To understand storage, we're going to start by studying the *raw ingredients* that compose storage systems, including hard drives, solid state drives, and system memory (see Figure 6-2). It's essential to understand the basic characteristics of physical storage technologies to assess the trade-offs inherent in any storage architecture. This section also discusses serialization and compression, key software elements of practical storage. (We defer a deeper technical discussion of serialization and compression to Appendix A.) We also discuss *caching*, which is critical in assembling storage systems.

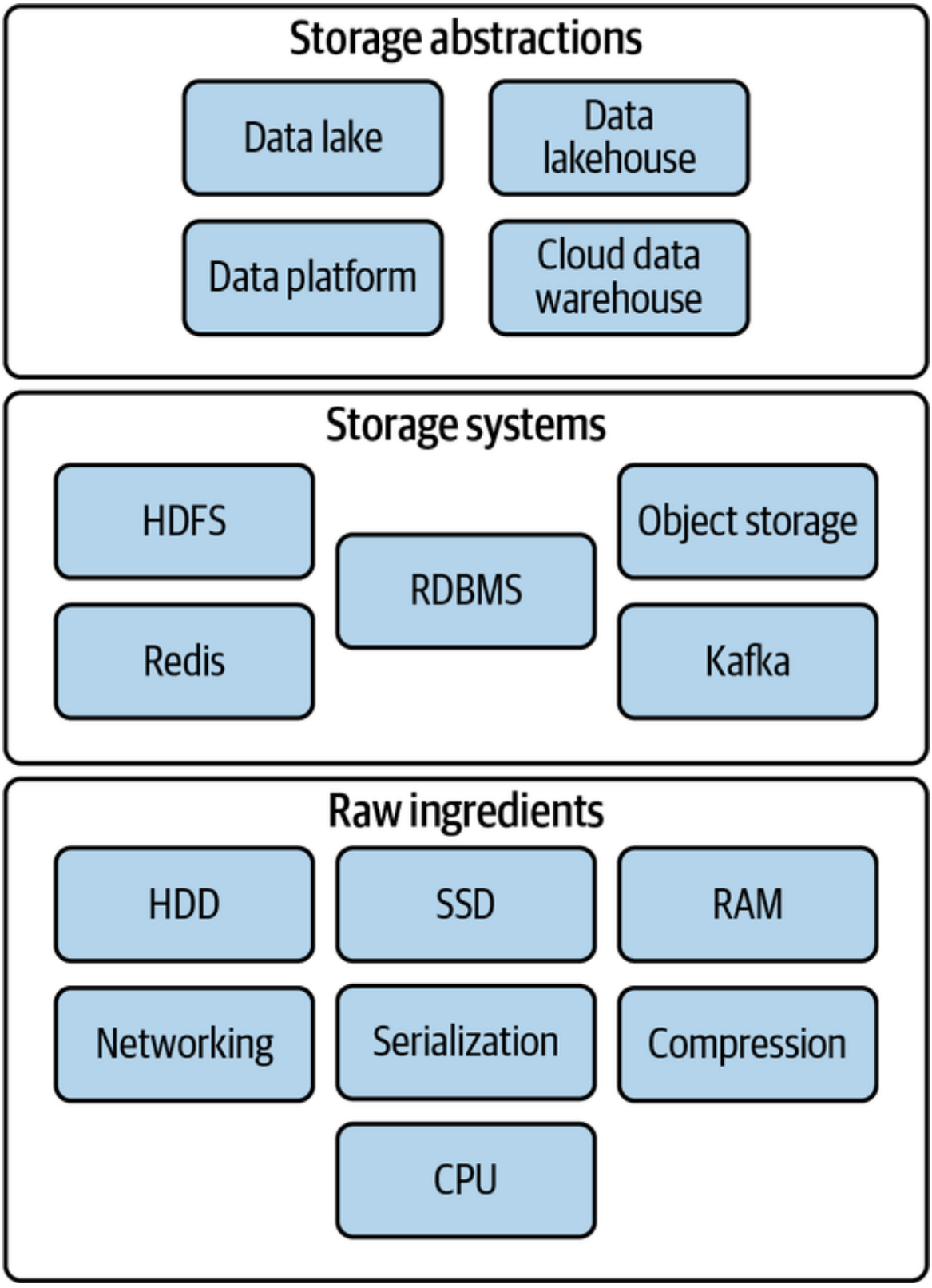

Figure 6-2. Raw ingredients, storage systems, and storage abstractions

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 access paradigms.

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.

Raw Ingredients of Data Storage

Storage is so common that it's easy to take it for granted. We're often surprised by the number of software and data engineers who use storage every day but have little idea how it works behind the scenes or the trade-offs inherent in various storage media. As a result, we see storage used in some pretty...interesting ways. 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.

Let's look at some of the raw ingredients of data storage: disk drives, memory, networking and CPU, serialization, compression, and caching.

Magnetic Disk Drive

Magnetic disks utilize spinning platters coated with a ferromagnetic film (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. Still, they form the backbone of bulk data storage systems because they are significantly cheaper than SSDs per gigabyte of stored data.

