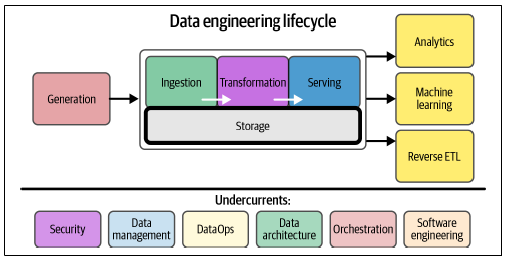
# Fundamentals of Data Engineering - Reis & Housley

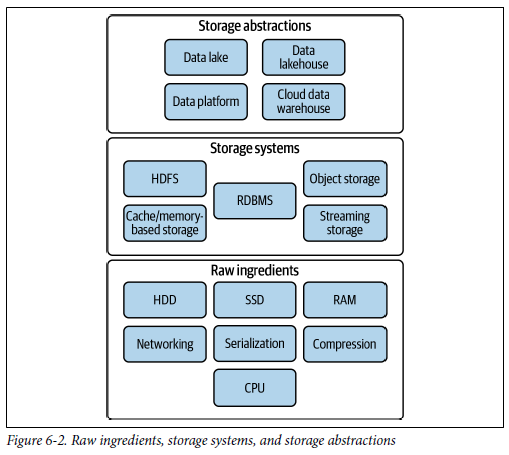
## Part II. The Data Engineering Lifecyle in Depth

### Chapter 6 – Storage

* **Storage is the cornerstone of the DE lifecycle + underlies its major stages of ingestion, transformation, + serving**



* **Data gets stored *many* times as it moves through the DE lifecycle** (*it’s storage all the way down*)
* Whether data is needed seconds, minutes, days, months, or years later, it ***must* persist in storage until systems are ready to consume it for further processing + transmission**
* **Knowing the use case of the data + the way you will retrieve it in the future is the first step to choosing the proper storage solutions for your data architecture**
* We discussed storage in Chapter 5 (Source Systems), but *w/ a difference in focus + domain of control*
* **Source systems are generally NOT maintained or controlled by DE’s**
* **The storage that DE’s *do* handle directly (this chapter) encompasses the DE lifecycle stages of ingesting data from source systems to serving data to *deliver value* w/ analytics, data science, ML, etc.**
* **Many forms of storage undercut the entire DE lifecycle in some fashion**
* To understand storage, we’re going to start by studying the **raw ingredients**that compose storage systems, including **hard drives, solid state drives, + system memory**
* 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**
* See a deeper technical discussion of serialization and compression in *Appendix A*)
* We also discuss **caching**, which is **critical in assembling storage systems**
* 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 + clusters that can ingest + retrieve data using various access paradigms**
* Finally, we’ll look at **storage abstractions**
* **Storage systems are assembled into a cloud DW, a data lake, etc.**
* **When building data pipelines, DE’s choose the appropriate abstractions for storing their data as it moves through the ingestion, transformation, + serving stages**



#### Raw Ingredients of Data Storage

* Storage is so common that it’s easy to take it for granted
* Often surprised by the number of SWE + DE’s who use storage every day but have little idea how it works BTS 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 DE’s from the complexities of managing servers, DE’s still need to be aware of underlying components’ essential characteristics, performance considerations, durability, + costs**
* In ***most* data architectures, data frequently passes through magnetic storage, SSDs, + memory** as it works its way through the various processing phases of a data pipeline
* **Data storage + query systems generally follow complex recipes involving distributed systems, numerous services, + multiple hardware storage layers**
* **These systems require the *right* raw ingredients to function correctly**
* Some raw ingredients of data storage: **disk drives, memory, networking + CPU, serialization, compression, + caching**

##### 1) Magnetic Disk Drive

* **Magnetic disks**utilize spinning platters coated with a ferromagnetic film, which is magnetized by a **read/write head** **during write operations to physically encode binary data**
* The **read/write head detects the magnetic field + outputs a bitstream during read operations**
* **Magnetic disk drives (HDD)** have been around for ages + still form the **backbone of bulk data storage systems because they are significantly cheaper than SSDs per GB of stored data**
* On the one hand, these disks **have seen extraordinary improvements in performance, storage density, + cost**
* Currently, commercial magnetic disk drives cost roughly 3 cents per GB of capacity
* **On the other hand, *SSDs dramatically outperform magnetic disks on various metrics***
* IBM developed magnetic disk drive (HDD) technology in the 1950s, + since then, HDD capacities have grown steadily
* The first commercial HDD, the IBM 350, had a capacity of 3.75 MB
* As of this writing, HDD 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), + 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 (i.e., HDDs have disadvantages)**
* 1) **Disk transfer speed** **(rate at which data can be read and written)** **does *NOT* scale in proportion w/ disk capacity**
* **Disk capacity scales w/** **areal density**(gigabits stored per square inch), whereas **transfer speed scales w/ linear density**(bits per inch)
* This means that if disk capacity grows by a factor of 4X, transfer speed increases by only a factor of 2X
* Consequently, current data center drives support maximum data transfer speeds of 200–300 MB/s
* To frame this another way, it takes > 20 hours to read the entire contents of a 30 TB magnetic drive, assuming a transfer speed of 300 MB/s
* 2) **Seek time** 🡪 To access data, the drive must *physically* relocate the read/write heads to the appropriate track on the disk
* 3) 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**, which **leads to** **rotational latency**
* Typical commercial drives spinning at 7,200 revolutions per minute (RPM) seek time, + rotational latency, leads to over 4ms of overall average **latency (time to access a selected piece of data)**
* 4) **Input/output operations per second (IOPS), critical for transactional databases**
* An HDD ranges from 50 to 500 IOPS
* **Various tricks can improve latency + transfer speed**
* Using a higher rotational speed can increase transfer rate + 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 HDDs remotely competitive w/ SSDs for random access lookups**
* **SSDs can deliver data w/ significantly lower latency, higher IOPS, + higher transfer speeds, partially because there is no physically rotating disk or magnetic head to wait for**
* As mentioned earlier, **HDDs are still prized in data centers for their low data storage costs**
* In addition, **HDDs 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 + CPUs are also key raw ingredients in storage systems** (seen later)

##### 2) Solid-State Drive

* **Solid-state drives (SSDs)** **store data as charges in flash memory cells**
* SSDs **eliminate the mechanical components of HDDs**, + the **data is read by purely electronic means**

SSDs can look up random data in < 0.1ms (100 microseconds) + can also **scale both data-transfer speeds + IOPS by slicing storage into partitions w/ numerous storage controllers running in parallel**

