O'REILLY

Building Medallion Architectures

Designing with Delta Lake and Spark

Piethein Strengholt

Chapter 1. The Evolution of Data Architecture

Creating a robust data architecture is one of the most challenging aspects of data management. The process of handling data—ranging from its collection to transformation, distribution, and final consumption—differs widely depending on a variety of factors. These factors include governance, tools used, the organization's risk profile, size, and maturity, the requirements of the use cases, and other needs, such as performance, flexibility, and cost management.

Despite these differences, every data architecture comprises several fundamental components. I frequently discuss these components using the metaphor of a three-layered architecture design, a concept I introduced in my previous work: *Data Management at Scale* (O'Reilly). This design has proven instrumental for organizations in conceptualizing and structuring their data management strategies. It features three layers: the first includes various data providers; the second serves as the distribution platform; and the third consists of data consumers. Additionally, an overarching metadata and governance layer is crucial for managing and overseeing the entire data architecture. You can see a reflection of this design in Figure 1-1.

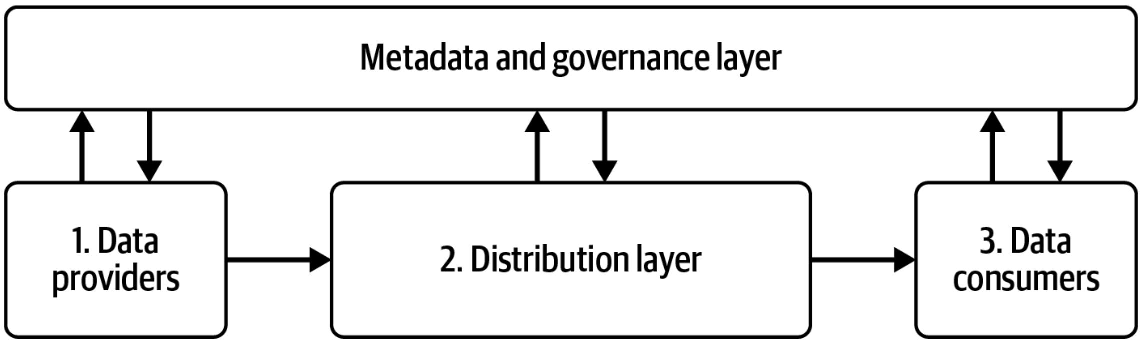

Figure 1-1. The three-layered architecture design

From left to right, here's a brief overview of each layer:

The first layer

This layer consists of various data providers, which represent the diverse sources from which data is extracted. This extracted data is characterized by a mixture of data types, formats, and locations spread across different organizations.

The second layer

This layer represents the distribution platform and is complex due to the vast array of tools and technologies available. Organizations face the challenging task of selecting from hundreds, if not thousands, of products and open source solutions for integration.

The third layer

This layer comprises data consumers, characterized by consuming data services. Data services leverage business intelligence, machine learning, and artificial intelligence (AI) to provide predictions, automation, and real-time insights. Other services manage basic storage and data processing. This layer includes a wide variety of technologies and application types, as each business problem demands a customized solution, making both types of services essential in modern data architectures.

To round off the high-level architecture, I typically draw an overarching layer in discussions, referred to as the metadata and governance layer. This layer plays a crucial role in overseeing and managing the entire data architecture.

The three-tiered diagram, with a particular focus on the inner architecture of the middle layer, illustrates the evolution of data platform management within organizations. It showcases a significant shift from traditional proprietary data warehouse systems to more adaptable, open source, and distributed data architectures. This transformation is driven by a collection of open source tools and frameworks, collectively known as the *modern data stack*.

The challenge is that the modern data stack does not represent a complete data platform on its own. It requires the integration of many independent services and tools, each designed to tackle specific elements of data processing and management. Each service or tool brings its own set of standards for data exchange, security protocols, and metadata management. Furthermore, the overlapping functionalities of many services complicate the deployment and usage. Therefore, to effectively leverage the modern data stack, one must carefully select the appropriate services and then meticulously integrate each component. This integration process poses a significant barrier to entry.

This issue isn't a failing of any one vendor; it's a failing of the market.[1]

Benn Stancil

Technology providers see this issue too. They have recognized the complexity of integrating and managing infrastructure, data storage, and computation. They've made significant progress in development and standardization, particularly in Apache Spark and open source table formats, such as Delta Lake. This has led to the creation of comprehensive software platforms, simplifying the way data can be handled. Many data engineers prefer these platforms due to their innovative features. In addition, organizations that leverage Spark and Delta Lake find the *Medallion architecture* (which I'll define in the next section) particularly advantageous as it fully exploits the strengths of a robust, scalable, and efficient framework for end-to-end data management and analytics.

What Is a Medallion Architecture?

A Medallion architecture is a data design pattern used to logically organize data, most often in a lakehouse, using three layers for the data platform, with the goal of incrementally and progressively improving the structure and quality of data as it flows through each layer of the data architecture (from Bronze ⇒ Silver ⇒ Gold layer). In Chapter 3, we delve into the details of each layer. For now, here's a brief overview of each layer:

Bronze layer

The Bronze layer stores raw data from various sources in its native structure, serving as a historical record and a reliable initial storage.

Silver layer

The Silver layer refines and standardizes raw data for complex analytics through quality checks, standardization, deduplication, and other transformations. It acts as a transitional stage for processed, granular data with improved quality and consistency.

Gold layer

The Gold layer optimizes refined data for specific business insights and decisions. It aggregates, summarizes, and enriches data for high-level reporting and analytics, emphasizing performance and scalability to provide fast access to key metrics and insights.

Such a design, as seen in Figure 1-2, offers an excellent opportunity for implementing applications or use cases for business growth and development.

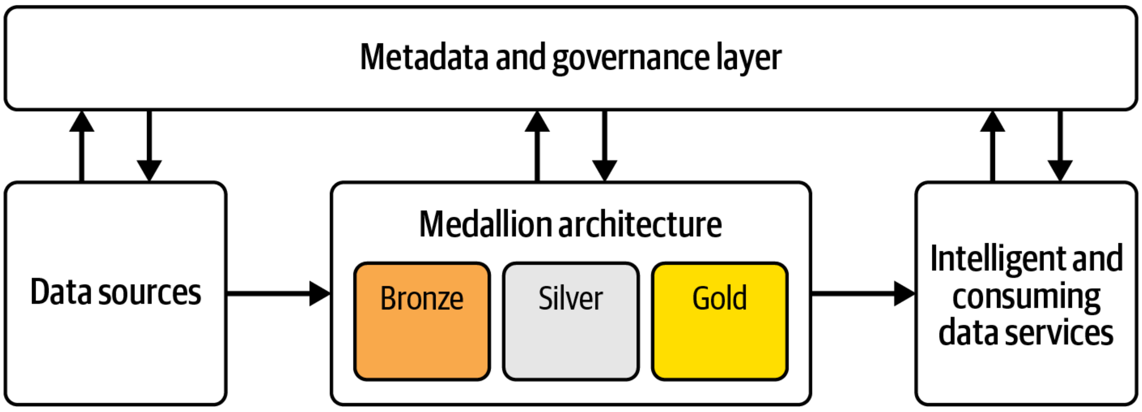
Metadata and governance layer
Data sources
Medallion architecture
Bronze
Silver
Gold
Intelligent and consuming data services

Figure 1-2. A Medallion architecture, which arranges data into three layers, enhancing the data's structure and quality as it progresses through the layers

The Medallion architecture offers business-friendly labels for different layers. However, many enterprises fail to grasp how to effectively layer and model their data, spending countless hours discussing issues such as selection, integration, overlapping features, and so on. They struggle with fitting objectives into different zones or understanding the meaning of "Bronze," "Silver," and "Gold". Questions also arise about governance and scaling strategies. For instance, what parts of the architecture can be made metadata-driven with flexible configuration and automation? Choosing a comprehensive platform does not automatically answer these questions or solve these issues.

Before diving into the specifics of answering these questions and designing a Medallion architecture, it's crucial to first comprehend the evolution of data architectures. Where did these platforms originate? What design principles persist and still must be applied today? Understanding the history and fundamental principles of data architectures will provide a solid foundation for effectively utilizing these end-to-end platforms. This chapter aims to provide an introduction by exploring past developments, observations, patterns, best practices, and principles. By equipping you with the necessary background information, reasoning, and lessons learned, you'll be better prepared for Part II, when you learn how to design and implement your own data architecture.

If you feel you're already well-versed in the fundamentals of data architecture, feel free to skip ahead to Chapter 3 on the detailed discussion of the Medallion architecture and its layers. If not, join me as we start by examining traditional data warehouses. Then we'll move on to explore the patterns behind the emergence of data lakes, including Hadoop. We'll discuss the pros, cons, and lessons learned from each architecture, and how these developments relate to modern best practices. Lastly, we'll delve into the lakehouse and Medallion architectures, which are closely intertwined. In that section, we'll also discuss different technology providers.

