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.

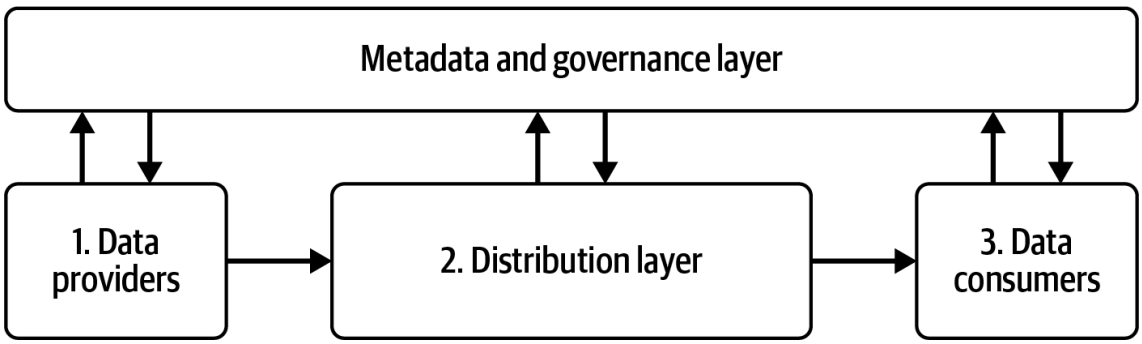

Figure 1-1. The three-layered architecture design

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.

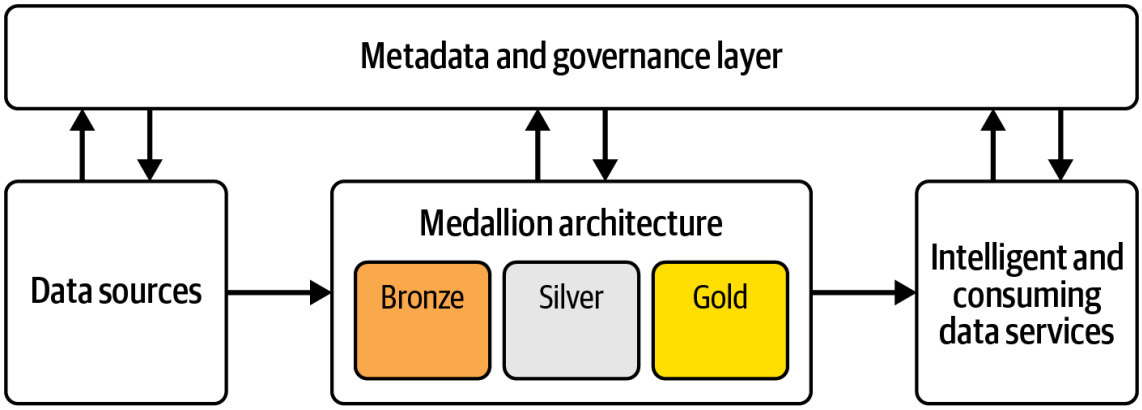

Figure 1-2. A Medallion architecture, which arranges data into three layers, enhancing the data's structure and quality as it progresses through the layers

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.

A Brief History of Data Warehouse Architecture

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.

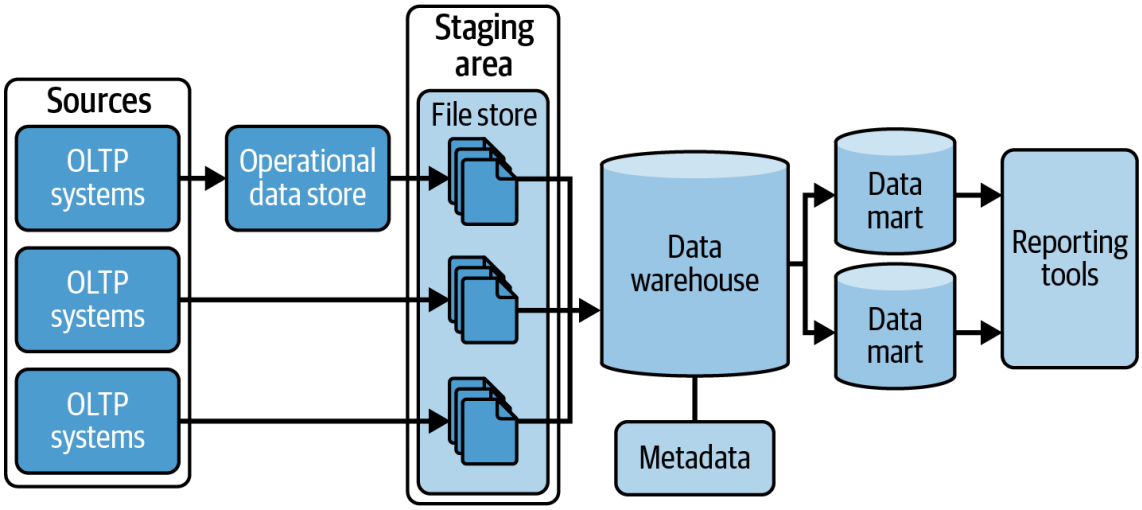

Figure 1-3. Typical data warehouse architecture

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.

