N.B.K.R.INSTITUTE OF SCIENCE &TECHNOLOGY, VIDYANAGAR

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DATA ENGINEERING

LECTURE NOTES

UNIT-IV

SUBJECT: DATA ENGINEERING

UNIT-IV

Storage: Raw Ingredients of Data Storage, Data Storage Systems, Data Engineering Storage Abstractions, Data warehouse, Data Lake, Data Lakehouse.

Ingestion: Data Ingestion, Key Engineering considerations for the Ingestion Phase, Batch Ingestion Considerations, Message and Stream Ingestion Considerations, Ways to Ingest Data.

Storage:

Storage is the one of the main concept of the data engineering lifecycle and underlies its major stages—ingestion, transformation, and serving. Data gets stored many times as it moves through the lifecycle. Whether data is needed seconds, minutes, days, months, or years later, it must persist in storage until systems are ready to use data for further processing and transmission. Knowing the use of the data and the way we will retrieve it in the future is the first step to choosing the proper storage solutions for our data Architecture.

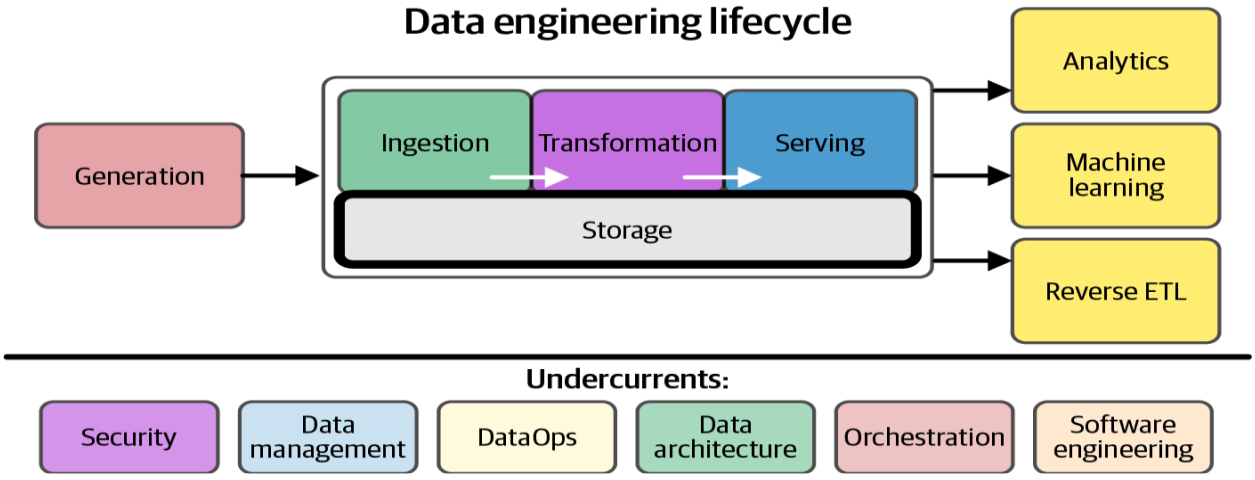


Figure 6-1. Storage plays a central role in the data engineering lifecycle

Source systems are generally not maintained or controlled by data engineers. The storage that data engineers handle directly, The data engineering lifecycle stages of ingesting data from source systems to serving data to deliver value with analytics, data science, etc.

To understand storage, we need to know the R***aw Ingredients***that represents storage systems, including hard drives, solid state drives, and system memory (see Figure 6-2). It's essential to understand the basic tcharacteristics of physical storage technologies to assess the any storage architecture. In This we will also discuss serialization, compression and caching.

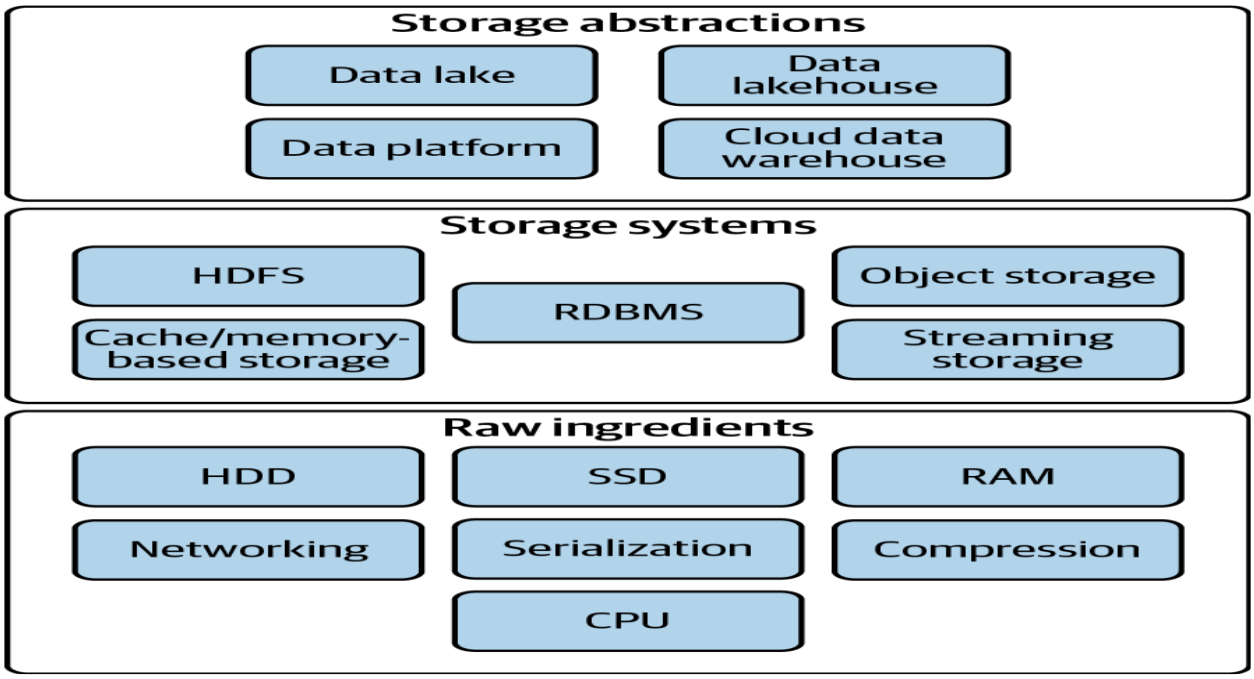


Figure 6-2. Raw ingredients, storage systems, and storage abstractions

Next, we'll look at ***Storage systems***. In practice, we don't directly access system memory or hard disks. These physical storage components exist inside servers and clusters that can ingest and retrieve data using various techniques.

Finally, we'll look at ***Storage Abstractions***. Storage systems are assembled into a cloud data warehouse, a data lake, etc. When building data pipelines, engineers choose the appropriate abstractions for storing their data as it moves through the ingestion, transformation, and serving stages.

I) Raw Ingredients of Data Storage:

Storage is so commonly used unit. That the number of software and data engineers who use storage every day but have little idea how it works in various storage media. Though current managed services potentially free data engineers from the complexities of managing servers, data engineers still need to be aware of underlying components' essential characteristics, performance considerations, durability, and costs.

In most data architectures, data frequently passes through magnetic storage, SSDs, and memory as it works its way through the various processing phases of a data pipeline. Data storage and query systems generally follow complex recipes involving distributed systems, numerous services, and multiple hardware storage layers. These systems require the right raw ingredients to function correctly. Some of the raw ingredients of data storage: disk drives, memory, networking and CPU, serialization, compression, and caching.

1) Magnetic Disk Drive:

Magnetic disks utilize spinning platters coated with a Magnetic material as the shape of Gramophone record (Figure 6-3). This film is magnetized by a read/write head during write operations to physically encode binary data. The read/write head detects the magnetic field and outputs a bitstream during read operations. Magnetic disk drives have been around for ages. They still form the backbone of bulk data storage systems because they are significantly cheaper than SSDs per gigabyte of stored data.