Let's take a journey back to the 1990s. Back then, data warehousing emerged as a common practice for collecting and integrating data into a unified collection. The aim was to create a consistent version of the truth for an organization, serving as key source for business decision making within the company.

This process of delivering data insights involves many steps, such as collecting data from various sources, transforming it into a consistent format, and loading it into a central repository. This process is covered in more detail in Chapter 3. For the moment, let's concentrate on the architecture of a data warehouse, as shown in Figure 1-3, which also includes sources and consuming services, such as reporting tools.

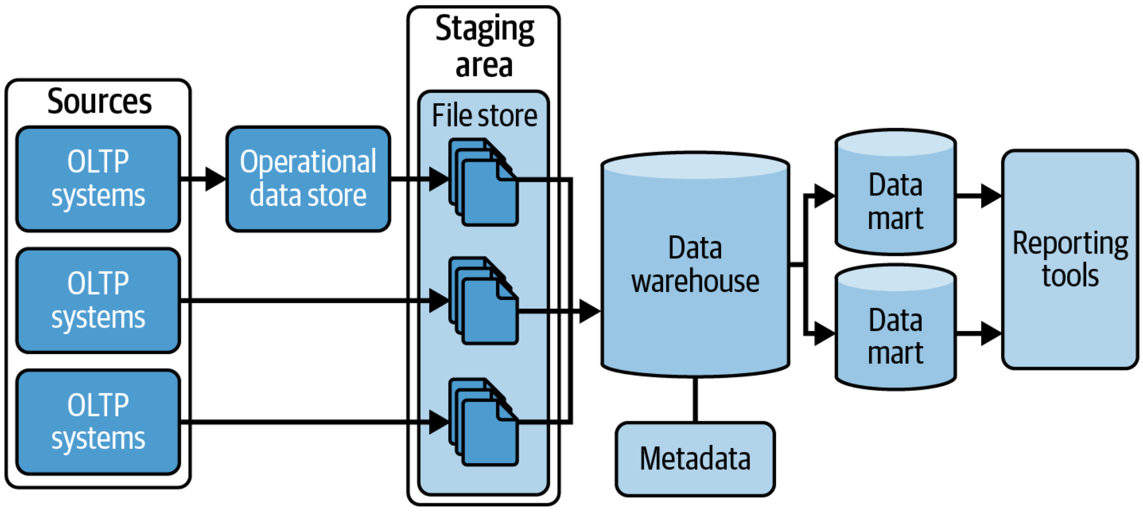

Figure 1-3. Typical data warehouse architecture

I'll start the analysis by exploring the various layers, beginning with OLTP systems on the left, then moving into the data warehouse in the middle. The data marts on the right are introduced and discussed in "Inmon Methodology".

OLTP Systems

Most of the source systems were designed for transactional or operational purposes, reflecting the early computing needs to manage transactions and maintain records. These sources, as seen in Figure 1-3 on the left, are often referred to as *online transaction processing* (OLTP) systems, reflecting their vital operational role.

If you look into an OLTP system, you'll observe that operational workloads are usually quite predictable. You should understand how OLTP systems are used and what typical loads are expected. Queries are relatively straightforward, and the data retrieved is relatively low in volume: reading a record, updating a record, deleting a record, etc. The underlying physical data model is designed (optimized) to facilitate these predictable queries. The result is that OLTP systems are usually normalized, aiming to store each attribute only once.

Database Normalization Versus Denormalization

In the context of relational databases, normalization is a process that restructures data to reduce redundancy and improve data integrity. It is usually performed using a set of rules known as *normal forms*, each addressing specific anomalies or redundancies. The 3NF, or third normal form, is the most commonly used in practice. Normalization allows data to be stored more efficiently and consistently, facilitating easier maintenance and retrieval of data. Thus, when we talk about normalized data, we refer to data that has been organized in a way that enhances its storage efficiency and integrity.

Denormalization is the process of reversing the effects of normalization, intentionally introducing redundancy into a database for the purpose of improving query performance and data retrieval speeds. Denormalization is often used in data warehousing and analytics to optimize data retrieval and processing. By denormalizing data, we can reduce the number of joins required to retrieve data, which can significantly improve query performance. However, denormalization can also introduce data integrity issues, as redundant data can become inconsistent if not properly managed.

Many OLTP systems prioritize maintaining integrity and stability. To achieve this, most OLTP systems use database management systems that adhere to the ACID properties—atomicity, consistency, isolation, and durability.[2] These properties are crucial for running business transactions effectively, as they help in managing and safeguarding data during operations. However, the OLTP design leads to several implications that are important to understand, especially during discussions with organizations.

First, operational systems are not designed to easily provide a complete and consolidated analytical view of what's happening in the business or specific domains. This is because extracting data from highly normalized data models for complex queries is often challenging because it puts a lot of strain on OLTP systems. To get the insights needed, complex queries are required. These queries involve more data and combinations of data, meaning that many tables must be joined or grouped together. Unfortunately, these types of queries are typically quite resource-intensive and can hit performance limits if executed too frequently, especially when dealing with large datasets.[3] If an operational system becomes unpredictable due to these issues, it could negatively impact the business. Therefore, it's essential to consider the potential implications of a normalized design carefully. While it may work well for certain purposes, it's not always the best approach for providing comprehensive analytical views.

Secondly, the stringent requirements for high integrity, performance, and availability often make OLTP systems costly. To optimize these systems, a typical strategy includes shifting unused data out and/or designing the systems to handle only the most recent data. This approach means updates to the data occur instantly without keeping older record versions. Engineers sometimes, in the context of data virtualization,[4] argue for keeping all historical data within these OLTP systems instead of moving it to a data warehouse, data lake, or lakehouse. However, this is often impractical due to the inherent design of OLTP systems. In some instances, storing vast amounts of historical data can bog down these systems, resulting in slower transaction processing and update times. Additionally, maintenance and adaptability challenges can arise.

Thirdly, OLTP systems were originally designed to be specifically optimized for particular businesses, isolated and independent. Each system stores its data differently. This isolation and diversity make it difficult for any single system to offer a unified view without significant data integration efforts.

By separating analytical loads from operational systems, organizations address many of these issues. This separation not only preserves the integrity of the historical data but also optimizes systems for better analytical processing. Furthermore, storing and processing data from various sources in a universal format offers a more cohesive view than a single system could provide. The standard practice is to move this data to a middle layer, such as a data warehouse.

Data Warehouses

The data warehouse serves as a central hub where everything converges. It is used to gather and organize data from various source systems, transforming it into a consistent format for *online analytical processing* (OLAP), which involves complex processing specific to the needs of analytical processing. As offline analyses are typically less business-critical, the integrity and availability requirements for those systems can be less stringent. While data in OLTP systems is stored and optimized for integrity and redundancy, in OLAP, you optimize for analytical performance. Given that, in OLAP, mainly repeated reads are performed, and few writes, it's common to optimize for more intensive data reading. Data can be duplicated to facilitate different read patterns for various analytical scenarios. Tables in OLAP databases are generally not heavily normalized, but preprocessed into denormalized data structures: tables are usually large, flattened, sparse copies of data.

Tip

Deciphering Data Architectures by James Serra (O'Reilly) provides a comprehensive overview of data architectures, including data warehousing, data lakes, and lakehouses. It's a valuable resource on the evolution of data architectures and the principles behind them.

To load data into the data warehouse, you need to extract it from the different source systems. The extraction process is the first step. This process involves understanding the source data and reading and copying the necessary data into an intermediate location, often called the staging area, for further downstream manipulation. The staging area, as you can see in Figure 1-3, lies between the operational source systems and the data integration and presentation areas. This area of the data warehouse is often both a storage area and a set of processes commonly referred to as extract, transform, and load (ETL).

The Staging Area

The staging area, sometimes called the *landing area* or *staging layer*, can be implemented in different ways, varying from relational databases and file stores. Relational databases are more flexible, but more expensive. File stores are cheap, but offer limited features. Staging areas are also typically used to retain historical copies. This is useful for reprocessing scenarios in cases where the data warehouse gets corrupted and needs to be rebuilt. The number of older data deliveries (historical copies) can vary between staging areas across organizations. I have seen use cases where all the data deliveries, including corrections, had to be kept for audits for several years. In other use cases, I have seen the staging area emptied (nonpersistent) after successful processing or after a fixed period of time. In this context, cleaning is done to reduce storage costs or may be required from a governance perspective.