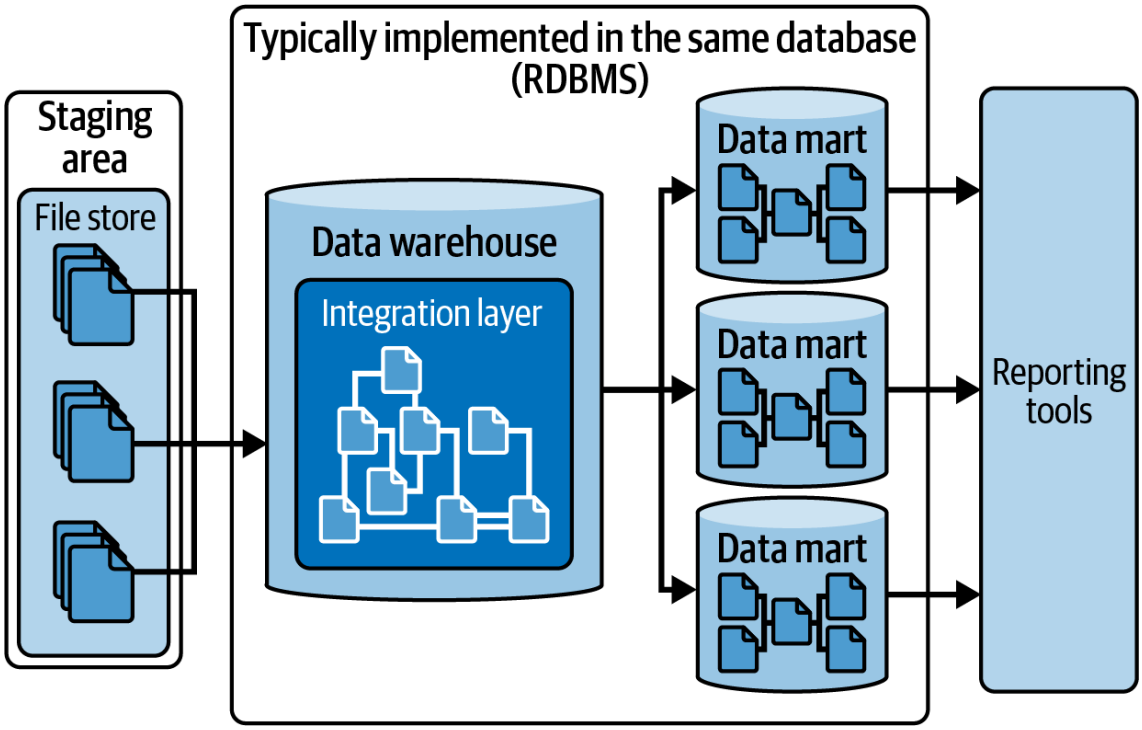

Figure 1-4. The Inmon approach; a top-down design where a centralized data warehouse is built first, and then data marts are created from this central warehouse

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.

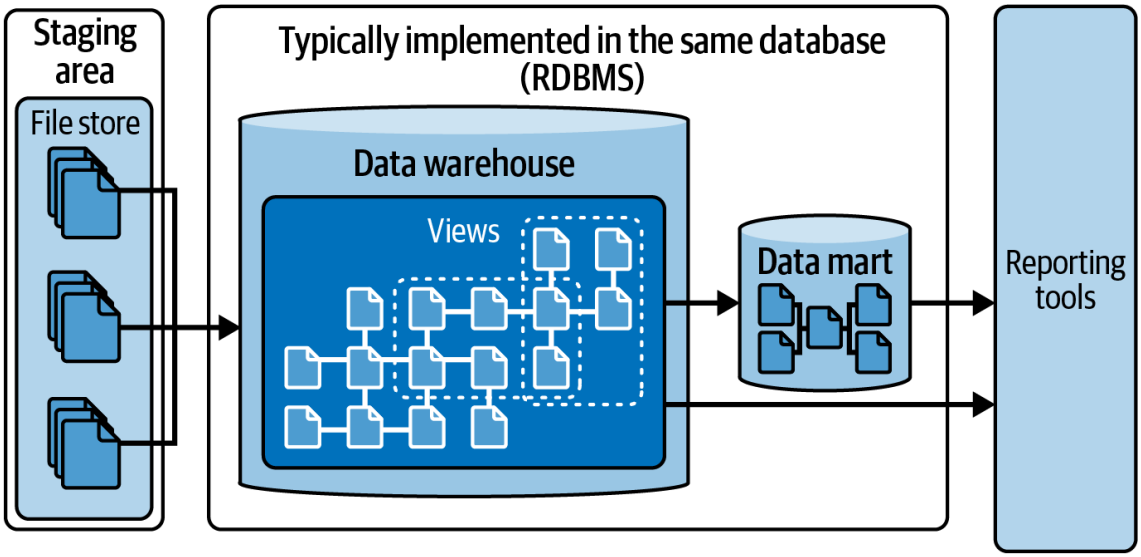

Figure 1-5. The Kimball methodology; a bottom-up approach to building the data warehouse

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.