On the one hand, these disks have seen extraordinary improvements in performance, storage density, and cost. On the other hand, SSDs dramatically outperform magnetic disks on various metrics. Currently, commercial magnetic disk drives cost roughly 3 cents per gigabyte of capacity.

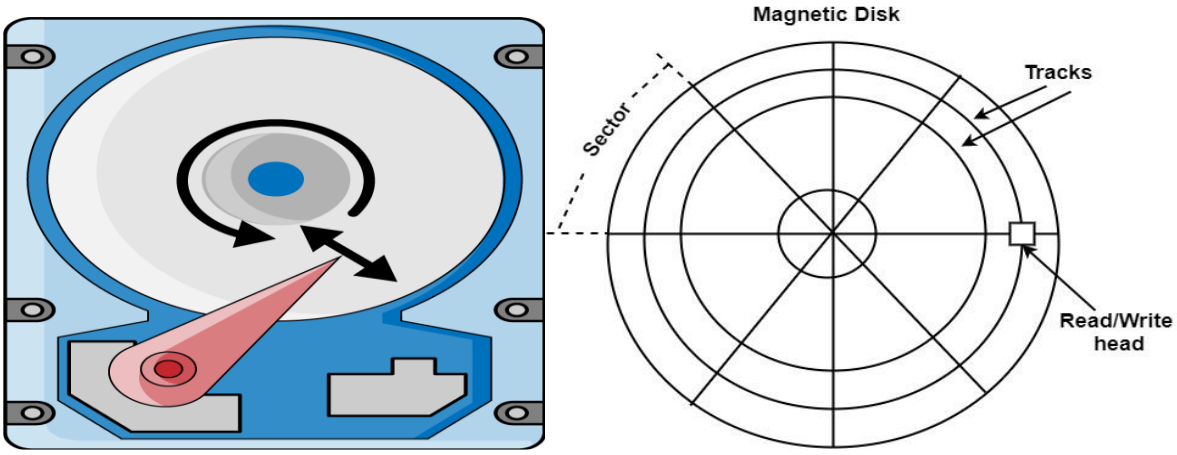


Figure 6-3. Magnetic disk head movement and rotation are essential in random access Latency

A magnetic disk is a storage device that can be assumed as the shape of a Gramophone record. This disk is coated on both sides with a thin film of Magnetic material. This magnetic material has the property that it can store either '1' or '0' permanently. The magnetic material has square loop hysteresis (curve) which can remain in one out of two possible directions which correspond to binary '1' or '0'.
Bits are saved in the magnetized surface in marks along concentric circles known as tracks. The tracks are frequently divided into areas known as sectors.
In this system, the lowest quantity of data that can be sent is a sector. The subdivision of one disk surface into tracks and sectors is displayed in the figure.

IBM developed magnetic disk drive technology in the 1950s. Since then, magnetic disk capacities have grown steadily. The first commercial magnetic disk drive, the IBM 350, had a capacity of 3.75 megabytes. As of this writing, magnetic drives storing 20 TB are commercially available. In fact, magnetic disks continue to see rapid innovation, with methods such as heat-assisted magnetic recording (HAMR), shingled magnetic recording (SMR), and helium-filled disk enclosures being used to realize ever greater storage densities. In spite of the continuing improvements in drive capacity, other aspects of HDD performance are hampered by physics.

First, ***disk transfer speed***, the rate at which data can be read and written, does not scale in proportion with disk capacity. Disk capacity scales with *areal density* (GB stored per square inch), whereas transfer speed scales with *linear density* (bits per inch). This means that if disk capacity grows by a factor of 4, transfer speed increases by only a factor of 2. Consequently, current data center drives support maximum data transfer speeds of 200–300 MB/s. To frame this another way, it takes more than 20 hours to read the entire contents of a 30 TB magnetic drive, assuming a transfer speed of 300 MB/s.

A second major limitation is **seek time**. The time required to arrange the read/write head at the desired track is called seek time. For example, suppose that the read/write head is on track 2 and the record to be read is on track 5, then the read/write head must move from track 2 to track 5. The average seeks time on a modern disk is 8 to 12 ms.

Various tricks can improve **latency and transfer speed**. The time required to position the read/write head on a specific sector when the head has already been placed on the desired track is called rotational delay. The rotational delay is based on the speed of rotation of the disk. On average the latency will be half of one revolution time. The average latency time on modern disks is 4.2 to 6.7ms.

As mentioned earlier, magnetic disks are still prized in data centers for their low data storage costs. In addition, magnetic drives can sustain extraordinarily high transfer rates through parallelism. This

is the critical idea behind cloud object storage: data can be distributed across thousands of disks in clusters. Data-transfer rates go up dramatically by reading from numerous disks simultaneously, limited primarily by network performance rather than disk transfer rate. Thus, network components and CPUs are also key raw ingredients in storage systems, and we will return to these topics shortly.

2) Solid-State Drive:

Solid-state drives (SSDs) store data as charges in flash memory cells. SSDs eliminate the mechanical components of magnetic drives; the data is read by purely electronic means. SSDs can look up random data in less than 0.1 ms (100 microseconds). In addition, SSDs can scale both data-transfer speeds and I/O PS by slicing storage into partitions with numerous storage controllers running in parallel. Commercial SSDs can support transfer speeds of many gigabytes per second and tens of thousands of I/O PS.

Because of these exceptional performance characteristics, SSDs have revolutionized transactional databases and are the accepted standard for commercial deployments of OLTP systems. SSDs allow relational databases such as PostgreSQL, MySQL, and SQL Server to handle thousands of transactions per second.

However, SSDs are not currently the default option for high-scale analytics data storage. Again, this comes down to cost. Commercial SSDs typically cost 20–30 cents (USD) per gigabyte of capacity, nearly 10 times the cost per capacity of a magnetic drive. Thus, object storage on magnetic disks has emerged as the leading option for large-scale data storage in data lakes and cloud data warehouses.

SSDs still play a significant role in OLAP systems. Some OLAP databases uses SSD caching to support high-performance queries on frequently accessed data. As low-latency OLAP becomes more popular.

3) Random Access Memory:

We commonly use the terms ***random access memory*** (RAM) and *memory* interchangeably. Strictly speaking, magnetic drives and SSDs also serve as memory that stores data for later random access retrieval, but

RAM has several specific characteristics:

• It is attached to a CPU and mapped into CPU address space.
• It stores the code that CPUs execute and the data that this code directly processes.
• It is *volatile*, while magnetic drives and SSDs are *nonvolatile*.

Though they may occasionally fail and corrupt or lose data, drives generally retain data when powered off. RAM loses data in less than a second when it is unpowered.

• It offers significantly higher transfer speeds and faster retrieval times than SSD storage. DDR5 memory—the latest widely used standard for RAM—offers data retrieval latency on the order of 100 ns, roughly 1,000 times faster than SSD. A typical CPU can support 100 GB/s bandwidth to attached memory and millions of IOPS. (Statistics vary dramatically depending on the number of memory channels and other configuration details.)
• It is significantly more expensive than SSD storage, at roughly $10/GB (at the time of this writing).
• It is limited in the amount of RAM attached to an individual CPU and memory controller.
• It is still significantly slower than CPU cache, a type of memory located directly on the CPU die or in the same package.

When we talk about system memory, we almost always mean *dynamic RAM*, a high-density, low-cost form of memory. Dynamic RAM stores data as charges in capacitors. These capacitors leak over time, so the data must be frequently *refreshed* (read and rewritten) to prevent data loss. The hardware memory controller handles these technical details

Other forms of memory, such as *static RAM*, are used in specialized applications such as CPU caches.

4) Networking and CPU:

Networking and CPU are increasingly, storage systems are distributed to enhance performance, durability, and availability. We mentioned specifically that individual magnetic disks offer relatively low-transfer performance, but a cluster of disks parallelizes reads for significant performance scaling. While storage standards such as redundant arrays of independent disks (RAID) parallelize on a single server, cloud object storage clusters operate at a much larger scale, with disks distributed across a network and even multiple data centers and availability zones.