* Commercial SSDs can support transfer speeds of many GB per second + tens of thousands of IOPS
* B/c of these **exceptional performance characteristics,** **SSDs have revolutionized transactional databases + are the accepted standard for commercial deployments of OLTP systems**
* **SSDs allow RDBs** such as PostgreSQL, MySQL, + MS 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 10X the cost per capacity of an HDD
* Thus, **object storage on HDDs has emerged as the leading option for large-scale data storage in data lakes + cloud DWs**
* **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, expect SSD usage in these systems to follow suit

##### 3) Random Access Memory

* We commonly use the terms **random access memory (RAM)** and **memory**interchangeably
* Strictly speaking, **HDDs + 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 + mapped into CPU address space**
* It **stores the code that CPUs execute + the data that this code directly processes**
* It is **volatile, while magnetic drives and SSDs are *nonvolatile***
* Though they may occasionally fail + corrupt or lose data, **drives generally retain data when powered off, while RAM loses data in < 1 second when unpowered**
* **RAM offers significantly higher transfer speeds + faster retrieval times than SSD storage**
* DDR5 memory (the latest widely used standard for RAM) offers data retrieval latency on the order of 100ns, roughly 1,000X faster than SSD
* A typical CPU can support 100 GB/s bandwidth to attached memory and also millions of IOPS (Statistics vary dramatically depending on the number of memory channels + other configuration details)
* It is **significantly more expensive than SSD storage**, at roughly $10/GB (as of 2022-2023)
* It is **limited in the amount of RAM attached to an individual CPU + memory controller**
* This **adds further to complexity and cost**
* High-memory servers typically utilize many interconnected CPUs on 1 board, each w/ a block of attached RAM
* It 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 + 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**
* **Capacitors leak over time, so the data must be frequently refreshed(read + rewritten) to prevent data loss**
* The **hardware memory controller handles these technical details** + **DE’s simply need to worry about bandwidth + 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 + 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 + data**
* **RAM is used in various storage + 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 + write performance**
* **In *these* applications, DE’s 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 + automatically dump all data to disk in the event of power loss

##### 4) Networking and CPU

* Why mention **networking** + **CPU** as raw ingredients for storing data?
* **Increasingly, storage systems are *distributed* to enhance performance, durability, + availability**
* Specifically, individual HDDs 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, w/ disks distributed across a network + even multiple data centers + availability zones** **(a standard cloud construct consisting of compute environments w/ independent power, water, + other resources )**
* **Multizonal storage enhances both the availability + durability of data**
* **CPUs handle the details of servicing requests, aggregating reads, + distributing writes**
* **Storage becomes a web app w/ an API, backend service components, + load balancing**
* **Network device performance + network topology are key factors in realizing high performance**
* **DE’s need to understand how networking will affect the systems they build + use**
* **DE’s constantly balance the durability + availability achieved by spreading out data geographically vs. the performance + cost benefits of keeping storage in a small geographic area + close to data consumers or writers**
* Appendix B covers cloud networking and major relevant ideas

##### 5) Serialization

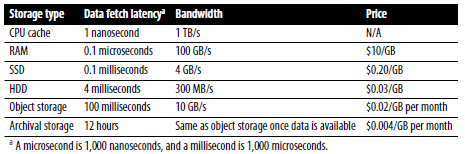
* **Serialization**is another raw storage ingredient + a **critical element of database design**
* **Decisions around serialization will inform how well queries perform across a network, CPU overhead, query latency, + more**
* Ex: Designing a data lake involves choosing a base storage system (e.g., Amazon S3) + standards for serialization that balance interoperability w/ 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 + packing data into a standard format that a reader will be able to decode**
* **Serialization formats provide a standard of data exchange**
* Might encode data in a row-based manner as an XML, JSON, or CSV file, + 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, + allows exchange between programming languages + CPUs**
* The serialization algorithm **also has rules for handling exceptions**
* Ex: 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 RDBs organize data as rows on disk to support speedy lookups + in-place updates**
* **Columnar databases organize data into column files to optimize for highly efficient compression + support fast scans of large data volumes**
* **Each serialization choice comes w/ a set of trade-offs, + DE’s tune these choices to optimize performance to requirements**
* See a more detailed catalog of common data serialization techniques + formats in Appendix A
* **DE’s should become familiar + common serialization practices + 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 w/ *other* details of storage systems in complex ways**
* Highly efficient compression has **3 main advantages in storage systems**
* **1)** The **data is smaller** + thus **takes up less space on the disk**
* **2)** Compression **increases the practical scan speed per disk**
* W/ 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
* **3) 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 + decompressing data = extra time + resource consumption to read or write data**
* There is a more detailed discussion of compression algorithms and trade-offs in Appendix A

##### 7) Caching

* 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 + the less storage space available**
* **Less frequently accessed data is stored in cheaper, slower storage**
* Caches are **critical for data serving, processing, + transformation**
* When analyzing storage systems, it’s helpful to put every type of storage inside a **cache hierarchy**



* **Most practical data systems rely on *many* cache layers assembled from storage w/ varying performance characteristics**
* This starts inside CPUs, where processors may deploy up to 4 cache tiers
* We move down the hierarchy to RAM and SSDs
* Cloud object storage is a lower tier that supports long-term data retention + durability while allowing for data serving + dynamic data movement in pipelines
* Can think of archival storage as a **reverse cache**
* Archival storage provides inferior access characteristics for low costs + is generally used for data backups + 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 + 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 DE
* **Storage systems exist at a level of abstraction above raw ingredients**
* Ex: Magnetic disks are a raw storage ingredient, while major cloud object storage platforms + HDFS are storage systems that *utilize* magnetic disks
* Still higher levels of storage abstraction exist, such as data lakes + lakehouses

##### Single Machine Vs. Distributed Computing

* As data storage + access patterns become more complex + outgrow the usefulness of a single server, **distributing data to more than 1 server becomes necessary**
* **Data can be stored on multiple servers, known as distributed storage**, **a distributed system whose purpose is to store data in a distributed fashion**
* Distributed storage coordinates the activities of multiple servers to **store, retrieve, + process data faster + 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 + scalability for large amounts of data**
* Ex: Object storage, Apache Spark, + cloud DW’s rely on distributed storage architectures
* **DE’s must always be aware of the consistency paradigms of the distributed systems**