The complexity with extracting and staging is that different source systems may each have a different type of data format. Therefore, the actual ingestion process will vary significantly depending on the data source type. Some systems allow direct database access, while others permit data to be ingested through APIs. Despite advancements, many data collection processes continue to depend on extracting files because extracting files is proven to be more cost-effective and simpler to implement for large volumes of data.

Once the technical data is extracted to the staging area, various potential transformations will be applied, such as data cleansing, enriching data, applying master data management, and assigning warehouse keys. These transformations are all preliminary steps before data from multiple sources is combined, transformed, and loaded into the data warehouse integration and presentation areas. In most cases, you need to rework heavily normalized and complex data structures. As you learned in the OLTP section, these structures come directly from our transactional source systems.

Note

It's crucial to understand that data transformation issues persist when constructing modern data architectures. There is no escaping this data transformation dilemma. The data must be cleaned and integrated to be useful for analytical processing.

So, the question remains: how do you model data in the integration and presentation areas? Let's discuss two common methodologies: Inmon and Kimball.

Inmon Methodology

Unfortunately, there remains some confusion among data engineers about whether, after extraction and transformation, the data should be modeled in physical normalized structures before being loaded into the presentation area for querying and reporting. The confusion stems from the different approaches to handling data.

Traditionally, data warehouses were expensive systems. Emerging in the early 1990s, the Inmon approach, named after its creator Bill Inmon and illustrated in Figure 1-4, was a widely used method based on a normalized data model, generally modeled in the third normal form.

This 3NF model structures data into tables with minimal redundancy, ensuring that each piece of data is stored only once and eliminating duplicate data. It also ensures referential integrity by ensuring that every nonprime attribute of the table is dependent on only the primary key. This method substantially reduces the required storage space. Furthermore, it involves creating a centralized and highly structured data warehouse, known as an enterprise data warehouse (EDW), which serves the entire organization.

For querying and better performance, the Inmon approach also incorporates a presentation layer: data marts. These are created after data has been efficiently stored in the integration layer. These additional data marts typically contain only a subset of the integration layer's data. They are designed to cater to a specific use case, group, or set of users. The data in these data marts is usually organized in star schemas, as it has been optimized for reading performance. The simplicity and denormalization of data structures in star schemas are the key reasons why they are well-suited for read-intensive operations. Consequently, it might be argued that data in data marts is stored less efficiently compared to the integration layer. Additionally, substantial effort is required to transform the data from the 3NF in the integration layer to a denormalized model in the data marts. This process often involves complex joins to reassemble the data, thereby restoring its full meaning for more effective analysis and querying.

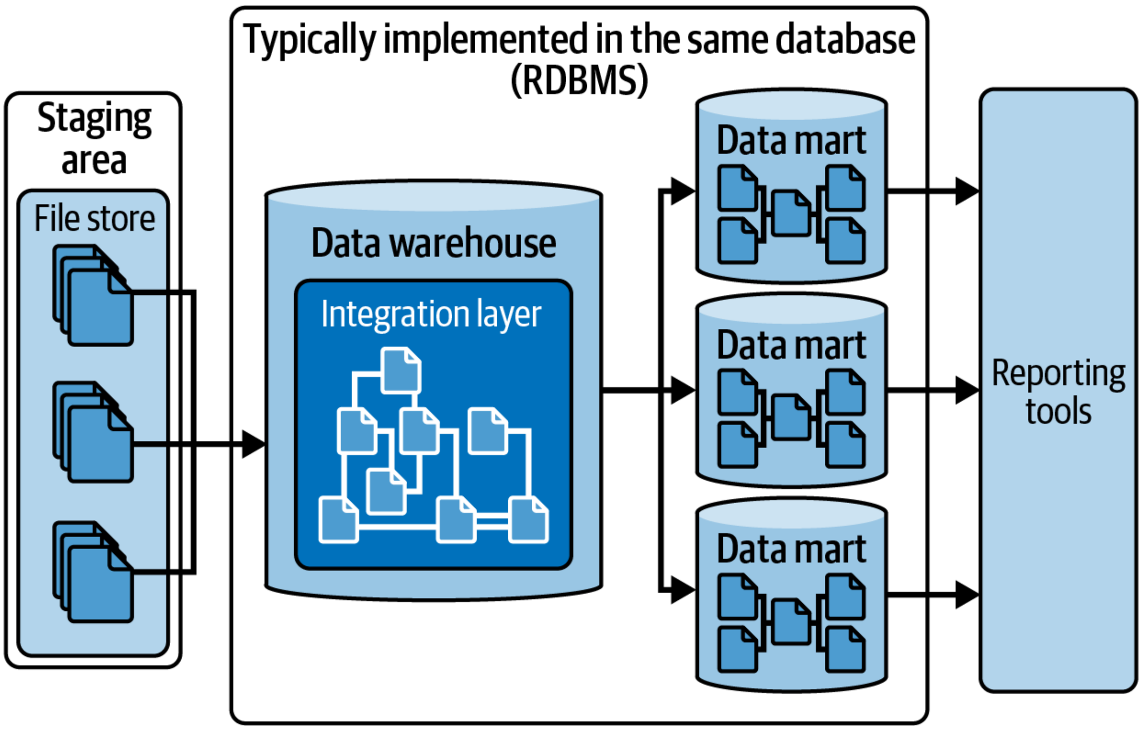

Figure 1-4. The Inmon approach; a top-down design where a centralized data warehouse is built first, and then data marts are created from this central warehouse

Many practitioners have reservations about the approach of using normalized structures in the integration layer and dimensional structures for presentation purposes. This is because data is extracted, transformed, and loaded twice. First, you extract, transform, and load the data into the normalized integration layer. Then, you do it all over again to ultimately load the data into the dimensional model. Obviously, this two-step process requires more time and resources for the development effort, more time for the periodic loading or updating of data, and more capacity to store the copies of the data. Another drawback is that if new data has to be added to a data mart, data always has to be added to the integration layer first. Since the development takes time and changes to the integration layer have to be made carefully, users requiring new data have to wait (longer).

Moreover, data redundancy—unnecessary duplication of data—is frequently pointed out as an issue in the Inmon integration layer. However, this argument carries little weight in the era of cloud computing. Cloud storage has become quite cost-effective, though computation costs can remain high. Because of this high costs, many experts now advocate for the Kimball data modeling approach.

Kimball Methodology

Kimball methodology, named after its creator Ralph Kimball, was introduced in 1996 as a data modeling technique, and is often used in data warehousing.[5] It focuses on the creation of dimension tables for efficient analytical processing. In this approach, dimensional data marts are built first to respond to business needs. For this, Kimball recommends a dimensional modeling technique using a star schema.

An abstract representation of the Kimball methodology is shown in Figure 1-5.

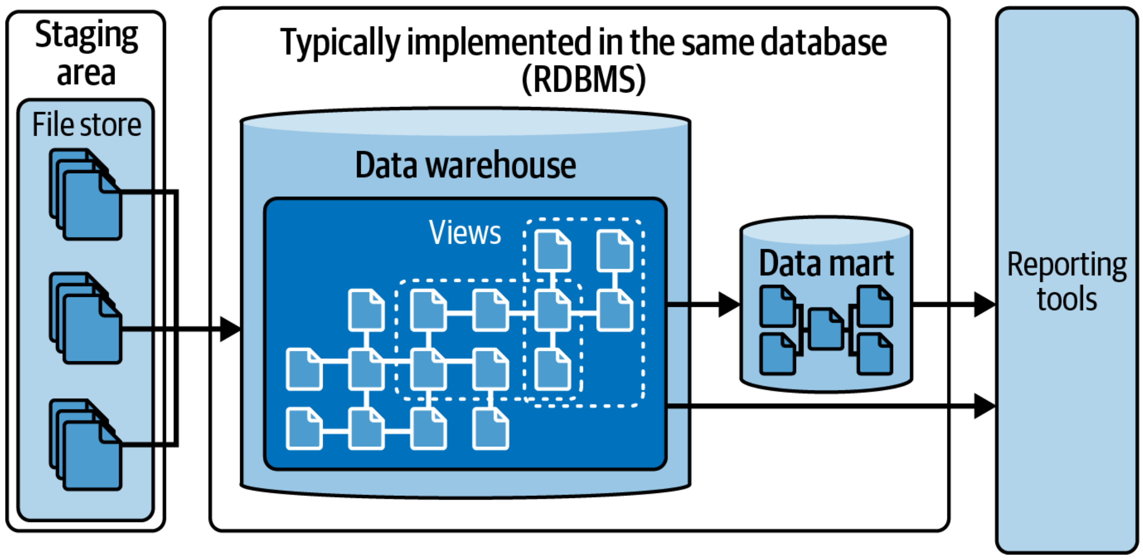

Figure 1-5. The Kimball methodology; a bottom-up approach to building the data warehouse

In this approach to data modeling, the integration layer of the data warehouse is seen as a conglomerate or collection of dimension tables, which are derived copies of transaction data from the source systems. Once data is transferred into the integration layer, it's already optimized for read performance. This data, being more flattened and sparse, closely resembles the data mart structures in the Inmon approach. Yet, unlike the Inmon model, Kimball's integration layer comprises dimension tables, which form the foundation for the data marts. Therefore, not only does Kimball recognize data marts, it also sees them as vital for enhancing performance and making subselects. Data marts permit aggregation or modification of persistent data copies based on user group requirements. Intriguingly, data marts can also be virtual. These are logical, dimensionally modeled views built on top of the existing dimension and fact tables in the integration layer, offering flexibility and efficiency in data handling.