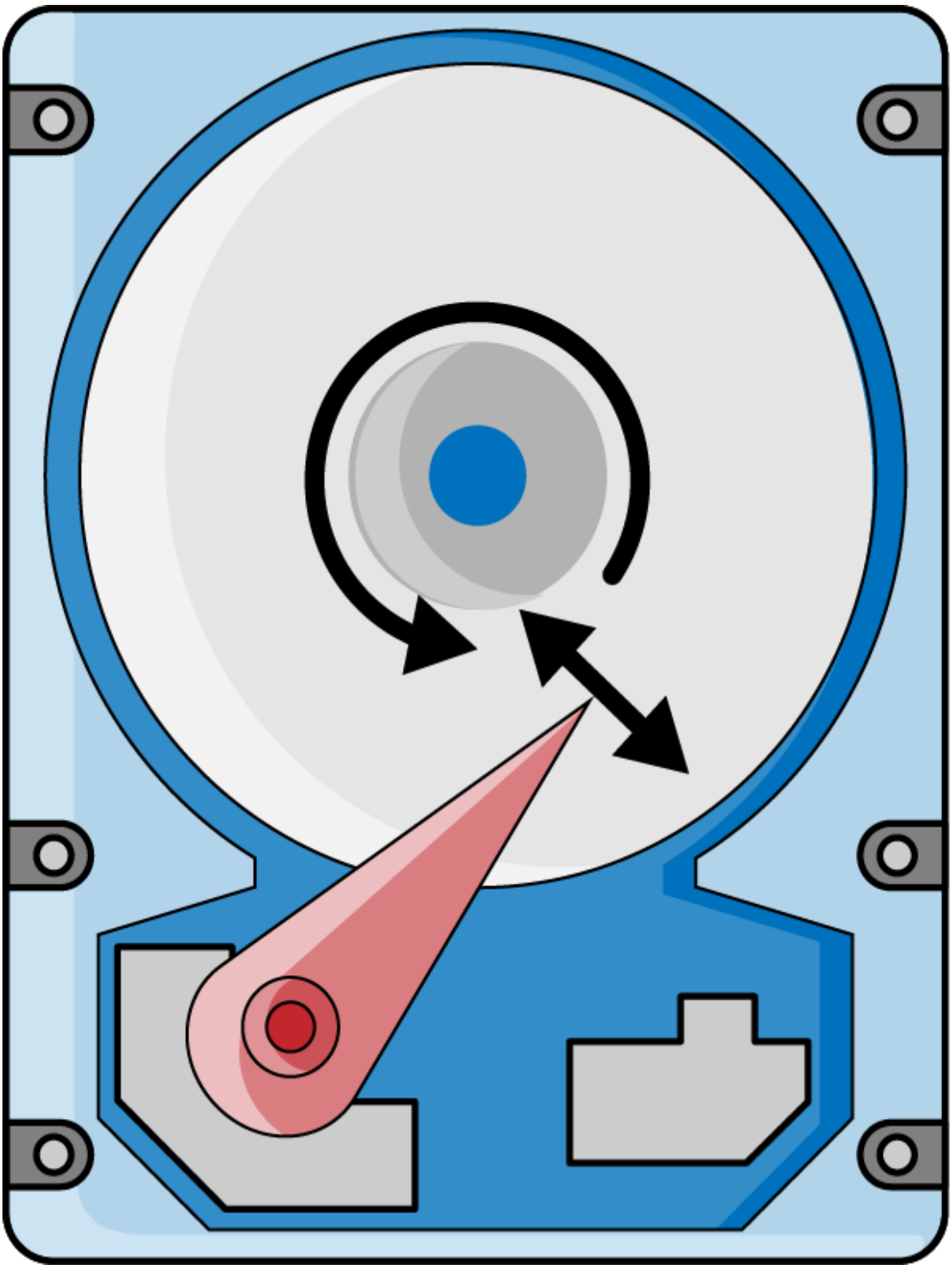

Figure 6-3. Magnetic disk head movement and rotation are essential in random access latency

On the one hand, these disks have seen extraordinary improvements in performance, storage density, and cost.[1] 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. (Note that we'll frequently use the abbreviations *HDD* and *SSD* to denote rotating magnetic disk and solid-state drives, respectively.)

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* (gigabits 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. To access data, the drive must physically relocate the read/write heads to the appropriate track on the disk. Third, in order to find a particular piece of data on the disk, the disk controller must wait for that data to rotate under the read/write heads. This leads to *rotational latency*. Typical commercial drives spinning at 7,200 revolutions per minute (RPM) seek time, and rotational latency, leads to over four milliseconds of overall average latency (time to access a selected piece of data). A fourth limitation is input/output operations per second (IOPS), critical for transactional databases. A magnetic drive ranges from 50 to 500 IOPS.

Various tricks can improve latency and transfer speed. Using a higher rotational speed can increase transfer rate and decrease rotational latency. Limiting the radius of the disk platter or writing data into only a narrow band on the disk reduces seek time. However, none of these techniques makes magnetic drives remotely competitive with SSDs for random access lookups. SSDs can deliver data with significantly lower latency, higher IOPS, and higher transfer speeds, partially because there is no physically rotating disk or magnetic head to wait for.

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.

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 IOPS 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 IOPS.

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 leverage SSD caching to support high-performance queries on frequently accessed data. As low-latency OLAP becomes more popular, we expect SSD usage in these systems to follow suit.

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:

- Is attached to a CPU and mapped into CPU address space.
- Stores the code that CPUs execute and the data that this code directly processes.
- 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.
- 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.)

- Is significantly more expensive than SSD storage, at roughly $10/GB (at the time of this writing).
- Is limited in the amount of RAM attached to an individual CPU and memory controller. This adds further to complexity and cost. High-memory servers typically utilize many interconnected CPUs on one board, each with a block of attached RAM.
- Is still significantly slower than CPU cache, a type of memory located directly on the CPU die or in the same package. Cache stores frequently and recently accessed data for ultrafast retrieval during processing. CPU designs incorporate several layers of cache of varying size and performance characteristics.

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; data engineers simply need to worry about bandwidth and retrieval latency characteristics. Other forms of memory, such as *static RAM*, are used in specialized applications such as CPU caches.

Current CPUs virtually always employ the *von Neumann architecture*, with code and data stored together in the same memory space. However, CPUs typically also support the option to disable code execution in specific pages of memory for enhanced security. This feature is reminiscent of the *Harvard architecture*, which separates code and data.

RAM is used in various storage and processing systems and can be used for caching, data processing, or indexes. Several databases treat RAM as a primary storage layer, allowing ultra-fast read and write performance. In these applications, data engineers must always keep in mind the volatility of RAM. Even if data stored in memory is replicated across a cluster, a power outage that brings down several nodes could cause data loss. Architectures intended to durably store data may use battery backups and automatically dump all data to disk in the event of power loss.

Networking and CPU

Why are we mentioning networking and CPU as raw ingredients for storing data? 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. Appendix B covers cloud networking and major relevant ideas.

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, for example, 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 provide a more detailed catalog of common data serialization techniques and formats in Appendix A. 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).

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. We undertake a more detailed discussion of compression algorithms and trade-offs in Appendix A.

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

i
s
p
l
a
y
i
n
g
s
t
o
r
a
g
e
t
y
p
e
s
w
it
h
a
p
p
r
o
x
i
m
a
t
e
p

r
i
c
i
n
g
a
n
d
p
e
r
f
o
r
m
a
n
c
e
c
h
a
r
a
c
t
e
r
i
s
ti
c
s

Storage type	Data fetch latency[a]	Bandwidth	Price
CPU cache	1 nanosecond	1 TB/s	
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	3 GB/s per instance	$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).

Data Storage Systems

This section covers the major data storage systems you'll encounter as a data engineer. 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 (which we cover in "Data Engineering Storage Abstractions").

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).

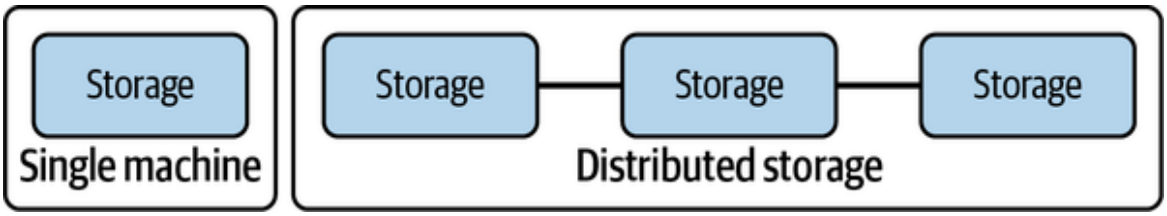

Figure 6-4. Single machine versus distributed storage on multiple servers

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.

