Building Data Lakes Successfully — Part 1 — Architecture, Ingestion, Storage and Processing

Published 7 October 2020 - ID G00723117 - 62 min read

By Analysts Sumit Pal

Initiatives: Data Management Solutions for Technical Professionals

Data lakes provide a flexible data ingestion, storage and processing platform but building a data lake is challenging. This research helps data and analytics technical professionals circumvent challenges and align use cases with technologies and architectural best practices.

Overview

Key Findings

- Enterprises are opting to build data lakes and leverage their data assets without thorough consideration to appropriate business use cases. This is often the main reason for data lakes turning into data swamps.
- Organizations underestimate both the technical complexities of developing and maintaining a data lake and the associated expertise needed to overcome those complexities, resulting in both a brittle environment that takes longer to solidify and higher costs.
- Organizations are thinking of using data lakes as a replacement for data warehouses. But data lakes are complementary to data warehouses; they do not replace them. Data warehouses are for questions, whereas data lakes are for when organizations don't know what questions to ask.
- Organizations that are successfully building and leveraging data lakes are employing them for use cases such as offloading extraction, transformation and loading (ETL) processing, data discovery/exploration by data scientists, stream processing, and storing and enriching unstructured and multistructured datasets.

Recommendations

Data and analytics technical professionals responsible for building data lakes should consider the following:

- "Begin with the end in mind." Improve the chances of success by building data lakes iteratively for the specific requirements of certain business groups, sets of users or analytics use cases, rather than taking a "big bang," enterprisewide approach.
- Learn lessons from data warehouse experiences. Do not build a data lake hoping that the enterprise will figure out how to use it. Build proof of value rather than proof of concept before

jumping into data lake implementations. Organizations with good engineering skills and the right use cases have implemented data lakes successfully. With data lakes, it is very easy to get into a “technology tail-chase trail.” Avoid that and stay focused on business outcomes from the data lake.

- Avoid doing ad hoc ingestion or data processing on a data lake. Build a framework or leverage third-party vendors to provide self-service-driven ingest, storage provisioning and processing.

Analysis

Organizations attempting to gain control over data that resides in multiple formats and locations across the enterprise are drowning in the data deluge. They are realizing that traditional technologies can’t meet all their new business needs, causing many to invest in a scale-out data lake architectural pattern. With the complexities of data orchestration, data discovery, data preparation and renewed focus on new regulations, privacy and governance requirements, data management is getting harder.

Organizations are investing massive amounts of effort to:

- Build and architect data lakes in response to today’s data challenges.
- Maintain, organize and manage the varied data resources.

Data lakes are built on distributed architecture at massive scale, with the ability to ingest, store and process structured, semistructured, unstructured or multistructured datasets.

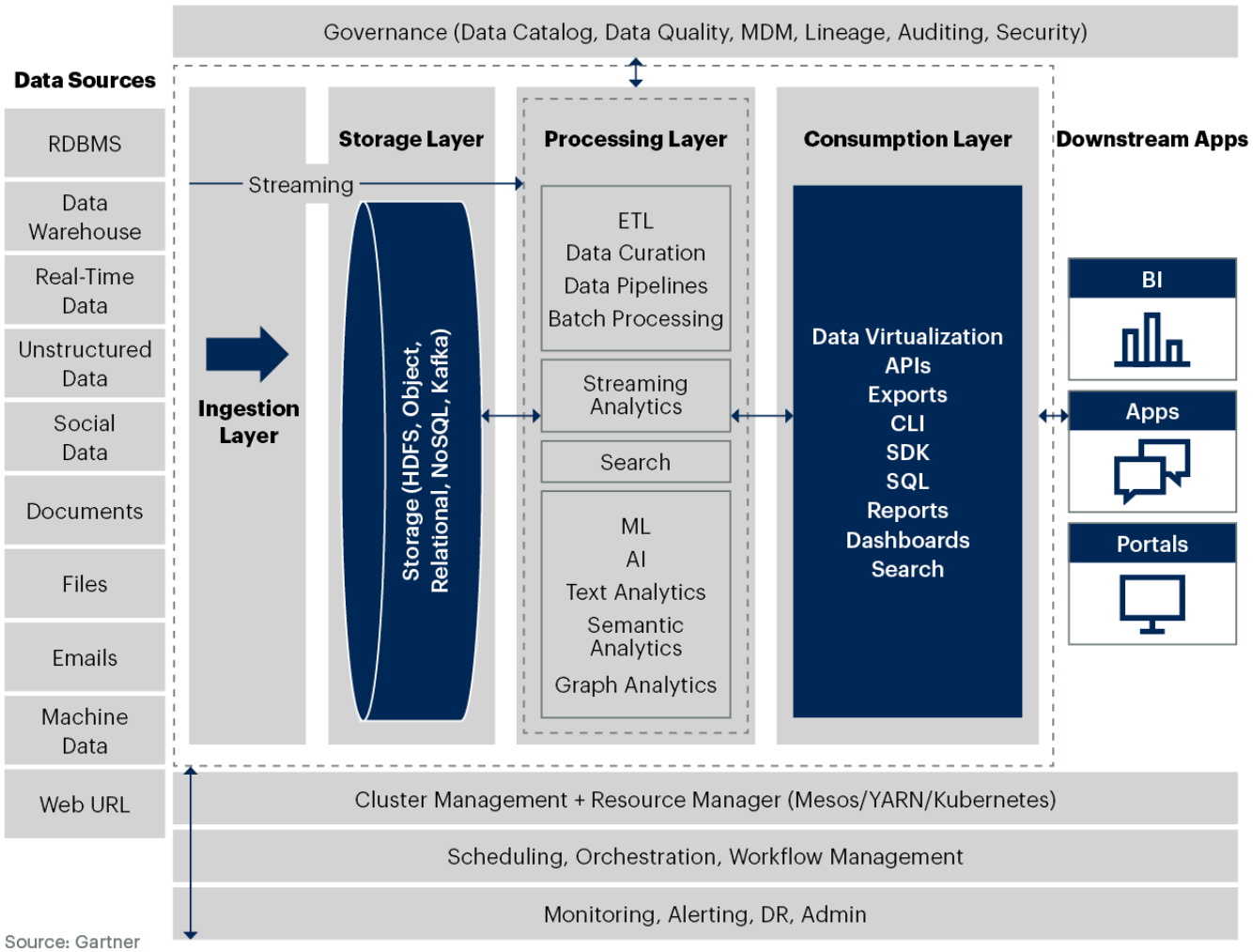
Data lakes need multiple components, frameworks and products from different vendors to work in tandem. Hence, enterprises need to architect data lakes with upfront planning and correct architecture. This is because these potentially critical business resources could be underutilized due to lacking governance and security, an inability to find the data, and redundant and poor-quality data. The data lake stack is complex and requires critical decisions around data ingestion, storage, processing, consumption and governance.

Architecture

Data lakes are complex, and building them successfully involves integrating different technologies because there is no one end-to-end product in the market to implement enterprisewide data lakes. A high-level data lake architecture, along with the key functional capabilities and major components needed to build a data lake, is shown in Figure 1.

Figure 1: Data Lake Reference Architecture

Data Lake Reference Architecture



Source: Gartner
723117_C

This document will cover the data ingestion, data storage and data processing sections of the data lake reference architecture. Please refer to Building Data Lakes Successfully — Part 2 — Consumption, Governance and Operationalization for details on data governance, operationalization and data consumption in a data lake.

When architecting a data lake, some of the key capabilities around ingestion, storage and processing are listed in Table 1.

Table 1: Key Capabilities Around Ingestion, Storage and Processing

Capability ↓	*Description* ↓

Capability	Description
Data Ingest	■ Provide flexibility to ingest diverse data formats and types into a data lake ■ Make new data sources, both static and streaming, available for ingestion and define schema during ingest
Data Storage	■ Accommodate a variety of data storage platforms based on volume and variety of data types with different read and write patterns across multiple hardware types
Data Processing — Batch/Streaming/Analytic Workloads	■ Provide multiple ways of processing the variety of data types ingested and stored in the data lake for analytics and building data-driven applications

Source: Gartner (October 2020)

Most organizations need to answer some key architectural and deployment questions before building a data lake. Should they:

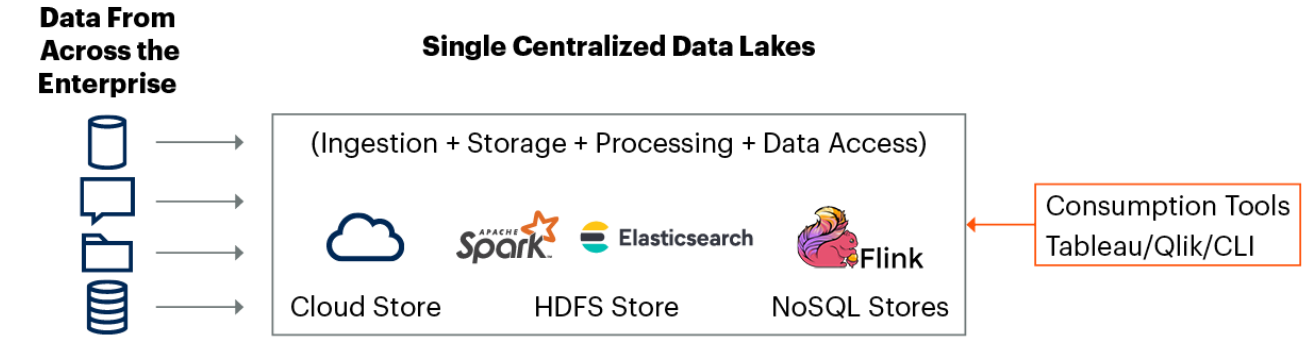
- Build a single centralized data lake for the entire organization?
- Build multiple data lakes that are localized to the data centers of the organization?
- Build an on-premises or cloud-based data lake or a hybrid or multicloud-based data lake?

These approaches to building a data lake are discussed below, along with their advantages and disadvantages.

Figure 2 shows a single centralized enterprise wide data lake.

Figure 2: Single Centralized Organizationwide Data Lake

Single Centralized Organizationwide Data Lake



Source: Gartner
723117_C

Advantages

- A single centralized data lake is easier to manage and govern compared to multiple decentralized lakes because everything, including governance, is in one place and there's a unified set of tools.
- All data is in one single location, which is easier for data access and for dealing with bandwidth constraints, and provides a single source of truth.
- Costs can be controlled around the proliferation of tools, storage and processing being centralized and more controlled.

Disadvantages

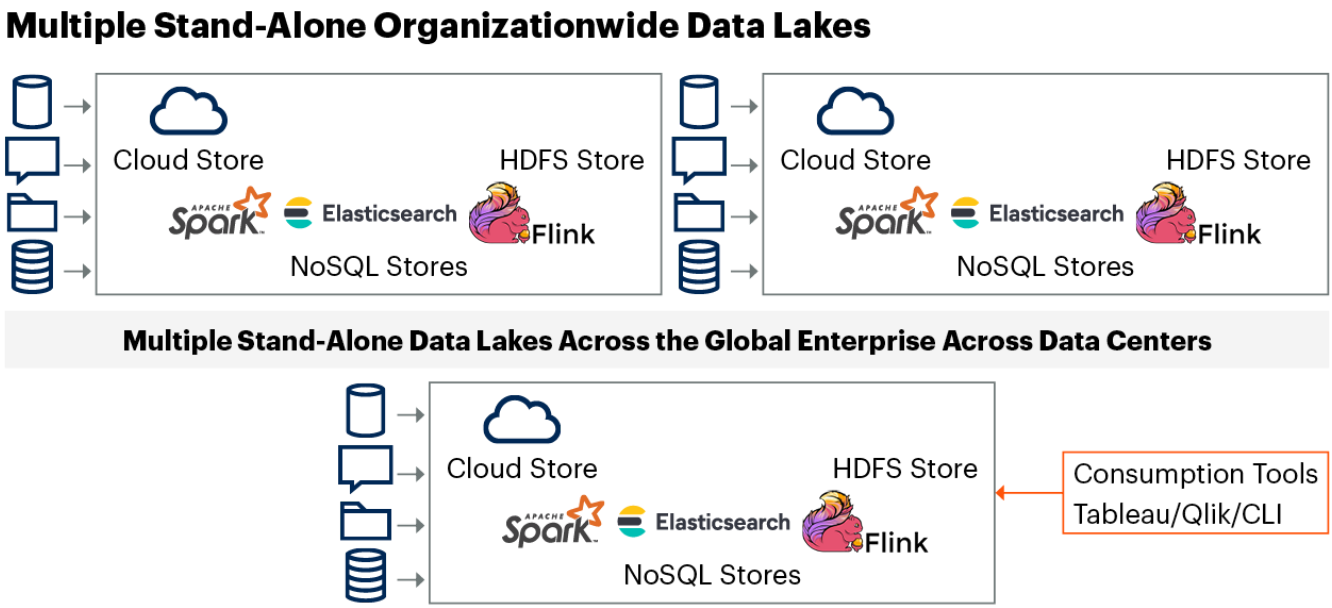
This approach, though simple, may not be practical from a global enterprise's perspective. Some of the problems associated with such an architectural model include:

- Conflicting legislation/sovereignty on data residency, movement and access can hamper usability.
- Very high network capacity and speed is required to move everything to one place.
- There can be a single point of failure and a security risk.
- Creating another copy of the data in the data lake creates a data silo.
- Creating another copy increases the chances of data divergence, and syncing needs to be properly coordinated.

- A new copy of the data increases the surface area of data breaches unless the data is correctly governed.

An alternative approach is to build multiple stand-alone data lakes within the enterprise, with each localized to a data center. Figure 3 shows the architecture of such a data lake.

Figure 3: Multiple Stand-Alone Organizationwide Data Lakes



Source: Gartner
723117_C

Advantages

- Data that is too big to move across the network into a single location can be handled within a local data lake.
- Legislation and regulations can be accommodated across geographical jurisdictions.