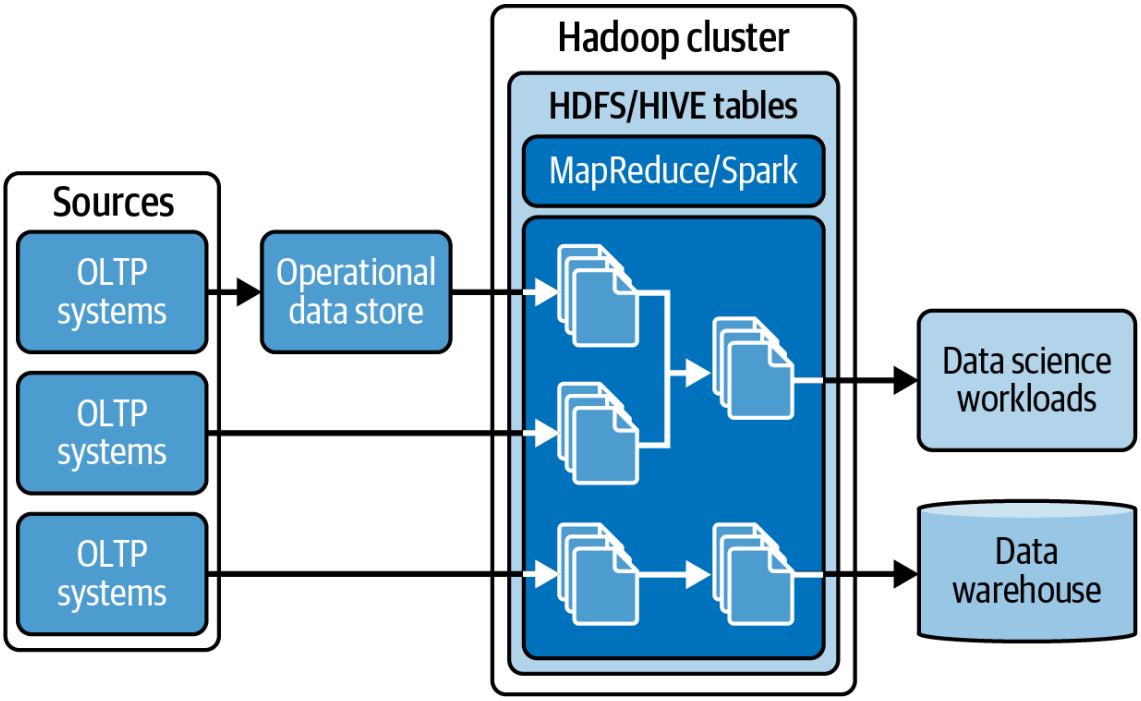

Figure 1-6. Typical data lake architecture with raw copies of data

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".