Data engineers must always be aware of the consistency paradigms of the distributed systems, which we'll explore next.

Eventual Versus Strong Consistency

A challenge with distributed systems is that your data is spread across multiple servers. How does this system keep the data consistent? Unfortunately, distributed systems pose a dilemma for storage and query accuracy. It takes time to replicate changes across the nodes of a system; often a balance exists between getting current data and getting "sorta" current data in a distributed database. Let's look at two common consistency patterns in distributed systems: eventual and strong.

We've covered ACID compliance throughout this book, starting in Chapter 5. Another acronym is *BASE*, which stands for *basically available, soft-state, eventual consistency*. Think of it as the opposite of ACID. BASE is the basis of eventual consistency. Let's briefly explore its components:

Basically available

Consistency is not guaranteed, but efforts at database reads and writes are made on a best-effort basis, meaning consistent data is available most of the time.

Soft-state

The state of the transaction is fuzzy, and it's uncertain whether the transaction is committed or uncommitted.

Eventual consistency

At *some* point, reading data will return consistent values.

If reading data in an eventually consistent system is unreliable, why use it? Eventual consistency is a common trade-off in large-scale, distributed systems. If you want to scale horizontally (across multiple nodes) to process data in high volumes, then eventually, consistency is often the price you'll pay. Eventual consistency allows you to retrieve data quickly without verifying that you have the latest version across all nodes.

The opposite of eventual consistency is *strong consistency*. With strong consistency, the distributed database ensures that writes to any node are first distributed with a consensus and that any reads against the database return consistent values. You'll use strong consistency when you can tolerate higher query latency and require correct data every time you read from the database.

Generally, data engineers make decisions about consistency in three places. First, the database technology itself sets the stage for a certain level of consistency. Second, configuration parameters for the database will have an impact on consistency. Third, databases often support some consistency configuration at an individual query level. For example, DynamoDB supports eventually consistent reads and strongly consistent reads. Strongly

consistent reads are slower and consume more resources, so it is best to use them sparingly, but they are available when consistency is required.

You should understand how your database handles consistency. Again, data engineers are tasked with understanding technology deeply and using it to solve problems appropriately. A data engineer might need to negotiate consistency requirements with other technical and business stakeholders. Note that this is both a technology and organizational problem; ensure that you have gathered requirements from your stakeholders and choose your technologies appropriately.

File Storage

We deal with files every day, but the notion of a file is somewhat subtle. A *file* is a data entity with specific read, write, and reference characteristics used by software and operating systems. We define a file to have the following characteristics:

Finite length

A file is a finite-length stream of bytes.

Append operations

We can append bytes to the file up to the limits of the host storage system.

Random access

We can read from any location in the file or write updates to any location.

Object storage behaves much like file storage but with key differences. While we set the stage for object storage by discussing file storage first, object storage is arguably much more important for the type of data

engineering you'll do today. We will forward-reference the object storage discussion extensively over the next few pages.

File storage systems organize files into a directory tree. The directory reference for a file might look like this:

```
/Users/matthewhousley/output.txt
```

When I provide this to the operating system, it starts at the root directory `/`, finds `Users`, `matthewhousley`, and finally `output.txt`. Working from the left, each directory is contained inside a parent directory, until we finally reach the file `output.txt`. This example uses Unix semantics, but Windows file reference semantics are similar. The filesystem stores each directory as metadata about the files and directories that it contains. This metadata consists of the name of each entity, relevant permission details, and a pointer to the actual entity. To find a file on disk, the operating system looks at the metadata at each hierarchy level and follows the pointer to the next subdirectory entity until finally reaching the file itself.

Note that other file-like data entities generally don't necessarily have all these properties. For example, *objects* in object storage support only the first characteristic, finite length, but are still extremely useful. We discuss this in "Object Storage".

In cases where file storage paradigms are necessary for a pipeline, be careful with state and try to use ephemeral environments as much as possible. Even if you must process files on a server with an attached disk, use object storage for intermediate storage between processing steps. Try to reserve manual, low-level file processing for one-time ingestion steps or the exploratory stages of pipeline development.

Local disk storage

The most familiar type of file storage is an operating system–managed filesystem on a local disk partition of SSD or magnetic disk. New Technology File System (NTFS) and ext4 are popular filesystems on Windows and Linux, respectively. The operating system handles the details

of storing directory entities, files, and metadata. Filesystems are designed to write data to allow for easy recovery in the event of power loss during a write, though any unwritten data will still be lost.

Local filesystems generally support full read after write consistency; reading immediately after a write will return the written data. Operating systems also employ various locking strategies to manage concurrent writing attempts to a file.

Local disk filesystems may also support advanced features such as journaling, snapshots, redundancy, the extension of the filesystem across multiple disks, full disk encryption, and compression. In "Block Storage", we also discuss RAID.

Network-attached storage

Network-attached storage (NAS) systems provide a file storage system to clients over a network. NAS is a prevalent solution for servers; they quite often ship with built-in dedicated NAS interface hardware. While there are performance penalties to accessing the filesystem over a network, significant advantages to storage virtualization also exist, including redundancy and reliability, fine-grained control of resources, storage pooling across multiple disks for large virtual volumes, and file sharing across multiple machines. Engineers should be aware of the consistency model provided by their NAS solution, especially when multiple clients will potentially access the same data.

A popular alternative to NAS is a storage area network (SAN), but SAN systems provide block-level access without the filesystem abstraction. We cover SAN systems in "Block Storage".

Cloud file system services

Cloud filesystem 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 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. Storage is exposed through the NFS 4 protocol, which is also used by NAS systems. EFS provides automatic scaling and pay-per-storage pricing with no advanced storage reservation required. The service also provides *local* read-after-write consistency (when reading from the machine that performed the write). It also offers open-after-close consistency across the full filesystem. In other words, once an application closes a file, subsequent readers will see changes saved to the closed file.

Block Storage

Fundamentally, *block storage* is the type of raw storage provided by SSDs and magnetic disks. In the cloud, virtualized block storage is the standard for VMs. These block storage abstractions allow fine control of storage size, scalability, and data durability beyond that offered by raw disks.

In our earlier discussion of SSDs and magnetic disks, we mentioned that with these random-access devices, the operating system can seek, read, and write any data on the disk. A *block* is the smallest addressable unit of data supported by a disk. This was often 512 bytes of usable data on older disks, but it has now grown to 4,096 bytes for most current disks, making writes less fine-grained but dramatically reducing the overhead of managing blocks. Blocks typically contain extra bits for error detection/correction and other metadata.

Blocks on magnetic disks are geometrically arranged on a physical platter. Two blocks on the same track can be read without moving the head, while reading two blocks on separate tracks requires a seek. Seek time can occur between blocks on an SSD, but this is infinitesimal compared to the seek time for magnetic disk tracks.