Disadvantages

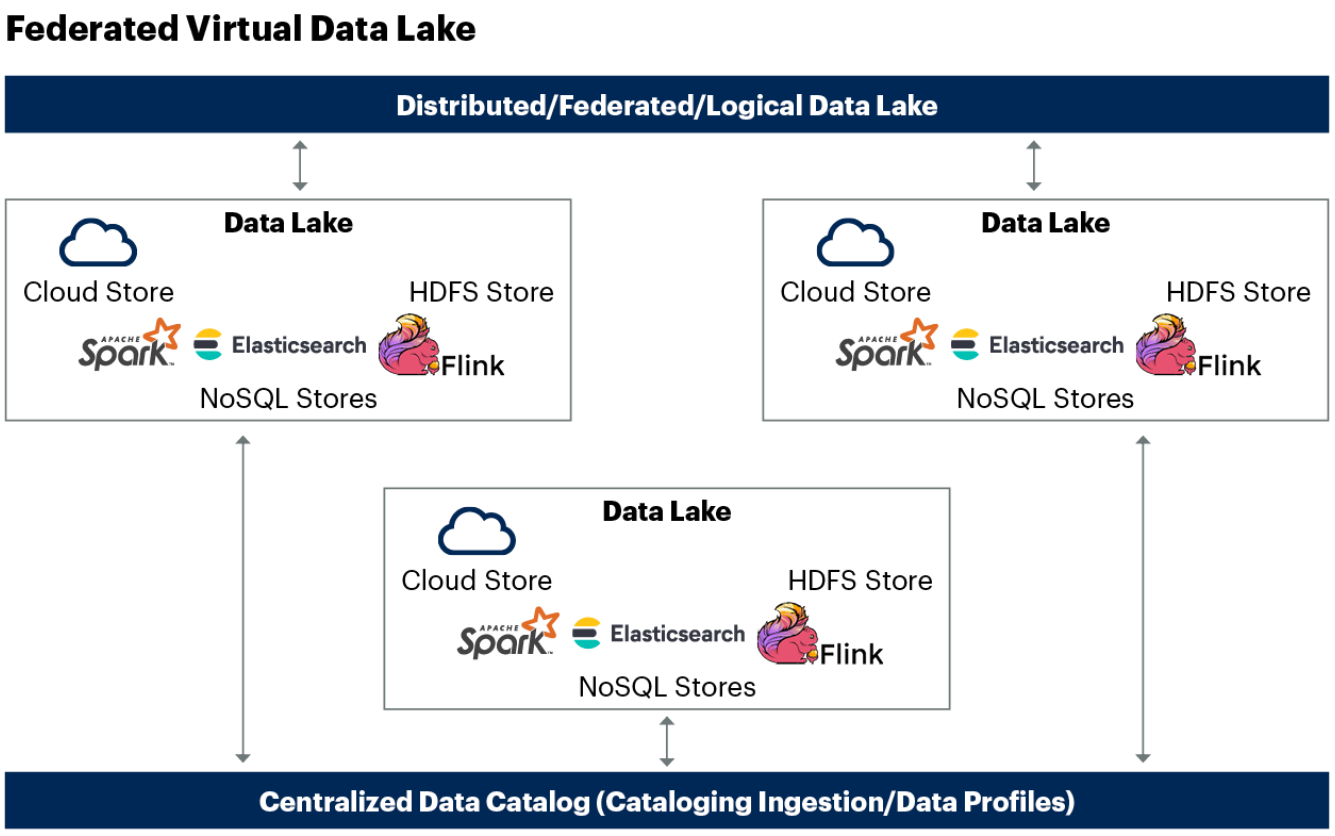
- This approach creates multiple silos of data lakes.
- Inconsistencies of data governance policies and tools can occur.
- Synchronization of common data across multiple data lakes can be very difficult.
- Multiple heterogeneous technologies may exist across different data lakes.
- Costs can be high due to network bandwidth requirements for data movement.
- Data access can be challenging.

Another architectural approach gaining traction and usage across large organizations is to build virtual/federated/logical data lakes.

The idea is to present siloed data lakes from across the organization as a single data lake while managing the architectural details and capabilities separately for each individual data lake. In order to accomplish this sort of a data lake architecture, organizations need support from data federation tools as well as enterprisewide catalogs that can span across data lakes, whether they are deployed on-premises or in the cloud. The catalog becomes the interface for searching and finding data within the data lake, and the data federation maps the request to whichever data lake is appropriate. Figure 4 shows the high-level architecture of a federated data lake across an organization.

Data lakes can also be part of a large logical data warehouse (LDW; see Adopt the Logical Data Warehouse Architecture to Meet Your Modern Analytical Needs).

Figure 4: Federated Virtual Data Lake



Source: Gartner
723117_C

Advantages

- Data movement problems associated with centralized data lakes are avoided.
- Data access and governance is easier when localized data is being processed and consumed.

Disadvantages

- Building such federated approaches for data lakes can be quite challenging.
- Data catalogs to help build such federated data lakes are still not widely available or capable.

Hybrid Data Lakes

There is growing concern among organizations regarding where to build their data lakes. Most greenfield data lake solutions are being built in the cloud. As much as the cloud vendors are encouraging customers to move all their data to the cloud, that is not likely to happen, especially as concerns about vendor lock-in continue and data gravity remains on-premises. This divide between on-premises and cloud data silos isn't going to go away any time soon.

The decoupled compute and storage architecture is becoming increasingly attractive, as it enables organizations to scale storage capacity independently to match the growth rate of compute, which reduces both capital expenditures and operating expenses. However, decoupled architecture introduces performance loss for certain types of workloads.

Vendors and organizations are innovating to prevent creation of data silos by exposing different views of the same data to different processing frameworks. The idea is to expose the same datasets to SQL engines, data processing and deep learning frameworks without moving the data.

Storage is getting cheaper, easier, bigger and better. Ten years ago, Hadoop Distributed File System (HDFS) made sense — with tight coupling between data and compute. Now the network is faster than the disk, and this has driven surges in decoupled compute and storage with organizations looking to leverage the cloud with object stores as the primary data storage.

Moving to the cloud using the "lift and shift" approach is rarely recommended. Cloud-based data lakes should take advantage of the cloud native architecture, elasticity and hardware fluidity. Cloud-based data lakes are good for agility and time to market, and they're suited for ephemeral workloads. The on-premises model provides a more predictable cost structure and is suited for long-running workloads.

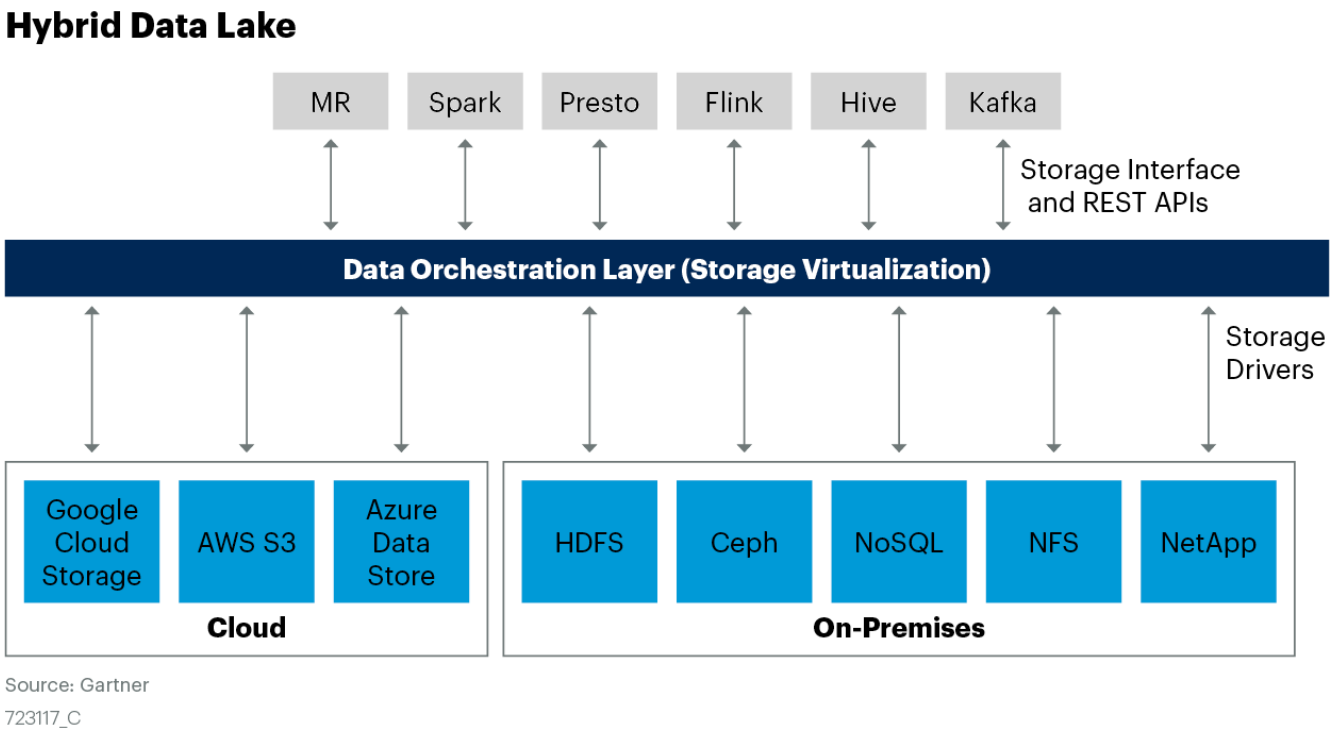
It may be worthwhile for organizations to rethink the philosophy of going all out and moving all data into cloud storage. Instead, it may be advisable to build applications with an API that could work with any underlying storage and embrace data silos wherever they reside.

Workloads can be migrated to the public cloud on demand without moving data between computing environments first, by bringing data to applications on demand. For example, Alluxio, a data orchestration platform architected as an in-memory file system, provides a "zero-copy" approach, where organizations can burst workloads to the cloud without copying the data. It provides an abstraction layer to the data sources, enabling them to read and write data without knowing where data is stored. It pulls the data only when the processing requests it and allows organizations to define pre-fetch policies and pin the loaded data into a tier-based in-memory file

system. It also provides a REST-based interface for moving data among on-premises, cloud or combination systems. And it simplifies the process of mounting object stores, HDFS and other storage repositories.

Figure 5 shows a high-level architecture of such a hybrid data lake.

Figure 5: Hybrid Data Lake



Source: Gartner
723117_C

Questions to Ask

Data lake architects and data engineers should ask the following questions when architecting a data lake:

- What are the workload processing requirements?

The choice of components for handling different types of workloads varies. Batch workloads need tools like Apache's Sqoop, Spark and Hive for ingestion and data processing, while streaming workloads will need tools like Apache's NiFi, Kafka and Flink for ingestion and data processing.

- What are the nonfunctional requirements (NFRs) for the data lake?

Understanding the NFRs for a data lake is extremely important in determining the overall architecture, selection of tools, frameworks, hardware and capacity planning for the data lake. Each organization and business should answer these questions, which are specific to their business, data storage and processing needs:

- What are the scaling requirements across compute, storage, networking or input/output (I/O)?
- What are the availability and recoverability requirements?
- What kind of compute and storage fault tolerance is desired?
- What are the performance metrics across workloads — SLAs/throughputs and latency?
- What are the disaster recovery requirements?
- What are the data replication requirements?
- What are the storage requirements based on throughput, latency and volume?
- What are the read/write access and patterns?
- What is the impact of data loss for the organization?

This research helps answer these questions by looking into best practices for establishing successful data lakes.

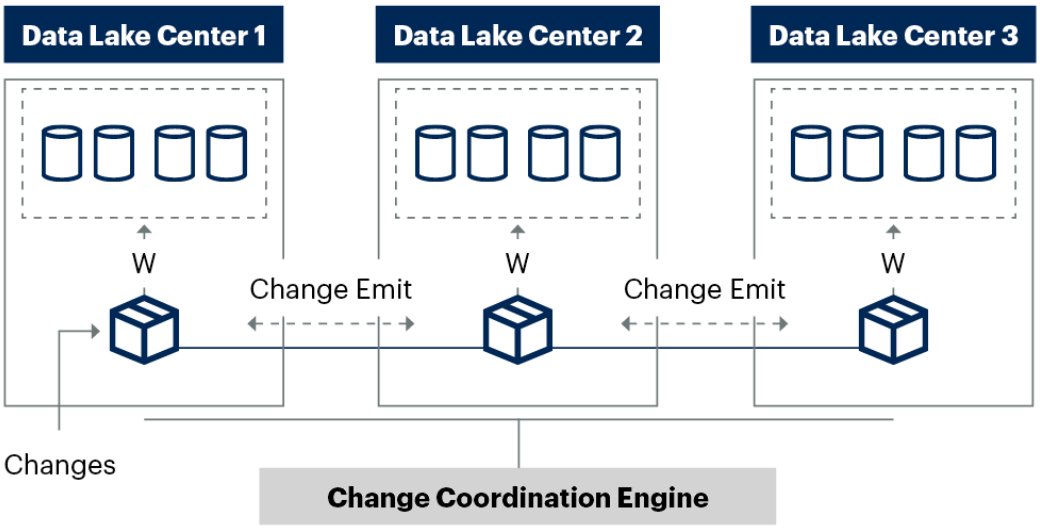
Best Practices

Some of the architectural best practices for data lakes are outlined below.

- For a highly available data lake, ensure each component within the data lake is highly available. For example, if you are using Kerberos distribution center, make sure it is set up for high availability (HA), or if you are using Hive metastore, make sure it is set up for HA (standby metastore with replication). If you are using Apache Hadoop-based components in a data lake, then use HDFS 2.0 to make NameNode Highly Available and scale using federated NameNode architecture.
- Avoid tight coupling of components, decouple components with a layer of abstraction and program to interfaces.
- For data lakes spanning across data centers, ensure active-active replication is set up, as shown in Figure 6, with tools and vendors like WANdisco.