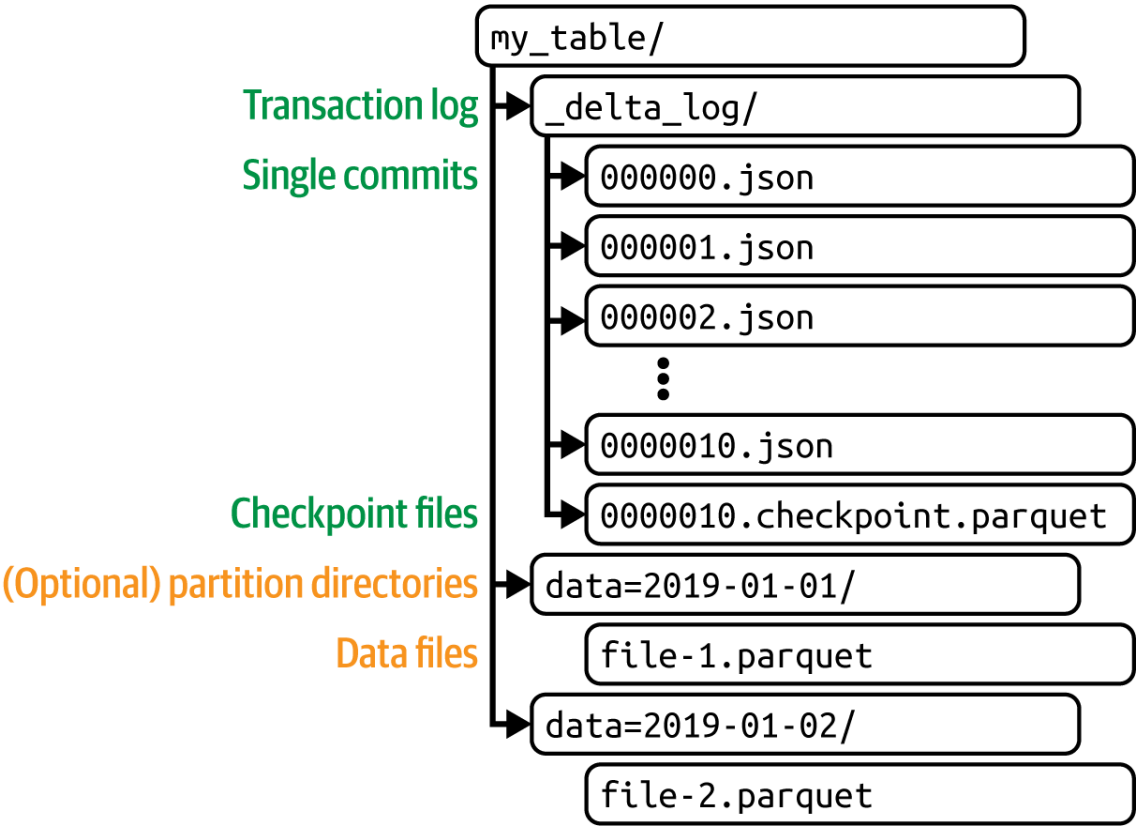

Figure 1-7. Example of how Delta Lake structures its data and transaction log

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.

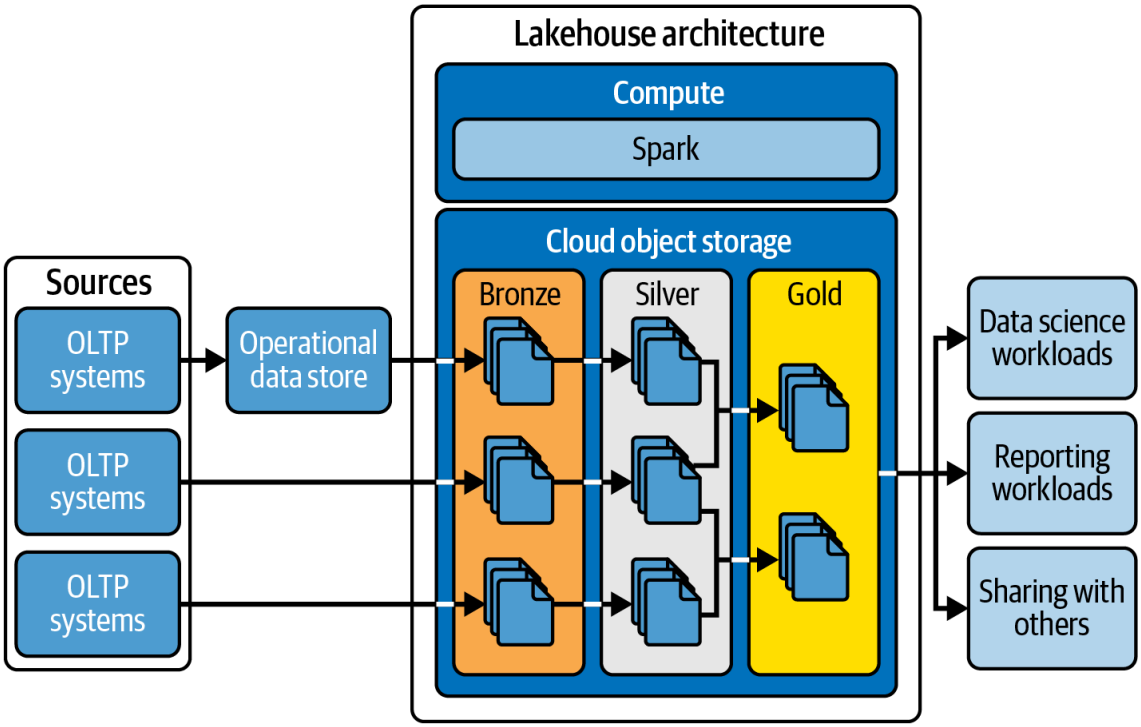

Figure 1-8. Typical lakehouse architecture, with a representation of the Bronze, Silver, and Gold layers included

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.

Chapter 2. Laying the Groundwork

Before building a house, you need to first establish a solid foundation. The same idea applies to Medallion architecture. This chapter serves as a preparatory bridge, introducing key components and patterns that recur throughout our discussions of Medallion architectures. It also sets the stage for Chapter 3, in which we'll dive deeply into the Medallion architecture and its layers.

In this chapter, we cover several core areas:

Extra landing zones

Preliminary areas where raw data is ingested before it lands in the Medallion architecture

Raw data

The unprocessed data collected from various sources, which forms the basis for further transformations and analysis

Batch processing

A method of data processing where data is collected, processed, and then output in batches at scheduled intervals

Real-time data processing

In contrast to batch processing, this involves processing data as soon as it becomes available, enabling immediate analysis and decision making

ETL and orchestration tools

Integral for extracting, transforming, and loading data, and play a crucial role in orchestrating and automating workflows within the data ecosystem

The inclusion of these components in the Medallion architecture varies. Some practitioners draw them directly within the main architectural framework, while others prefer to position them outside the diagrams or at the periphery, emphasizing their supportive role.