Block storage applications

Transactional database systems generally access disks at a block level to lay out data for optimal performance. For row-oriented databases, this originally meant that rows of data were written as continuous streams; the situation has grown more complicated with the arrival of SSDs and their associated seek-time performance improvements, but transactional databases still rely on the high random access performance offered by direct access to a block storage device.

Block storage also remains the default option for operating system boot disks on cloud VMs. The block device is formatted much as it would be directly on a physical disk, but the storage is usually virtualized. (see "Cloud virtualized block storage".)

RAID

RAID stands for *redundant array of independent disks*, as noted previously. RAID simultaneously controls multiple disks to improve data durability, enhance performance, and combine capacity from multiple drives. An array can appear to the operating system as a single block device. Many encoding and parity schemes are available, depending on the desired balance between enhanced effective bandwidth and higher fault tolerance (tolerance for many disk failures).

Storage area network

Storage area network (SAN) systems provide virtualized block storage devices over a network, typically from a storage pool. SAN abstraction can allow fine-grained storage scaling and enhance performance, availability, and durability. You might encounter SAN systems if you're working with on-premises storage systems; you might also encounter a cloud version of SAN, as in the next subsection.

Cloud virtualized block storage

Cloud virtualized block storage solutions are similar to SAN but free engineers from dealing with SAN clusters and networking details. We'll look at Amazon Elastic Block Store (EBS) as a standard example; other

public clouds have similar offerings. EBS is the default storage for Amazon EC2 virtual machines; other cloud providers also treat virtualized object storage as a key component of their VM offerings.

EBS offers several tiers of service with different performance characteristics. Generally, EBS performance metrics are given in IOPS and throughput (transfer speed). The higher performance tiers of EBS storage are backed by SSD disks, while magnetic disk-backed storage offers lower IOPS but costs less per gigabyte.

EBS volumes store data separate from the instance host server but in the same zone to support high performance and low latency (Figure 6-5). This allows EBS volumes to persist when an EC2 instance shuts down, when a host server fails, or even when the instance is deleted. EBS storage is suitable for applications such as databases, where data durability is a high priority. In addition, EBS replicates all data to at least two separate host machines, protecting data if a disk fails.

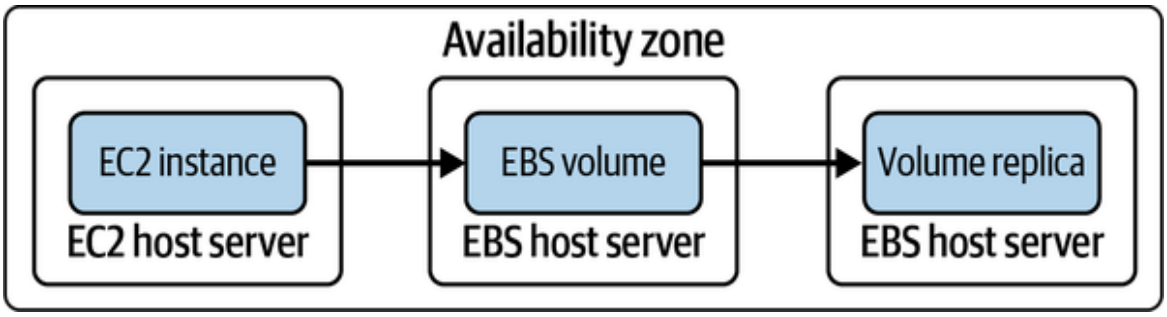

Figure 6-5. EBS volumes replicate data to multiple hosts and disks for high durability and availability, but are not resilient to the failure of an availability zone

EBS storage virtualization also supports several advanced features. For example, EBS volumes allow instantaneous point-in-time snapshots while the drive is used. Although it still takes some time for the snapshot to be replicated to S3, EBS can effectively freeze the state of data blocks when the snapshot is taken, while allowing the client machine to continue using the disk. In addition, snapshots after the initial full backup are differential; only changed blocks are written to S3 to minimize storage costs and backup time.

EBS volumes are also highly scalable. At the time of this writing, some EBS volume classes can scale up to 64 TiB, 256,000 IOPS, and 4,000 MiB/s.

Local instance volumes

Cloud providers also offer block storage volumes that are physically attached to the host server running a virtual machine. These storage volumes are generally very low cost (included with the price of the VM in the case of Amazon's EC2 instance store) and provide low latency and high IOPS.

Instance store volumes (Figure 6-6) behave essentially like a disk physically attached to a server in a data center. One key difference is that when a VM shuts down or is deleted, the contents of the locally attached disk are lost, whether or not this event was caused by intentional user action. This ensures that a new virtual machine cannot read disk contents belonging to a different customer.

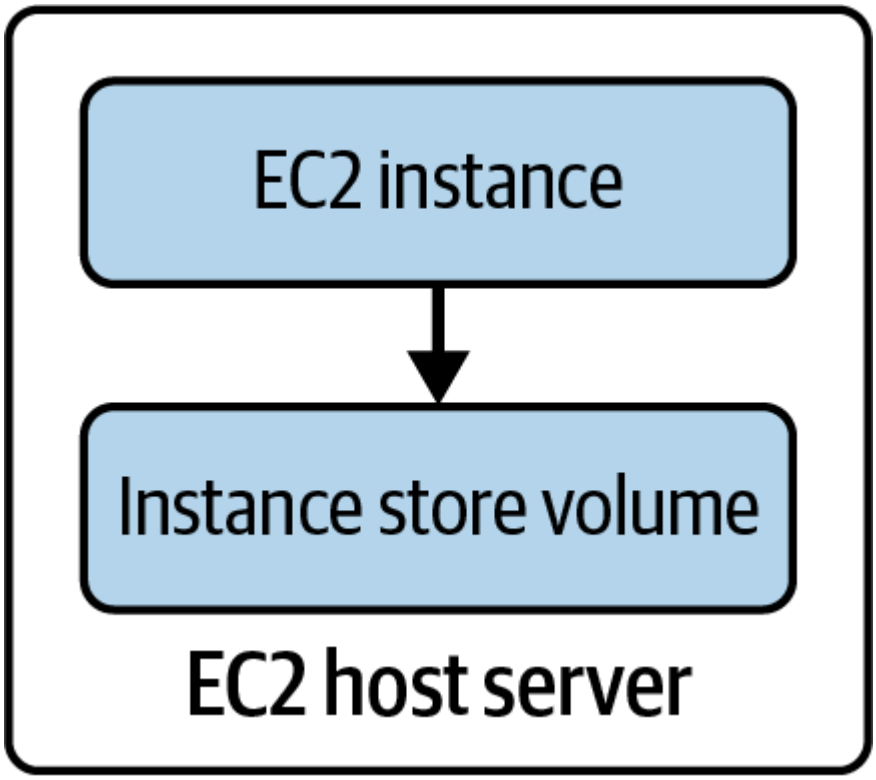

Figure 6-6. Instance store volumes offer high performance and low cost but do not protect data in the event of disk failure or VM shutdown

Locally attached disks support none of the advanced virtualization features offered by virtualized storage services like EBS. The locally attached disk is not replicated, so a physical disk failure can lose or corrupt data even if the host VM continues running. Furthermore, locally attached volumes do not support snapshots or other backup features.