Availability zones are a standard cloud construct consisting of compute environments with independent power, water, and other resources. Multizonal storage enhances both the availability and durability of data.

CPUs handle the details of servicing requests, aggregating reads, and distributing writes. Storage becomes a web application with an API, backend service components, and load balancing. Network device performance and network topology are key factors in realizing high performance.

Data engineers need to understand how networking will affect the systems they build and use. Engineers constantly balance the durability and availability achieved by spreading out data geographically versus the performance and cost benefits of keeping storage in a small geographic area and close to data consumers or writers.

5) Serialization:

Serialization is another raw storage ingredient and a critical element of database design. The decisions around serialization will inform how well queries perform across a network, CPU overhead, query latency, and more. Designing a data lake, forexample, involves choosing a base storage system (e.g., Amazon S3) and standards for serialization that balance interoperability with performance considerations.

What is serialization, exactly? Data stored in system memory by software is generally not in a format suitable for storage on disk or transmission over a network. Serialization is the process of flattening and packing data into a standard format that a reader will be able to decode. Serialization formats provide a standard of data exchange. We might encode data in a row-based manner as an XML, JSON, or CSV file and pass it to another user who can then decode it using a standard library. A serialization algorithm has logic for handling types, imposes rules on data structure, and allows exchange between programming languages and CPUs. The serialization algorithm also has rules for handling exceptions. For instance, Python objects can contain cyclic references; the serialization algorithm might throw an error or limit nesting depth on encountering a cycle.

Low-level database storage is also a form of serialization. Row-oriented relational databases organize data as rows on disk to support speedy lookups and in-place updates. Columnar databases organize data into column files to optimize for highly efficient compression and support fast scans of large data volumes. Each serialization choice comes with a set of trade-offs, and data engineers tune these choices to optimize performance to requirements.

We suggest that data engineers become familiar with common serialization practices and formats, especially the most popular current formats (e.g., Apache Parquet), hybrid serialization (e.g., Apache Hudi), and in-memory serialization (e.g., Apache Arrow).

6) Compression:

Compression is another critical component of storage engineering. On a basic level, compression makes data smaller, but compression algorithms interact with other details of storage systems in complex ways.

Highly efficient compression has three main advantages in storage systems. First, the data is smaller and thus takes up less space on the disk. Second, compression increases the practical scan speed per disk. With a 10:1 compression ratio, we go from scanning 200 MB/s per magnetic disk to an effective rate of 2 GB/s per disk.

The third advantage is in network performance. Given that a network connection between an Amazon EC2 instance and S3 provides 10 gigabits per second (Gbps) of bandwidth, a 10:1 compression ratio increases effective network bandwidth to 100 Gbps.

Compression also comes with disadvantages. Compressing and decompressing data entails extra time and resource consumption to read or write data.

7) Caching:

We've already mentioned caching in our discussion of RAM. The core idea of caching is to store frequently or recently accessed data in a fast access layer. The faster the cache, the higher the cost and the less storage space available. Less frequently accessed data is stored in cheaper, slower storage. Caches are critical for data serving, processing, and transformation.

As we analyze storage systems, it is helpful to put every type of storage we utilize inside a *cache hierarchy* (Table 6-1). Most practical data systems rely on many cache layers assembled from storage with varying performance characteristics. This starts inside CPUs; processors may deploy up to four cache tiers. We move down the hierarchy to RAM and SSDs. Cloud object storage is a lower tier that supports long-term data retention and durability while allowing for data serving and dynamic data movement in pipelines.

Table 6-1. A heuristic cache hierarchy displaying storage types with approximate pricing and performance characteristics

Storage type	Data fetch latency[a]	Bandwidth	Price
CPU cache	1 nanosecond	1 TB/s	N/A
RAM	0.1 microseconds	100 GB/s	$10/GB
SSD	0.1 milliseconds	4 GB/s	$0.20/GB
HDD	4 milliseconds	300 MB/s	$0.03/GB
Object storage	100 milliseconds	10 GB/s	$0.02/GB per month
Archival storage	12 hours	Same as object storage once data is available	$0.004/GB per month

[a] A microsecond is 1,000 nanoseconds, and a millisecond is 1,000 microseconds.

We can think of archival storage as a *reverse cache*. Archival storage provides inferior access characteristics for low costs. Archival storage is generally used for data backups and to meet data-retention compliance requirements. In typical scenarios, this data will be accessed only in an emergency (e.g., data in a database might be lost and need to be recovered, or a company might need to look back at historical data for legal discovery).

II) Data Storage Systems:

Storage systems exist at a level of abstraction above raw ingredients. For example, magnetic disks are a raw storage ingredient, while major cloud object storage platforms and HDFS are storage systems that utilize magnetic disks. Still higher levels of storage abstraction exist, such as data lakes and lakehouses.

1) Single Machine Versus Distributed Storage

As data storage and access patterns become more complex and outgrow the usefulness of a single server, distributing data to more than one server becomes necessary. Data can be stored on multiple servers, known as *distributed storage*. This is a distributed system whose purpose is to store data in a distributed fashion (Figure 6-4).

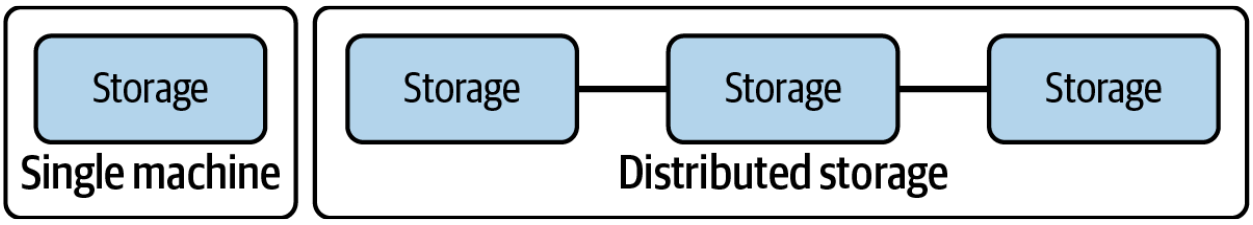


Figure 6-4. Single machine versus distributed storage on multiple servers

Distributed storage coordinates the activities of multiple servers to store, retrieve, and process data faster and at a larger scale, all while providing redundancy in case a server becomes unavailable. Distributed storage is common in architectures where you want built-in redundancy and scalability for large amounts of data. For example, object storage, Apache Spark, and cloud data warehouses rely on distributed storage
architectures.

2) Cloud file system services

Cloud file system services provide a fully managed filesystem for use with multiple cloud VMs and applications, potentially including clients outside the cloud environment. Cloud filesystems should not be confused with standard storage attached to VMs—generally, block storage with a filesystem managed by the VM operating system. Cloud filesystems behave much like NAS (Network Attached Storage) solutions, but the details of networking, managing disk clusters, failures, and configuration are fully handled by the cloud vendor.

For example, **Amazon Elastic File System (EFS)** is an extremely popular example of a cloud filesystem service provided by Amazon Web Services (AWS) designed to provide scalable, elastic, concurrent with some restrictions and encrypted file storage for use with both AWS cloud services and on-premises resources. Amazon EFS is built to be able to grow and shrink automatically as files are added and removed. Amazon EFS supports Network File System (NFS) versions 4.0 and 4.1 (NFSv4) protocol, and control access to files through Portable Operating System Interface (POSIX) permissions.

3) Object Storage

Object storage contains *objects* of all shapes and sizes which is Unstructured data (Figure 6-8). It could be any type of file—TXT, CSV, JSON, images, videos, or audio. widely used object storages are Amazon S3, Azure Blob Storage, and Google Cloud Storage (GCS). Itis a technology that manages data as objects. All data is stored in one large repository which may be distributed across multiple physical

storage devices, instead of being divided into files or folders. It is easier to understand object-based storage when you compare it to more traditional forms of storage – file and block storage.