We will also explore the management of Delta tables. Understanding how to effectively manage these tables is crucial for maintaining the integrity and efficiency of your data processes across the different Medallion layers.

By the end of this chapter, you should have a solid foundational understanding of these key elements, preparing you for a deeper dive into the Medallion architecture and its operational layers in subsequent chapters.

Foundational Preconditions

Before you can dive into designing and implementing any architecture, you need data. Therefore, to kick off designing and implementing a new architecture, the first step always begins with pinpointing target source systems and figuring out the best way to gather data from them. In this context, a major challenge involves whether it's necessary to use intermediate landing zones for data ingestion, which can facilitate better organization and transformation before the data integrates into the Medallion architecture. Following that, you need to decide on the data ingestion methods—should you go with batch processing or

real-time processing? This decision will influence the tools and techniques you choose for data integration, orchestration, and table management.

In the following sections, we'll explore these decisions in the order previously mentioned. This discussion will build a solid foundation for you to understand the Bronze, Silver, and Gold layers that we will cover in detail in Chapter 3. Let's start by discussing the potential need for additional landing zones.

Extra Landing Zones

A "landing zone" is often used as a preliminary area where raw data is ingested before it lands in the Medallion architecture. Various factors influence this choice, such as the characteristics of the data source.

For example, when dealing with external services or SaaS vendors, you might need a secure landing zone to initially store data before moving it to the Bronze layer. Similarly, certain application teams with strict data ingestion requirements may require their own dedicated landing zones. These teams manage the extraction process, ensuring there are no unexpected errors or data inconsistencies. They might also require specific ETL tools or deployments of integration runtimes for this purpose. In such cases, it is common to have multiple landing areas between your sources and the lakehouse architecture.

Conversely, direct data ingestion into the Bronze layer is feasible depending on the requirements of the source system. For example, if the source system has robust security measures and stable direct ingestion techniques, it may be more efficient to bypass intermediate landing zones and feed data directly into the Bronze layer. In these instances, there is no need to stage the data before it is ingested into the Bronze layer. Consequently, the approach to data ingestion can differ significantly depending on the characteristics of the data sources and the particular needs of the data management teams.

Landing Zones Takeaway

While the Bronze layer mainly serves as the area for data collection, you may need an extra preliminary landing zone in larger or more complex environments. It's up to you to decide whether to label this layer as Bronze. Some organizations might call it a "tin layer," "pre-Bronze," "pre-refinement," "staging," or "landing area" and place it outside the typical Medallion architecture. Others might include it in the Bronze layer design. The important thing is to ingest and store data in a way that meets your organization's specific functional and nonfunctional requirements.

Let's now explore some more nuances, focusing on handling raw data, followed by the different types of ingestion methods.

Raw Data

When discussing data collection with the goal of having the data land in the Bronze layer, using terms like "raw" or "as-is," you need to be careful. Let's consider complex transactional systems that involve vendor packages with thousands of tables. In these cases, an engineer might need to step in and mediate during the data extraction process.

Take an example from my experience as an architect. I had to handle data extraction from Temenos T24, which is a core banking system. This system stores all its data in XML format, with each table having just two columns: RECID for the primary key and XMLRECORD for

the data. Extracting this data directly would lead to a very complex and unmanageable dataset. To solve this, we used the T24 Data Extractor. This tool pulls the data from the T24 database and stores it in a more manageable data structure. At a later stage, we transitioned to using Apache Flink to decode and process the data in motion, allowing us to transform and enrich the XML records as they were streamed from T24 to other environments.

Similarly, complex enterprise resource planning systems, with their tens of thousands of specialized and interconnected tables, pose significant challenges for direct data extraction. Organizations often use extraction services or tap into semantic models provided by these commercial packages to mediate this process.

What can you learn from these examples? There could be mediation applied. The first step from technical system tables to ingestion sometimes involves a middleware or custom-built software component. Its job is to map complex system tables or proprietary data structures into a simpler and more manageable data model.

Raw Data Takeaway

Preprocessing complex data or applying mediation happens before the data reaches the the first layer. This is not considered a transformation within the Medallion architecture. Therefore, while the data in the first layer is typically "raw," preprocessing can be crucial for manageability and clarity in cases involving complex systems.

Now that we've covered the initial essentials, let's shift our focus to another crucial aspect of data management—whether to opt for batch or real-time ingestion. Understanding the differences between these methods will further enhance our approach to handling data efficiently. We'll start with batch processing, followed by a longer discussion on real-time data and streaming ingestion because of its complexity.