##### Eventual Vs. Strong Consistency

* A challenge w/ distributed systems is that data is spread across multiple servers, so *how does this system keep the data* ***consistent****?*
* Unfortunately, **distributed systems pose a dilemma for storage + query accuracy, as it takes time to replicate changes across the nodes of a system + often a balance exists between getting current data + getting “sort of” current data in a distributed database**
* Let’s look at **2 common consistency patterns in distributed systems: eventual and strong**
* We’ve covered **ACID compliance**, + another acronym is **BASE** = **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 database reads + 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, + 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**
* W/ strong consistency, the **distributed database ensures that writes to any node are first distributed w/ a consensus + that any reads against the database return consistent values**
* **Use strong consistency when you can tolerate higher query latency + require correct data *every time* you read from the database**
* Generally, DE’s make decisions about consistency in 3 places
* **1) The database technology itself sets the stage for a certain level of consistency**
* **2) Configuration parameters for the database will have an impact on consistency**
* **3) Databases often support some consistency configuration at an individual query level**
* Ex: DynamoDB supports eventually consistent reads *and* strongly consistent reads
* **Strongly consistent reads are slower + consume more resources, so it is best to use them sparingly, but they are available when consistency is required**
* **Understand how your database handles consistency**
* Again, **DE’s are tasked w/ understanding technology deeply + using it to solve problems appropriately**
* A **DE might need to negotiate consistency requirements w/ other technical + business stakeholders**
* Note that **this is both a tech *and* organizational problem, so ensure you have gathered requirements from your stakeholders + choose your technologies appropriately**

##### File Storage

* We deal w/ files every day, but the notion of a file is somewhat subtle
* A **file**is **a** **data entity w/ specific read, write, + reference characteristics used by software + OS’s**
* 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 storagebehaves 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 DE you do today**
* **File storage systems organize files into a directory tree**, + the directory reference for a
* file might look like this: */Users/matthewhousley/output.txt*
* When this file reference is passed to the OS, it starts at the **root directory** */*, finds *Users*, *matthewhousley*, + 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 + directories that it contains**
* This **metadata consists of the name of each entity, relevant permission details, + a pointer to the actual entity**
* To find a file on disk, the OS looks at the metadata at each hierarchy level + 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.
* Ex: **Objectsin object storage support only the 1st characteristic, finite length, but are still extremely useful**
* **In cases where file storage paradigms are necessary for a pipeline, be careful w/ state + 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**

###### a) Local Disk Storage

* The **most familiar type of file storage is an OS-managed filesystem on a local disk partition of SSD or HDD**
* New Technology File System (NTFS) and ext4 are popular filesystems on Windows and Linux, respectively
* The **OS handles the details of storing directory entities, files, + 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**, where reading immediately after a write will return the written data
* OS’s 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, + compression**

###### b) 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**, which quite often ship w/ 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 + reliability, fine-grained control of resources, storage pooling across multiple disks for large virtual volumes, + file sharing across multiple machines**
* **DE’s 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**

###### c) Cloud Filesystem Services

* **Cloud filesystem services** provide a **fully managed filesystem for use w/ multiple cloud VMs + applications, potentially including clients outside the cloud environment**
* Cloud filesystems **should not be confused w/ standard storage attached to VMs (generally, block storage w/ a filesystem managed by the VM’s OS)**
* 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**
* Ex: 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 + pay-per-storage pricing w/ no advanced storage reservation required**
* Also provides ***local* read-after-write consistency (when reading from the machine that performed the write)**
* 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 HDDs, + in the cloud, *virtualized* block storage is the standard for VMs**
* These **block storage abstractions allow fine control of storage size, scalability, + data durability beyond that offered by raw disks**
* In earlier discussions of SSDs and HDDs, we mentioned that **w/ these random-access devices, the OS can seek, read, 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 + other metadata**
* Blocks on HDDs are geometrically arranged on a physical platter
* 2 blocks on the same track can be read w/out moving the head, while reading 2 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 HDDs**

###### a) 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 w/ the arrival of SSDs + 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 OS 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

###### b) RAID

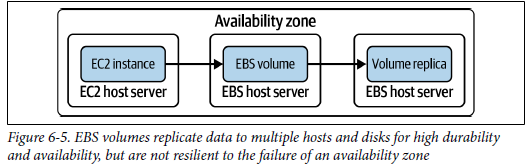
* **RAID****= redundant array of independent disks**
* RAID **simultaneously controls multiple disks to improve data durability, enhance performance, + combine capacity from multiple drives**
* An **array can appear to the OS as a single block device**
* Many encoding + parity schemes are available,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

###### c) 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 + enhance performance, availability, + durability**
* Might encounter SAN systems if working w/ on-prem storage systems, + might also encounter a cloud version of SAN

###### d) Cloud Virtualized Block Storage

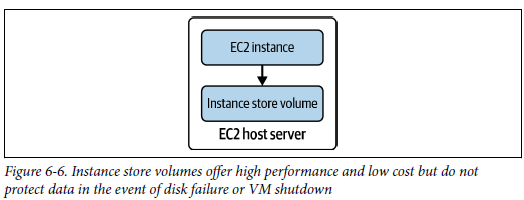
* **Cloud virtualized block storagesolutions are similar to SAN but free DE’s from dealing w/ SAN clusters + networking details**
* Amazon Elastic Block Store (EBS) is a standard example, + other public clouds have similar offerings
* EBS = the default storage for Amazon EC2 VMs (other cloud providers also treat virtualized object storage as a key component of their VM offerings)
* EBS offers several tiers of service w/ different performance characteristics
* Generally, EBS **performance metrics are given in IOPS + throughput (transfer speed)**
* Higher performance tiers of EBS storage are backed by SSD disks, while HDD-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 + low latency**



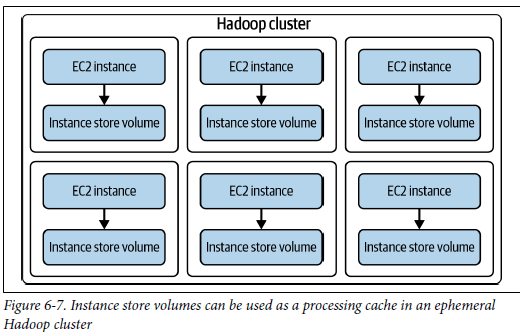
* This **allows EBS volumes to persist when an EC2 instance shuts down, when a host server fails, or even when the instance is deleted**
* EBS storage is suitable for applications such as **databases, where data durability is a high priority**
* In addition, EBS **replicates all data to at least 2 separate host machines, protecting data if a disk fails**
* EBS **storage virtualization also supports several advanced features**
* Ex: 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 + backup time)
* EBS volumes are also **highly scalable**
* As of 2022-2023, some EBS volume classes can scale up to 64 TiB, 256,000 IOPS, + 4,000 MiB/s

###### e) Local Instance Volumes

* Cloud providers also offer **block storage volumes that are *physically attached* to the host server running a VM**
* *These* storage volumes are **generally very low cost** (included w/ the price of the VM in the case of Amazon’s EC2 instance store) + **provide low latency and high IOPS**
* **Instance store volumes** (see below) **behave essentially like a disk physically attached to a server in a data center**