Figure 6: Active-Active Replication Across Data Lakes

Active-Active Replication Across Data Lakes



Source: Gartner
723117_C

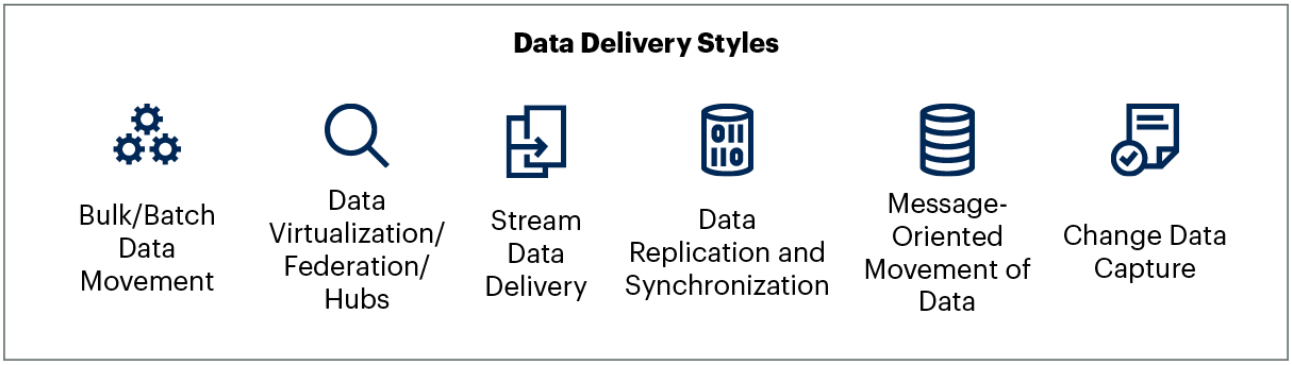
Ingestion

Before the data lake becomes useful for data analysis, the data lake needs to connect to various data sources and acquire data from them. The ingestion layer in the data lake enables the functionalities to achieve this. Data ingestion lays the foundation for data extraction from source data systems and the orchestration of different ingestion strategies in a data lake.

Figure 7 shows the different ways data can be delivered into a data lake.

Figure 7: Data Delivery Techniques

Data Delivery Techniques



Source: Gartner
723117_C

Table 2 lists some of the major considerations before setting up data ingestion.

Table 2: Considerations Before Data Ingestion

Category	Type	Description
Technology	Schema	Identify the source and the associated schema. Have tools in place to identify data drift and schema drift.
Technology	Security	Classify the data and perform personally identifiable information (PII) data masking, data tokenization and data encryption.
Technology	Governance — metadata and lineage	Capture metadata at ingest and also have hooks in place to capture data lineage.
Organization	Owner	Define the owner of the data at the source as well as the data lake.
Organization	Steward	Define the privacy, regulatory and compliance requirements of the ingested data.
Process	Monitoring	Ensure proper monitoring and alerting is in place to detect failures and problems during ingestion.
Process	Orchestration	Ensure proper workflows and scheduling is in place for timely data ingestion.

Source: Gartner (October 2020)

Data catalogs can help define some of the above requirements. Building a data ingestion strategy requires a clear understanding of source systems and SLAs expected from the ingestion framework. Ingestion mode can be batch, real-time or a change data capture (CDC) — depending on the data transfer granularity and cadence:

- **Batch-Based** — Use batch-based ingestion techniques when throughput is more critical for your business decisions. Use batch-based techniques when SLAs are in the hours or days range. When using batch-based ingestion, pay particular attention to the batch size or batch granularity and the batch cadence. Often, batch size is an important criterion for tuning and performance optimization of batch ingestion jobs.

- **Real-Time** — Use real-time ingestion techniques when latency is more critical for your business decisions than throughput. Organizations typically use real-time ingestion when SLAs are in the minutes or less range. Handling backpressure, data loss and the ability to autoscale based on incoming data flow are important when designing real-time data ingests.
- **CDC** — CDC has a few fundamental advantages over batch-based replication techniques, as it:

- Enables faster and accurate decisions based on more recent data.
- Minimizes disruptions to production workloads by reading the logs instead of the operational tables.
- Reduces costs and network bandwidth consumption because only incremental changes are transmitted.

Data Migration From On-Premises to Cloud

For organizations moving to the cloud data lake from on-premises deployments, there are multiple challenges to be resolved. The growing divide between on-premises and cloud data silos isn't going to go away soon. Some of these challenges include:

- Keeping data synchronized, which is one of the biggest challenges faced by hybrid solutions. Challenges of synchronizing data are a way to identify when data has changed and a mechanism to propagate changes to the corresponding copies.
- Selecting the right tools for the data movement.

Some considerations and best practices when migrating data from on-premises to cloud include:

- Always make copies of data read-only if possible.
- Avoid maintaining more than two copies of data. Keep only one copy on-premises and another only in the cloud.
- Leverage tools like Apache Airflow and Oozie to help manage your data synchronization.
- If a dataset is accessed in both environments, organizations need to establish a primary storage location for it in one environment and maintain a synchronized copy in the other.
- Develop a strategy to resolve conflicts.
- When large datasets are transferred to the cloud, ensure data always goes through a dedicated network line with a certain speed from the organization data center to the cloud data center.

Choices for Tool Selection for Data Movement

- Leverage cloud-provider native tools such as:
  - AWS: AWS Storage Gateway and AWS DataSync
  - Azure: Azure Storage Explorer, AzCopy, Azure PowerShell, Azure CLI and ExpressRoute
  - Azure Data Box Edge and Data Box Gateway
  - Google Cloud: Dataproc Cluster with DistCp, gsutil, Google Cloud Storage connector and Google Cloud Data Fusion
- Use framework-specific tools such as DistCp or Brooklin.
- Leverage commercial third-party vendor tools or managed services — NetApp/DataSync/WANdisco (LiveMigrator) — to move on-premises file system data to object storage in a public cloud in an automated way.
- Ship data offline for PB-sized data migrations using cloud-native tools.
- Adopt a hybrid data lake approach.

Performance Considerations When Using Apache Hadoop DistCp

- Increase the number of mappers, as DistCp's lowest granularity is a single file.
- Use more than one DistCp job.
- Consider splitting files. For example, split data by time stamp, jobs or ownership.
- Use the "strategy" command-line parameter.
- Increase the number of threads.
- Use the output committer algorithm.

Questions to Ask

Below are some of the questions data lake architects should ask before architecting and building the data lake and choosing the components, frameworks and products for ingesting data into the data lake:

- What style of ingestion is required?

Organizations should understand the type of ingestion required for their data lakes, whether it is batch-based ingestion, real-time data ingestion or incremental, CDC-based ingestion required to populate the data lake. It is possible that an organization requires all these kinds of ingestion

capabilities. Tools and framework selection should be based on the ingestion types because not all ingestion tools support all types of ingestion capabilities.

- What data formats and type of data should be supported for ingestion?

Understanding the type of data that is being ingested is important because not all tools can support all types of data ingestion. Some tools, for example, cannot support ingested binary data like video and audio, and some tools can work only with structured datasets.

- What is the data velocity in terms of throughput and volume?

Setup, configuration and capacity planning of the ingestion framework and cluster and scalability of the ingestion tool should be guided by the above requirements.

- What is the ingestion frequency?

Orchestration and workflow handling of the ingestion should be tied to the frequency. Ingestion processes should be deployed with the right scale to ensure each ingestion batch is properly sized for throughput.

- What is the source location of the source data systems and the location of the data lake?

Determine whether the data ingest will happen on-premises, from on-premises to cloud, from cloud to cloud, or from cloud to on-premises. Understanding the origin and destination for the data to be ingested is important for choosing the right toolsets, designing the right network bandwidth and architecting failure-handling mechanisms. This is an important question to answer in order to select the right framework, tools and bandwidth requirements for data ingestion.

- How will you handle data ingestion failures and interruptions? There are different types of failures and errors:
  - Wrong data
  - Wrong schema
  - Hardware failure/software failure
  - Handling exception records incorrectly
  - Not detecting file integrity (Is the file complete? Was it tampered with in transit?)
  - Compromising data integrity (Is the data as expected? Are all fields present? Is there any data drift?)

Each organization should define policies and rules to handle these failures and errors. These can be handled at the framework level or hooked into the orchestration engine to schedule data ingestion. The Details section of this research deals with ingestion and outlines an architectural approach to handle these situations:

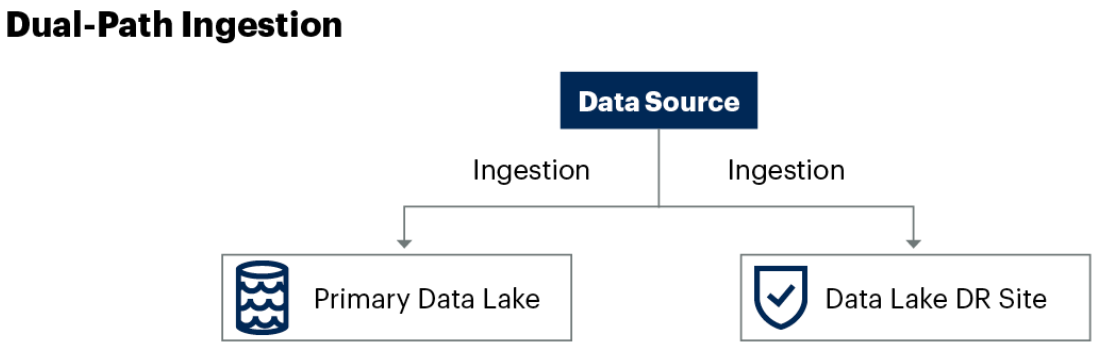
- **Scheduling** — How often does the data arrive? Was the data received on time? Does data delivery require acknowledgment?
- **Schema management** — Is there an associated schema with the data? Can the schema be inferred or, if not, how is the schema communicated and verified? How does the ingestion framework handle schema evolution?

Best Practices

- The ingestion framework should be able to handle additions and upgrades to existing and new data sources across a variety of data types and data velocity, such as streaming video and audio.
- For maintenance, it should be easy to add new ingestion jobs and to update, modify, debug, troubleshoot and provide traceability for an already running job.
- The ingestion framework must be scalable on demand based on varying load conditions. It should be able to scale on volume and velocity to minimize latency and to improve throughput to maintain the SLA.
- The ingestion framework must be capable of being fault-tolerant with fail-safety (recovery) as well as failover (resiliency) support.
- Ideally, the framework should support multithread and multievent execution.
- It should be able to quickly transform the acquired data structure into the target data formats as needed by the downstream processing layers.
- It should be capable of merging small files into larger files to reduce metadata management problems and enable faster downstream data processing.
- Avoid data lake indigestion by fixing data quality at ingestion time and by continuously monitoring data quality as data ingests into the data lake.
- Architect your ingestion framework to handle changes to incoming data — for example, fields getting reordered. The framework should be able to detect these kinds of changes and self-correct or issue alerts and notifications.
- You should have a clear strategy to handle bad and corrupt data during ingestion.

- Integrate data ingestion with the data catalog to ensure data quality, data discovery and data classification at ingest time.
- Ensure automation of ingestion workflow and orchestration and scheduling of ingest workloads.
- Handle security at the source and set up handling PII on data that lands within the raw zone.
- Ensure the ingestion framework can throttle the source or apply backpressure when the data lake components are not able to keep up with the data flow.
- When architecting the ingestion layer, take into account the problems associated with distributed computing, like networks, which can be unreliable from data loss and data corruption perspectives.
- The framework should be integrated with the data catalog and able to detect and gracefully handle data and schema drifts.
- Minimize dependency on the data engineers when onboarding new datasets, preventing the nonavailability of data engineers from becoming a blocker.
- If economics permit, ensure dual-path ingestion, as shown in Figure 8.

Figure 8: Dual-Path Ingestion



Source: Gartner
723117_C

Storage

Data and cloud storage follow Moore's law: the world's data doubles every two years, while the cost of storing that data declines at roughly the same rate. As organizations churn away, data accumulates quickly en masse as a byproduct of routine tasks.

Following data ingestion, the data needs to be stored. Data storage within the data lake is the most critical component. Data storage choices and architecture determine the performance, scalability and accessibility of the data in a data lake. With myriad complex data types, growing data

volumes and data variety, there is no single storage platform or choice that can suffice all storage requirements across different selection criteria.

Data lakes will typically have storage across any or many of these types of data stores:

- NoSQL data stores
- Relational data stores
- Streaming message/event stores
- New SQL data stores
- In-memory data stores
- Distributed file systems
- Object data stores

For detailed coverage of these data stores, see the following Gartner research:

- Assessing the Optimal Data Stores for Modern Architectures
- An Introduction to Graph Data Stores and Applicable Use Cases
- Time Series Database Architectures and Use Cases

Depending on the consumption use cases, either the existing storage can be used or data can be moved to another form of storage like NoSQL data stores based on use cases and read and write patterns. Most organizations use HDFS — on-premises and cloud or object store on cloud, for storing the data postingestion and postprocessing. Object stores such as Amazon Simple Storage Service (Amazon S3) and Google Cloud Storage are overtaking HDFS deployments rapidly. Data lake architects must understand the nuances of replacing a file system with an object store. Object stores have a different architecture than file systems. Patterns developed around HDFS primitives may not translate one-to-one when moving into an object store. Here are some of things to be careful about when working with object stores:

Some advantages of object stores:

- **Lower costs:** Object stores use a pay-for-what-is-used model instead of buying or provisioning hardware upfront and estimating storage requirements.
- **Decoupled compute and storage:** Data in an object store can be assessed from any other cluster, which makes it easier to tear down and set up new clusters with no data movement.

- **Interoperability:** Because of the decoupling, object stores enable interoperability between different processing engines and data stores.
- **HA:** Object stores are architected to be highly available and globally replicated. They are not vulnerable to NameNode as a single point of failure.
- **Management overhead:** Object stores require very minimal maintenance regarding upgrades or rollbacks to previous versions of the file system and other routine administrative tasks.

Some the object store issues include:

- An object store has a higher I/O variance than HDFS. When applications or data stores need consistent I/O requirements, like Apache HBase or another NoSQL database, object stores are a bad choice, although products like HBase have recently been ported to S3.
- Object stores do not support file appends and file truncation. Objects are immutable, and once uploaded, they cannot be changed.
- Object stores are not Portable Operating System Interface (POSIX)-compliant. Over the last couple of releases, HDFS has incorporated many POSIX-like features.
- Object stores do not expose all file system information because it abstracts the entire underlying storage management layer.
- Object storage may have greater request latency than HDFS.
- The directory listing is slow.
- They will eventually be consistent.
- Bucket renaming is not atomic because they are metadata pointers, not true directories.
- Bucket metadata and permissions will not work as they do in HDFS.

Choose HDFS for workloads that require:

- Low performance and fast metadata access or small file read.
- POSIX-like features like strictly atomic directory renames, fine-grained HDFS permissions or HDFS symlinks.