Batch Processing

Batch processing is a method in which data is collected over a period and then processed all at once. This approach remains popular today for several reasons. First, it's cost-effective because it allows you to process large datasets in a single operation. This reduces the need for continuous processing and keeps components from running nonstop. Additionally, many existing systems and infrastructures widely support batch processing. It's also particularly advantageous when building historical perspectives, as it processes all accumulated data in one go.

When setting up batch data ingestion within your Bronze layer, you need to keep a few key considerations in mind:

- Maintaining data integrity is crucial. Use validation methods like row counts, checksums, or hash totals to ensure data accuracy and completeness during transfers. Also, track auditing information such as file counts, data volume, copy duration, activity run IDs, and outcomes to facilitate monitoring and troubleshooting.
- Unlike continuous processing, batch data processing occurs at set intervals. Select these intervals carefully, considering data source availability, the processing sequence, and the urgency of data analysis.

- Despite best efforts, batch processes might face failures. Therefore, setting up robust error handling mechanisms is essential for the resilience of the Bronze layer. Consider creating an "error layer" or "data orphanage" dedicated to monitoring and managing these issues. This layer acts as a safety net, catching any problems that arise during batch processing, ensuring smooth and reliable data management.

- Selecting the right tools for data extraction and ingestion is crucial for effective data management. Consider the various tools and methods available to ensure seamless data integration and processing. For instance, Azure Data Factory is widely used for data extraction, workflow orchestration, and pipeline management, offering over 200 connectors. Evaluating and choosing the appropriate tools can significantly enhance the efficiency and reliability of your data collection processes. We'll connect back to tools and considerations in "ETL and Orchestration Tools".

Before exploring modern data ingestion methods, it's important to acknowledge that data collection remains a significant dilemma for many organizations. This is because of source systems implications, the variety of formats and network designs, and the diversity of technologies and platforms. All of these challenges make it difficult to extract data in a consistent and standardized manner.

Batch Processing Takeaway

Data collection challenges are not new; historically, organizations collected data for data warehouses. Consequently, many organizations often reuse existing interfaces, such as custom data extraction scripts or traditional batch deliveries, to import data into their lakehouses. Although batch processing might seem straightforward, it often remains a complex task for enterprises because each source usually receives its own unique treatment.

As organizations advance, they may consider adopting more contemporary interfaces, such as real-time data ingestion. This versatility is where modern lakehouse architectures excel, as they efficiently manage both batch and real-time ingestion.

Real-Time Data Processing

As highlighted in Chapter 1, modern lakehouse architectures efficiently handle streaming or real-time data. By processing data the moment it's generated, you enable immediate insights and timely decisions, crucial for applications such as fraud detection, personalized customer interactions, and dynamic system adjustments. This ability significantly boosts the responsiveness and effectiveness of your applications, making it a potent tool for businesses in fast-paced environments.

To set up real-time data ingestion in your system, you'll usually need to add extra components or tweak your existing setup. This is because most applications don't automatically generate events. As a developer or engineer, you might have to modify your application's architecture to handle event distribution. For instance, if you're working with a Node.js application using Express, you might install a library and write code to send events to a service like Azure Event Hubs.

In some situations, you might also need to add software or components to read transaction database logs or to make API calls. Once you've established a steady stream of events, your

data platform can begin to capture and process this data using services or frameworks such as Spark.

Warning

In data movements, it's essential to differentiate between state-carrying events and simple notifications. State-carrying events provide a snapshot of an application's state at a specific time, useful for processing, analysis, or triggering further events. They are crucial in data architectures for monitoring data changes or updates. Conversely, notifications are basic alerts that inform systems or users about occurrences, prompting immediate action, such as notifying a user of a new email or alerting a system administrator about a potential issue. They typically do not include detailed state information.

Technology providers support multiple approaches to managing real-time or streaming data. This variety is necessary due to the complexity and the wide range of integration options available. So, in the next sections we'll look at the most common options at the time of writing. We start by discussing Spark Structured Streaming, a stream processing engine that allows you to read streams and transform data while it's still flowing. After that we'll discuss change data feed and change data capture, followed by considerations and other learning resources.

Spark Structured Streaming

Robust support for Structured Streaming within the Apache Spark engine greatly improves real-time data processing. As an integral part of Spark, it facilitates the continuous processing of data streams, allowing data ingestion from diverse sources such as event log files, IoT devices, and real-time messaging systems like Apache Kafka. Spark Structured Streaming also excels in handling data in its raw, unstructured form, which is typical in many real-time data scenarios.

At this stage, transformations become critical as they convert raw data into a more usable format. For unstructured data, these transformation steps might include flattening nested JSON, extracting fields from XML, or applying complex functions to derive new columns. These operations are essential for cleansing and preparing data for downstream analytics and storage, ensuring that the data can be effectively processed and analyzed.