****

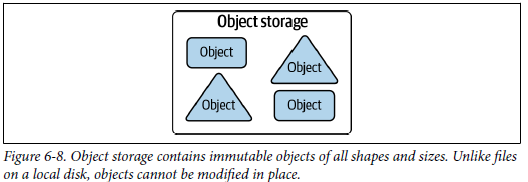
* **One key difference is that when a VM shuts down or is deleted, the contents of the locally attached disk are lost, whether or not this event was caused by intentional user action**
* This **ensures that a new VM cannot read disk contents belonging to a different customer**
* **Locally-attached disks 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
* Ex: 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, + writes the results back to S3
* The EMR filesystem builds in replication + redundancy + is serving as a cache rather than permanent storage
* The EC2 instance store is a perfectly suitable solution in this case + can enhance performance since data can be read + processed locally w/out flowing over a network



* **DE’s should 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 + 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**

**

* The term “object storage*”* is somewhat confusing because “object*”* has several meanings in CS

In *this* context, talking about **a specialized file-like construct** **that could be *any* type of file** (TXT, CSV, JSON, images, videos, or audio)

* **Object stores have grown in importance + popularity w/ the rise of big data + the cloud**
* Amazon S3, Azure Blob Storage, + Google Cloud Storage (GCS) are widely used **object stores**
* **In addition, many cloud DW’s (and a growing number of databases) utilize object storage as their storage layer, + cloud data lakes generally sit on object stores**
* Although many on-prem object storage systems can be installed on server clusters, we 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 + use**
* **Object storage was arguably one of the first “serverless” services, as DE’s 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**
* **Lose much of the writing flexibility we expect w/ 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, as **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, + this parallelism is hidden from DE’s, who can simply deal w/ the stream rather than communicating with individual disks**
* **In a cloud environment, *write* speed scales w/ the number of streams being written up to quota limits set by the vendor**.
* ***Read* bandwidth can scale w/ the number of parallel requests, number of VMs employed to read data, + 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 + availability are built into the cost**, + cloud storage vendors offer other storage classes at discounted prices in exchange for reduced durability or availability
* **Cloud object storage is a key ingredient in separating compute vs. storage, allowing DE’s to process data w/ ephemeral clusters + scale these clusters up + 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.
* But still, the **general trend is that most organizations will move data processing to the cloud, using an object store as essential storage + 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 (Though *in practice*, the primary limit on storage space for public cloud customers is *budget*)**
* From a practical standpoint, DE’s can quickly store massive quantities of data for projects w/out planning months in advance for necessary servers + disks

###### a) Object Stores for Data Engineering Applications

* **From the standpoint of DE, object stores provide excellent performance for large batch reads + batch writes**
* This **corresponds well to the use case for massive OLAP systems**
* A bit of DE 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 w/ *many small updates* every second**
* Such 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 + files than the limitations of HDFS + object stores**
* Since then, systems such as Apache Hudi + Delta Lake have emerged to manage this complexity, + privacy regulations such as GDPR + CCPA have made deletion + update capabilities imperative
* **Update management for object storage is the central idea behind the data lakehouse concept**
* **Object storage is an ideal repository for unstructured data in any format beyond these structured data applications**
* Object storage **can house any binary data w/ no constraints on type or structure + frequently plays a role in ML pipelines for raw text, images, video, + audio**

###### b) Object Lookup

* As mentioned, **object stores are key-value stores**, *but what does this mean for DE’s?*
* **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) + references objects by *key***
* Simple S3 example might look like: *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**
* Ex: Might store an object w/ 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**
* Many cloud console interfaces allow users to view the objects inside a “directory,” + cloud CLI 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 DE’s need to understand that certain “directory”-level operations are costly in an object store**
* To run “*aws ls S3://oreilly-data-engineering-book/projectdata/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

###### c) 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 DE’s 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**
* Ex: Until recently, S3 was **eventually consistent (after a new version of an object was written under the same key, object store might sometimes return old version of an 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, where **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, + standard methods are used to achieve this**
* **1 approach = add a strongly consistent database (e.g., PostgreSQL) to the mix**
* Writing an object is now a 2-step process:
* 1. Write the object
* 2. Write 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 w/ 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 + 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, + 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 +** also **maintain a version list so that clients can get a list of all object versions + 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 w/ a version, the consistency issue w/ some object storage systems disappears** (the key + 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 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 DE’s need to consider w/ 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
* DE’s 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**, + the archival process can be managed using lifecycle policies

###### d) 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 b/c many of these storage tiers still make data highly available, but w/ **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
* 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 possibility of a zonal outage
* Amazon still claims extremely high data durability, w/ 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, which promises a dramatic reduction in long-term storage costs for much higher access costs
* Users have various retrieval speed options, from minutes to hours, w/ higher retrieval costs for faster access
* Ex: As of 2022-2023, S3 Glacier Deep Archive discounts storage costs even further (Amazon advertises that storage costs start at $1 per TB per month, + in exchange, data restoration takes 12 hours)
* In addition, this storage class is designed for data that will be stored 7–10 years + be accessed only 1-2 times per year
* **Be aware of how you plan to utilize archival storage, as it’s easy to get into + 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*

###### e) 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 + how these will interact w/ the characteristics + pricing of object storage**
* Ex: File Gateway 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, **RAM offers excellent latency + transfer speeds**
* *However*, ***traditional* RAM is extremely vulnerable to data loss b/c 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 + high bandwidth**
* **Data should generally be written to a more durable medium for retention purposes**
* These **ultra-fast cache systems are useful when DE’s need to serve data w/ ultra-fast retrieval latency**
* **Example: Memcached and lightweight object caching**
* **Memcachedis a key-value store designed for caching database query results, API call responses, + more**
* It **uses simple data structures**, supporting either string or integer types
* **Can deliver results w/ very low latency while also taking the load off backend systems**
* **Example: Redis, memory caching with optional persistence**
* **Redis**is **another key-value store, but supports somewhat more complex data types (such as lists or sets)**
* Redis also **builds in multiple persistence mechanisms, including snapshotting + journaling**
* W/ a typical configuration, Redis writes data roughly every 2 seconds
* Redis is thus **suitable for extremely high-performance applications but can tolerate a small amount of data loss**