Confusion About the Functions of Data Warehouse Layers

It's crucial to clarify that there can be confusion about the functions of data warehouse layers, which correlates to the same confusion that can arise when layering a Medallion architecture. These different layers, which form the middle part of the larger overall architecture, are designed to assign contrasting responsibilities to various stages, mirroring common practices in software architecture. Typically, there's a staging or ingestion layer where raw data is stored, effectively separating the source systems from the data warehouse. Following this is an integration or transformation layer, where data is integrated after meeting all the acceptance criteria of the staging area. Here, cleansed, corrected, enriched, and transformed data is stored in a unified model. It's harmonized, meaning formats, types, names, structures, and relations have been standardized. This layer also contains historical data processed to show changes over time. Finally, there's a presentation layer where relevant data is selected for specific use cases. The data is remodeled to meet the specific requirements of the use case.

However, it's essential to note that there could be valid reasons to deviate from this traditional three-layered data warehouse design. For auditability or flexibility, some organizations implement additional layers. For instance, an extra layer might be added for auditability, where sources are first mapped to the target model before merging with other sources. Alternatively, the staging area could be split into a low-cost file store holding all data deliveries and a relational database only containing the most recent validated data. The key takeaway is that the number of layers or zones depends on your requirements. There's no universally correct answer. It's about making the right trade-offs to meet your specific needs.

To facilitate this approach to dimensional modeling, Kimball introduces the concept of *conformed dimensions*. These are key dimensions that are shared across and used by various user groups. Additionally, Kimball introduces the technique of historizing data through *slowly changing dimensions* (SCDs).

SCDs are tables that capture all historical changes slowly and predictably. In other words, an SCD is a type of dimension that has attributes to show change over time. SCD1, SCD2, and SCD3 are the most commonly used methods for handling these changes:

SCD1

This type, also known as "overwrite," involves simply updating the existing record in the data warehouse with the new information. This method works well when historical data is not important and only the most current data is needed. However, SCD type 1 does not allow for tracking historical changes, as the old data is overwritten with the new data.

SCD2

This type, also known as "add new row," involves creating a new record in the data warehouse for each change that occurs, while still retaining the original record. This method is useful when historical data is important and needs to be preserved. SCD2 allows for tracking changes over time by creating a new record with a new primary key value, but retaining the original record with a separate primary key value. This way, the data warehouse can maintain a complete history of changes over time.

SCD3

This type, also known as "add new attribute," involves adding a new attribute to the existing record in the data warehouse to track changes. This method is useful when only a few attributes need to be tracked over time. However, SCD3 has limitations, primarily that it only tracks a limited amount of history (typically just one previous state).

In the era of modern data architectures, data modeling, such as the Inmon and Kimball methodologies, continues to play a vital role in managing and harnessing the power of data effectively. It aids in understanding and utilizing complex data efficiently by separating the concerns of ingestion, integration and harmonization, and consumption. By creating solid representations of data and its interrelationships, data usage becomes easier for both technical and nontechnical stakeholders. Additionally, data modeling facilitates better performance and query optimization. With a well-structured data model, it's easier to locate and retrieve specific data, thereby improving the system's speed and performance. Finally, it supports better data governance and security. With clear data models, organizations can implement better data management policies, ensuring the right access control and data usage.

Key Takeaways from Traditional Data Warehouses

This brings us to the end of our discussion on traditional data warehouses. We've explored the Inmon and Kimball methodologies, which are still relevant today. As a review, we'll cover the key takeaways from traditional data warehouses that you can apply moving forward.

Firstly, the concept of layering data isn't new. It's been proven to be an effective strategy for separating different concerns, which helps in organizing and managing data more efficiently.

Secondly, data modeling is crucial. It plays a significant role in flexibility, reducing data redundancy, boosting performance, and serving as an interface for the business. Getting data modeling right is essential for any data management system to be effective.

Lastly, traditional data warehousing highlights the tight integration between software and hardware. Typically hosted on-premises, these systems integrate compute and storage tightly, making data handling quick and efficient. They include sophisticated pieces of software that maximize the performance of the hardware they run on. You can scale these systems vertically by boosting the physical infrastructure. Despite their potential costliness and limitations, these systems were once the preferred choice for many organizations and still hold value for specific use cases today.

Data warehouses are invaluable to businesses because they deliver high-quality, standardized data, essential for informed decision making. The key to their effectiveness lies in their expert data modeling and the tight integration of hardware and storage, ensuring fast and efficient data retrieval. This makes them an essential tool for business operations.

However, traditional data warehouse architectures that rely on relational database management systems (RDBMSs) face difficulties in handling rapidly increasing data volumes. They encounter storage and scalability issues that can lead to substantial costs. The main issue is that scaling vertically (adding more power to a single machine) has limits and can get expensive. Apart from cost, other issues preventing the scalability of data warehouse architecture to meet current demands include a lack of flexibility in supporting various types of workloads, such as handling unstructured data and performing machine learning tasks. Therefore, engineers have begun to explore other architectures that can address these challenges. This leads us nicely into the next section where we'll explore data lake architectures.

A Brief History of Data Lakes

Data lakes emerged as a solution to the shortcomings of traditional data warehouses. They started gaining traction in the mid-2000s, alongside the rise of open source software. Unlike their predecessors, data lakes introduced a new distributed architecture that could manage vast amounts of data in various states: unstructured, semi-structured, and structured. This flexibility expanded the usability of data.

Data lakes use open source software and, therefore, can run on any standard or affordable commodity hardware. This shift away from proprietary RDBMSs marked a significant change in how big data solutions were built, away from costly hardware clusters. Additionally, the integration of machine learning technologies into data lakes provides them with capabilities beyond the traditional reporting uses of data warehouses. For a visual understanding of how a data lake is structured, refer to the diagram in Figure 1-6.

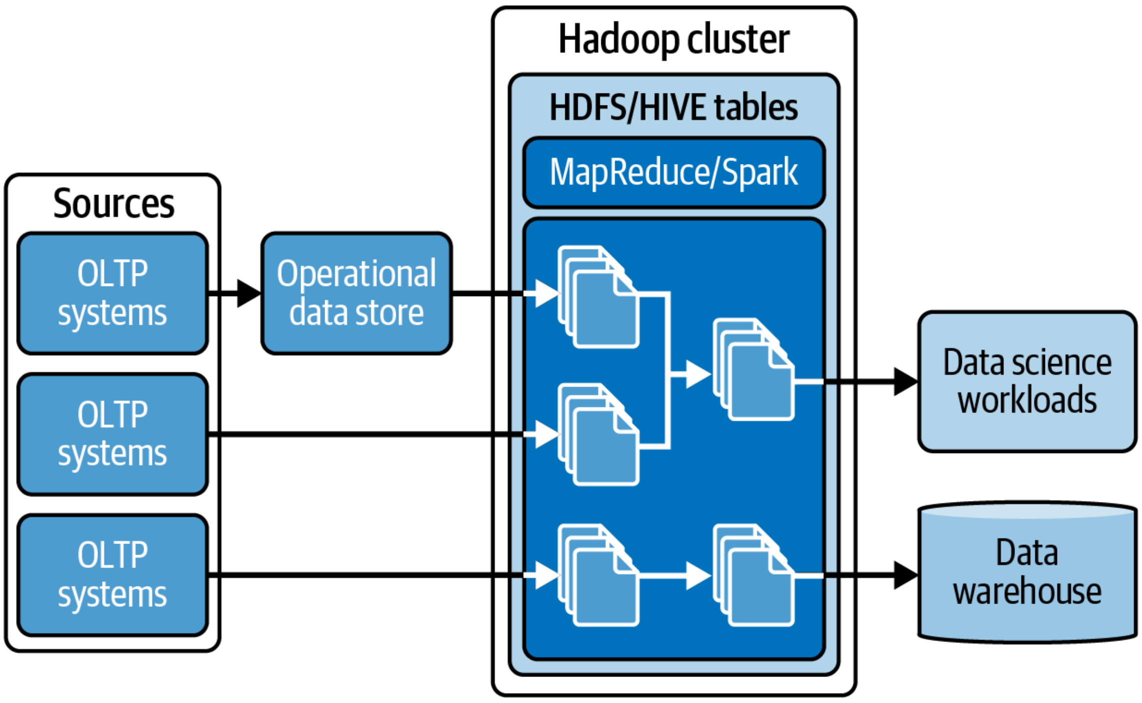

Figure 1-6. Typical data lake architecture with raw copies of data

The first generation of data lakes primarily used Hadoop, a well-known open source software framework that includes a variety of tools and services. At the heart of Hadoop are the MapReduce programming framework and Hadoop Distributed File System (HDFS),[6] which enables the processing of large datasets using a distributed algorithm across a cluster. Additionally, Hadoop comes with several other utilities that enhance its capability to store, process, and analyze massive amounts of data.