Once the data has been transformed, Structured Streaming provides the capability to perform various output operations. This includes writing the processed data to persistent storage systems for further analysis or future use. Supported destinations can vary widely depending on the application requirements and might include Delta Lake for ACID-compliant storage and versioning, traditional databases for relational storage, NoSQL databases for schema-less storage options, or even directly back into real-time message buses or event queues to enable further processing or real-time analytics.

Note

Real-time ingestion with Spark requires an "always on" compute cluster, which can result in higher costs. If latency is not a concern, consider triggering (every five minutes) and executing these workloads to lower the cost.

Additionally, Structured Streaming's integration with other components of the Spark ecosystem, such as MLlib for machine learning, allows for the development of

sophisticated analytical pipelines. This integration empowers organizations to derive real-time insights from their data, facilitating immediate decision making and enabling reactive and proactive business strategies. In summary, the support for Structured Streaming in Spark is a powerful feature for organizations looking to leverage real-time data processing.

Change Data Feed

Another pattern in the context streaming data processing worth discussing is change data feed. This feature allows you to capture changes to Delta tables as they occur, enabling you to process these changes in (near) real time.

To enable change data feed, you need to set the delta.enableChangeDataFeed property to true on the Delta table. Here's an example of how you can enable the change data feed on a Delta table:

```
ALTER TABLE myDeltaTable

SET TBLPROPERTIES (delta.enableChangeDataFeed = true)
```

The change data feed can serve as an input data source for Spark Structured Streaming. The combination of these two provides a powerful mechanism for processing real-time data. It allows you to process, for example, only the changes after an initial MERGE comparison, thereby accelerating and simplifying ETL/ELT operations. With this same pattern, you can also stream changes downstream to other systems like Kafka or an RDBMS, which can then use this incremental data for processing in later stages of data pipelines.

For a practical guide on how Delta Lake is integrated with Spark Structured Streaming, you can visit the Delta Lake documentation on "Table Streaming Reads and Writes". This resource provides detailed insights and examples to help you implement streaming data ingestion effectively.

Change Data Capture

Change data capture (CDC) is a technique employed to identify and capture changes made to data in real time. CDC tools actively monitor database modifications and record these changes as they occur, enabling the replication of data to other systems. This method is especially effective for real-time data capture of your source systems, facilitating the seamless integration of these changes into data platforms. We'll discuss CDC in more detail in Chapter 5. Let's conclude the section about real-time data processing with considerations and learning resources.

Considerations and Learning Resources

Spark Structured Streaming, change data feed, and CDC are integral components that enable powerful, real-time data integration and analytics pipelines. For a detailed comparison of how each component contributes to these processes, see Table 2-1.

Option	Description
Spark Structured Streaming	Supports complex operations and integrates with various sources and sinks; handles continuous data processing.
Change data feed	Captures changes to Delta tables in real time; enables efficient data processing by streaming only the increments and difference between states (also known as deltas).
CDC	Tracks and captures changes in database transaction logs effectively, allowing for real-time data replication and integration.

Table 2-1. Overview of real-time data processing options

When discussing real-time data replication, stream processing, and the design of your Bronze layer, nuances come into play. For instance, consider using Microsoft Fabric's mirroring capability to replicate data in near real time from a cloud native Azure SQL application into the lakehouse. This replication technique uses the database's CDC technology, transforms it into appropriate Delta tables, and lands it in the lakehouse architecture (OneLake). The question of whether this data belongs to the Bronze layer depends on your specific needs.

If your primary goal is to stack full data extracts, the real-time replicated data might be better classified as part of an intermediate (or landing) area. This setup allows you to query the data directly, as if it were part of the source, without impacting the source application. However, it doesn't organize the Bronze layer to accumulate raw data copies for historical analysis effectively. In this scenario, the initial layer acts as a replication or pre-Bronze layer.

Conversely, if your aim is to maintain a queryable Bronze layer that mirrors the source data "as-is," then real-time replicated data can be considered part of the Bronze layer. Ultimately, the classification hinges on your specific needs and how you plan to utilize the replicated data within your lakehouse architecture.

Real-Time Data and Streaming Ingestion Takeaway

The key takeaway for streaming and real-time data ingestion is that the decision to implement real-time data ingestion or replication, and to classify this data within the Bronze layer or an intermediate area, is fundamentally driven by distinct usage requirements and strategic objectives.

It's also important to understand that with real-time data ingestion, updates often occur across multiple layers simultaneously. For instance, you might use Spark Structured Streaming to process streaming data from Azure Event Hubs directly. Then, while processing, you can immediately perform transformations such as removing unwanted

columns, aggregating data, or even adding preliminary sentiment scores. You could store the raw data in the Bronze layer but also send the transformed, cleaned data immediately to the Silver layer, or even the Gold layer. In such scenarios, the Bronze layer acts more like an archive zone rather than a layer from which to read. This approach of parallel processing is especially useful for real-time analytics and insights, enabling you to quickly access and analyze incoming data.