##### The Hadoop Distributed File System

* In the recent past, “Hadoop” was virtually synonymous with “big data.”
* **The Hadoop Distributed File System (HDFS)** is based on **Google File System (GFS)** + was initially engineered to process data with the **MapReduce programming model**
* **Hadoop is similar to object storage but w/ a key difference: Hadoop combines compute + storage on the *same* nodes, where object stores typically have limited support for internal processing**
* **Hadoop breaks large files into blocks (chunks of data < a few hundred MB in size)**
* The **filesystem is managed by the NameNode, which maintains directories, file metadata, + a detailed catalog describing the location of file blocks in the cluster**
* In **a typical configuration, each block of data is replicated to 3 nodes**, which **increases both the durability + 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 w/ storage nodes to allow in-place data processing**
* This was **originally achieved using the MapReduce programming model** (see Chapter 8)
* Often see claims that Hadoop is dead, which 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 + primarily used to run legacy jobs, the pure MapReduce programming model has fallen by the wayside
* *BUT* HDFS remains widely used in various applications + 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 + have the resources to maintain on-prem systems effectively
* Smaller companies may want to reconsider the cost overhead + 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

##### Streaming Storage

* **Streaming data has different storage requirements than non-streaming data**
* In the case of **message queues, stored data is temporal + 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, + Google Cloud Pub/Sub) also support long data retention
* Closely related to data retention in these systems is the notion of **replay**, which **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** (see Chapter 7)
* 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 + 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 + 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 table PKs (for unique identification of rows) and FKs (for joins w/ 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.**
* Rather than cover transactional database records in depth, we are interested in the **evolution *away* from indexes in analytics-oriented storage systems + some new developments in indexes for analytics use cases**

###### The Evolution from Rows to Columns

* **An early DW was typically built on the same type of RDBMS used for transactional applications**
* **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 + condition checking**
* Remember **columnar serialization**, which **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**, which **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 DE’s was to denormalize data, using wide schemas, arrays, + 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**

###### 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 sub-tables by splitting it on a field**
* It is quite common in analytics + 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 w/in a columnar database sorts data by one or a few fields, collocating 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 + evolution in approaches to columnar storage
* **Micro partitions**are **sets of rows between 50 and 500 MB 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 pruningof queries based on predicates**
* Ex: A WHERE clause might stipulate the following: ***WHERE*** *created\_date=‘2022-01-02’*
* 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 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**, 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 RDB

#### Data Engineering Storage Abstractions

* **Data engineering storage abstractions**are **data organization and query patterns that sit at the heart of the DE lifecycle + are built atop the data storage systems discussed previously**
* The main types of abstractions we’ll concern ourselves with are those that support data science, analytics, + reporting use cases 🡪 data warehouse, data lake, data lakehouse, data platforms, + data catalogs
* The storage abstraction you require as a DE boils down to a few key considerations:
* **Purpose and use case**: 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
* 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 DW’s 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 + technically be very similar under the hood

##### The Data Warehouse

* **DWs 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 w/in a company**
* **In terms of storage trends, we’ve evolved from building DWs atop conventional transactional databases, row-based MPP systems** (e.g., Teradata and IBM Netezza), **+ columnar MPP systems** (e.g., Vertica and Teradata Columnar) **to cloud DWs and data platforms**
* **In practice, cloud DWs 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
* **Cloud DWs can handle massive amounts of raw text *and* complex JSON documents**
* The limitation is that **cloud DWs cannot handle *truly* unstructured data, such as images, video, or audio, unlike a *true* data lake.**
* **Cloud DWs can be coupled w/ 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 w/out the cost overhead of a proprietary MPP system.
* **The last 5 years (2018-2023) have seen 2 major developments in the evolution of data lake storage**
* **1) A major migration toward separation of compute and storagehas occurred**
* In practice, this means **a move away from Hadoop toward cloud object storage for long-term retention of data**
* **2)** DE’s discovered that **much of the functionality offered by MPP systems** (schema management; update, merge + delete capabilities) + 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 DW + 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 DW**
* This means **robust table and schema support + features for managing incremental updates and deletes**
* Lakehouses **typically also support table history and rollback** (accomplished by retaining old versions of files + metadata)
* **A lakehouse system is a metadata + file-management layer deployed w/ data management + transformation tools**
* Databricks has heavily promoted the lakehouse concept w/ Delta Lake, an OSS storage management system
* The **architecture of the data lakehouse is similar to the architecture used by various commercial data platforms, including BigQuery + Snowflake**
* These systems **store data in object storage + provide automated metadata management, table history, + 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 = overhead in processing, time, + cost**
* **In a data lakehouse architecture, various tools can connect to the metadata layer + 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 DW 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, DE’s must ensure the tools offered meet their needs**
* Tools not directly provided in the platform can still interoperate, w/ extra data overhead for data interchange
* Platforms also emphasize close integration with object storage for unstructured use cases, as mentioned w/ cloud DWs
* **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 DE + 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 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 + 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**, which is automatically reserialized into columnar object storage
* The query engine supports seamless querying of both the streaming buffer + 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 discuss some big ideas in storage (key considerations you need to keep in mind as you build out your storage architecture)
* Many of these considerations are part of larger trends
* Ex: **Data catalogs** fit under the trend toward “enterprise-y” DE + data management
* **Separation of compute + storage** = 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 w/ various systems + abstractions**
* **Typically work across operational *and* analytics data sources, integrate data lineage and presentation of data relationships, + allow user editing of data descriptions**
* **Often used to provide a central place where people can view their data, queries, + data storage**
* **As a DE, you’ll likely be responsible for setting up + maintaining the various data integrations of data pipeline + storage systems that will integrate w/ the data catalog + the integrity of the data catalog itself**

###### a) Catalog Application Integration

* ***Ideally*, data applications are designed to integrate w/ 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

###### b) 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, DW’s, + operational databases**
* Data catalogs can collect existing metadata + **may also use scanning tools to infer metadata such as key relationships or the presence of sensitive data**

###### c) Data Portal and Social Layer

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

###### d) Data Catalog Use Cases

* Data catalogs have both **organizational *and* technical use cases**
* **Data catalogs make metadata easily available to systems**
* Ex: A data catalog is a key ingredient of a data lakehouse, allowing table discoverability for queries
* *Organizationally*, data catalogs allow business users, analysts, data scientists, + engineers to search for data to answer questions
* Data catalogs streamline cross-organizational communications + collaboration

##### Data Sharing

* **Data sharing****allows organizations and individuals to share specific data + carefully defined permissions *with specific entities***
* Data sharing **allows data scientists to share data from a sandbox w/ their collaborators within an organization**
* Across organizations, data sharing **facilitates collaboration between partner businesses**
* Ex: 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 w/ 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