Questions to Ask

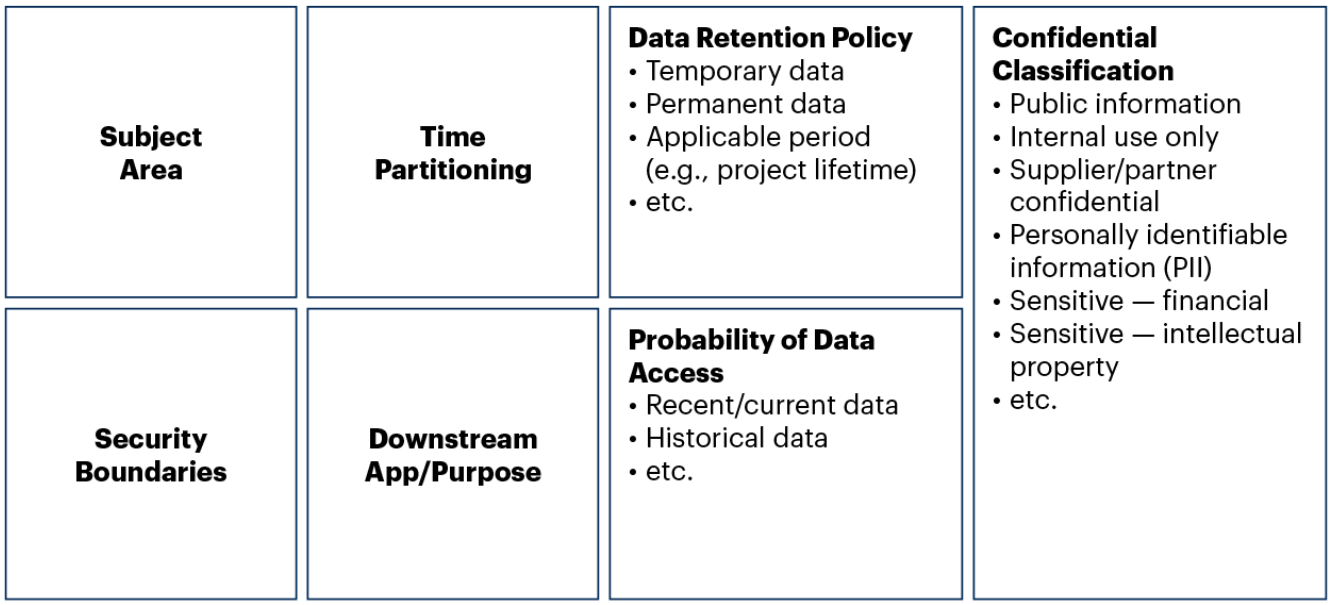
Below are some of the questions data architects should ask before architecting and building the data lake and choosing the components, frameworks and products for storing data in the lake:

- What are the data type requirements for storage in terms of the type of data being stored?
- What are the velocity and throughput requirements of storage?
- What are the data storage volume requirements?
- What type of storage should be chosen — distributed files systems or object stores?
- What are the different storage zones?
- What are the enterprisewide data format, compression and data splitability requirements? Is there an evaluation framework to choose the data format? Some questions to answer would be:
  - Is the data storage going to be row-based, column-based or both?
  - Does the data format need to be human-readable?
  - Does the data to be stored have an associated schema?
  - Can the schema be versioned and updated, and is it subject to change and evolution?
  - How does the data format affect the size of the data stored?
  - Is the data format splittable across the different nodes in a distributed system?
  - What compression capabilities should be supported?
  - What are the ways in which it can be integrated with other tools? How is it interoperable with other tools in the ecosystem?
  - Is the data format optimized for read- and/or write-based patterns?
  - What are the read and write performance requirements? Is it sequential read/random reads or sequential write/random write?
  - What are the hardware requirements on which the data is stored?
  - What are the data partitioning requirements?
  - How is the data storage organization to be done?

Some storage organization approaches are shown in Figure 9.

Figure 9: Storage Organization Approaches

Storage Organization Approaches



Source: Gartner
723117_C

Best Practices

As organizations are moving to the cloud, there is an increased move to adopting cloud-based object stores as the primary data storage for data lakes. For organizations that have decided to stick with HDFS, it is best to move to HDFS 3.0, which uses erasure coding that results in better data storage savings and removes the need to replicate data three times but with a higher data access overhead.

Also, evaluate Apache Ozone, a distributed key-value store for uniform access to data stores irrespective of whether the underlying storage is HDFS or object store. Ozone is designed to work well with both existing HDFS deployments and object stores using an S3 gateway. It can manage large and small files and thousands of nodes equally well, and it provides strong consistency. Ozone can also speak multiple protocols like Hadoop API, S3 API, iSCSI block and NFS. It also provides a storage interface for Kubernetes and YARN. It became general availability (GA) in August 2020.

One of the reasons that HDFS has survived is that competing architectures can't deliver its performance at scale. That is no longer true. Modern, cloud-native storage systems like MinIO have improved the performance problems for object stores.

There are a variety of vendors like — MinIO, Ceph, SwiftStack, Cloudian and OpenIO that offer S3-compatible storage that can be installed anywhere. The advantage of deploying your own object store is the usability of storage at hand — that is, using commodity storage servers. These compatible storage engines operate seamlessly with tools like Spark and Presto.

It is important to evaluate the file sizes and where they originate from. Contrary as it sounds, big data can be made up of a lot of small files, especially those originating from event-based streams from Internet of Things (IoT) devices, servers or applications, which typically arrive in kilobit (Kb)-sized files. This can easily add up to hundreds of thousands of files being ingested in the data lake. Writing small files to an object storage is easy and fast; however, querying these small-sized files either using an SQL engine or for data processing engines like Spark drastically reduces performance and throughput.

Also, metadata collection — like file size, date of creation and so on — on small files is tedious and can reduce performance. Small files have the overhead of opening the file, reading metadata and closing it. Many files lead to noncontiguous disk seeks, which object storage is not optimized for. The remedy for the small-file problem is to merge them into larger files on a regular basis, which is called small-file compaction. Defining these compaction windows is also something that must be wisely tuned and scheduled. Compacting too often would be wasteful because files will still be pretty small and performance improvements will be very small. Files should be deleted postcompaction to save space and storage costs.

In distributed file systems, there is a lot of internode communication. Maximum transmission unit (MTU) is an important metric that indicates the packet size that can be sent over TCP/IP. The default size for MTU is 1,500, which is small for most big data workloads. It is advisable to increase the MTU value to 6,000 to 9,000 to improve performance.

Leverage the performance and throughput improvements with data compaction, compression, partitioning and the right file formats for data processing and query latency.

Architect your applications to be portable between on-premises storage and cloud storage by leveraging either data orchestration solutions like Alluxio, storage-agnostic APIs like Apache Ozone or abstraction layers like those provided by MinIO.

Data Processing

Other than the use case where data lakes are used just for data storage, data lakes provide value only when the data that has been ingested and stored can be processed to build data-driven applications. The versatility of data lakes is evident by the different data processing workloads they can support. For example:

- Batch
- Stream Workloads
- Machine Learning Workloads
- Graph Processing Workloads
- BI Workloads — SQL Query Processing

The data processing landscape is seeing a massive shift toward a decoupled storage and compute-based architecture. Around 2008 to 2009, Hadoop was the game changer with storage and compute tightly coupled. Back then, the disk was still faster than the network. It made perfect sense with commodity hardware to store and process data and move the compute to the data. This was the right choice back then. However, it may not be the right choice now, as the network has become faster than the disk and memory is becoming cheaper by the day. Network switch throughputs have improved considerably to 10 Gbps, 40 Gbps and 100 Gbps interfaces, with better flow control for data-intensive workloads, unlike a decade ago.

However, the challenge with data processing in a data lake is in the variety of available data processing engines. Each of the different data processing engines has a different goal and is best suited for different use cases. The criteria for selecting processing engines depend on the internal architecture of the engines, the best use case fit for these engines, business SLAs, team expertise and other components used in the data lake.

Questions to Ask

When looking to architect the data processing layer of the data lake, try to answer the following questions before choosing a data processing platform or engine:

- How does the engine perform data processing and execute the plans?

Directed acyclic graphs (DAGs) and execution plans express the underlying tasks and dependencies that a distributed engine performs. DAGs can be managed external to the execution engine — which is the simplest architecture (for example, Hadoop MapReduce) — or tightly coupled with the execution engine. Most of today's processing engines no longer use the external model, though. MapReduce is used heavily in some of the core products in the Hadoop ecosystem, but is getting overshadowed by more modular and efficient engines like Apache Spark.

- How does the engine's system architecture distribute the processing for multiple users and handle resource allocation for concurrency and computation?
- How fast, both in terms of throughput and latency, is the engine at executing different types of use cases?
- How does the processing engine manage task and node failures? Different processing engines behave differently when they encounter either hardware or software failures. Analyze and understand these before choosing a processing engine for your use case.
- How is the processing engine architected — a large number of records or in single groups of events?
- What type of workloads does your organization plan to operate in the data lake? Are they batch, streaming, a combination of batch and streaming, machine learning, SQL access or BI workloads?

Best Practices

This section outlines some of the recommended best practices to follow when choosing data processing frameworks and engines for a data lake.

Where and When to Use MapReduce: MapReduce is a low-level framework that puts the onus on developers for handling minute details of operation and data flow, resulting in a lot of setup and boilerplate code. Some operations, such as file compaction and distributed file-copy, nicely fit into the MapReduce paradigm. MapReduce should be used by experienced developers who are comfortable with the paradigm, and they should be applied to only those problems where minute control of the execution has significant advantages.

Organizations are using either an AWS Lambda- or a Kappa-based architecture, with more moving to Kappa's stream-first architecture as the data processing paradigm (see Stream Processing: The New Data Processing Paradigm).

There is a plethora of SQL processing engines built for large-scale data lakes that support use cases related to SQL data processing, BI, ad hoc workloads and so on. However, most of the SQL engines degrade in terms of performance latency, as the concurrency increases with complex analytic queries on large datasets. It is recommended not to use SQL processing engines on the data lake for extremely large SQL and BI workloads with high concurrency and highly complex analytical queries.

Data pipeline debt is a technical debt for data pipelines that is prevalent in all data-driven systems. Debt in data pipelines is a little different from other software systems — most pipeline complexity lives in the data, not the code. In order to nullify the effects of pipeline debt, invest in pipeline tests, which should encompass not just code testing but data testing as well because the oddities and complexities are latent and inherent in both places. The idea is not to test just when new code is written, but whenever data drift or schema drift happens. Validate data quality early in the data pipeline and at relevant points before provisioning and starting up lengthy ETL processes that take a long time to complete and consume significant resources.

Strengths

Some of the strengths of a data lake are outlined below:

- Data lakes provide the ability to work with multiple data types and data formats, which allows the best-of-breed selection of data formats, depending on workload SLAs.
- Data lakes decouple schema from data. Schema on read facilitates agility and flexibility and allows data lake users to be schema-free or define multiple schemas for the same data.
- Data lakes provide capabilities to support a much broader range of possible use cases for enterprises. These include data archival, data integration across a variety of data sources, ETL offloading from data warehouses, and complex data processing across batch, streaming and ML workloads.

- Data lakes can store and derive value from unlimited types of data — structured, semistructured unstructured, multistructured and so on.

Weaknesses

Some of the major weakness and shortcomings of a data lake are outlined below:

- Because of the architectural technical complexity and the multiple moving pieces required to create a data lake, it often takes enormous effort to make data lakes work end to end, especially when leveraging open-source technologies. Hence, data lakes have a slow time to value. No end-to-end tools exist to do everything in a data lake, and a huge burden falls on development, operations, engineering and architecture.
- Data lake technologies are advancing at a rapid pace, and when trying to keep up with this pace, it can be extremely challenging to keep skill sets relevant and applicable.
- Data lakes cannot be used for online transaction processing (OLTP) or operational workloads. Advances and developments are happening in this space, but the adoption curve is below the critical mass.
- Unlike data warehouses, which have been around for over 25 years, there are no industry-wide-accepted best practices.

Guidance

To be successful with data lakes, technical professionals need to clearly define, understand and outline their requirements across the dimensions discussed in this section.

Cloud

For building data lakes on the cloud, keep in mind the following:

- For greenfield projects, it is best to start building data lakes on the cloud, unless data movement and data compliance regulations prevent cloud data management.
- Realize that though cloud PaaS solutions can jump-start your data lake implementation, cloud providers will not give a highly automated, integrated and abstracted system on which an organization can manage its end-to-end data and analytics activity. Cloud provides a plethora of options for data ingestion, data storage and data processing — and a wider choice of third-party solutions via cloud marketplaces. However, it is up to the customer to connect all the pieces together.
- Organizations that find all the moving pieces of building a data lake from the ground up very challenging have two options. They can use cloud-native solutions to build their data lake — in

which case they have to stitch together and configure, coordinate and orchestrate all the pieces by themselves. The other choice is to use data lake PaaS vendors like Qubole.

Technology

Define the strategy and the high-level architectural components required in each layer needed. For example:

- What are the components, tools and frameworks being used across each of the layers in the data lake? For example, is the data lake going to be based on a distributed file system or an object store?
- What analytic tools will be required at each layer?
- What is the data ingestion and data integration strategy? How do you handle data drift and data schema changes and merge incremental CDC into the full load?
- What is the I/O and memory model, and what are the compute, storage, latency and throughput requirements?
- How do you achieve code reusability across the different layers?

People

Build the right technical skill sets within the organization and continually develop those skill sets. Open-source technologies are continuously moving, changing and morphing.

Define, develop and hire across all the skill sets, including a data engineer, a data architect, data scientists, DevOps, DataOps and data analysts, which are required to successfully architect and build data lake solutions.