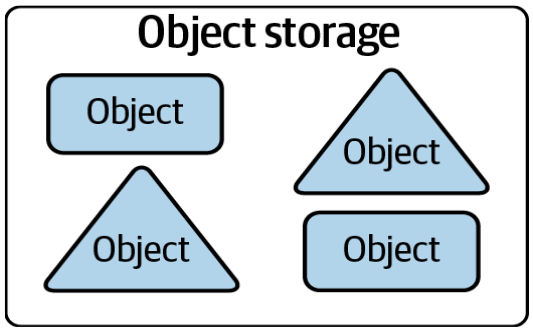


Figure 6-8. Object storage contains immutable objects of all shapes and sizes. Unlike files on a local disk, objects cannot be modified in place.

Although many on-premises object storage systems can be installed on server clusters, we'll focus mostly on fully managed cloud object stores. One of the most important characteristics of cloud object storage is that it is straightforward to manage and use. Object storage was one of the first "serverless" services; engineers don't need to consider the characteristics of underlying server clusters or disks.

An object store is a key-value store for immutable data objects. We lose much of the writing flexibility we expect with file storage on a local disk in an object store. Objects don't support random writes or append operations; instead, they are written once as a stream of bytes. After this initial write, objects become immutable. To change data in an object or append data to it, we must rewrite the full object. Object stores generally support random reads through range requests.

For a software developer used local random access file storage, the characteristics of objects might seem like constraints, object stores don't need to support locks or change synchronization, allowing data storage across massive disk clusters. Object stores support extremely performant parallel stream writes and reads across many disks, and this parallelism is hidden from engineers, who can simply deal with the stream rather than communicating with individual disks.

In a cloud environment, **write speed scales with the number of streams** being written up to quota limits set by the vendor. Read bandwidth **can scale with the number of parallel requests**, the number of virtual machines employed to read data, and the number of CPU cores. These characteristics make object storage ideal for serving high-volume web traffic or delivering data to highly parallel distributed query engines.

Typical cloud object stores save data in several availability zones, dramatically reducing the odds that storage will go fully offline or be lost in an unrecoverable way. This durability and availability are built into the cost; cloud storage vendors offer other storage classes at discounted prices in exchange for reduced durability or availability.

Object storage is a key ingredient in separating compute and storage, allowing engineers to process data with ephemeral clusters and scale these clusters up and down on demand. This is a key factor in making big data available to smaller organizations that can't afford to own hardware for data jobs that they'll run only occasionally. Some major tech companies will continue to run permanent Hadoop clusters on their hardware. Still, the general trend is that most organizations will move data processing to the cloud, using an object store as essential storage and serving layer while processing data on ephemeral clusters.

In object storage, available storage space is also highly scalable, an ideal characteristic for big data systems. Storage space is constrained by the number of disks the storage provider owns, but these providers handle exabytes of data. In a cloud environment, available storage space is virtually limitless; in

practice, the primary limit on storage space for public cloud customers is budget. From a practical standpoint, engineers can quickly store massive quantities of data for projects without planning months in advance for necessary servers and disks.

4) Cache and Memory-Based Storage Systems

As discussed in "Raw Ingredients of Data Storage" RAM offers excellent latency and transfer speeds. However, traditional RAM is extremely volatile data. RAM-based storage systems are generally focused on caching applications, presenting data for quick access and high bandwidth. Data should generally be written to a more durable medium for retention purposes. These ultra-fast cache systems are useful when data engineers need to serve data with ultra-fast retrieval latency.

Example: Memcached and lightweight object caching

Memcached is a free, open-source, high-performance, distributed memory object caching system. It stores data in RAM as key-value pairs, making retrieval very fast. *Memcached* is designed for caching database query results, API call responses, and more. Memcached uses simple data structures, supporting either string or integer types. Memcached can deliver results with very low latency.

Example: Redis (REmote DIctionary Server) **, memory caching with optional persistence**

Like Memcached, *Redis* is a NoSQLkey-value store, but it supports somewhat more complex data types (such as lists or sets). Redis also builds in multiple persistence mechanisms, including snapshotting and journaling. With a typical configuration, Redis writes data roughly every two seconds. Redis is thus suitable for extremely high-performance applications but can tolerate a small amount of data loss.

5) The Hadoop Distributed File System

The Hadoop Distributed File System is based on Google File System (GFS) and was initially engineered to process data with the MapReduce programming model. Hadoop is similar to object storage but with a key difference: Hadoop combines compute and storage on the same nodes, where object stores typically have limited support for internal processing.

Hadoop breaks large files into *blocks*, chunks of data less than a few 100MB's in size. The filesystem is managed by the *NameNode*, which maintains directories, file metadata, and a detailed catalog describing the location of file blocks in the cluster. In a typical configuration, each block of data is replicated to three nodes. This increases both the durability and availability of data. If a disk or node fails, the replication factor for some file blocks will fall below 3. The NameNode will instruct other nodes to replicate these file blocks so that they again reach the correct replication factor. Thus, the probability of losing data is very low.

We often see claims that Hadoop is dead. This is only partially true. Hadoop is no longer a hot, bleeding-edge technology. Many Hadoop ecosystem tools such as Apache Pig are now on life support and primarily used to run legacy jobs. The pure MapReduce programming model has fallen by the wayside. HDFS remains widely used in various applications and organizations.

Hadoop still appears in many legacy installations. Many organizations that adopted Hadoop during the peak of the big data craze have no immediate plans to migrate to newer technologies. This is a good choice for companies that run massive (thousandnode) Hadoop clusters and have the resources to maintain on-premises systems effectively. Smaller companies may want to reconsider the cost overhead and scale limitations of running a small Hadoop cluster against migrating to cloud solutions.

In addition, HDFS is a key ingredient of many current big data engines, such as Amazon EMR. In fact, Apache Spark is still commonly run on HDFS clusters.

6) Streaming Storage

Streaming data has different storage requirements than nonstreaming data. In the case of message queues, stored data is temporarly and expected to disappear after a certain duration. However, distributed, scalable streaming frameworks like **Apache Kafka** (a distributed, open-source streaming platform used for building real-time data pipelines and streaming applications, enabling high-throughput, low-latency data processing and storage)now allow extremely long-duration streaming data retention. Kafka supports indefinite data retention by pushing old, infrequently accessed messages down to object storage. Kafka competitors (including Amazon Kinesis, Apache Pulsar, and Google Cloud Pub/Sub) also support long data retention.

Closely related to data retention in these systems is the notion of replay. *Replay* allows a streaming system to return a range of historical stored data. Replay is the standard data-retrieval mechanism for streaming storage systems. Replay can be used to run batch queries over a time range or to reprocess data in a streaming pipeline.

Other storage engines for real-time analytics Query Engine is Transactional databases. Data becomes visible to queries as soon as it is written. Storage well-known scaling and locking limitations, especially for analytics queries that run across large volumes of data.

III)Data Engineering Storage Abstractions:

Data engineering storage abstractions are data organization and query patterns that sit at the heart of the Data Engineering Lifecycle and are built on top of the data storage systems.

The main types of abstractions are support data science, analytics, and reporting use cases. These include data warehouse, data lake, data lakehouse, data platforms, and data catalogs.

Few key considerations for storage abstraction:

Purpose and use case

You must first identify the purpose of storing the data. What is it used for?

Update patterns

Is the abstraction optimized for bulk updates, streaming inserts, or upserts?

Cost

What are the direct and indirect financial costs? The time to value? The opportunity costs?

Separate storage and compute

We should know that the popularity of separating storage from compute means the lines between OLAP databases and data lakes are increasingly blurring. Major cloud data warehouses and data lakes are on a collision course. In the future, the differences between these two may be in name only since they might functionally and technically be very similar under the hood.

IV)The Data Warehouse:

Data warehouses are a standard OLAP data architecture. The term *data warehouse* refers to technology platforms (e.g., Google BigQuery and Teradata),a Centralized repository that stores and organizes data from various sources within a company. In terms of storage trends, building data warehouses atop conventional transactional databases, row-based MPP (Massively Parallel Processing) systems (e.g.,Teradata and IBM Netezza), and column-based MPP systems (e.g., Vertica and Teradata Columnar) to cloud data warehouses and data platforms.