Despite these limitations, locally attached disks are extremely useful. In many cases, we use disks as a local cache and hence don't need all the advanced virtualization features of a service like EBS. For example, suppose we're running AWS EMR on EC2 instances. We may be running an ephemeral job that consumes data from S3, stores it temporarily in the distributed filesystem running across the instances, processes the data, and writes the results back to S3. The EMR filesystem builds in replication and redundancy and is serving as a cache rather than permanent storage. The EC2 instance store is a perfectly suitable solution in this case and can enhance performance since data can be read and processed locally without flowing over a network (see Figure 6-7).

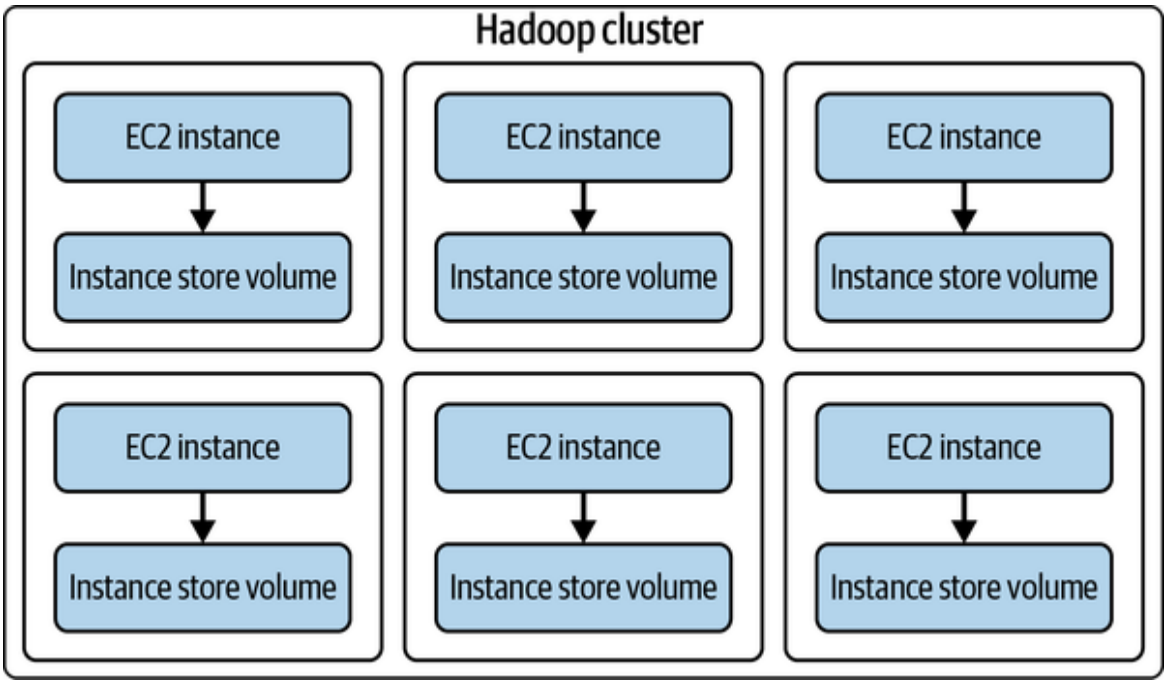

Figure 6-7. Instance store volumes can be used as a processing cache in an ephemeral Hadoop cluster

We recommend that engineers think about locally attached storage in worst-case scenarios. What are the consequences of a local disk failure? Of an accidental VM or cluster shutdown? Of a zonal or regional cloud outage? If none of these scenarios will have catastrophic consequences when data on locally attached volumes is lost, local storage may be a cost-effective and performant option. In addition, simple mitigation strategies (periodic checkpoint backups to S3) can prevent data loss.

Object Storage

Object storage contains *objects* of all shapes and sizes (Figure 6-8). The term *object storage* is somewhat confusing because *object* has several meanings in computer science. In this context, we're talking about a specialized file-like construct. It could be any type of file—TXT, CSV, JSON, images, videos, or audio.