##### Schema

* ***What is the expected form of the data?***
* What is the file format? Is it structured, semi-structured, or unstructured? What data types are expected? How does the data fit into a larger hierarchy? Is it connected to other data through shared keys or other relationships?
* Note that **schema** **need *not* be *relational***
* Rather, **data becomes more useful when we have *as much information about its structure and organization as possible***
* For images stored in a data lake, this schema information might explain the image format, resolution, + the way the images fit into a larger hierarchy
* **Schema can function as** a sort of Rosetta stone, **instructions that tell us how to read the data**
* **2 major schema patterns exist**: **schema on write** and **schema on read**
* **Schema on write** is essentially **the traditional DW pattern, where a table has an integrated schema + *any writes to the table must conform***
* To support schema on write, a **data lake must integrate a schema metastore**
* With **schema on read**, the **schema is dynamically created when data is written, + a reader must determine the schema when reading the data**
* **Ideally**, schema on read is **implemented using file formats that implement built-in schema information, such as Parquet or JSON**
* **CSV files are notorious for schema inconsistency + are *NOT* recommended in this setting**
* The **principal advantage of schema on write** is that **it enforces data standards, making data easier to consume and utilize in the future**
* **Schema on read emphasizes flexibility, allowing virtually any data to be written**
* This **comes at the cost of greater difficulty consuming data in the future**

##### Separation of Compute from Storage

* A key idea in modern DE is the **separation of compute from storage**, which **has emerged as a standard data access and query pattern in today’s cloud era**
* **Data lakes, as discussed, store data in object stores and spin up *temporary* compute capacity to read + process it**
* **Most fully-managed OLAP products now rely on object storage behind the scenes**
* To understand the motivations for separating compute + storage, we should first look at the **colocation of compute and storage**

###### a) Colocation of Compute and Storage

* **Colocation of compute + storage has long been a standard method to improve database performance**
* For ***transactional databases*, data colocation allows fast, low latency disk reads + high bandwidth**
* Even when we virtualize storage (e.g., using Amazon EBS), data is located relatively close to the host machine
* The same basic idea applies for analytics query systems running across a cluster of machines
* Ex: W/ HDFS + MapReduce, the standard approach is to locate data blocks that need to be scanned in the cluster, and then push individual mapjobs out to these blocks
* The data scan + processing for the map step are strictly local
* The reducestep involves shuffling data across the cluster, but *keeping map steps local* effectively preserves more bandwidth for shuffling, delivering better overall performance
* Map steps that filter heavily also dramatically reduce the amount of data to be shuffled

###### b) Separation of Compute and Storage

* **If colocation of compute and storage delivers high performance, why the shift toward separation of compute and storage?**
* Several motivations exist:
* **Ephemerality and scalability**
* In the cloud, we’ve seen a dramatic shift toward **ephemerality**
* ***In general*, it’s cheaper to buy + host a server than to rent it from a cloud provider,** ***provided that you’re running it 24 hours a day nonstop for years on end***
* ***In practice,* workloads vary dramatically, + significant efficiencies are realized w/ a pay-as-you-go model if servers can scale up + down**
* This is true for web servers in online retail, and it is also true for big data batch jobs that may run only periodically
* **Ephemeral compute resources allow DEs to spin up massive clusters to complete jobs on time + then delete clusters when these jobs are done**
* The **performance benefits of temporarily operating at ultra-high scale can outweigh the bandwidth limitations of object storage**
* **Data durability and availability**
* **Cloud object stores significantly mitigate the risk of data loss + generally provide extremely high uptime (availability)**
* Ex: S3 stores data across multiple zones, + if a natural disaster destroys a zone, data is still available from the remaining zones
* Having multiple zones available also reduces the odds of a data outage
* If resources in a zone go down, DEs can spin up same resources in a different zone
* The **potential for a misconfiguration that destroys data in object storage is still somewhat scary, but simple-to-deploy mitigations are available**
* **Copying data to multiple cloud regions** reduces this risk since configuration changes are generally deployed to only one region at a time
* **Replicating data to multiple storage providers can further reduce the risk**
* **Hybrid separation and colocation**
* **Practical realities of separating compute from storage are more complicated than implied**
* **In practice, we constantly *hybridize* colocation and separation to realize the benefits of *both* approaches**
* This **hybridization is typically done in 2 ways**: **multitier caching** and **hybrid object storage**.
* With **multitier caching**, we **utilize object storage for long-term data retention + access but spin up local storage to be used during queries + various stages of data pipelines**
* Both Google + Amazon offer versions of hybrid object storage (object storage that is tightly integrated with compute)
* Let’s look at examples of how some popular processing engines hybridize separation and colocation of storage and compute.
* Example: AWS EMR with S3 and HDFS
* Big data services like Amazon EMR spin up temp HDFS clusters to process data
* DEs have the option of referencing *both* S3 and HDFS as a filesystem
* A common pattern is to stand up HDFS on SSD drives, pull from S3, + save data from intermediate processing steps on local HDFS
* Doing so can realize significant performance gains over processing directly from S3
* Full results are written back to S3 once the cluster completes its steps, + the cluster + HDFS are deleted
* Other consumers read the output data directly from S3
* Example: Apache Spark
* In practice, Spark generally runs jobs on HDFS or some other ephemeral distributed filesystem to support performant storage of data between processing steps
* Spark also relies heavily on in-memory data storage to improve processing
* The problem w/ owning the infrastructure for running Spark is that dynamic RAM (DRAM) is extremely expensive
* **By separating compute + storage in the cloud, we can rent large quantities of memory + then release that memory when the job completes**
* Example: Apache Druid
* Apache Druid relies heavily on SSDs to realize high performance
* Since SSDs are significantly more expensive than HDDs, Druid keeps only *one* copy of data in its cluster, reducing “live” storage costs by a factor of 3
* Of course, maintaining data durability is still critical, so Druid uses an object store as its durability layer
* When data is ingested, it’s processed, serialized into compressed columns, and written to cluster SSDs *and* object storage
* In the event of node failure or cluster data corruption, data can be automatically recovered to new nodes
* In addition, the cluster can be shut down + then fully recovered from SSD storage
* Example: **Hybrid object storage**
* Google’s Colossus file storage system supports fine-grained control of data block location, although this functionality is not exposed directly to the public
* BigQuery uses this feature to co-locate customer tables in a single location, allowing ultra-high bandwidth for queries in that location
* We refer to this as **hybrid object storage**b/c it **combines the clean abstractions of object storage w/ some advantages of co-locating compute + storage**
* Amazon also offers some notion of hybrid object storage through S3 Select, a feature that allows users to filter S3 data directly in S3 clusters before data is returned across the network
* We **speculate that public clouds will adopt hybrid object storage more widely to improve the performance of their offerings + make more efficient use of available network resources**
* *Some may be already doing so without disclosing this publicly*
* The **concept of hybrid object storage underscores that there can still be advantages to having low-level access to hardware rather than relying on someone else’s public cloud**
* Public cloud services do not expose low-level details of hardware + systems (e.g., data block locations for Colossus), but **these details can be extremely useful in performance optimization and enhancement**
* See a discussion of cloud economics in Chapter 4
* **While now seeing a mass migration of data to public clouds, many hyper-scale data service vendors that currently run on public clouds provided by other vendors may build their data centers in the future, albeit with deep network integration into public clouds**