Other Criteria:

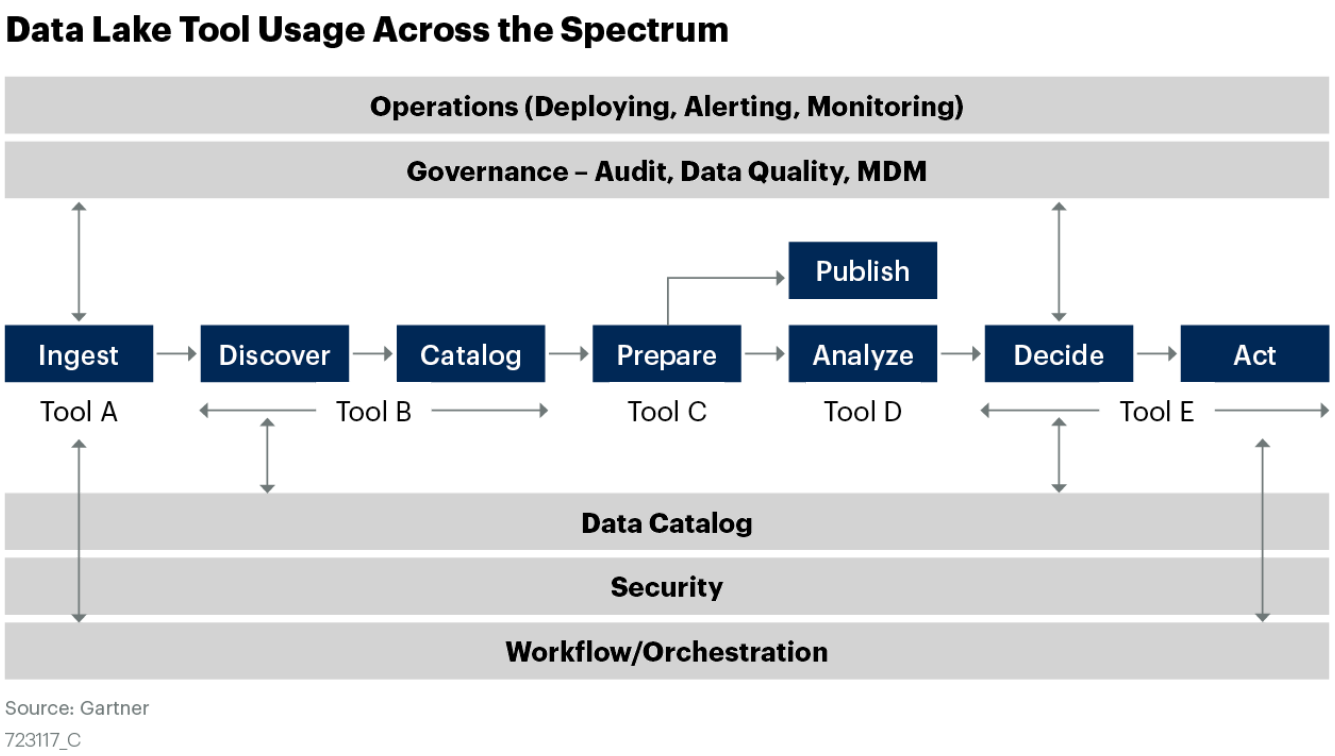
- Manage, minimize and isolate complexity to reduce coupling, and build a configuration-driven system.
- Select the best-of-breed tools. There is no single end-to-end tool for building data lakes across its myriad layers and requirements. Develop components of the toolchain by selecting the best-of-breed tools at each layer and by integrating them with APIs and scripts to develop an end-to-end system.

Building a data lake is a complex undertaking. It is unlike a database server where you provision a server, install a database and the system is up and running. Organizations undertaking building a data lake, but that lack deep engineering, architectural and development skills, should invest in

data-lake-based PaaS solutions on the cloud with vendors like Qubole, Cazena, Zaloni and Upsolver.

Figure 10 shows how different tools should integrate across the different layers of a data lake.

Figure 10: Data Lake Tool Usage Across the Spectrum



Source: Gartner
723117_C

The Details

This section takes a deeper dive into each of the components of a data lake and discusses architectural underpinnings of these components.

Ingestion

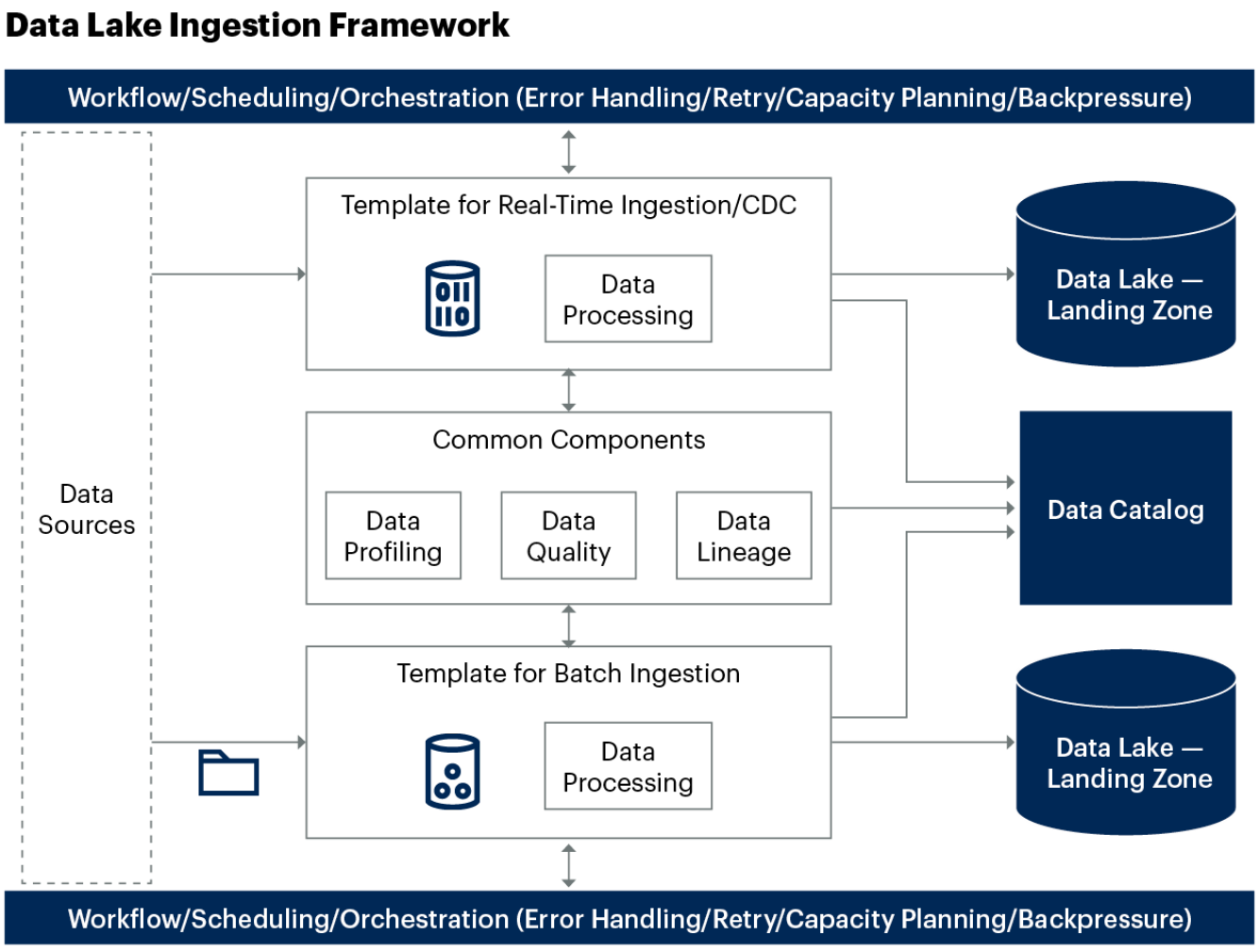
A data ingestion framework should have the following characteristics:

- A single framework to perform all data ingestions consistently into the data lake
- Metadata-driven architecture that captures the metadata of what datasets will be ingested, when they will be ingested and how often they need to be ingested; how to capture the metadata of datasets; and the credentials needed to connect to the data source systems
- Template design architecture to build generic templates that can read the metadata supplied in the framework and automate the ingestion process for different formats of data, both in batch and real time
- Tracking metrics, events and notifications for all data ingestion activities

- A single consistent method to capture all data ingestion along with technical metadata, data lineage and governance
- Proper data governance with “search and catalog” to find data within the data lake
- Data profiling to collect the anomalies in the datasets so that data stewards can look at them and come up with data quality and transformation rules

Figure 11 shows the high-level framework for ingestion based on the above characteristics.

Figure 11: Data Lake Ingestion Framework



Source: Gartner
723117_C

Batch-based ingestion tools include Sqoop, NiFi, Oracle Copy to BDA (Big Data Appliance), Pivotal Greenplum (gphdfs) and HDFS Copy.

Real-time streaming, ingestion-based tools include Apache Flume, Kafka, NiFi and StreamSets.

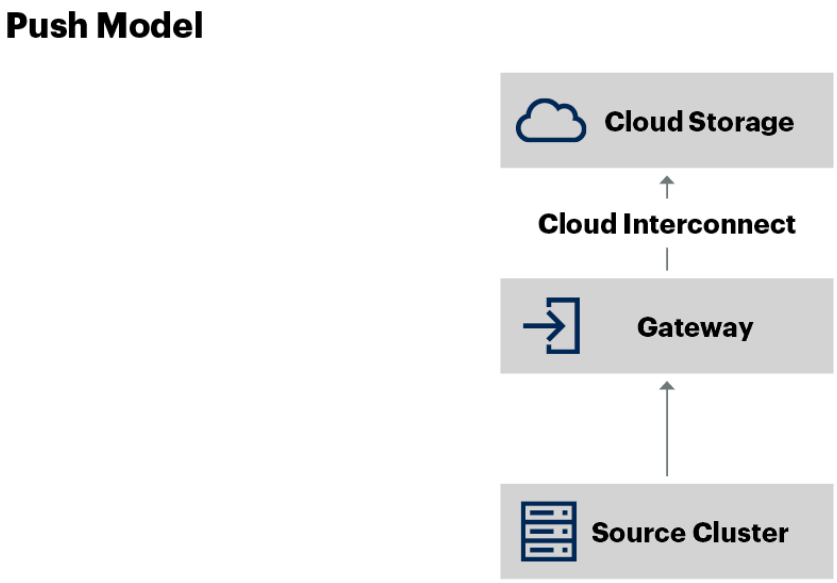
Data Migration From On-Premises to Cloud

There are primarily two different architectural migration patterns for transferring data from on-premises to the cloud:

- Push-based models
- Pull-based models

The push model is the simplest model. Source cluster runs the data migration jobs on its data nodes and pushes files directly to cloud storage through the gateway. Figure 12 shows a high-level setup for the push mode.

Figure 12: Push Model

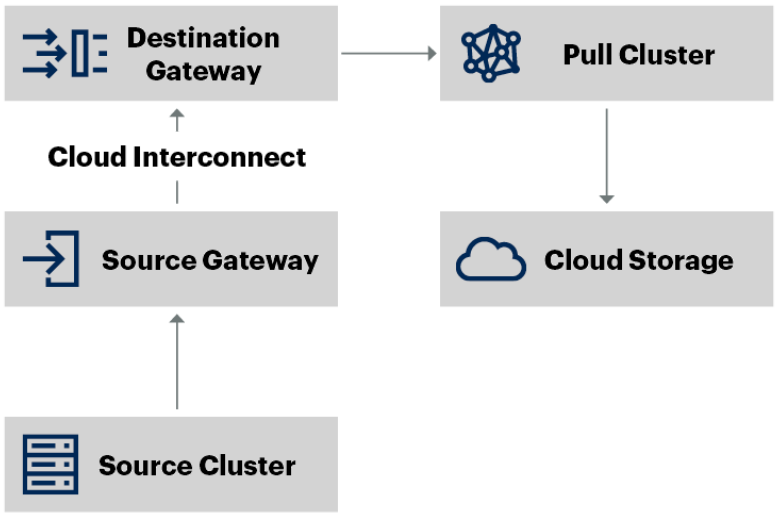


Source: Gartner
723117_C

The pull model is more complex. Ephemeral cluster runs the data migration jobs on its data nodes, pulls files from the source cluster and copies them to the cloud store. Figure 13 shows a high-level setup for the pull-based model.

Figure 13: Pull Model

Pull Model



Source: Gartner
723117_C

Advantages of the Pull Model:

- Impact on the source cluster's resources like CPU and RAM is minimized, and source nodes are used only for serving blocks out of the cluster.
- It is easier to fine-tune the specifications and load balance for the pull cluster's resources to handle the data copy jobs and tear down the pull cluster when the migration is complete.
- Traffic on the source cluster's network is reduced, which allows for higher outbound bandwidths and faster transfers.

CDC

The different ways CDC can be accomplished are listed in Table 3.