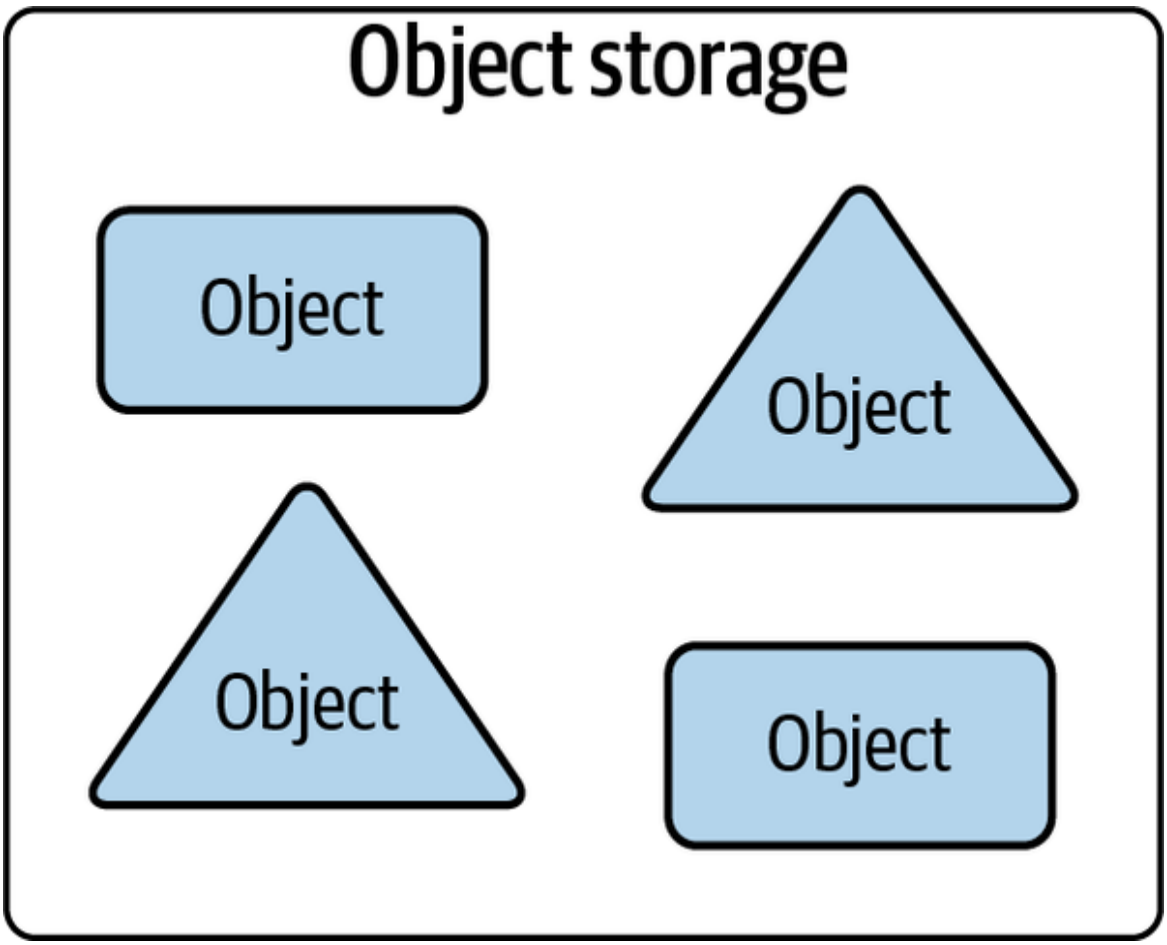

Figure 6-8. Object storage contains immutable objects of all shapes and sizes. Unlike files on a local disk, objects cannot be modified in place.

Object stores have grown in importance and popularity with the rise of big data and the cloud. Amazon S3, Azure Blob Storage, and Google Cloud Storage (GCS) are widely used object stores. In addition, many cloud data warehouses (and a growing number of databases) utilize object storage as their storage layer, and cloud data lakes generally sit on object stores.

Although many on-premises object storage systems can be installed on server clusters, we'll focus mostly on fully managed cloud object stores. From an operational perspective, one of the most attractive characteristics of cloud object storage is that it is straightforward to manage and use. Object storage was arguably 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, but these lookups may perform much worse than random reads from data stored on an SSD.

For a software developer used to leveraging local random access file storage, the characteristics of objects might seem like constraints, but less is more; 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. We'll discuss this in "Storage classes and tiers".

Cloud 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.

Object stores for data engineering applications

From the standpoint of data engineering, object stores provide excellent performance for large batch reads and batch writes. This corresponds well to the use case for massive OLAP systems. A bit of data engineering folklore says that object stores are not good for updates, but this is only partially true. Object stores are an inferior fit for transactional workloads with many small updates every second; these use cases are much better served by transactional databases or block storage systems. Object stores work well for a low rate of update operations, where each operation updates a large volume of data.

Object stores are now the gold standard of storage for data lakes. In the early days of data lakes, write once, read many (WORM) was the operational standard, but this had more to do with the complexities of managing data versions and files than the limitations of HDFS and object stores. Since then, systems such as Apache Hudi and Delta Lake have emerged to manage this complexity, and privacy regulations such as GDPR and CCPA have made deletion and update capabilities imperative. Update management for object storage is the central idea behind the data lakehouse concept, which we introduced in Chapter 3.

Object storage is an ideal repository for unstructured data in any format beyond these structured data applications. Object storage can house any

binary data with no constraints on type or structure and frequently plays a role in ML pipelines for raw text, images, video, and audio.

Object lookup

As we mentioned, object stores are key-value stores. What does this mean for engineers? It's critical to understand that, unlike file stores, object stores do not utilize a directory tree to find objects. The object store uses a top-level logical container (a bucket in S3 and GCS) and references objects by key. A simple example in S3 might look like this:

```
S3://oreilly-data-engineering-book/data-example.json
```

In this case, `S3://oreilly-data-engineering-book/` is the bucket name, and `data-example.json` is the key pointing to a particular object. S3 bucket names must be unique across all of AWS. Keys are unique within a bucket. Although cloud object stores may appear to support directory tree semantics, no true directory hierarchy exists. We might store an object with the following full path:

```
S3://oreilly-data-engineering-book/project-
data/11/23/2021/data.txt
```

On the surface, this looks like subdirectories you might find in a regular file folder system: `project-data`, `11`, `23`, and `2021`. Many cloud console interfaces allow users to view the objects inside a "directory," and cloud command-line tools often support Unix-style commands such as `ls` inside an object store directory. However, behind the scenes, the object system does not traverse a directory tree to reach the object. Instead, it simply sees a key (`project-data/11/23/2021/data.txt`) that happens to match directory semantics. This might seem like a minor technical detail, but engineers need to understand that certain "directory"-level operations are costly in an object store. To run `aws ls S3://oreilly-data-engineering-book/project-data/11/` the object store must filter keys on the key prefix `project-data/11/`. If the bucket contains

millions of objects, this operation might take some time, even if the "subdirectory" houses only a few objects.

Object consistency and versioning

As mentioned, object stores don't support in-place updates or appends as a general rule. We write a new object under the same key to update an object. When data engineers utilize updates in data processes, they must be aware of the consistency model for the object store they're using. Object stores may be eventually consistent or strongly consistent. For example, until recently, S3 was *eventually consistent*; after a new version of an object was written under the same key, the object store might sometimes return the old version of the object. The *eventual* part of *eventual consistency* means that after enough time has passed, the storage cluster reaches a state such that only the latest version of the object will be returned. This contrasts with the *strong consistency* model we expect of local disks attached to a server: reading after a write will return the most recently written data.

It might be desirable to impose strong consistency on an object store for various reasons, and standard methods are used to achieve this. One approach is to add a strongly consistent database (e.g., PostgreSQL) to the mix. Writing an object is now a two-step process:

1. Write the object.
2. Write the returned metadata for the object version to the strongly consistent database.

The version metadata (an object hash or an object timestamp) can uniquely identify an object version in conjunction with the object key. To read an object, a reader undertakes the following steps:

1. Fetch the latest object metadata from the strongly consistent database.
2. Query object metadata using the object key. Read the object data if it matches the metadata fetched from the consistent database.

3. If the object metadata does not match, repeat step 2 until the latest version of the object is returned.

A practical implementation has exceptions and edge cases to consider, such as when the object gets rewritten during this querying process. These steps can be managed behind an API so that an object reader sees a strongly consistent object store at the cost of higher latency for object access.

Object versioning is closely related to object consistency. When we rewrite an object under an existing key in an object store, we're essentially writing a brand-new object, setting references from the existing key to the object, and deleting the old object references. Updating all references across the cluster takes time, hence the potential for stale reads. Eventually, the storage cluster garbage collector deallocates the space dedicated to the dereferenced data, recycling disk capacity for use by new objects.

With object versioning turned on, we add additional metadata to the object that stipulates a version. While the default key reference gets updated to point to the new object, we retain other pointers to previous versions. We also maintain a version list so that clients can get a list of all object versions, and then pull a specific version. Because old versions of the object are still referenced, they aren't cleaned up by the garbage collector.