###### c) Zero-copy cloning

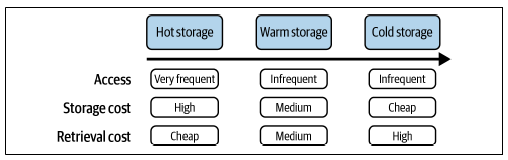
* Cloud-based systems based around object storage support **zero-copy cloning**, which **typically means that a new virtual copy of an object is created (e.g., a new table) w/out necessarily physically copying the underlying data**
* Typically, **new pointers are created to the raw data files, + future changes to these tables will not be recorded in the old table**
* For those familiar w/ the inner workings of OOP languages such as Python, this type of “shallow” copying is familiar from other contexts
* **Zero-copy cloning is a compelling feature, but DE’s must understand its strengths and limitations**
* Ex: Cloning an object in a data lake environment + then deleting the files in the original object might also wipe out the new object
* **For fully managed object-store-based systems (e.g., Snowflake and BigQuery), DE’s need to be extremely familiar w/ the exact limits of shallow copying**
* DE’s have more access to underlying object storage in data lake systems such as Databricks, which is a blessing and a curse
* DE’s should **exercise great caution before deleting any raw files in the underlying object store**
* Databricks + other data lake management technologies sometimes also support a notion of **deep copying**, whereby ***all* underlying data objects are copied**
* This is a **more expensive** process, but **also more robust in the event that files are unintentionally lost or deleted**

##### Data Storage Lifecycle and Data Retention

* **Storing data isn’t as simple as just saving it to object storage or disk + forgetting about it**
* **Need to think about the data storage lifecycle and data retention**
* **When thinking about access frequency and use cases, ask, “How important is the data to downstream users, + how often do they need to access it?”** 🡪 **the data storage lifecycle**
* **Another question you should ask is, “How long should I keep this data? Indefinitely, or am I fine discarding it past a certain time frame?”** 🡪 **data retention**

###### a) Hot, Warm, and Cold Data

* Did you know that data has a temperature?
* **Depending on how frequently data is accessed, we can roughly bucket the way it is stored into 3 categories of persistence: hot, warm, + cold**
* Query access patterns differ for each dataset



* **Typically, newer data is queried more often than older data**
* **Hot data**
* Hot datahas **instant or frequent access requirements**
* The **underlying storage** for hot data **is suited for fast access + reads, such as SSD or memory**
* B/c of the type of hardware involved with hot data, storing hot data is **often the most expensive form of storage**
* Example use cases for hot data include retrieving product recommendations + product page results
* **The cost of storing hot data is the highest of these 3 storage tiers, but retrieval is often inexpensive**
* **Query results cache** is another example of hot data
* **When a query is run, some query engines will persist the query results in the cache**
* For a limited time, when the *same* query is run, **instead of rerunning the same query against storage, the query results cache serves the cached results**
* Allows for much faster query response times vs. redundantly issuing the same query repeatedly
* **Warm data**
* Warm datais **accessed semi-regularly**, say, once per month
* **No hard + fast rules indicate how often warm data is accessed, but it’s less than hot data + more than cold data**
* Major cloud providers offer object storage tiers that accommodate warm data
* Ex: S3 offers an Infrequently Accessed Tier, + Google Cloud has a similar storage tier called Nearline
* Vendors give their models of recommended access frequency, + DE’s can also do their cost modeling + monitoring
* **Storage of warm data is cheaper than hot data, w/ slightly more expensive retrieval costs**
* **Cold data**
* On the other extreme, **cold data is infrequently accessed data**
* **Hardware used to archive cold data is typically cheap + durable, such as HDD, tape storage, + cloud-based archival systems**
* Cold data is **mainly meant for long-term archival**, when there’s **little to no intention to access the data**
* **Though storing cold data is cheap, retrieving cold data is often expensive**

###### b) Storage Tier Considerations

* When considering the storage tier for data, consider the costs of each tier
* If you store all data in hot storage, all data can be accessed quickly, but this comes at a tremendous price!
* Conversely, storing all data in cold storage to save on costs will certainly lower storage costs, but at the expense of prolonged retrieval times + high retrieval costs if you need to access data
* **The storage price goes down from faster/higher performing storage to lower storage**
* **Cold storage is popular for archiving data**
* Historically, cold storage involved physical backups + often mailing this data to a 3rd-party that would archive it in a literal vault
* **Cold storage is increasingly popular in the cloud**
* Every cloud vendor offers a cold data solution, **+ you should weigh the cost of pushing data into cold storage vs. the cost + time to retrieve the data**
* **DE’s need to account for spillover from hot to warm/cold storage**
* **Memory is expensive and finite**
* Ex: If hot data is stored in memory, it can be spilled to disk when there’s too much new data to store + not enough memory
* **Some databases may move infrequently accessed data to warm or cold tiers,** offloading the data to either HDD or object storage
* The latter is increasingly more common because of the cost-effectiveness of object storage
* **If in the cloud + using managed services, disk spillover will happen automatically**
* **If using cloud-based object storage, create automated lifecycle policies for your data**
* This **will drastically reduce your storage costs**
* Ex: If your data needs to be accessed only once a month, move the data to an infrequent access storage tier
* If your data is 180 days old + not accessed for current queries, move it to an archival storage tier
* In both cases, you can automate the migration of data away from regular object storage, + you’ll save money
* **That said, consider the retrieval costs (both in time and money) of using infrequent or archival style storage tiers.**
* Access, retrieval times, + costs may vary depending on the cloud provider
* Some cloud providers make it simple + cheap to migrate data into archive storage, but it is costly + slow to retrieve your data