In practice, Cloud data warehouses are often used to organize data into a data lake, a storage area for massive amounts of unprocessed raw data. Cloud data warehouses can handle massive amounts of raw text and complex JSON documents.

The limitation is that cloud data warehouses cannot handle truly unstructured data, such as images, video, or audio, unlike a true datalake. Cloud data warehouses can be combined with object storage to provide a complete data-lake solution.

V) The Data Lake:

The *data lake* was originally a massive store in the form of raw data, unprocessed form. Initially, data lakes were built primarily on Hadoop systems, where cheap storage allowed for retention of massive amounts of data without the cost overhead of MPP system.

The last five years have seen two major developments in the evolution of data lake storage.

-First, a major migration toward *separation of compute and storage* has occurred. In practice, this means a move away from Hadoop toward cloud object storage for long-term retention of data.

-Second, data engineers discovered that much of the functionality offered by MPP systems (schema management; update, merge and delete capabilities).

VI) The Data Lakehouse:

The *data lakehouse* is an architecture that combines aspects of the data warehouse and the data lake. As it is generally conceived, the lakehouse stores data in object storage just like a lake. However, the lakehouse adds to this arrangement features designed to streamline data management and create an engineering experience similar to a data warehouse. This means robust table and schema support and features for managing incremental updates and deletes. Lakehouses typically also support table history and rollback; this is accomplished by retaining old versions of files and metadata.

A lakehouse system is a metadata and file-management layer deployed with data management and transformation tools. Databricks has heavily promoted the lakehouse concept with Delta Lake, an open source storage management system.

We would be remiss not to point out that the architecture of the data lakehouse is similar to the architecture used by various commercial data platforms, including BigQuery and Snowflake. These systems store data in object storage and provide automated metadata management, table history, and update/delete capabilities. The complexities of managing underlying files and storage are fully hidden from the user.

The key advantage of the data lakehouse is interoperability means exchange data between tools when stored in an open file format. Reserializing data from a proprietary database format incurs overhead in processing, time, and cost. In a data lakehouse architecture, various tools can connect to the metadata layer and read data directly from object storage.

It is important to emphasize that much of the data in a data lakehouse may not have a table structure imposed. We can create data warehouse features where we need them in a lakehouse, leaving other data in a raw or even unstructured format.

VII) 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).

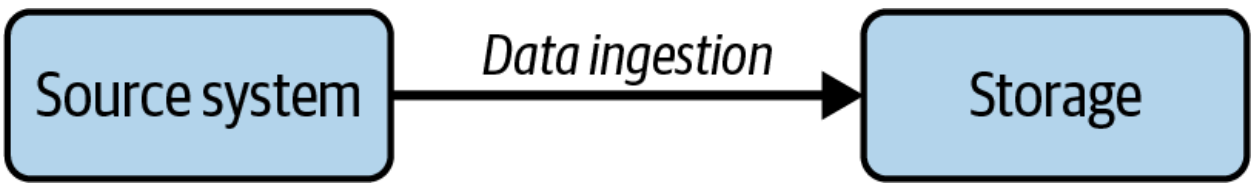


Figure 7-2. Data from system 1 is ingested into system 2

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 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.

VIII) Key Engineering Considerations for the Ingestion Phase

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

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

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

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

1) Bounded Versus Unbounded Data

Data comes in two forms: bounded and unbounded (Figure 7-3). *Unbounded data* is data as it exists in reality, as events happen continuously, ongoing and flowing. *Bounded data* is a convenient way of bucketing data across some sort of boundary, such as time. Bounded data is finite and unchanging data. Whereas Unbounded data is Infinite and change the data.

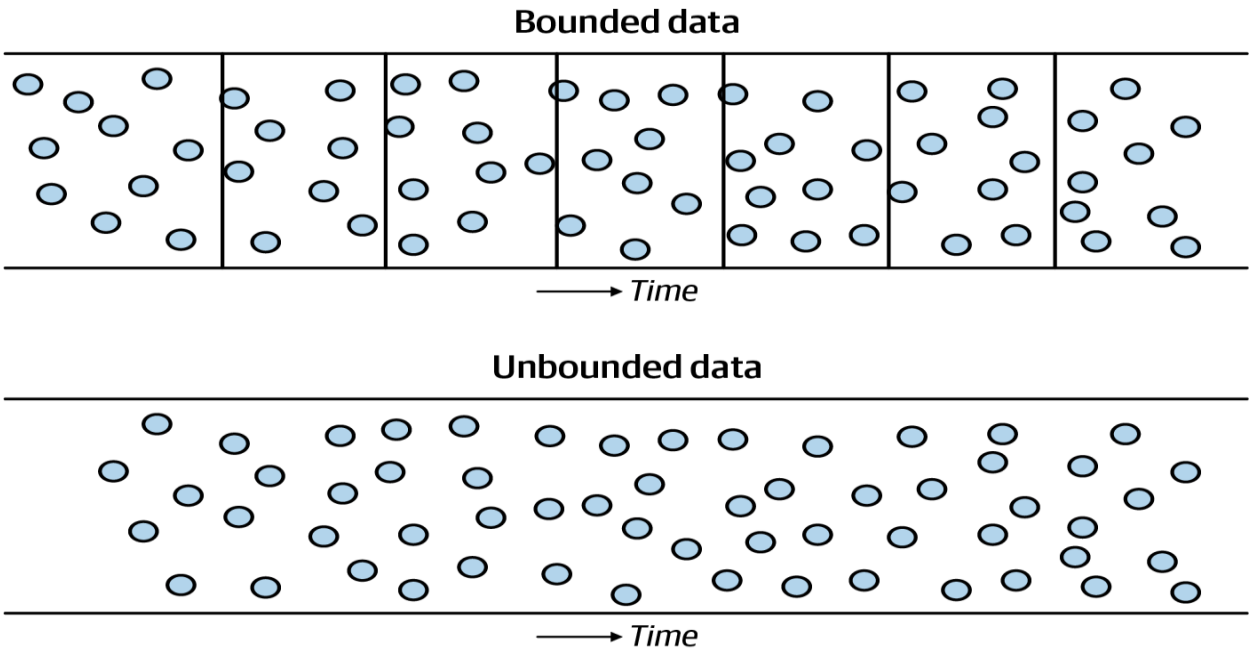


Figure 7-3. Bounded versus unbounded data

Processing unbounded data often requires that events are ingested in a specific order, such as the order in which events occurred, to be able to reason about result completeness. Bounded streams have a defined start and end. Bounded streams can be processed by ingesting all data before performing any computations.

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.

2) 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 (Customer Relationship Management) 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.

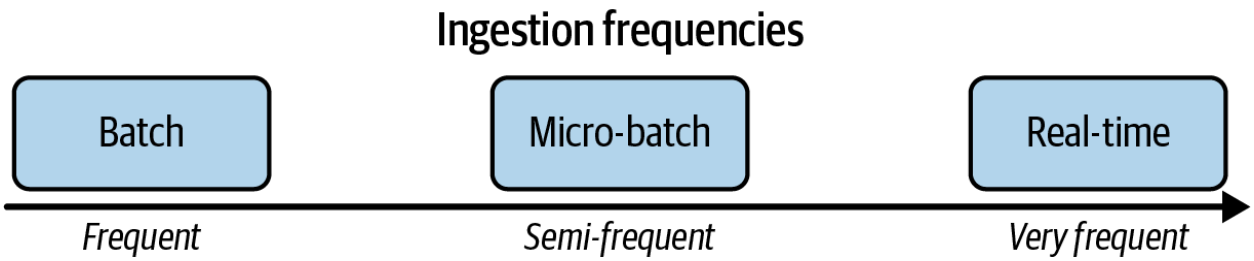


Figure 7-4. The spectrum batch to real-time ingestion frequencies

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.