Table 3: Different Ways to Accomplish CDC

Capture Method	Description	Production Impact
Log Reader	Scans recovery transaction logs; use when access to logs is available	Minimal
Query	Flag new transaction with time stamp and version number	Low
Trigger	Source transaction triggers CDC; use when no access to logs is available	Medium

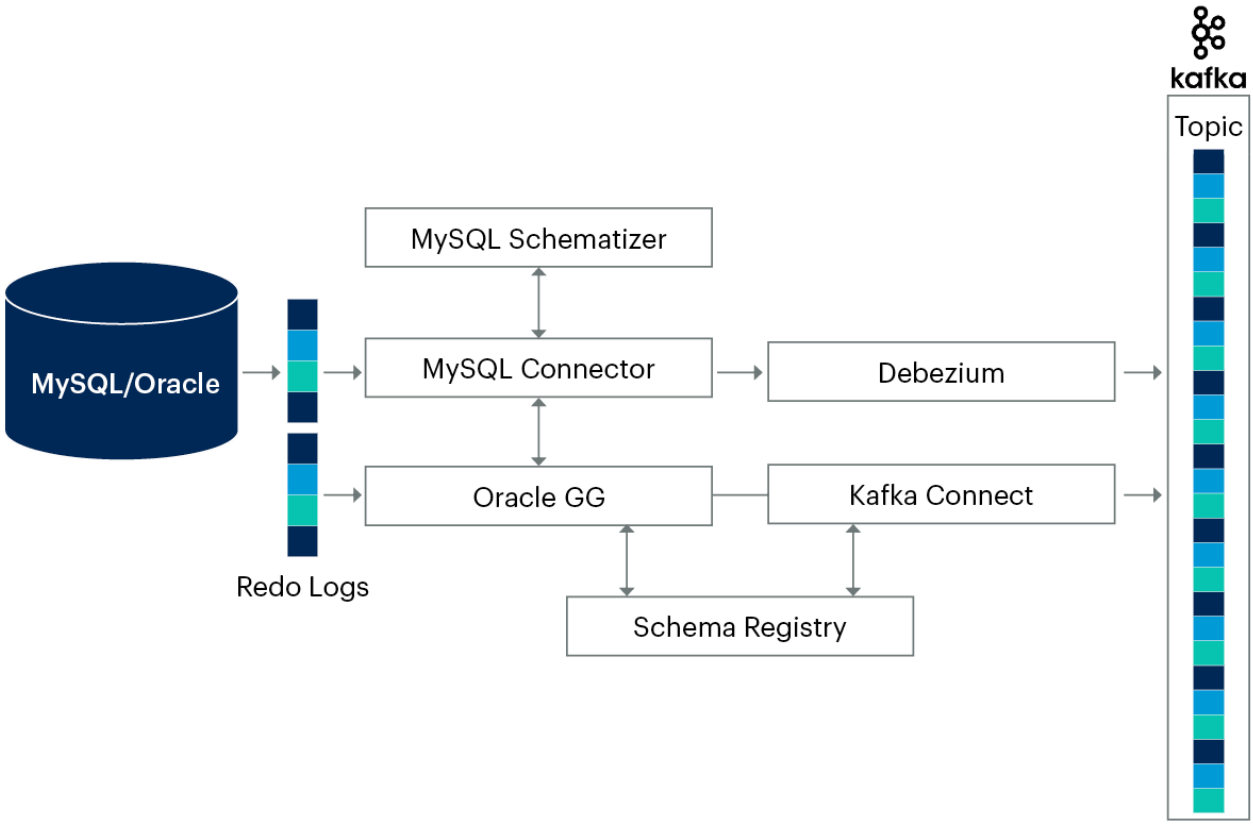
Source: Gartner (October 2020)

CDC Delivery Methods

- **Transactional** — CDC copies data updates/transactions in the same sequence in which they were applied to the source. This is appropriate when sequential integrity is more important to the analytics user than ultrahigh performance.
- **Aggregated/batch** — CDC bundles multiple source updates and sends them together to the target. This facilitates processing of high volumes of transactions when performance is more important than sequential integrity.
- **Stream-optimized** — Stream-optimized CDC replicates source updates into a message stream that is managed by streaming platforms such as Kafka, Microsoft Azure Event Hubs, MapR-ES or Amazon Kinesis. This is data in motion. It supports a variety of new use cases, including real-time, location-based customer offers and analysis of continuous stock-trading data.
- **Open-source** — CDC can also be done with open-source tools like Debezium to ingest data from change logs into a distributed messaging platform like Kafka. This architecture is shown in Figure 14.

Figure 14: CDC Architecture to Integrate With Data Lakes

CDC Architecture to Integrate With Data Lakes

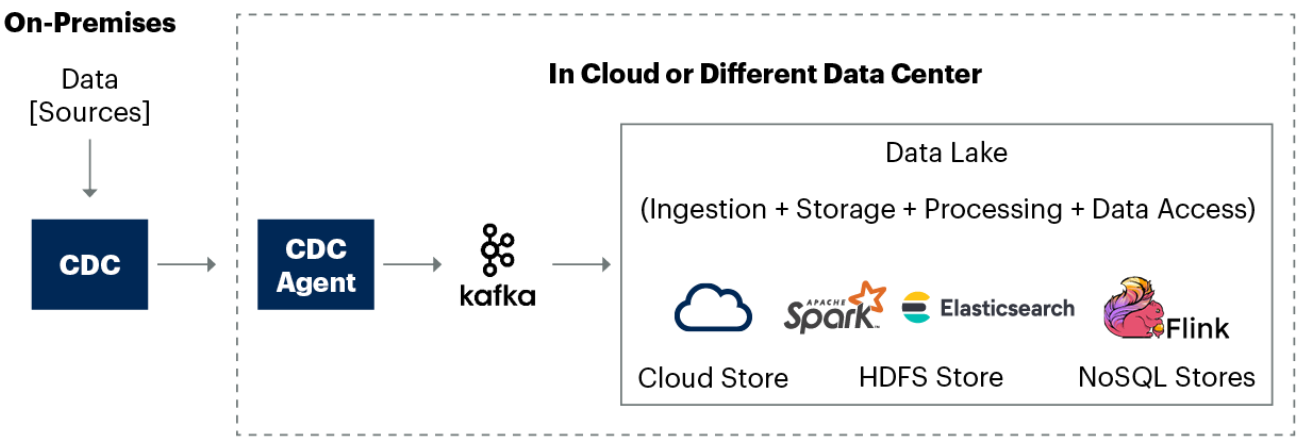


Source: Gartner
723117_C

A very common customer question is: How do you ingest data from applications generating data deployed on-premises to a data lake in the cloud? Figure 15 shows a high-level architecture on how to handle this. Tools like Attunity Replicate are architected for doing this kind of data ingestion.

Figure 15: Using CDC to Ingest Data From On-Premises to Cloud

Using CDC to Ingest Data From On-Premises to Cloud



Source: Gartner

Table 4 shows some of the products, vendors and frameworks for data ingestion into a data lake.

Table 3: Products, Vendors and Frameworks for Data Ingestion Into Data Lake

Batch	Sqoop, StreamSets, Apache Gobblin, Netflix (Suro)
Real-Time	Apache NiFi, Apache Kafka, Apache Flume, StreamSets, Amazon Kinesis, Azure Event Hub, Google Cloud Pub/Sub, Splunk-Streamlio, Gobblin, Netflix (Suro), Mozilla (Heka),
Log Files	Netflix (Suro), Elastic Filebeat, ELK Stack
CDC	LinkedIn (Databus), Debezium, Qlik Data Integration Platform, IBM InfoSphere Data Replication (IIDR), Oracle GoldenGate for Big Data
Replication	Qlik Data Integration Platform, SQData

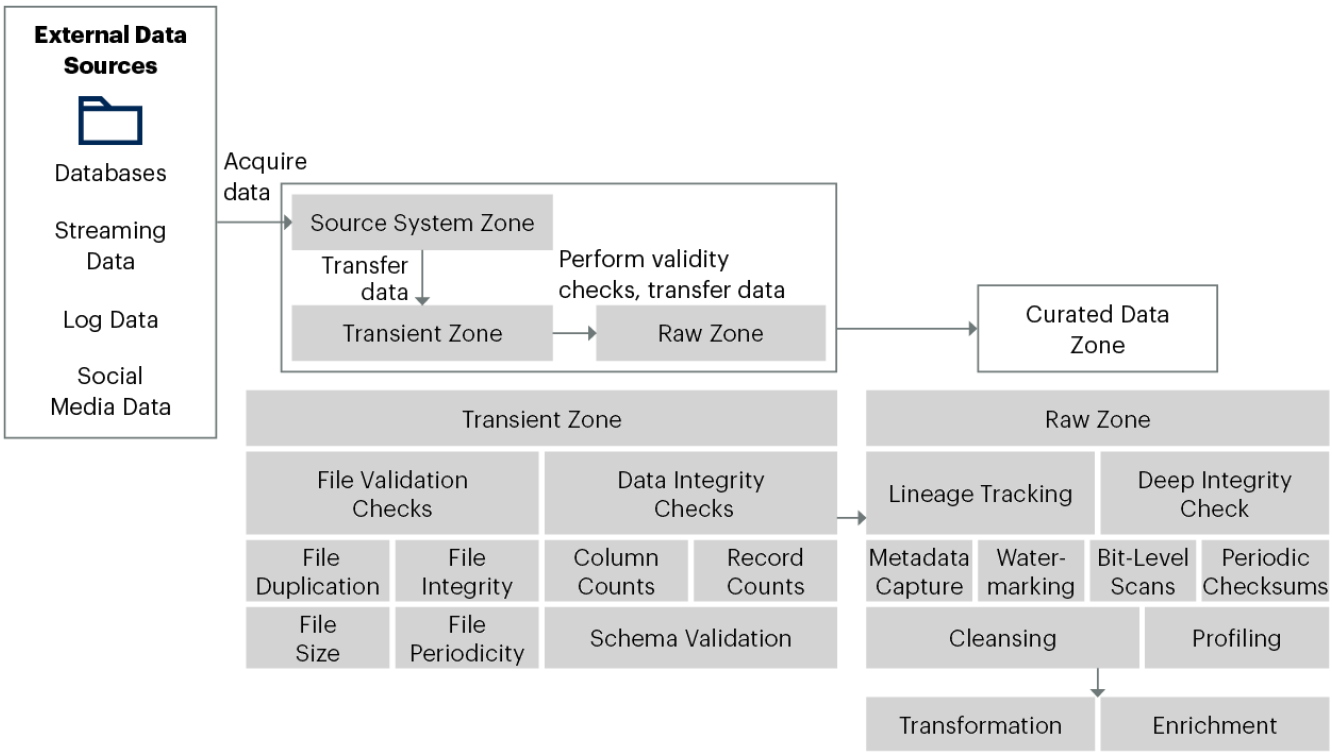
Source: Gartner (October 2020)

Storage

The ingested data needs a landing place. This is where the data lake architects should design the zones of a data lake. Figure 16 shows the different zones in a data lake and some example functionalities that are performed in those zones.

Figure 16: Storage Zones in a Data Lake

Storage Zones in a Data Lake



Source: Gartner
723117_C

Source System Zone: This zone establishes connectivity to the external data sources. It accesses the data, extracts the required data and transfers the data into the transient landing zone for it to perform basic validity checks. After the validity checks have been performed, the data is moved into the Raw Zone to make it accessible for further processing. Once the data goes through data cleaning, data profiling, transformation and enrichment, the data is moved to the Curated Data Zone.

Transient Zone: This performs basic data quality checks just to make sure that the data ingested passes the minimum data quality thresholds before the expensive downstream data pipelines are started.

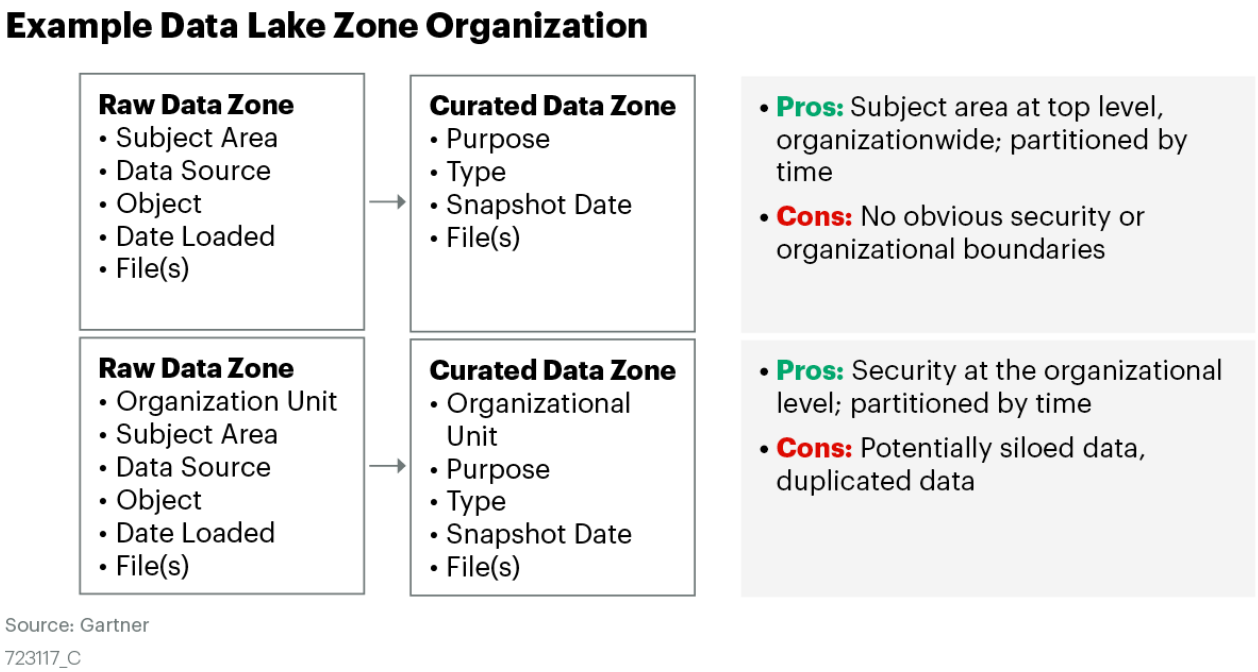
Raw Zone: The data in the Raw Zone should be immutable. It should retain the original data to accommodate future needs. It should be able to support any data, including batch, streaming, incremental and full load.

Curated Data Zone: This is the cleansed, organized, enriched data for the data consumption layer. Most downstream self-service data access should use the data from the Curated Data Zone.

Data in each of the zones can be organized in any of the ways shown in Figure 17. The objective is to avoid a chaotic data swamp. Plan the structure based on optimal data retrieval. The

organization pattern should be self-documenting. The zones could be implemented either as a directory or as an object store bucket structure.

Figure 17: Example Data Lake Zone Organization



Source: Gartner
723117_C

Storage Organization

Organizations are advised to spend time planning out how the data will be organized so that finding the data is as straightforward as possible. The data lake should be organized for optimal data retrieval. The metadata capabilities of data lakes greatly influence how to handle data organization.