A key component worth mentioning is Apache Hive.[7] Developed on top of Hadoop, Hive is a data warehouse system that offers a SQL-like interface for querying and analyzing large datasets stored in the HDFS. One of Hadoop's strengths in the context of data lakes is its flexibility with data formats. Unlike traditional databases that demand a predefined schema, Hadoop with Hive supports a schema-on-read approach. This approach allows you to ingest and store data without a fixed structure and only define the schema when you read the data. This flexibility is invaluable for handling various types of data. We explore Hive in more detail in the section "Apache Hive".

Understanding Hadoop is crucial when comprehending modern data architectures because many Hadoop components, or at least concepts, are still present today. In the upcoming sections, we will cover the most relevant essentials. We'll start with the HDFS, move on to MapReduce, and then dive into Hive. We'll also discuss the limitations of using the HDFS and MapReduce. Lastly, we will discuss the emergence of Apache Spark, as it forms the backbone of many modern lakehouse architectures.

Hadoop's Distributed File System

The Hadoop Distributed File System is renowned for its fault tolerance and capacity to handle large datasets. It's important to note that the HDFS scales horizontally,[8] contrasting with the vertical scaling required by RDBMSs, thereby addressing issues of load and separating compute from storage.

Unlike the data management of RDBMS in data warehouses, the HDFS operates differently. It divides data into large chunks (blocks), which are then distributed and replicated across nodes within a network of computers. Each block is typically 128 MB or 256 MB in size, and the default replication factor is three. Consequently, reading and writing data (input/output, I/O, operations) in the HDFS can be time-consuming, especially if the data is not appropriately aligned and distributed across the nodes. Additionally, processing data in these files can present challenges; numerous small files can lead to excessive processing tasks, causing significant overhead. Because Hadoop is optimized for large files, logically grouping data enhances the efficiency of storage and processing. So, yes, it's advisable to use (denormalized) data models with Hadoop. Its data distribution process is very powerful but can be extremely inefficient if not managed properly.

In the HDFS, blocks are immutable, meaning you can only insert and append records, not directly update data. This differs from how data warehouse systems process individual mutations. Hadoop systems store all changes to data in an immutable write-ahead log before an asynchronous process updates the data in the data files. Then, how does this affect historic data within our data models? You may recall the concept of slowly changing dimensions from our discussion of the Kimball methodology. SCDs optionally preserve the history of changes to attributes, allowing us to query data at a specific point in time. If you would like to achieve this with Hadoop, you must develop a workaround that physically (re)creates a new version of the dimension table, including all the historic changes. You can do this by loading all data and creating a new table with the updated data. This process is resource-intensive and can be complex to manage. Therefore, it's crucial to consider the implications of using Hadoop for traditional data warehouse workloads.

Now, transitioning from how the HDFS manages data to how it's processed, let's talk about MapReduce.

MapReduce

MapReduce is a programming model designed to process data in parallel across a distributed cluster of computers. For many years, it served as the primary engine for big data processing in Hadoop. MapReduce uses three phases—map, shuffle, and reduce—to process jobs:

Map phase

During the map phase, the input data is divided into smaller chunks, which are processed in parallel across the nodes in the cluster. However, if the data is not evenly distributed across the nodes, some nodes may complete their tasks faster than others, potentially reducing overall performance.

Shuffle phase

During the shuffle phase, the output data from the map phase is sorted and partitioned before being transferred to the reduce phase. If the output data is voluminous and needs to be transferred across the network, this phase can be time-consuming.

Reduce phase

During the reduce phase, the previously shuffled data is aggregated and further processed in parallel across the nodes in the cluster. Similar to the map phase, if the reduce tasks are not evenly distributed across the nodes, some nodes may finish processing faster than others, which can lead to slower overall performance.

The map, shuffle, and reduce processes can lead to performance issues in Hadoop if the data is not evenly distributed across the nodes. Since data needs to be transferred across the network, it is crucial that tasks run efficiently. While MapReduce may not be directly used in the latest modern platforms, its concepts and approach to big data computation still form the basis for many of today's modern data architectures. Next, let's explore Apache Hive, which was originally built on the foundation laid by MapReduce.

Apache Hive

Initially developed by Facebook, Apache Hive provides a SQL layer over Hadoop, enabling users to query and analyze large datasets stored in Hadoop using a SQL-like language known as HiveQL. Hive employs MapReduce as its underlying execution engine to process queries and analyze data. It stores its data in the HDFS. For data querying, Hive translates HiveQL queries into MapReduce jobs, which are then executed on the Hadoop cluster. These jobs read data from the HDFS, process it, and write the output back to the HDFS, a process that involves significant disk I/O and network data transfer.

Hive differs significantly from traditional data warehouses in terms of data storage and querying. Unlike traditional data warehouses, where data is stored in a proprietary format and queries are executed directly on this data, Hive stores data on the HDFS and executes queries using MapReduce jobs. Additionally, the file formats in Hive use open formats that are available under open source licenses. To better understand how Hive stores and manages data, let's discuss the pattern of external and internal tables, followed by the Hive Metastore.

External and internal tables

In Apache Hive, a key distinction exists between external and internal tables. External tables link to data stored outside of Hive, typically in CSV or Parquet files on the HDFS. Parquet files on the HDFS. Hive does not control this data; it merely provides direct file-level access, allowing you to analyze the files. For example, you can mount a CSV (comma-separated values) file as an external table and query it directly.

Note

When you drop an external table in Hive, it only removes the metadata, leaving the underlying data intact. In contrast, dropping a managed table results in the deletion of both the table's metadata and its underlying data.

On the other hand, internal tables, also known as managed tables, are fully controlled by Hive. These tables often use columnar storage formats like ORC (Optimized Row Columnar) and Parquet, which are prevalent in many modern Medallion architectures. These formats are particularly beneficial for analytical queries involving aggregations, filtering, and sorting of large datasets. They enhance performance and efficiency by drastically reducing I/O operations and the amount of data loaded into memory. Additionally, columnar formats offer better data compression, which saves storage space and reduces the costs associated with managing large volumes of data.

Hive Metastore

A key component in Hive is the Hive Metastore, a central repository that stores metadata about the tables, columns, and partitions in an HDFS cluster. This metadata includes the data schema, the data location in the HDFS, and other information necessary for querying and processing the data. This component is still present in many of today's Medallion architectures.

Hive, through its metastore, allows for data to be ingested without the strict requirement of first defining a schema. Instead, the schema is applied dynamically when the data is accessed for reading. This method is also known as "schema on read." This flexibility is in stark contrast to the "schema-on-write" methodology prevalent in traditional databases, where data must conform to a predefined schema at the time of writing.

Warning

The schema-on-read approach, still present in modern data architectures, often leads to misunderstandings. Some engineers mistakenly believe that schema on read eliminates the need for data modeling. This is a significant misconception! Without proper data modeling, data will be incomplete or of low quality, and integrating data from multiple sources becomes challenging. Inadequate data modeling can also lead to poor performance. While the schema-on-read approach is helpful for quickly storing and discovering raw data, it's still necessary to ensure data quality, integration, and performance.

Hive, along with its metadata, the HDFS, and MapReduce, initially faced some challenges. The first issue concerned the efficient handling of many small files. In the HDFS, data is spread across multiple machines and is replicated to enhance parallel processing. Each file, regardless of its size, occupies a minimum default block size in memory because data and metadata are stored separately. Small files, which are smaller than one HDFS block size (typically 128 MB), can place excessive pressure on the NameNode.[9] For instance, if you are dealing with 128 MB blocks, you would have around 8,000 files for 1 TB of data, requiring 1.6 MB for metadata. However, if that 1 TB were stored as 1 KB files, you would need 200 GB for metadata, placing a load on the system that is 1,280 times greater. Such issues can drastically reduce the read performance of the entire data lake if not managed correctly.