3) 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.

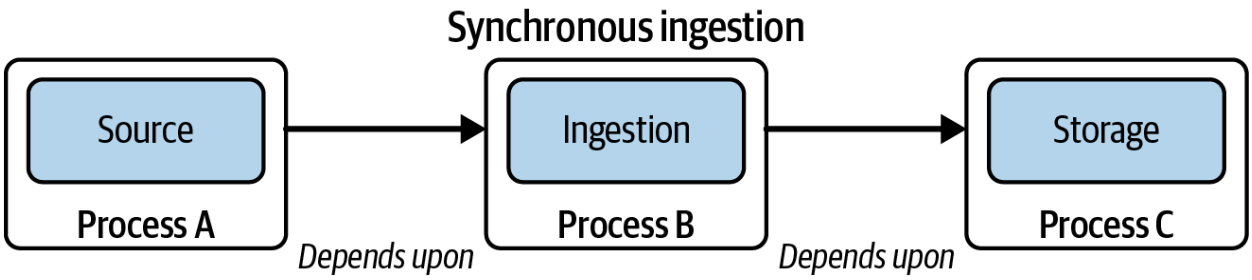


Figure 7-5. A synchronous ingestion process runs as discrete batch steps

Synchronous ingestion involves a system waiting for a response from a data source before proceeding, while asynchronous ingestion allows the system to continue without waiting for a response. Synchronous methods are simpler to implement and provide immediate feedback but can lead to delays if the data source is slow or unavailable. Asynchronous methods offer improved scalability and resource utilization, allowing for more efficient data processing. The choice between them depends on the specific needs and priorities of the data ingestion process.

Synchronous Ingestion: The system waits for a response from the data source before proceeding to the next step in the ingestion process.

Advantages: Simpler to implement and debug. Provides immediate feedback on the success or failure of the data transfer. Guarantees data consistency and order.

Disadvantages: Can be slower if the data source is slow or unresponsive. Potentially blocks other tasks if the system is waiting for a response. May lead to increased latency if the data source is under heavy load.

Asynchronous Ingestion: The system continues to the next step in the ingestion process without waiting for a response from the data source.
Advantages: Improved scalability and resource utilization. Can handle large volumes of data efficiently. Allows for more flexible and efficient data processing pipelines.

Disadvantages: More complex to implement and debug. May require additional error handling mechanisms to ensure data integrity. May introduce some level of latency depending on the specific implementation.

4) Serialization and Deserialization

Serialization and deserialization are fundamental processes in data ingestion, enabling the transfer and storage of objects by converting them into a byte stream and back.

Serialization converts an object into a byte stream, making it possible to store or transmit the object's state.

Deserialization: This is the reverse of serialization. It takes the byte stream (created during serialization) and reconstructs the original object, restoring its state.\

5) 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.

Throughput and scalability are crucial aspects of efficient data ingestion, enabling systems to handle large volumes of data without performance degradation. Throughput refers to the rate at which data is processed, while scalability refers to the ability of a system to accommodate increasing data loads and user demands.

Throughput measures the amount of data ingested or processed within a specific timeframe, often expressed in bytes per second or similar units. High throughput ensures that data is ingested quickly and efficiently, preventing bottlenecks and delays in data processing.

Scalability refers to the ability of a data ingestion system to handle increasing data volumes and user demands without compromising performance or functionality. Scalability ensures that data ingestion pipelines can accommodate growing data needs and prevent bottlenecks or data loss as data volume increases.

6) Reliability and Durability

Reliability and durability are crucial aspects of data ingestion, ensuring data integrity and availability. Reliability focuses on the consistency and dependability of data throughout its lifecycle, ensuring accuracy and completeness, while durability focuses on the longevity and accessibility of stored data, guarding against loss or corruption.

Reliability: Data reliability refers to the dependability and consistency of data. It ensures that data remains accurate, complete, and consistent throughout its lifecycle.

Reliable data is essential for making informed decisions, optimizing operations, and maintaining compliance. It also improves risk management, enhances customer satisfaction, and increases efficiency.

Durability: Data durability ensures that stored data remains intact, complete, and uncorrupted over time, guaranteeing long-term accessibility.

Durability is critical for protecting data from various threats, including hardware failures, natural disasters, human errors, cybersecurity breaches, and software.

7) 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)

8) 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.

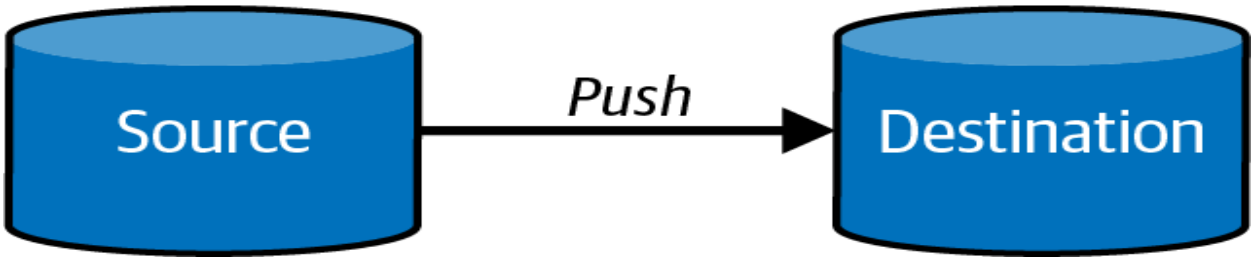


Figure 7-7. Pushing data from source to destination

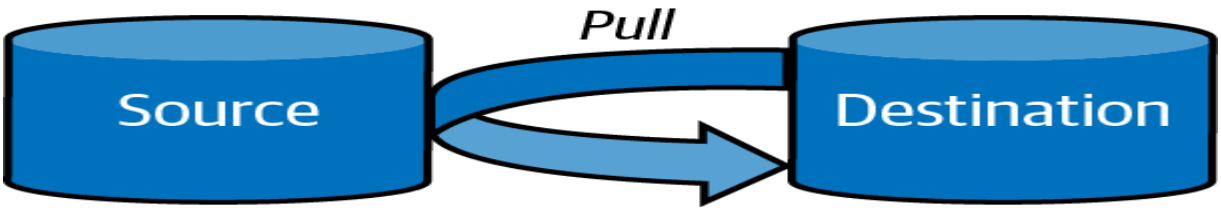


Figure 7-8. A destination pulling data from a source

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.

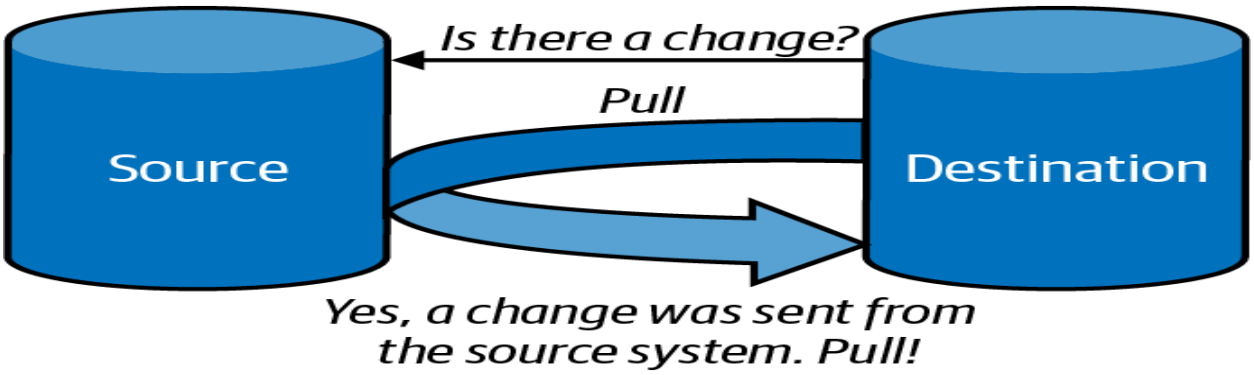


Figure 7-9. Polling for changes in a source system

IX) **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)