Organization of the data lake is influenced by one or more of the following factors:

- Subject matter
- Data source (this comes with a trade-off: a data source table may be duplicated if used by different organizational units; therefore, a different type of structure might make more sense depending on needs)
- Security boundaries
- Downstream applications and uses
- Data load pattern (including real-time, streaming/incremental/full load/one-time)
- Time partitioning and probability of data access (including recent/current data/historical data)
- Data classification

- Public information
- Internal use only
- Supplier/partner confidential
- PII
- Sensitive — financial
- Sensitive — intellectual property

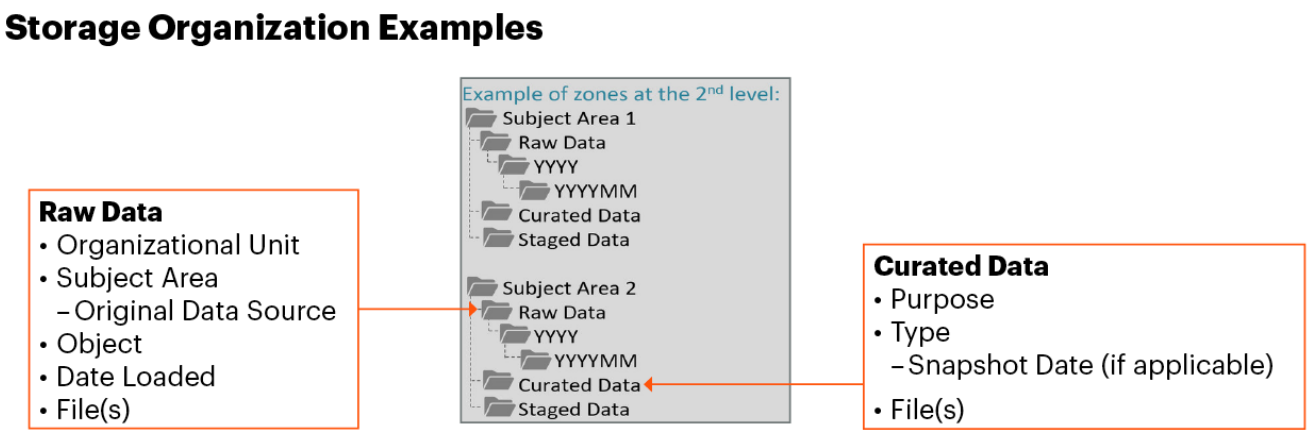
- Business impact (high/medium/low)
- Data retention policy (temporary data/permanent data/applicable period)

Some of the best practices include:

- Translate data lake zones to a top-level folder structure.
- Create zones within a subject area and allow business users access to the prepared data in the Curated Data Zone.
- Zones like raw data and staged data are “kitchen areas” that have restricted access. That’s why putting the zones at the topmost level is very common.
- If the objective is to make all the data available in an easier way, then putting zones underneath a subject area makes most sense. It is less common to separate zones beyond just top-level folders.

Some examples are shown in Figure 18:

Figure 18: Storage Organization Examples



Source: Gartner
723117 C

Some Key Points to Consider About the Raw Data Layer:

- It is usually meant for immutable data, which allows time travel if necessary.
- It should be the lowest-granularity, most atomic data you are able to obtain.
- It is possible that bad data is deleted from the raw data layer but generally deletes are rare.
- The choice of data format is particularly crucial in the raw data layer since the performance of writes for low-latency data is critical. Remember that formats like comma-separated values (CSV) are easy to use and human-readable, but they don't perform well at a larger scale. Choose formats like Parquet or ORC or Apache Arrow. When selecting data formats, think about file sizes, if a self-describing schema is desired (like JSON or XML), data type support, if schema changes over time are expected, performance needed for write and read and integration with other systems.
- The raw data layer is often partitioned by date. If there's an insert/update date in the source that can be relied upon, organize incremental data loads by that date. If there's no reliable date to detect changes, then decide if the source is copied over in its entirety every time the data load executes. This correlates with what is required to detect the history of changes over time.
- When partitioning by date, generally the dates should go at the end of the path rather than at the beginning. This makes it much easier to organize and secure content by department, subject area and so on.
- Rule of thumb is to not mix data schemas in a single folder so that all files in a folder can be traversed with the same script. If a script is looping through all the files in a folder, depending on the technique being used, a script might fail if it finds a different format.
- Raw data should be stored the same as what's contained in the source. This is much like staging data for a data warehouse without any transformations, data cleansing or standardization. The only exception would be to add metadata such as a time stamp, or a source system reference, directly within the data itself using additional fields.
- Very few people have access to the raw data layer. Just like staging for a data warehouse is considered the back-end "kitchen" area, the raw data layer is usually hands-off for most users except for highly skilled analysts or data scientists.
- Some architectural designs call for a transient or temporary landing zone prior to the Raw Data Zone. An intermediate zone prior to the Raw Data Zone is appropriate if validity checks need to be performed before the data can hit the Raw Data Zone. It's helpful to separate "new data" for a period (for instance, to ensure that jobs pulling data from the Raw Data Zone always pull consistent data).

Key Points About the Curated Data Layer:

- This layer contains data for specific, known, purposes.
- The curated data layer is considered “schema on write” because its structure is predefined.
- Make sure the curated data can be regenerated on demand. There could be a number of reasons for this. To recover from an error, depending on how schema changes over time are handled, regenerating curated data can help.
- Assign friendly names rather than cryptic names from a source system to improve usability.
- The curated data layer could be set up like a data warehouse.
- Some organizations find that using multiple logical data layers to deliver curated data makes sense, like the concept of different data marts.

Another aspect of data lake storage design and architecture is the capacity planning around storage. Capacity planning becomes a function of some of these parameters:

- **Initial data size** — Historical and current data that will be moved into data lake
- **Year-over-year growth** — Annual data growth rate
- **Compression ratio** — The factor by which the data gets compressed; measurement of compression factor varies by data types
- **Replication factor** — The number of replicas in a cluster (a higher replication factor impacts query performance and data availability; a replication factor of 3 is an optimum number that can balance availability with performance)
- **Intermediate data size** — Hadoop creates multiple temporary files during intermediate stages; temporary data size accounts for 30% to 40% of raw data size; the following formula is a good rule of thumb:
  - Total storage required = (initial data size + year-over-year growth + intermediate data size) * replication factor * 1.2

Storage formats are an important consideration for optimum data lake performance in terms of storage footprint, I/O, and query and data processing performance. Table 5 outlines some of the storage formats and their usage patterns.

Table 4: Data Formats

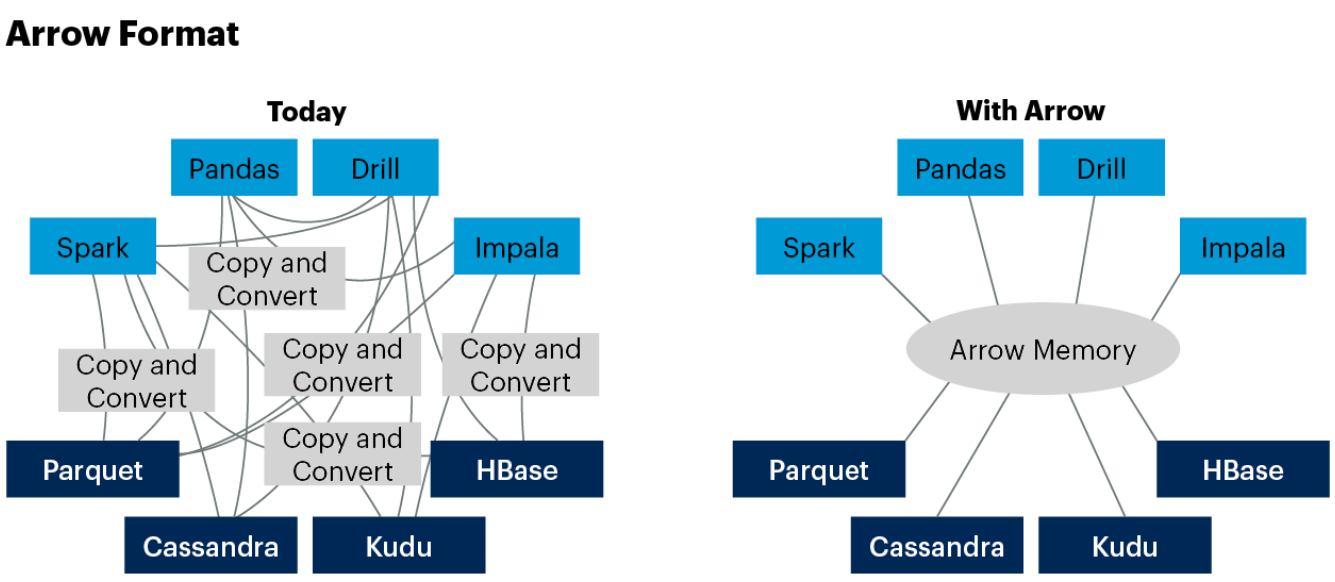
Format Type	Features	Use
Apache Parquet	■ Columnar and nested data support	■ Slower write/update performance ■ Optimum for analytic query
Apache Avro	■ Row format data, nested data support, supports schema evolution	■ Supports schema embedded within the file ■ Supports splitting and block compression
Apache ORC	■ Optimized record columnar files ■ Row format data as key value ■ Hybrid of row and columnar format	■ High compression possible ■ Slow write/update performance ■ Used for analytic queries ■ No schema evolution
SequenceFile	■ Native support with MapReduce ■ Format is splittable ■ Transparent compression	■ Limited schema evolution ■ Supports block compression ■ Used for intermediate file in MR
CSV/TSV Files	■ Regular flat files with or without header information	■ Easy to parse ■ No block compression support ■ No schema evolution

Format Type	Features	Use
JSON	■ Key-value-based, semistructured, nested and hierarchical data	■ No block compression support ■ Schema evolution is better than CSV

Source: Gartner (October 2020)

Apache Arrow is a new storage format gaining more usage across the industry. It is an in-memory columnar data format with O(1) random access performance. It is highly optimized for numeric data and does zero-copy reads. Its layout is highly cache-efficient for analytics workloads and is efficient for fast data interchange between systems without serialization costs such as Apache Thrift, Avro and Google's protocol buffers. It can support complex data structures (hierarchical/nested) and dynamic schemas, and it works very well with modern CPU architectures — cache prefetching and avoiding CPU stalls. As shown in Figure 19, using Arrow's in-memory format avoids any overhead for cross-system and cross-platform communication. More details on Arrow data format can be found on the Apache Arrow website.

Figure 19: Arrow Format



Source: Gartner
723117_C

Partitioning

Partitioning plays a vital role when querying large-scale datasets since it limits the volume of data scanned, dramatically accelerating queries and reducing costs. Good data-partitioning strategies are essential for long-running data pipelines. Object storage doesn't provide indexing, and partitioning is the closest you can get to indexing in a cloud data lake.

It is always advisable to define good strategies for partitioning datasets within the data lake. Good data-partitioning strategies can turn ETL pipelines from consuming hours of clock and machine time and computing power to compact jobs requiring little or no communication among the processing units. It is hard to decide on a silver bullet partitioning strategy, as future jobs might need to query the data in different ways from when partitioning was first applied.

Questions to ask for optimal partitioning include:

- How to manage data retention? To prune data the easiest way is to delete partitions, so deciding which data you want to retain can determine how data is partitioned.
- Is data being ingested continuously, close to the time it is being generated? If so, partition by processing time.
- Is the overall number of partitions too large? This could be detrimental to performance.
- Is there a field besides the time stamp that is always being used in queries? If so a multilevel partitioning by a custom field needs to be defined.

Partitioning Best Practices

- Avoid data shuffling with a good partitioning strategy
- Partitioning by a custom field and by time (multilevel partitioning). Use multilevel partitioning when there is a need to create a distinction between types of events — such as when ingesting logs with different types of events — and have queries that always run against a single event type.
- Monthly subpartitions are often used when also partitioning by custom fields in order to improve performance.
- Each partition adds metadata to catalog and processing metadata always adds to latency.
- It is usually recommended *not* to use daily subpartitions with custom fields since the total number of partitions will be too high.
- Data engineering teams should revisit data partitioning strategies often.
- Partitioning by date is common. When partitioning by date, dates should be at the end of the path rather than the beginning. It's much easier to organize and secure content by department,

subject area and so on because there is often a need to set security at specific folder levels (such as by subject area), and it's rarely based on time. For example,

- Optimal for folder security: \SubjectArea\DataSource\YY\MM\DD\FileData_YY_MM_DD.csv

- Tedious for folder security: \YY\MM\DD\SubjectArea\DataSource\FileData_YY_MM_DD.csv

Data Processing

This section discusses some of the data processing paradigms and patterns that are applied for data processing in the data lake. These include MapReduce, Lambda architecture and Kappa architecture.

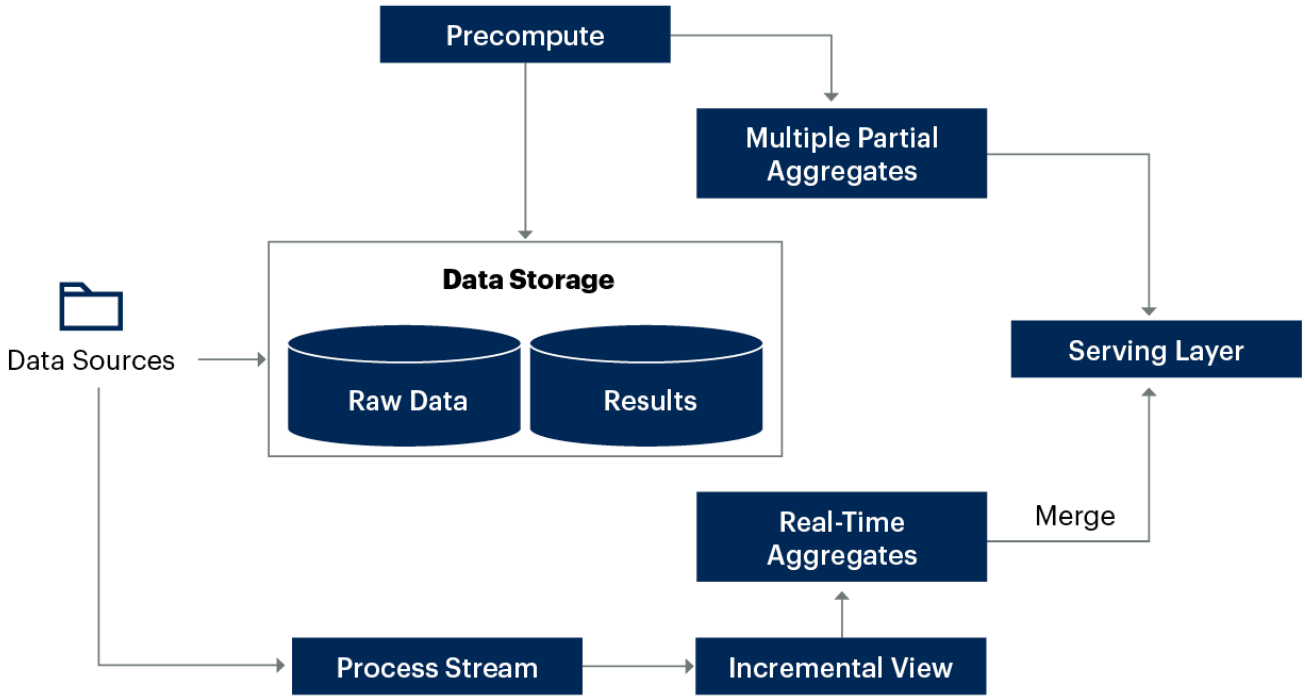
Lambda Architecture

Lambda architecture tries to mitigate the latencies of MapReduce, trying to balance latency, throughput and fault tolerance. The three primary layers are:

- **Batch layer:** This precomputes results using a distributed processing system output to the read-only data store and updates views by replacing the existing precomputed views. Data accuracy in the views is high with batch jobs (accuracy over latency).
- **Speed/real-time layer:** This processes data streams in real time, and the views are almost instantaneous, but they may have less data accuracy (latency over accuracy). However, those views can be updated later by batch methods (accuracy over latency).
- **Serving layer:** This stores outputs from the batch and speed layers to respond to ad hoc queries either by precomputed views or by new views from the processed data. A high-level Lambda architecture is shown in Figure 20.