###### c) Data retention

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

i) Value

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

ii) Time

* The **value to downstream users also depends upon the age of the data**
* **New data is typically more valuable + frequently accessed than older data**
* **Technical limitations may determine how long you can store data in certain storage tiers**
* Ex: If you store hot data in cache or memory, you’ll likely need to set a **time to live (TTL)**, so you can expire data after a certain point or persist it to warm or cold storage
* Otherwise, hot storage will become full, + queries against the hot data will suffer from performance lags

iii) Compliance

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

iv) Cost

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

##### Single-Tenant Vs. Multitenant Storage

* Chapter 3 covered the trade-offs between **single-tenant and multitenant architecture**
* To recap, w/ **single-tenant architecture, each group of tenants (e.g., individual users, groups of users, accounts, or customers) gets its own dedicated set of resources such as networking, compute, + storage**
* A **multitenant architecture** **inverts this and shares these resources among groups of users**
* Both architectures are widely used, + this section looks at the implications of single-tenant and multitenant *storage*
* **Adopting single-tenant storage means that every tenant gets their dedicated storage**
* Each tenant gets a database, no data is shared among these databases, + storage is totally isolated
* An example of using single-tenant storage is that each customer’s data *must* be stored in isolation and *cannot* be blended w/ any other customer’s data
* In this case, each customer gets their own database.
* **Separate data storage implies separate + independent schemas, bucket structures, + everything related to storage**
* This means you **have the liberty of designing each tenant’s storage environment to be uniform or let them evolve however they may**
* **Schema variation across customers can be an advantage and a complication, so as always, consider the trade-offs**
* **If each tenant’s schema isn’t uniform across all tenants, this has major consequences if you need to query multiple tenants’ tables to create a unified view of all tenant data**
* **Multitenant storage** **allows for the storage of *multiple tenants within a single database***
* Ex: Instead of the single-tenant scenario where customers get their own database, multiple customers may reside in the same database schemas or tables in a multitenant database
* **Storing multitenant data means each tenant’s data is stored in the same place**
* You **need to be aware of querying both single and multitenant storage** (see Chapter 8)

#### Whom You’ll Work With

* **Storage is at the heart of the DE infrastructure**
* You’ll **interact w/ the people who own your IT infrastructure** (typically, DevOps, security, + cloud architects)
* **Defining domains of responsibility between DE and other teams is critical**
* Do DE’s have the authority to deploy their infrastructure in an AWS account, or must another team handle these changes?
* **Work w/ other teams to define streamlined processes so that teams can work together efficiently + quickly**
* The **division of responsibilities for data storage will depend significantly on the maturity of the organization involved**
* The DE will likely manage the storage systems + workflow if the company is early in its data maturity
* If the company is later in its data maturity, the DE will probably manage a *section* of the storage system
* This DE will also likely interact with engineers on either side of storage (ingestion + transformation)
* **The DE needs to ensure the storage systems used by downstream users are securely available, contain high-quality data, have ample storage capacity, + perform when queries and transformations are run**

#### Undercurrents

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

##### Security

* While DE’s often view security as an impediment to their work, they should **embrace the idea that security is a key enabler**
* **Robust security at rest + in motion w/ fine-grained data access control allows data to be shared and consumed more widely w/in a business**
* The value of data goes up significantly when this is possible
* As always, **exercise the principle of least privilege**
* **Don’t give full database access to anyone unless required**
* This means **most DE’s don’t need full database access in practice**
* Also, pay attention to the **column, row, and cell-level access controls** in your database
* **Give users only the information they need and no more**

##### Data Management

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

###### i) Data Catalogs and Metadata Management

* **Data is enhanced by robust metadata**
* **Cataloging** enables data scientists, analysts, + MLE’s by **enabling data discovery**
* **Data lineage accelerates the time to track down data problems + allows consumers to locate upstream raw sources**
* **As you build out your storage systems, invest in your metadata**
* **Integration** of a **data dictionary** **w/ these other tools allows users to share + record institutional knowledge robustly**
* Metadata management also significantly **enhances data governance**
* Beyond simply enabling *passive* data cataloging + lineage, **consider implementing analytics over these systems to get a clear, active picture** of what’s happening w/ your data

###### ii)Data Versioning in Object Storage

* Major cloud object storage systems enable **data versioning**, which **can help w/ error recovery when processes fail, + data becomes corrupted**
* Versioning is **also beneficial for tracking the history of datasets used to build models**
* Just as code version control allows developers to track down commits that cause bugs, data version control can aid MLE’s in tracking changes that lead to model performance degradation

###### iii) Privacy

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

##### DataOps

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

###### i) Systems Monitoring

* **DE’s must monitor storage in a variety of ways**
* Includes monitoring infrastructure storage components, where they exist, but also monitoring object storage and other “serverless” systems
* **DE’s should take the lead on FinOps (cost management), security monitoring, + access monitoring**

###### ii) Observing and Monitoring Data

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

##### Data Architecture

* Chapter 3 covers the basics of data architecture, as **storage is the critical underbelly of the DE lifecycle**
* Consider the following data architecture tips
* **Design for required reliability + durability**
* **Understand the upstream source systems** + how that **data, once ingested, will be stored + accessed**
* **Understand the types of data models + queries that will occur downstream.**
* If data is expected to grow, can you negotiate storage w/ your cloud provider?
* **Take an active approach to FinOps, and treat it as a central part of architecture conversations**
* Don’t prematurely optimize, but prepare for scale if business opportunities exist in operating on large data volumes
* **Lean toward fully managed systems, + understand provider SLAs**
* **Fully managed systems are generally far more robust + scalable than systems you have to babysit**

##### Orchestration

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

##### Software Engineering

* Can think about **SWE in the context of storage in two ways**
* 1) The **code** you write **should perform well w/ your storage system**
* Make sure your code stores the data correctly + doesn’t accidentally cause data, memory leaks, or performance issues
* **2) Define your storage infrastructure *as code* + use ephemeral compute resources when it’s time to process your data**
* B/c **storage is increasingly distinct from compute**, you **can automatically spin resources up and down while keeping your data in object storage**
* This **keeps your infrastructure clean and avoids coupling storage and query layers**

#### Conclusion

* Storage is everywhere and underlays many stages of the DE lifecycle
* We learned about the raw ingredients, types, abstractions, + big ideas around storage systems
* **Gain deep knowledge of the inner workings + limitations of the storage systems you’ll use**
* **Know the types of data, activities, and workloads appropriate for your storage**

#### Additional Resources

* “Column-Oriented DBMS” Wikipedia page
* “The Design + Implementation of Modern Column-Oriented Database Systems” by Daniel Abadi et al
* **Designing Data-Intensive Applicationsby Martin Kleppmann (O’Reilly)**
* “Diving Into Delta Lake: Schema Enforcement and Evolution” by Burak Yavuz et al.
* “Hot Data vs. Cold Data: Why It Matters” by Afzaal Ahmad Zeeshan
* IDC’s “Data Creation and Replication Will Grow at a Faster Rate than Installed
* Storage Capacity, According to the IDC Global DataSphere and StorageSphere Forecasts” press release
* **“Row-wise vs. Columnar Database? Theory and in Practice” by Mangat Rai Modi**
* **“Snowflake Solution Anti-Patterns: The Probable Data Scientist” by John Aven**
* “What Is a Vector Database?” by Bryan Turriff
* **“What Is Object Storage? A Definition and Overview” by Alex Chan**
* **“The What, When, Why, and How of Incremental Loads” by Tim Mitchell**