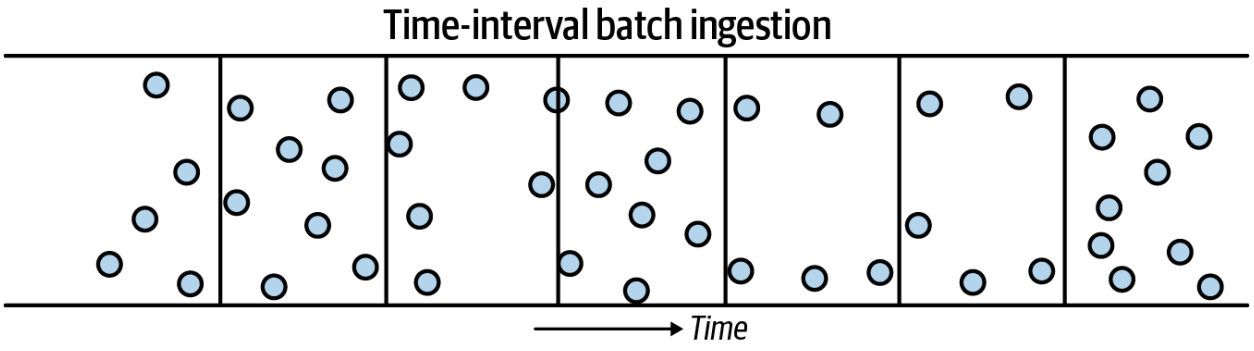


Figure 7-10. Time-interval batch ingestion

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.

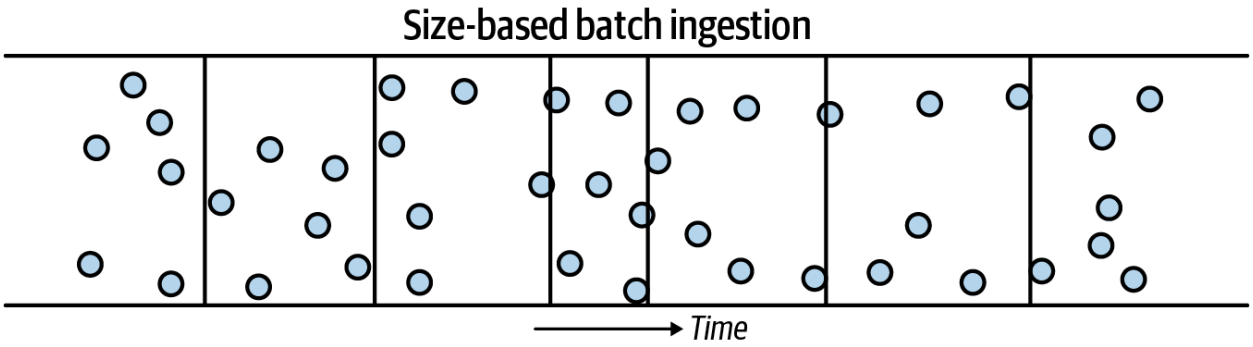


Figure 7-11. Size-based batch ingestion

Some commonly used batch ingestion patterns, which we discuss in this section,

include the following:

• Snapshot or differential extraction

• File-based export and ingestion

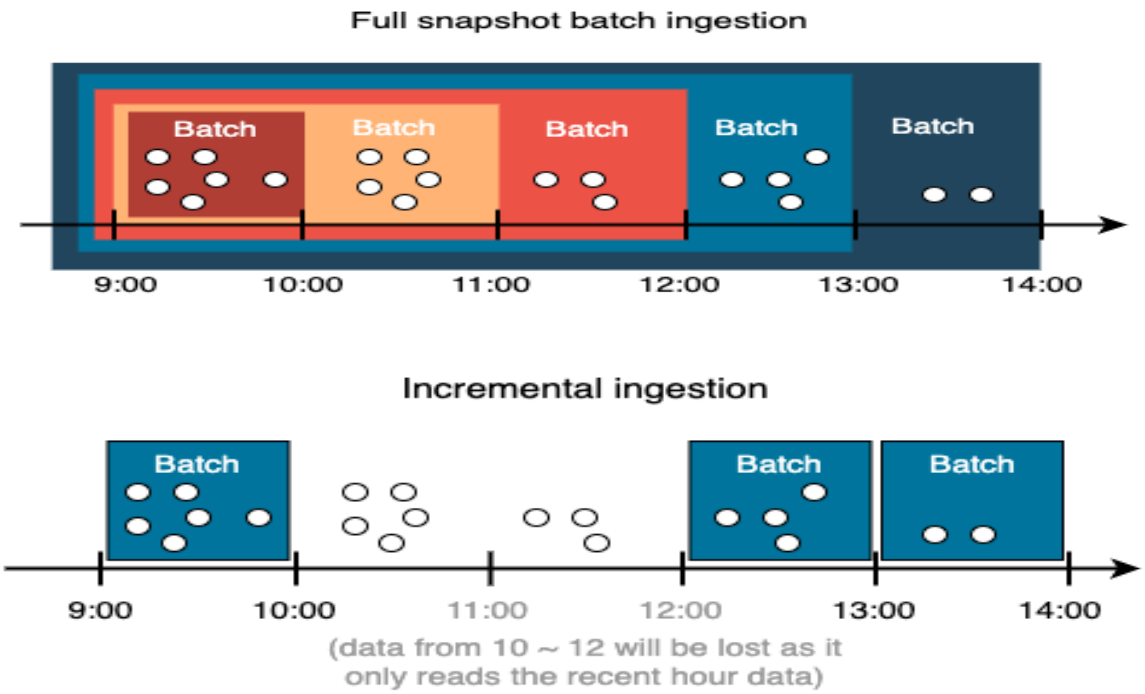
• ETL versus ELT

• 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

ETL and ELT, both extremely common ingestion, storage, and transformation patterns.

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

Load Once data is extracted, it can be either 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.

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 we'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.

X) **Message and Stream Ingestion Considerations:**

Schema Evolution

Schema evolution is common when handling event data; the process of adapting data schemas as they change over time, ensuring data pipelines remain functional and data quality is maintained. This involves managing changes like adding, removing, or renaming columns or types in data ingested into a system. Schema evolution is a crucial aspect of data management, particularly in big data systems where data often evolves over time.

To solve issues related to schema evolution. First, if our event-processing framework has a **schema registry** (a centralized repository for storing and managing schema definitions and metadata. It allows data producers and consumers to register, share, and evolve schemas), use it to version your schema changes. Next, a **dead-letter queue** (DLQ - a specialized message queue used to store messages that cannot be processed by the intended recipient or system due to various reasons like errors, invalid formats, or destination unavailability) can help you investigate issues with events that are not properly handled. Finally, the **low-fidelity route** (streamlined or simplified data processing path)(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

Ordering refers to ensuring data is processed and delivered in a specific sequence, often based on timestamps or other criteria. Multiple delivery in ingestion, or "at-least-once" delivery, means that messages or data chunks might be delivered to consumers more than once, potentially out of order.

Ingestion Replay

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

Time to Live

How long will you preserve your event record? A key parameter is *maximum message retention time*, also known as the *time to live* (TTL). TTL is usually a configuration we'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.

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 frame work 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.

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*.

Dead-letter queue segregates problematic events from events that can be accepted by the consumer. 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.

Consumer Pull and Push

A consumer subscribing to a topic can get events in two ways: push and pull. 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.

XI) **Ways to Ingest Data:**

Now we can focus on ways we can ingest data. The 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 (Open Data Base Connectivity) is a standard API that allows applications to access data from different database management systems (DBMS) using SQL as a standard for accessing the data. It enables applications to connect to diverse databases with a single interface.

JDBC (Java Data Base Connectivity) is a Java API that allows Java applications to interact with databases. It enables developers to connect to various databases, execute SQL queries, and retrieve and process data. JDBC acts as a bridge between Java applications and databases, simplifying data access and management.