Figure 20: Lambda Architecture With Batch Layer and Speed Layer

Lambda Architecture With Batch Layer and Speed Layer



Source: Gartner
723117_C

Lambda architecture has the following pain points:

- The architecture requires coding and executing the same logic twice — once for the batch layer and once for the streaming layer, possibly on different frameworks and underlying engines. Due to so many moving pieces in the architecture, it is extremely complex to build, maintain, troubleshoot and debug. Keeping sync between these two layers incurs cost and effort, and this has to be handled in a careful, controlled manner. A very low code reuse across the layers is an inherent problem with this architecture.
- Building a Lambda-architecture-based data pipeline requires expertise in a number of technologies. Getting the all-encompassing skill sets can be extremely challenging for an organization.
- Implementing a Lambda architecture with open-source technologies and then deploying in the cloud can be troublesome. To avoid this, you could use cloud technologies to implement Lambda architecture, but by doing so, the enterprise automatically gets itself tied to a particular cloud provider, and that is inherently a disadvantage.
- Even though the architecture pattern has been around for quite some time, the tools are still immature and evolving. Cloud evolution has surely accelerated and innovated in this space, so it won't be long before mature solutions and tools appear in this space.

- The serving layer can be complex, as it has the additional logic to merge the results from the batch and streaming paths.

Kappa Architecture

The Kappa architecture is based on the premise that batch is a special case of streaming. The batch layer results can be simulated using the streaming layer. Kappa architecture completely removes the batch layer and eliminates the need for the merging layer, thereby simplifying the overall architecture.

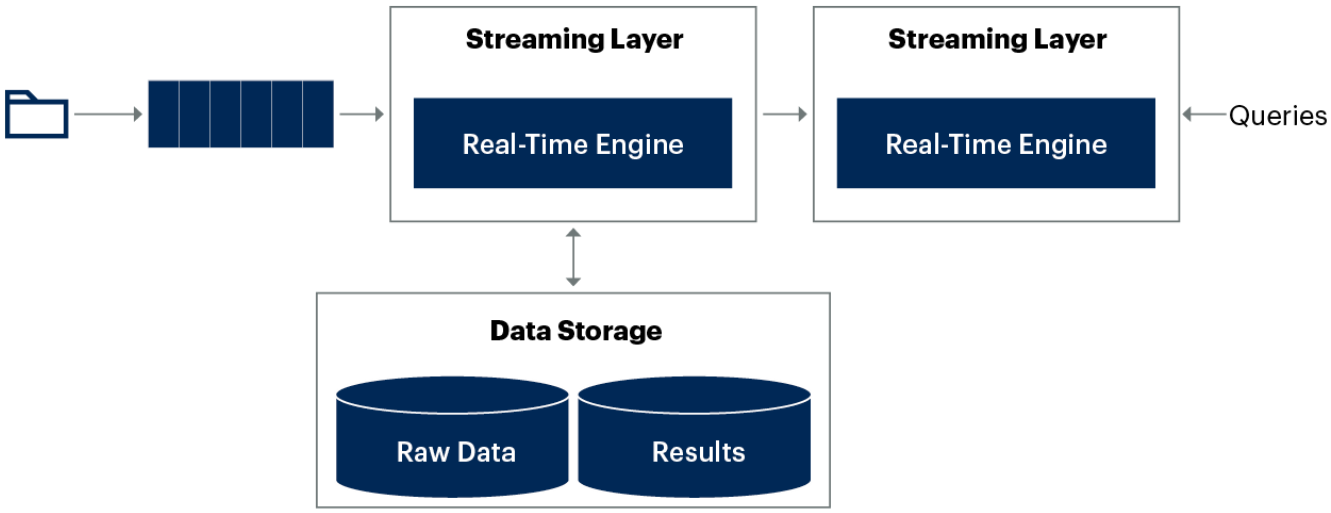
There are four underlying philosophies behind the Kappa architecture:

- **Everything is a stream:** Batching is a subset of stream, and hence, unifies everything as a stream.
- **Immutable data:** Raw data is stored, and computations or views are derived. Because the raw data is stored and it never changes, the computations can be recomputed on demand.
- **Simplicity of the framework:** Kappa architecture relies on a single engine. There is no code duplication or working with multiple data pipelines within the same architecture and data flow.
- **Guarantee deterministic computation:** In the Kappa architecture, data processing can be rerun on the historical data, and the data pipeline must guarantee that the order of the events stays the same. This is the core to guaranteeing consistent results. For this reason, Kappa architecture requires a core component — a distributed messaging platform like Kafka — that provides event ordering and delivery guarantees at least once. Kafka can connect the output of one process. Apache Flink is a streaming data flow engine that provides the second component in the Kappa architecture: the stream processing engine. Flink provides data distribution, interprocess communication and fault tolerance for distributed streaming computations over event streams.

A high-level Kappa architecture is shown in Figure 21.

Figure 21: Kappa Architecture

Kappa Architecture



Source: Gartner
723117_C

Kappa architecture has the following pain points:

- Kappa architecture needs the messaging platform and the streaming platform. Each of these components are complex, both in terms of architecture and engineering.
- Advanced concepts — such as backpressure handling, watermarking and windowing — need to be clearly understood, implemented and deployed.
- Building a Kappa-architecture-based data pipeline requires expertise across multiple products and frameworks.

Transactional Data Lakes

Data lakes have been primarily used in the industry for analytics. However, transactional data lakes are gaining traction and being used to provide an efficient and reliable solution for updates, upserts and deletions of data in data lakes. The traditional approach for updates and deletions has been to overwrite the data at a partition level. This required a rewrite of large amounts of data for even a few changed rows and failed to scale efficiently.

Some of the use cases driving the innovation and adoption of transactional data lakes are given below:

- Implementing regulations like General Data Protection Regulation (GDPR) and the California Consumer Privacy Act (CCPA) that require that organizations remove PII from their databases on request, which requires finding and deleting these records in data lake

- Implementing CDC based on logged changes to an existing dataset instead of a full table copy, enabling organizations to act in near real time on data in their operational databases to be replicated from legacy systems to data lakes
- Implementing slowly changing dimensions (SCD), as dimensional data is updated in source systems
- Removing wrong or fraudulent transactions from datasets, since using a batch-based approach would take hours or days between the time of the event and the time in which the updates are required

Challenges of Doing Updates and Deletes in a Data Lake

Typically, databases have indexes and it is easy to zero in on a specific record to be changed or deleted. Data in a lake is stored as formatted files (like Parquet, Avro or Arrow) in object storage such as Amazon S3, Azure Blob Storage or HDFS. Object storage is used for append-only analysis, where the data is partitioned by time or subject area. It is not easy to point to a specific record or field for update and/or delete operations. Everything is a file in folders without indexes or consistent schemas. Finding a record that needs to be updated or deleted requires full scans, which is resource-intensive and delays responses to change requests.

Users don't get a consistent view of the data like they are used to from databases.

Solutions

There are several open-source projects that have innovated to tackle the problem of transactional updates and deletes in a data lake. Each of them provides different abstractions on top of the different file format of the data in a lake.

- Apache Hive ACID
- Apache KUDU
- Databricks Delta Lake
- Apache Iceberg (formerly from Netflix)
- Uber (Apache Hudi)

This section briefly discusses the implementation of some of the above solutions.

Operational Capabilities With Hive

Hive's support of ACID semantics and transactions are the basis of implementation of updates and inserts. Hive's DML statement supports the ability to append, update and delete data in transactional tables in ORC File format and append data in Parquet format. Under the covers, Hive

transactional updates work by maintaining subfolders to store different versions and update and/or delete changes for a table, and Hive metastore is used to track different versions.

However, there are challenges with Hive's implementation, since Hive was primarily implemented with HDFS as the storage layer in mind. Using cloud data lakes runs into issues due to the differences in the semantics of cloud storage and HDFS. Renaming files and subfolders, which is the basis of the underlying implementation, is expensive on cloud storage. Hive writes data to temporary locations first and then renames it to the final location in a final commit step. Renaming folders and buckets is not atomic on cloud storage. This can cause partial written data to become visible in the destination bucket. To mitigate this, Hive runs compaction in the background, and it is unsafe to run compactions concurrently with a read operation. However, launching a tablewide operation when compactions are running can lead to errors due to delta files being deleted. To avoid these concurrency-related issues, it is advisable to separate the inserts from the query with a table for ACID operations (for example, real-time ingestion and a materialized view built on top of the table for querying). Compactions, however, have to be paused during the rebuilding of the materialized views.

Databricks Delta

Delta Lake from Databricks brings ACID transaction capabilities, version control and indexing to a data lake. It provides snapshot isolation for concurrent read/write operations and enables efficient inserts, updates, deletes and rollbacks. It optimizes the entire workflow and steps around this using compaction and z-order partitioning to achieve better performance. It has the best integration with Spark ecosystem.

ACID transactions on Spark support serializable isolation levels preventing inconsistent data from being read. It supports time travel with data versioning and enables rollbacks and audit trails. It supports merge, update and delete operations to enable CDC and SCD.

Schema enforcement is used to prevent the insertion of bad records during ingestion. However, there are performance challenges during upserts when multiple partitions are updated. Table 6 lists a short comparison between the different approaches to implement transactional data lakes.

Table 5: Transactional Data Lake Implementation Comparison

Approach	*Update/Delete*	*Language*	*Cleanup/Compaction*

Approach	Update/Delete	Language	Cleanup/Compaction
Databricks Delta	Yes (Update/Delete/Merge)	Scala/Java/Python	Manual Cleanup
Apache Iceberg	No	Java/Python	API
Apache Hudi	Yes (Upsert)	Java/Python	Manual and Automatic
Hive ACID	Yes	HQL	Automatic

Source: Gartner (October 2020)

Handling Small Files

Too many small files — smaller than the 128MB block size — can kill the performance of any data flow pipeline. Most Amazon S3 slowdowns are due to the number of files you're trying to write. It is more efficient to write larger files than lots of small files. This can happen due to a number of reasons, as outlined below.

- Overhead of opening, reading metadata and closing adds a substantial delay in starting the processes to download or process the files in parallel. There is also an increase of API calls to the metastore and the number of partition entries in the metastore for that dataset.
- Many files imply noncontiguous disk seeks, which most storage systems like object storage or HDFS are not optimized for.
- Too many small files lead to inefficient compression and ingestion process.
- Ingesting data via streaming, say, every five minutes can cause issues. Data ingested incrementally in small batches can end up creating many small files over a period of time.
- Large number of reducers with not enough data being written to HDFS will dilute the result set to files that are small because each reducer writes one file.

- Data skew has similar effects whereby most data is routed to few reducers, leaving other reducers with little data to write, resulting in small files.
- An overpartitioned table, a partitioned dataset with a small amount of data (<128MB) per partition, can be a reason.
- Overparallelizing can result depending on the degree of parallelism specified and the number of partitions and tasks in Spark worker, with a new file written per partition. The more the degree of parallelism, the more the number of partitions and more files are written.

Solutions to Avoid Small Files

- A good size to consider for files is 256MB or greater.
- Control the number of partitions and review the partition design to reduce the partition granularity to curb the generation of small files, for example, from daily to monthly partitions.
- Perform regular compaction. When implementing compaction, define compaction windows wisely. Compacting too often will be wasteful and compacting too infrequently will result in long processing times. Reconfigure table partitions once compaction is completed so that it reads the compacted partitions rather than the original files. Delete uncompacted fields to save space and storage costs. Always have a copy of the data in its original state for replay and event sourcing. Compaction is prone to data loss and data duplication if done incorrectly.
- Keep file size as big as possible but still small enough to fit in memory uncompressed. Use a 500MB file size limit to stay within comfortable boundaries.
- Reduce the number of partitions in the following ways.
  - **Repartition** — Repartition is one mechanism for changing the number of partitions in your dataset. It can be used to increase or decrease the number of partitions.
  - **Coalesce** — Coalesce can only be used to decrease the number of partitions. In this use case, it fits perfectly.

Other benefits of not using multiple small files:

- Lowering the authentication checks across multiple files
- Reduced open file connections
- Faster copying/replication
- Fewer files to process when updating data lake permissions

For more details on data processing, see:

- An Introduction to and Evaluation of Apache Spark for Modern Data Architectures
- Streaming Architectures With Kafka
- Stream Processing: The New Data Processing Paradigm

This research will not be discussing SQL workload processing on data lakes and the associated products, tools and engines because these have been extensively discussed in Selecting SQL Engines for Modern Data Workloads

Document Revision History

Building Data Lakes Successfully - 11 July 2019

Recommended by the Author

Building Data Architecture Using Amazon Web Services

Building a Data and Analytics Architecture Using Azure

Building Data and Analytics Architecture in Google Cloud Platform

The Practical Logical Data Warehouse: A Strategic Plan for a Modern Data Management Solution for Analytics

Working With Semistructured and Unstructured Datasets

Assessing the Optimal Data Stores for Modern Architectures

An Introduction to Graph Data Stores and Applicable Use Cases

Time Series Database Architectures and Use Cases

Stream Processing: The New Data Processing Paradigm

An Introduction to and Evaluation of Apache Spark for Modern Data Architectures

Streaming Architectures With Kafka

Recommended For You

Selecting the Optimal Technical Architecture for Data Ingestion

Selecting SQL Engines for Modern Data Workloads

Demystifying the Data Fabric

Building Data Architecture Using Amazon Web Services

An Introduction to and Evaluation of Apache Spark for Modern Data Architectures

Supporting Initiatives

Data Management Solutions for Technical Professionals

Track