Second, the first version of Hive didn't support ACID transactions and full-table updates, risking that the database would get into an inconsistent state. Fortunately, this issue has been addressed in later versions. In "Emergence of Open Table Formats", I'll come back to this point when covering the Delta table format.

Third, MapReduce can be quite slow. At every stage of processing, data is read from and written back to the disk. This process of disk seeking is time-consuming and significantly slows down the overall operation. This performance issue brings us to Apache Spark,[10] which tries to overcome this performance challenge.

Spark Project

MapReduce, despite its benefits, presented certain inefficiencies, particularly when it came to large-scale applications. For instance, a typical machine learning algorithm might need to make multiple passes over the same dataset, and each pass had to be written as a unique MapReduce job. These jobs had to be individually launched on the cluster, requiring the data to be loaded from scratch each time.

To address these challenges, researchers at UC Berkeley, specifically from the AMPLab, initiated a research project in 2009 to explore ways to speed up processing jobs in Hadoop systems. They developed an in-memory computing framework known as Spark. This framework was designed to facilitate large-scale data processing more efficiently by storing data in memory rather than reading it from disk for every operation. This team also developed Shark,[11] an extension of Spark designed to handle SQL queries, thereby enabling more interactive use by data scientists and analysts. Shark's architecture was based on Hive. It converted the physical plan generated by Hive into an in-memory program, enabling SQL queries to run significantly faster (up to 100 times) than they would on Hive using MapReduce.

As Spark evolved, it became apparent that incorporating new libraries could greatly enhance its capabilities. Consequently, the project began to adopt a "standard library" approach.[12] Around this time, the team began to phase out Shark in favor of Spark SQL, a new component that maintained compatibility with Hive by using the Hive Metastore.

However, the speed improvements offered by Spark come with certain preconditions. For instance, Spark needs to read data from the disks to bring it into memory, and this is not an instantaneous process. So, after data is written to the HDFS, an additional loading process is required to read it from the disks and bring it into Spark's memory. This principle still holds true today. For example, when restarting the Spark cluster, all in-memory data is lost, and the data must be reloaded to regain the speed benefits. For building modern architectures, this means that there's typically a startup time before resources become available,[13] and during this period, data isn't immediately present in Spark. The data only becomes readily available for quick usage when querying and/or caching processes are initiated.

Let's park the discussion on Spark for now and move on to the learnings from data lakes and their evolution. We'll pick up the Spark discussion again when we delve into lakehouse architecture in the next section.

Moving Forward with Data Lakes

What can you learn from data lakes and their evolution? Let's consider a few key points.

Data lakes, powered by Hadoop, are robust solutions for storing massive volumes of raw data in various formats, both structured and unstructured. This data is readily available for processing in data science and machine learning applications, accommodating data formats that a traditional data warehouse cannot handle. Unlike traditional data warehouses, data lakes are not restricted to specific formats. They rely on open source formats like Parquet, which are widely recognized by numerous tools, drivers, and libraries, ensuring seamless interoperability. Moreover, many of the core concepts, such as external and managed tables, still exist in modern data architectures.

However, as data lakes gain popularity and broader usage, organizations have begun to notice some challenges. While ingesting raw data is straightforward, transforming it into a form that can deliver business value is quite complex. Traditional data lakes struggle with latency and query performance, necessitating a different approach to data modeling to take advantage of the distributed nature of data lakes and their ability to handle various data types flexibly.

Furthermore, traditional data lakes face their own set of challenges, such as handling large numbers of small files or providing transactional support. As a result, organizations have often relied on feeding data back into the traditional data warehouse, a two-tier architecture pattern in which a data lake stores data in a format compatible with common machine learning tools, from which subsets are loaded into the data warehouse.

To tackle these challenges, the industry has been shifting toward integrating the two-tier architecture into a single solution. This new architecture combines the best of both worlds, offering the scalability and flexibility of a data lake along with the reliability and performance of a data warehouse. To better understand how this integration has evolved, it's essential to explore the history and development of the lakehouse architecture.

A Brief History of Lakehouse Architecture

Now that we've discussed the history of data warehouses and data lakes, we've come to the last part of our discussion on the evolution of data architectures. Let's take a look at today's architectures that use lakehouses as the foundation with open source data table formats, such as Delta Lake. For this, we look at the evolution of Spark after it was launched. After that, we'll discuss the origin of Databricks, its role in the data space, and its relationship to other technology providers. Then, finally, we'll look at the Medallion architecture.

Founders of Spark

By 2013, the Spark project, with contributions from over 100 contributors across 30 organizations, had significantly grown in popularity. To ensure its long-term sustainability and vendor independence, the team decided to contribute Spark as open source to the Apache Software Foundation. Accordingly, Spark became Apache Spark: an Apache top-level project.

In 2013, the creators of Spark founded a company named Databricks to support and monetize Spark's rapid growth. Databricks aims to simplify big data processing, making it more accessible to data engineers, data scientists, and business analysts. Following this, the Apache Spark community launched several versions: Spark 1.0 in 2014, Spark 2.0 in 2016, Spark 3.0 in 2020, and Spark 4.0 in 2025. They continue to enhance Spark by regularly introducing new features.

Interestingly, Databricks adopted a different market strategy from its Hadoop competitors. Instead of focusing on on-premises deployments, like Cloudera and Hortonworks, Databricks opted for a cloud-only distribution called Databricks Cloud. At that time, there was even a free Community Edition. Databricks started first with Amazon Web Services, the most popular cloud service at that time, and later included Microsoft Azure and Google Cloud Platform. In November 2017, Databricks was announced as a first-party service on Microsoft Azure via its integration, Azure Databricks. Over the years, the pace of cloud adoption accelerated, and cloud-based solutions with decoupled storage and compute gained popularity, leading to the decline of traditional on-premises Hadoop deployments. Today, we can confidently say that Databricks made a smart strategic move.

What does this mean for Hadoop? Has Hadoop become obsolete? No, it's still alive in the cloud ecosystem, though it has undergone significant changes. Vendors replaced the HDFS with cloud-based object storage services. With object storage, the data blocks of a file are kept together as an object, together with its relevant metadata and a unique identifier. This differs from the HDFS, where data is stored in blocks across different nodes, and a separate metadata service (like the NameNode in the HDFS) tracks the location of these blocks.

The switch to cloud-based object storage brings several advantages. Not only is it generally less expensive for storing large volumes of data, but it also scales more efficiently, even up to petabytes. Every major cloud provider offers such services, complete with robust service-level agreements (SLAs) and options for geographical replication. For example, Microsoft has introduced Azure Data Lake Storage, an object storage solution that maintains compatibility with the HDFS interface while modernizing the underlying storage architecture. To sum it up: while the HDFS interface is still in place, the underlying storage architecture has undergone a major overhaul.

The same evolution has happened with Spark. It received many contributions, predominantly from Databricks. Nowadays, it can operate independently in a cluster of virtual machines or within containers managed by Kubernetes. This flexibility means you aren't tied to a single, massive Hadoop cluster. Instead, you can create multiple Spark clusters, each with its own compute configuration and size. Depending on your needs, these clusters can all interact with the same object storage layer, making Spark both elastic and dynamic.

Databricks is the leading force behind Apache Spark's roadmap and development. It offers a managed platform, whereby users get the full benefits from Spark by not having to learn complex cluster management concepts or perform endless engineering tasks. Instead, users navigate through a user-friendly interface. Companies using Databricks will also benefit from its latest innovations.

Let's shift gears a little bit and move on to today's modern architectures, which have seen significant advancements thanks to the development of open source table format standards like Hudi, Iceberg, and Delta Lake.

Emergence of Open Table Formats

Recognizing the critical need for improved transactional guarantees, enhanced metadata handling, and stronger data integrity within columnar storage formats, several projects were developed and later became open source. Apache Hudi, initiated by Uber, was one of the first to emerge in 2017. It was designed to simplify the management of large datasets on Hadoop-compatible filesystems, focusing on efficient upserts, deletes, and incremental processing. This project set the stage for further innovations in handling big data. Moreover, Hudi offers seamless integration with existing storage solutions and supports popular columnar file formats such as Parquet and ORC.

Following closely, in 2018, Netflix released Apache Iceberg to tackle performance and complexity issues in large-scale analytics data systems. Iceberg introduced a table format that improved slow operations and error-prone processes, enhancing data handling capabilities. It has gained popularity due to its rich feature set and ability to support Parquet, ORC, and Avro file formats.

In 2019, Databricks launched Delta Lake to further address the challenges in traditional data lakes, such as the lack of transactional guarantees and consistency problems. Delta Lake brought ACID transactions, scalable metadata handling, and unified streaming and batch data processing, all while ensuring data integrity through schema enforcement and evolution. Delta Lake exclusively utilizes the Parquet file format for data storage and employs Snappy as the default compression algorithm.