If we reference an object with a version, the consistency issue with some object storage systems disappears: the key and version metadata together form a unique reference to a particular, immutable data object. We will always get the same object back when we use this pair, provided that we haven't deleted it. The consistency issue still exists when a client requests the "default" or "latest" version of an object.

The principal overhead that engineers need to consider with object versioning is the cost of storage. Historical versions of objects generally have the same associated storage costs as current versions. Object version costs may be nearly insignificant or catastrophically expensive, depending on various factors. The data size is an issue, as is update frequency; more object versions can lead to significantly larger data size. Keep in mind that

we're talking about brute-force object versioning. Object storage systems generally store full object data for each version, not differential snapshots.

Engineers also have the option of deploying storage lifecycle policies. Lifecycle policies allow automatic deletion of old object versions when certain conditions are met (e.g., when an object version reaches a certain age or many newer versions exist). Cloud vendors also offer various archival data tiers at heavily discounted prices, and the archival process can be managed using lifecycle policies.

Storage classes and tiers

Cloud vendors now offer storage classes that discount data storage pricing in exchange for reduced access or reduced durability. We use the term *reduced access* here because many of these storage tiers still make data highly available, but with high retrieval costs in exchange for reduced storage costs.

Let's look at a couple of examples in S3 since Amazon is a benchmark for cloud service standards. The S3 Standard-Infrequent Access storage class discounts monthly storage costs for increased data retrieval costs. (See "A Brief Detour on Cloud Economics" for a theoretical discussion of the economics of cloud storage tiers.) Amazon also offers the Amazon S3 One Zone-Infrequent Access tier, replicating only to a single zone. Projected availability drops from 99.9% to 99.5% to account for the possibility of a zonal outage. Amazon still claims extremely high data durability, with the caveat that data will be lost if an availability zone is destroyed.

Further down the tiers of reduced access are the archival tiers in S3 Glacier. S3 Glacier promises a dramatic reduction in long-term storage costs for much higher access costs. Users have various retrieval speed options, from minutes to hours, with higher retrieval costs for faster access. For example, at the time of this writing, S3 Glacier Deep Archive discounts storage costs even further; Amazon advertises that storage costs start at $1 per terabyte per month. In exchange, data restoration takes 12 hours. In addition, this storage class is designed for data that will be stored from 7–10 years and be accessed only one to two times per year.

Be aware of how you plan to utilize archival storage, as it's easy to get into and often costly to access data, especially if you need it more often than expected. See Chapter 4 for a more extensive discussion of archival storage economics.

Object store–backed filesystems

Object store synchronization solutions have become increasingly popular. Tools like s3fs and Amazon S3 File Gateway allow users to mount an S3 bucket as local storage. Users of these tools should be aware of the characteristics of writes to the filesystem and how these will interact with the characteristics and pricing of object storage. File Gateway, for example, handles changes to files fairly efficiently by combining portions of objects into a new object using the advanced capabilities of S3. However, high-speed transactional writing will overwhelm the update capabilities of an object store. Mounting object storage as a local filesystem works well for files that are updated infrequently.

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 vulnerable to data loss because a power outage lasting even a second can erase 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 key-value store 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 while also taking the load off backend systems.

Example: Redis, memory caching with optional persistence

Like Memcached, *Redis* is a key-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.

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 hundred megabytes 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, barring a *correlated failure* (e.g., an asteroid hitting the data center).

Hadoop is not simply a storage system. Hadoop combines compute resources with storage nodes to allow in-place data processing. This was originally achieved using the MapReduce programming model, which we discuss in Chapter 8.

Hadoop is dead. Long live Hadoop!

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 (thousand-node) 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. We discuss this in more detail in “Separation of Compute from Storage”.

Streaming Storage

Streaming data has different storage requirements than nonstreaming data. In the case of message queues, stored data is temporal and expected to disappear after a certain duration. However, distributed, scalable streaming frameworks like Apache Kafka 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. Chapter 7 covers replay in more depth.

Other storage engines have emerged for real-time analytics applications. In some sense, transactional databases emerged as the first real-time query engines; data becomes visible to queries as soon as it is written. However, these databases have well-known scaling and locking limitations, especially for analytics queries that run across large volumes of data. While scalable versions of row-oriented transactional databases have overcome some of these limitations, they are still not truly optimized for analytics at scale.

Indexes, Partitioning, and Clustering

Indexes provide a map of the table for particular fields and allow extremely fast lookup of individual records. Without indexes, a database would need to scan an entire table to find the records satisfying a `WHERE` condition.

In most RDBMSs, indexes are used for primary table keys (allowing unique identification of rows) and foreign keys (allowing joins with other tables). Indexes can also be applied to other columns to serve the needs of specific applications. Using indexes, an RDBMS can look up and update thousands of rows per second.

We do not cover transactional database records in depth in this book; numerous technical resources are available on this topic. Rather, we are interested in the evolution away from indexes in analytics-oriented storage systems and some new developments in indexes for analytics use cases.

The evolution from rows to columns

An early data warehouse was typically built on the same type of RDBMS used for transactional applications. The growing popularity of MPP systems meant a shift toward parallel processing for significant improvements in scan performance across large quantities of data for analytics purposes. However, these row-oriented MPPs still used indexes to support joins and condition checking.

In "Raw Ingredients of Data Storage", we discuss columnar serialization. *Columnar serialization* allows a database to scan only the columns required for a particular query, sometimes dramatically reducing the amount of data read from the disk. In addition, arranging data by column packs similar values next to each other, yielding high-compression ratios with minimal compression overhead. This allows data to be scanned more quickly from disk and over a network.

Columnar databases perform poorly for transactional use cases—i.e., when we try to look up large numbers of individual rows asynchronously. However, they perform extremely well when large quantities of data must be scanned—e.g., for complex data transformations, aggregations, statistical calculations, or evaluation of complex conditions on large datasets.

In the past, columnar databases performed poorly on joins, so the advice for data engineers was to denormalize data, using wide schemas, arrays, and nested data wherever possible. Join performance for columnar databases has improved dramatically in recent years, so while there can still be performance advantages in denormalization, this is no longer a necessity. You'll learn more about normalization and denormalization in Chapter 8.

From indexes to partitions and clustering

While columnar databases allow for fast scan speeds, it's still helpful to reduce the amount of data scanned as much as possible. In addition to scanning only data in columns relevant to a query, we can partition a table into multiple subtables by splitting it on a field. It is quite common in analytics and data science use cases to scan over a time range, so date- and time-based partitioning is extremely common. Columnar databases generally support a variety of other partition schemes as well.

Clusters allow finer-grained organization of data within partitions. A clustering scheme applied within a columnar database sorts data by one or a few fields, colocating similar values. This improves performance for filtering, sorting, and joining these values.

