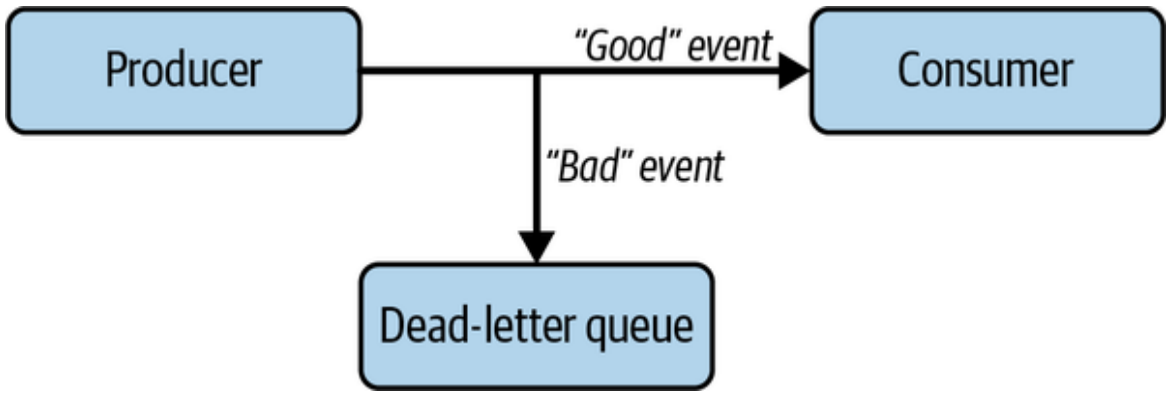

Figure 7-12. "Good" events are passed to the consumer, whereas "bad" events are stored in a dead-letter queue

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 and 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 re-encoded as string data to be sent over the wire, and columns from columnar databases must be re-serialized 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.

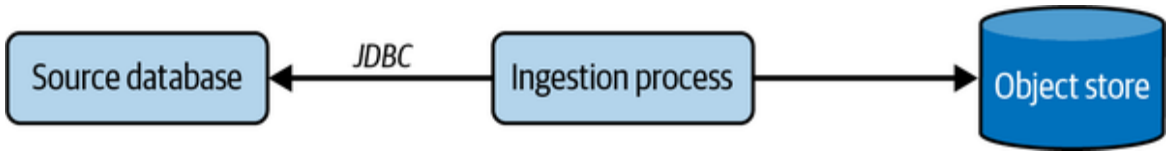

Figure 7-13. An ingestion process reads from a source database using JDBC, and then writes objects into object storage. A target database (not shown) can be triggered to ingest the data with an API call from an orchestration system.

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.