The JVM (Java Virtual Machine) is standard, portable across hardware architectures and operating systems, and provides the performance of compiled code through a just-in-time (JIT) compiler. The JVM is an extremely popular compiling VM for running code in a portable manner.

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

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

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

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).

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), 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.

Batch-oriented CDC

Batch-Oriented CDC in ingestion refers to a strategy where data changes are captured and loaded into a destination system in a batch-wise manner, rather than in real-time or near real-time. This means that changes are not immediately reflected in the destination, but are instead processed in periodic intervals or when a certain threshold of changes is reached.

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.

Continuous CDC

Continuous Change Data Capture (CDC) in data ingestion involves capturing and delivering data changes from a source database in real-time to a downstream system. This contrasts with traditional batch ingestion, which pulls all data at once. CDC is highly efficient and avoids data loss, making it suitable for real-time analytics and cloud data warehouses.

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. 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.

CDC and database replication

Change Data Capture (CDC) and database replication are related but distinct concepts in data ingestion. CDC is a method for tracking and capturing data changes in a source database, while replication is the process of copying data from one database to another.

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.

Change Data Capture (CDC) considerations in data ingestion focus on minimizing impact on source systems, optimizing data processing speed, and ensuring system responsiveness. CDC provides a way to track and replicate data changes as they happen, offering advantages over traditional batch ingestion for real-time analytics and data synchronization.

APIs

APIs, or Application Programming Interfaces, play a crucial role in data ingestion, allowing systems to seamlessly exchange information and load data from various sources. Ingestion APIs facilitate the transfer of data from external sources, such as databases, third-party services, and other systems, into a target system for processing and storage.

How APIs are used in Ingest Data:

Data Transfer:

APIs provide a standardized way for applications to request data from each other. This includes sending data to be ingested into a system, like loading data from a cloud service or a database into a data warehouse.

Streaming and Bulk Ingestion:

APIs can support both streaming (real-time) and bulk (batch) ingestion patterns. Streaming APIs allow for the continuous flow of data, while bulk APIs enable the loading of large datasets at once.

Data Transformation:

Ingestion APIs often work in conjunction with pipelines and processors to transform data from various formats into a consistent format suitable for the target system.

Customization and Flexibility:

APIs allow for the creation of custom solutions for data ingestion, catering to specific needs and requirements.

Message Queues and Event-Streaming Platforms

Message queues and event streaming platforms serve as essential components in data ingestion, providing solutions for different real-time data handling needs. Message queues facilitate reliable, point-

to-point communication between systems, while event streaming platforms enable real-time, continuous data flow and multi-consumer scenarios.

Message Queues Message queues act as intermediaries, ensuring reliable and ordered delivery of messages between producers and consumers. They are particularly useful when systems need to be decoupled and asynchronous communication is required.

Key Features:

Reliable Delivery: Guarantees that messages are delivered to the correct consumers, even if there are temporary issues.

Order Preservation: Maintains the order of messages, ensuring that they are processed in the same sequence as they were sent.

Point-to-Point Communication: Each message is typically delivered to a single consumer.
Examples: RabbitMQ and Amazon SQS.

Event Streaming Platforms Event streaming platforms enable the continuous flow of events, allowing for real-time data processing and analysis. They are designed for scenarios where high volume and velocity of data need to be handled.

Key Features:

Real-time Processing: Facilitates immediate processing of events as they occur.

Pub/Sub Model: Allows multiple consumers to subscribe to the same event stream.

Continuous Data Flow: Handles a continuous stream of events, rather than discrete messages.

Replayability: Consumers can rewind and replay messages within the stream.

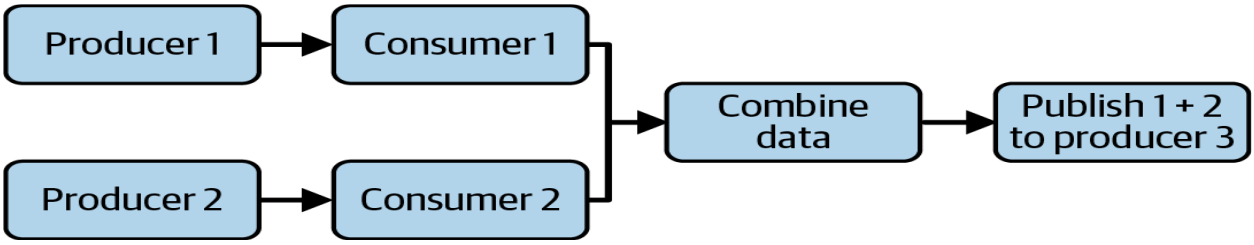


Figure 7-14. Two datasets are produced and consumed (producers 1 and 2) and then combined, with the combined data published to a new producer (producer 3)

Managed Data Connectors
These days, if you're considering writing a data ingestion connector to a database or API, ask yourself: has this already been created? Furthermore, is there a service that will manage the nitty-gritty details of this connection for me? "APIs" on page 254 mentions the popularity of managed data connector platforms and frameworks. These tools aim to provide a standard set of connectors available out of the box to spare data engineers building complicated plumbing to connect to a particular source. Instead of creating and managing a data connector, you outsource this service to a third party.

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

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

Moving Data with Object Storage

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

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

EDI

Another practical reality for data engineers is *electronic data interchange* (EDI). The term is vague enough to refer to any data movement method. It usually refers to somewhat archaic means of file exchange, such as by email or flash drive. Data engineers will find that some data sources do not support more modern means of data transport, often because of archaic IT systems or human process limitations.

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

Databases and File Export

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

For example,

Snowflake, BigQuery, Redshift, and others support direct export to object storage in various formats.

Practical Issues with Common File Formats

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

CSV is by no means a uniform format. Engineers must stipulate the delimiter, quote characters, and escaping to appropriately handle the export of string data. CSV also doesn't natively encode schema information or directly support nested structures. CSV file encoding and schema information must be configured in the target system to ensure appropriate ingestion. Autodetection is a convenience feature provided in many cloud environments but is inappropriate for production ingestion. As a best practice, engineers should record CSV encoding and schema details in file metadata.

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

format is designed to map data directly into processing engine memory, providing high performance in data lake environments.

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

Shell

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

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

SSH

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

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

SFTP and SCP

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

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

SCP is a file-exchange protocol that runs over an SSH connection. SCP can be a secure file-transfer option if it is configured correctly. Again, adding additional network access control (defense in depth) to enhance SCP security is highly recommended.

Webhooks

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

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

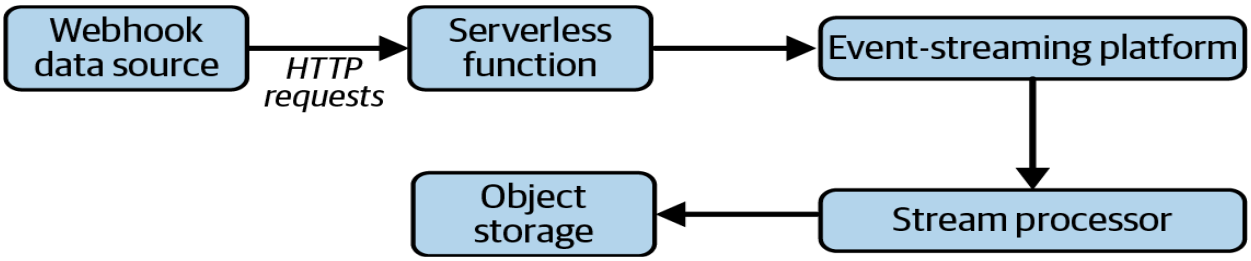


Figure 7-15. A basic webhook ingestion architecture built from cloud services

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

You'll notice that this architecture does much more than simply ingest the data. This underscores ingestion's entanglement with the other stages of the data engineering lifecycle; it is often impossible to define your ingestion architecture without making decisions about storage and processing.

Web Interface

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

Web Scraping

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

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

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

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

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

Transfer Appliances for Data Migration

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

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

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

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

Data Sharing

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

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