Apache Hudi, Apache Iceberg, and Delta Lake

In 2024, Databricks made a strategic decision by acquiring Tabular, a company that supports the Apache Iceberg initiative, one of the leading open source lakehouse table formats. The main objective of this acquisition is to enable compatibility between various lakehouse platforms. To achieve this compatibility in the short term, Delta Lake has introduced UniForm, which allows you to write data primarily to Delta Lake and then asynchronously generate the metadata for Apache Iceberg or Hudi. In the long run, Databricks is committed to developing a single, open, and common standard of interoperability across platforms, promising a more cohesive and streamlined approach to data management. Beside this development, there's also another initiative called Apache XTable. This originated from the founders of Apache Hudi, another lakehouse table format. Apache XTable provides true unidirectional conversion between the three formats (Delta, Hudi, and Iceberg).

Delta Lake provides ACID transactions through a transaction log (also known as the DeltaLog). Whenever a user performs an operation to modify a table (such as an insertion, update, or deletion), Delta Lake breaks that operation down into a series of discrete steps composed of one or more of the actions below. Those actions are then recorded in the transaction log as ordered, atomic units known as *commits*. The transaction log is automatically stored in a *_delta_log/* subdirectory among the Parquet files for a particular table. In Figure 1-7, you can see an example of the DeltaLog.

In Delta Lake, every commit is recorded in a JSON file, beginning with *000000.json* and continuing sequentially. As you update your tables, Delta Lake preserves all previous versions.[14] This feature, known as "time travel," allows you to view the state of a table at any specific point in time. For instance, you can easily check how a table appeared before it was updated or see its state at a particular moment. To learn more, I encourage you to read "Diving Into Delta Lake: Unpacking the Transaction Log".

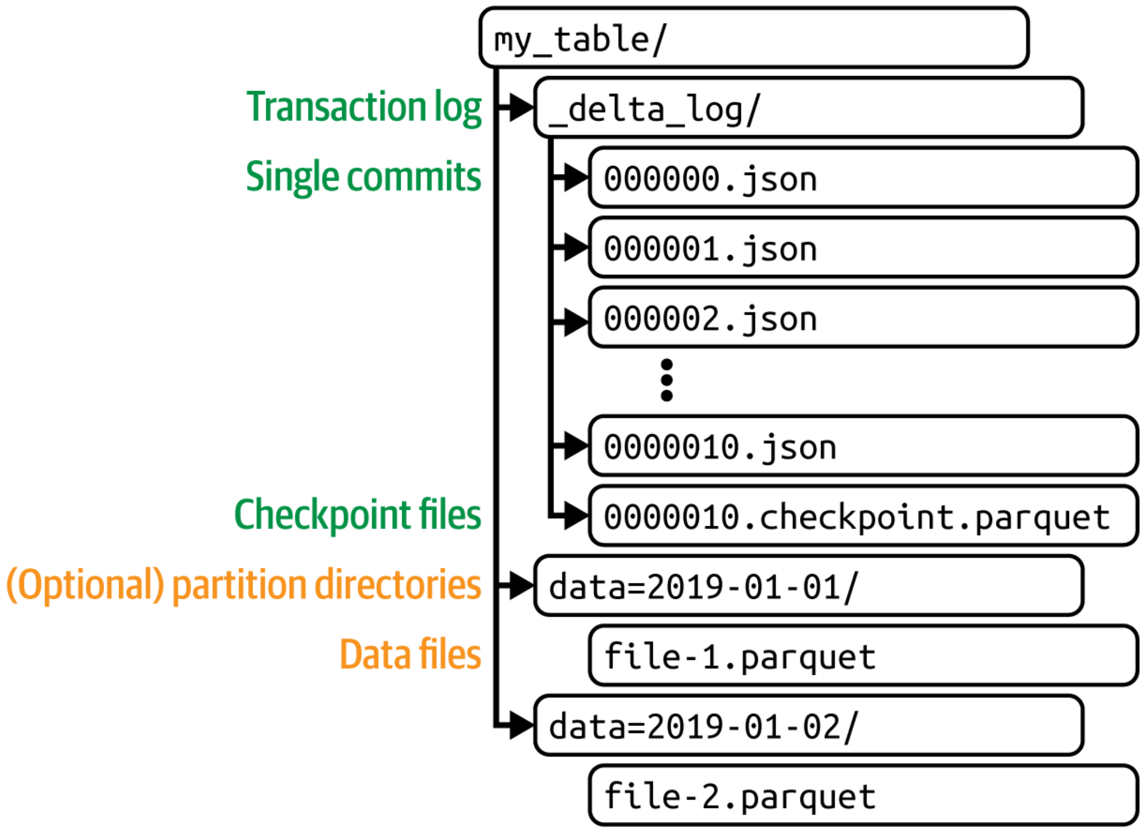

Figure 1-7. Example of how Delta Lake structures its data and transaction log

The Rise of Lakehouse Architectures

As Delta Lake made its debut, the *lakehouse architecture* concept began to gain traction. This innovative model combines the benefits of both data lakes and data warehouses. It allows organizations to operate on a unified data platform, primarily using open source software as its foundation. While the concept of the lakehouse is not inherently tied to any specific technology, the most popular implementations of the lakehouse architecture are built around Apache Spark and Delta Lake. So, Spark delivers the compute for big data processing, while Delta Lake provides an open source storage layer. Figure 1-8 offers an overview of what a lakehouse entails.

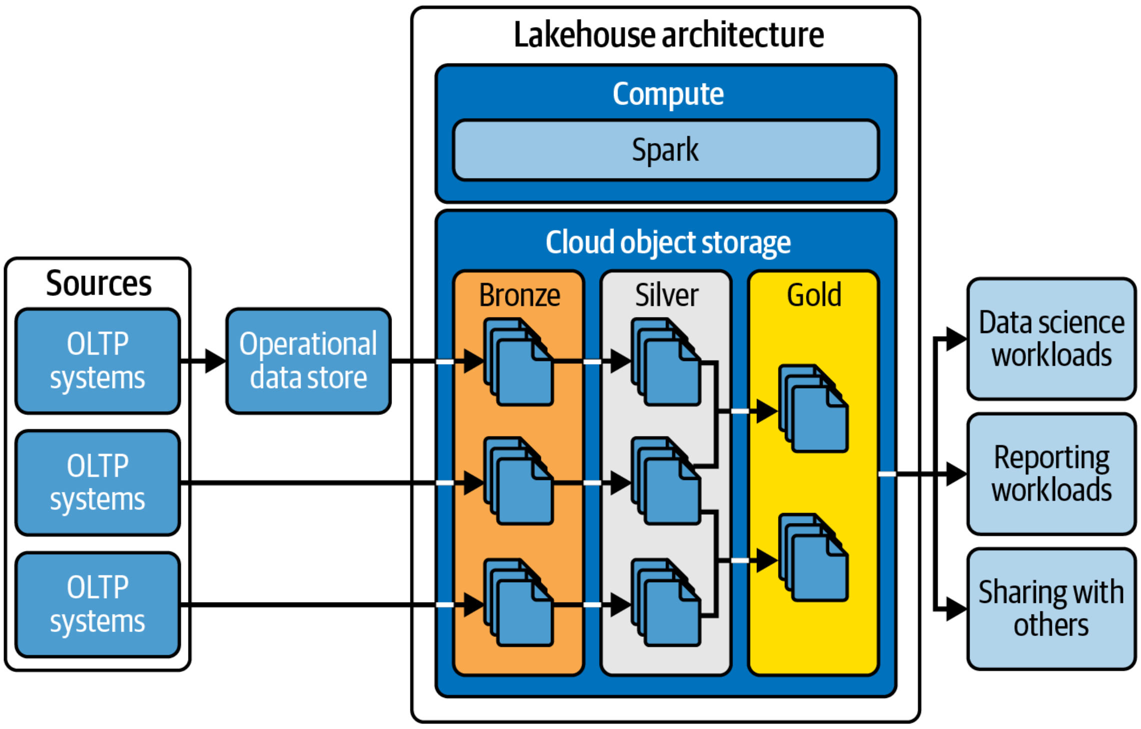

Figure 1-8. Typical lakehouse architecture, with a representation of the Bronze, Silver, and Gold layers included

The lakehouse architecture distinguishes itself from previous architectures by supporting low-cost cloud object storage while simultaneously providing ACID transactions. Moreover, it significantly enhances performance compared to traditional data lakes, largely due to innovations stemming from Apache Spark. Databricks was the pioneer in coining the term *lakehouse* and positioning itself within this product space. Subsequently, other major players followed quickly. Here is an overview of the current landscape as of 2025:

Databricks

As a strong advocate for lakehouse architecture, Databricks integrates Delta Lake, which supports ACID transactions and scalable metadata management. This table format pairs seamlessly with Apache Spark, enhancing big data processing and analytics for improved performance and reliability.