Example: Snowflake micro-partitioning

We mention Snowflake micro-partitioning because it's a good example of recent developments and evolution in approaches to columnar storage. *Micro partitions* are sets of rows between 50 and 500 megabytes in uncompressed size. Snowflake uses an algorithmic approach that attempts to cluster together similar rows. This contrasts the traditional naive approach to partitioning on a single designated field, such as a date. Snowflake specifically looks for values that are repeated in a field across many rows. This allows aggressive *pruning* of queries based on predicates. For example, a `WHERE` clause might stipulate the following:

```
WHERE created_date='2017-01-02'
```

In such a query, Snowflake excludes any micro-partitions that don't include this date, effectively pruning this data. Snowflake also allows overlapping micro-partitions, potentially partitioning on multiple fields showing significant repeats.

Efficient pruning is facilitated by Snowflake's metadata database, which stores a description of each micro-partition, including the number of rows and value ranges for fields. At each query stage, Snowflake analyzes micro-partitions to determine which ones need to be scanned. Snowflake uses the term *hybrid columnar storage*,[2] partially referring to the fact that its tables are broken into small groups of rows, even though storage is fundamentally columnar. The metadata database plays a role similar to an index in a traditional relational database.

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 atop the data storage systems discussed previously (see Figure 6-3). We introduced many of these abstractions in Chapter 3, and we will revisit them here.

The main types of abstractions we'll concern ourselves with are those that support data science, analytics, and reporting use cases. These include data warehouse, data lake, data lakehouse, data platforms, and data catalogs. We won't cover source systems, as they are discussed in Chapter 5.

The storage abstraction you require as a data engineer boils down to a few key considerations:

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

The trend is toward separating storage and compute, but most systems hybridize separation and colocation. We cover this in "Separation of Compute from Storage" since it affects purpose, speed, and cost.

You 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.

The Data Warehouse

Data warehouses are a standard OLAP data architecture. As discussed in Chapter 3, the term *data warehouse* refers to technology platforms (e.g., Google BigQuery and Teradata), an architecture for data centralization, and an organizational pattern within a company. In terms of storage trends, we've evolved from building data warehouses atop conventional transactional databases, row-based MPP systems (e.g., Teradata and IBM Netezza), and columnar MPP systems (e.g., Vertica and Teradata Columnar) to cloud data warehouses and data platforms. (See our data warehousing discussion in Chapter 3 for more details on MPP systems.)

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, as originally conceived by James Dixon.[3] 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 data lake. Cloud data warehouses can be coupled with object storage to provide a complete data-lake solution.

The Data Lake

The *data lake* was originally conceived as a massive store where data was retained in raw, 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 a proprietary 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) and initially dismissed in the rush to data lakes was, in fact, extremely useful. This led to the notion of the data lakehouse.

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 over proprietary tools is interoperability. It's much easier to 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 impose data warehouse features where we need them in a lakehouse, leaving other data in a raw or even unstructured format.

The data lakehouse technology is evolving rapidly. A variety of new competitors to Delta Lake have emerged, including Apache Hudi and

Apache Iceberg. See Appendix A for more details.

Data Platforms

Increasingly, vendors are styling their products as *data platforms*. These vendors have created their ecosystems of interoperable tools with tight integration into the core data storage layer. In evaluating platforms, engineers must ensure that the tools offered meet their needs. Tools not directly provided in the platform can still interoperate, with extra data overhead for data interchange. Platforms also emphasize close integration with object storage for unstructured use cases, as mentioned in our discussion of cloud data warehouses.

At this point, the notion of the data platform frankly has yet to be fully fleshed out. However, the race is on to create a walled garden of data tools, both simplifying the work of data engineering and generating significant vendor lock-in.

Stream-to-Batch Storage Architecture

The stream-to-batch storage architecture has many similarities to the Lambda architecture, though some might quibble over the technical details. Essentially, data flowing through a topic in the streaming storage system is written out to multiple consumers.

Some of these consumers might be real-time processing systems that generate statistics on the stream. In addition, a batch storage consumer writes data for long-term retention and batch queries. The batch consumer could be AWS Kinesis Firehose, which can generate S3 objects based on configurable triggers (e.g., time and batch size). Systems such as BigQuery ingest streaming data into a streaming buffer. This streaming buffer is automatically reserialized into columnar object storage. The query engine supports seamless querying of both the streaming buffer and the object data to provide users a current, nearly real-time view of the table.

Big Ideas and Trends in Storage

In this section, we'll discuss some big ideas in storage—key considerations that you need to keep in mind as you build out your storage architecture. Many of these considerations are part of larger trends. For example, data catalogs fit under the trend toward "enterprisey" data engineering and data management. Separation of compute from storage is now largely an accomplished fact in cloud data systems. And data sharing is an increasingly important consideration as businesses adopt data technology.

Data Catalog

A *data catalog* is a centralized metadata store for all data across an organization. Strictly speaking, a data catalog is not a top-level data storage abstraction, but it integrates with various systems and abstractions. Data catalogs typically work across operational and analytics data sources, integrate data lineage and presentation of data relationships, and allow user editing of data descriptions.

Data catalogs are often used to provide a central place where people can view their data, queries, and data storage. As a data engineer, you'll likely be responsible for setting up and maintaining the various data integrations of data pipeline and storage systems that will integrate with the data catalog and the integrity of the data catalog itself.

Catalog application integration

Ideally, data applications are designed to integrate with catalog APIs to handle their metadata and updates directly. As catalogs are more widely used in an organization, it becomes easier to approach this ideal.

Automated scanning

In practice, cataloging systems typically need to rely on an automated scanning layer that collects metadata from various systems such as data lakes, data warehouses, and operational databases. Data catalogs can collect

existing metadata and may also use scanning tools to infer metadata such as key relationships or the presence of sensitive data.

Data portal and social layer

Data catalogs also typically provide a human access layer through a web interface, where users can search for data and view data relationships. Data catalogs can be enhanced with a social layer offering Wiki functionality. This allows users to provide information on their datasets, request information from other users, and post updates as they become available.

Data catalog use cases

Data catalogs have both organizational and technical use cases. Data catalogs make metadata easily available to systems. For instance, a data catalog is a key ingredient of the data lakehouse, allowing table discoverability for queries.

Organizationally, data catalogs allow business users, analysts, data scientists, and engineers to search for data to answer questions. Data catalogs streamline cross-organizational communications and collaboration.

Data Sharing

Data sharing allows organizations and individuals to share specific data and carefully defined permissions with specific entities. Data sharing allows data scientists to share data from a sandbox with their collaborators within an organization. Across organizations, data sharing facilitates collaboration between partner businesses. For example, an ad tech company can share advertising data with its customers.

A cloud multitenant environment makes interorganizational collaboration much easier. However, it also presents new security challenges. Organizations must carefully control policies that govern who can share data with whom to prevent accidental exposure or deliberate exfiltration.

Data sharing is a core feature of many cloud data platforms. See Chapter 5 for a more extensive discussion of data sharing.