Azure HDInsight

Microsoft's cloud-based service, Azure HDInsight, offers a managed Apache Hadoop and Apache Spark service, providing a scalable and efficient environment for big data processing. It supports various table formats and integrates with other Azure services for enhanced data analytics.

Azure Synapse Analytics

A Microsoft service, Azure Synapse Analytics merges big data solutions with data warehousing into a unified analytics service. It offers flexible querying options through SQL serverless on-demand or provisioned resources, optimizing the management and analysis of extensive datasets.

Microsoft Fabric

This analytics and data platform is provided as a software as a service (SaaS) experience. Microsoft Fabric utilizes Apache Spark and Delta Lake, along with a suite of other services, to facilitate a wide range of data operations and analytics.

Cloudera

Cloudera provides a robust platform supporting various table formats and data processing frameworks. With strong integration capabilities for Apache Hadoop and Apache Spark, Cloudera offers a flexible environment suitable for constructing diverse lakehouse architectures.

Dremio

Leveraging Apache Arrow, Dremio enhances in-memory data operations across multiple languages. This platform excels at efficient data retrieval and manipulation, making it ideal for direct data lake exploration and analysis.

Starburst

Specializing in Trino, an open source distributed SQL query engine, Starburst delivers fast and scalable data analytics across numerous data sources. It supports a range of table formats and integrates seamlessly with other lakehouse technologies to boost query performance.

In addition to the vendors previously mentioned, other major players like Amazon Web Services, Google Cloud Platform, and Snowflake have also started to incorporate the term "lakehouse" within their offerings. This trend underscores the increasing recognition and adoption of lakehouse architecture within the data management industry. Tech giants recognize the value of blending the best aspects of data lakes and warehouses, leading to more comprehensive and efficient data solutions. As more organizations look to optimize their data handling and analytics capabilities, the lakehouse model continues to gain traction as a preferred architecture, shaping the future of data management.

Finally, Databricks and Microsoft have endorsed the Medallion architecture as a best practice for layering data within architectures based on Spark and Delta Lake. This approach is also the central theme of this book. This design pattern organizes data within a lakehouse, aiming to enhance the structure and quality of data incrementally as it moves through the architecture's layers—from Bronze to Silver to Gold. In the following section, we will delve deeper into the practical challenges encountered in implementing the Medallion architecture. After addressing these challenges, we will conclude this chapter and move on to explore the fundamentals of the Medallion architecture in Chapter 2 and design patterns in Chapter 3.

Medallion Architecture and Its Practical Challenges

The Medallion architecture, originally coined by Databricks, is not an evolution of any existing architectures. Instead, it is a data design pattern that offers a logical and structured approach to organizing data in a lakehouse. The term derives from its three distinct layers: Bronze, Silver, and Gold, which are user-friendly labels for managing data, similar to layering data in data warehouses or data lakes. The progression from Bronze to Gold signifies not only an improvement in data quality but also in structure and validation.

Despite the intuitive labels of the Medallion architecture—Bronze for raw data, Silver for cleaned data, and Gold for consumer-ready data—practical guidance is missing. This gap highlights a broader issue: there is no consensus on the specific roles of each layer, and the terms themselves lack descriptive precision. As we conclude that data modeling remains crucial, it's clear that while the naming convention offers a starting point, the real challenge lies in its practical application and the variability between theoretical guidance and actual execution, a central topic explored in this book.

In Chapter 2, we go over some of the foundational concepts that will help you navigate the Medallion architecture, including landing zones, raw data, batch processing, and ETL and orchestration tools. Then, in Chapter 3, we delve deeper into the Medallion architecture by discussing every layer in detail. This thorough examination will provide a clearer understanding of how data progresses through these layers and the challenges and considerations involved in effectively applying this architecture in real-world scenarios.

Conclusion

We began our exploration of the evolution of data architectures with traditional data warehouses and OLTP systems. From there, we moved on to the emergence of Hadoop and data lakes, and finally to the innovative lakehouse model. Each step in this evolution has been driven by the need to address specific limitations of the previous architectures, particularly in handling the scale, diversity, and complexity of modern data.

This evolution has also changed the way we can manage and control today's data architectures. With traditional data warehouses running on-premises, we had the ability to change the hardware configuration, the network, and the storage. However, with the rise of cloud computing, we have seen a shift toward fully distributed and managed services, which has made it easier to scale up and down but has also made it harder to control the underlying infrastructure. Proper configuration, layering, and data design are crucial in this context. For example, we have already emphasized the importance of data modeling in architecture design, and it remains a fundamental requirement for the successful design of modern data architectures. In the next chapters, we will delve deeper into these aspects.

Another important evolution is the speed at which business users expect new projects and insights to be delivered to them. This presents a challenge for the delivery of modern data architectures, as development teams face constant pressure to deliver rapid insights. Many organizations fail to recognize the indispensable need for data modeling. Data mesh approaches often overlook this necessity,[15] leading to a recurring problem where distributed teams repeatedly create slightly varied, incompatible models. These variations become entrenched in analytics models, ETL pipelines, data products, and application codes, turning what was once a clear and explicit design into something obscured and siloed.

The Medallion architecture within these platforms recognizes these issues but falls short of offering a definitive solution. The practical application of this design pattern exposes a gap between theoretical models and their real-world implementations and examples. This discrepancy underscores the ongoing need for precise data modeling and governance strategies tailored to the specific needs of organizations.

Moving forward, it is crucial for data architects and engineers to continue exploring these models, understanding their intricacies, and applying them thoughtfully to meet the increasing demands of big data environments. Chapter 2 will provide a detailed overview of the foundational preconditions required for building modern data architectures. By establishing a strong foundation, we prepare ourselves for more advanced discussions in Chapter 3, which will delve deeper into the Medallion architecture, providing more detailed insights into each layer.

1 This quote is from Benn Stancil's article, "Microsoft Builds the Bomb", which discusses market-wide challenges in data platform solutions.

2 ACID principles ensure reliable database transactions by making them indivisible, consistent, isolated, and durable, which is crucial for maintaining data integrity, especially in critical systems like finance and operations.

3 Applications, such as web services, that retrieve extensive data for a single observation do not always necessarily pose a problem.

4 Data virtualization is a technology that allows you to manage and manipulate data without needing to copy and export the data. Essentially, it creates a virtual layer that separates users from the technical details of data, such as its location, structure, or origin.

5 Ralph Kimball introduced the data warehouse/business intelligence industry to dimensional modeling in 1996 with his seminal book, *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses* (John Wiley & Sons).

6 Hadoop originated from Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung's paper, "The Google File System", published in 2003. This was followed by a second influential paper, "MapReduce: Simplified Data Processing on Large Clusters", by Jeffrey Dean and Sanjay Ghemawat.

7 Apache Hive was developed by Facebook based on the ideas presented in the research paper by Ashish Thusoo et al. titled "Hive: A Warehousing Solution Over a Map-Reduce Framework".

8 Horizontal scaling involves adding more machines or nodes to a system to handle increased load, distributing the workload across multiple servers. Vertical scaling involves adding more resources (such as CPUs, RAM, or storage) to an existing machine to enhance its capacity and performance.

9 The NameNode in Hadoop is responsible for managing the filesystem namespace, storing metadata for all files and directories, and regulating access to files by clients. It also manages the mapping of file blocks to DataNodes, ensuring data reliability and availability.

10 Modern Spark runtimes have features that carefully handle the small files problem. Miles Cole has posted an optimization guide for Microsoft Fabric, providing more background information.

11 See the research document from AMPLab at UC Berkeley on Apache Shark.

12 See the 2014 post by Reynold Xin, "Shark, Spark SQL, Hive on Spark, and the Future of SQL on Apache Spark".

13 In modern lakehouse architectures, the cold start time for querying or processing data can be significantly reduced by using sidecar files. These files contain metadata such as

schema information, statistics, and indexing data, enabling more efficient data management and query execution.

14 Every previous version is recorded in Delta Lake. However, when you perform an upsert or delete, the older versions stay put until the vacuum process kicks in. But Delta Lake isn't actually deleting any data immediately; it simply removes data that the current snapshot of the table no longer references. You can configure the intervals for both vacuuming and deleting. For a detailed explanation, check out Chapter 5 of *Delta Lake: The Definitive Guide* (O'Reilly), titled "Maintaining your Delta Lake."

15 Data mesh is a decentralized approach to data architecture and organizational design. It treats data as a product, focusing on domain-oriented ownership, data as a product, self-serve data infrastructure, and computational governance.