Streaming is a complex topic thoroughly explored in *Delta Lake: The Definitive Guide*. If you're interested in delving deeper into this subject, I also encourage you to follow the comprehensive guide, "Structured Spark Streaming with Delta Lake: A Comprehensive Guide". These learning resources provide detailed insights and guidance on implementing streaming within the Delta Lake framework, enhancing your understanding and ability to manage real-time data effectively.

Ingestion management, whether dealing with batch or streaming data, is a complex subject that requires clear guidance to navigate effectively. Choosing the right methods is crucial to ensure efficient data handling. To implement these methods successfully, leveraging the right tools becomes essential, as they directly influence the efficiency and effectiveness of data ingestion processes.

ETL and Orchestration Tools

Discussing the tools for extracting, transforming, loading, and orchestration is crucial as it can significantly influence the design of the data pipeline. In Chapter 5, I'll come back with examples, but for now, know that the following popular tools are available, each with unique features that cater to different needs within data architectures:

Apache Airflow

This is an open source platform that allows you to programmatically author, schedule, and monitor workflows. It's particularly useful for orchestrating complex data pipelines and managing dependencies between tasks. We discuss Apache Airflow further in Chapter 6.

Azure Data Factory

This is widely used among organizations that have adopted Synapse Analytics, Azure Databricks, and Microsoft Fabric. It is effective at creating, scheduling, and orchestrating data workflows. Within Microsoft Fabric, Azure Data Factory (ADF) is simply referred to as Data Factory, but the functionality largely remains the same. Another important feature of ADF is its support for numerous connectors, which enable the extraction of data from a wide variety of sources.

Databricks Auto Loader

This is specifically designed for the Databricks ecosystem. It excels at incrementally processing new files as they arrive in cloud object storage. One of its standout features is its handling of schema evolution. We discuss Auto Loader further in Chapter 5.

Databricks LakeFlow Connect

This is another Databricks-specific service. It provides built-in connectors for various sources, including Salesforce and SQL Server, facilitating data ingestion.

Databricks Workflows

This is a managed orchestration service as part of Databricks. It lets you define, manage, and monitor multitask workflows for ETL, analytics, and machine learning pipelines.

For broader compatibility and feature sets, third-party tools such as Fivetran, Qlik, StreamSets, Syncsort, Informatica, and Stitch are also popular choices. These tools offer extensive connectors and orchestration capabilities and are often used in conjunction with other tools to enhance functionality.

When designing your Medallion architecture, it is essential to consider the specific capabilities and limitations of the chosen ETL tools. The complexity of data loading steps enabled by these tools might necessitate more preprocessing steps within your architecture. Thus, the choice of ETL tools can dictate not only the design of the first layers but also influence the architecture as a whole. So, careful consideration is required to ensure that your data architecture adheres to the organizational requirements.

Assuming you have successfully captured the data through ingestion tools, it's time to shift our focus to managing those tables.

Managing Delta Tables

Table management is a concern across all layers, considering its effect performance, usability, and costs. Techniques such as *-ordering and liquid clustering are recommended methods for improving the way data is managed, making retrieval processes more efficient and effective. In the next sections, we'll look at these techniques in more detail. We'll start with Z-ordering and V-ordering, and then move on to partitioning. We close this section with a discussion on liquid clustering, table compaction and optimized writes, and the DeltaLog.

Z-Ordering

Z-ordering is a technique that boosts data retrieval efficiency by grouping related information within the same files. This method is particularly efficient because it improves locality, reduces I/O operations, and supports multidimensional data management—making it two to four times more efficient than regular data retrieval. Z-ordering, especially, performs better in scenarios that require efficient filtering and joins. While liquid clustering, which merges the advantages of Z-ordering and table partitioning, has taken over in some scenarios, you might still find Z-ordering relevant depending on your setup.

In practice, Delta Lake automatically takes advantage of this technique through its data-skipping algorithms. To implement Z-ordering, simply specify the columns you want to organize by using the ZORDER BY clause. This method ensures that your data is optimally arranged for quicker access and processing.

OPTIMIZE events

WHERE date >= current_timestamp() - INTERVAL 1 day

ZORDER BY (eventType)

This technique significantly reduces the volume of data that needs to be scanned, enhancing overall performance. However, Z-ordering is particularly beneficial for large tables, such as those spanning hundreds of gigabytes, terabytes, or more. This focus on large-scale datasets ensures that Z-ordering effectively optimizes data retrieval in environments where managing vast amounts of data is a challenge.

V-Ordering

V-ordering is another technique used to optimize the retrieval of data. Unlike Z-ordering, V-ordering is a write-time optimization for the Parquet file format. Basically, it's an optimization that logically organizes data based on the same storage algorithm used in Power BI's VertiPaq engine. This enables faster reads under Microsoft Fabric compute engines, such as Power BI and Warehouse. Note that V-ordering is 100% compliant with the open source Parquet format, so all Spark engines can read it as regular Parquet files.

According to Microsoft's Delta Lake table optimization and V-order documentation, the performance gains with V-ordering depends on the engine and scenario. In summary, performance with V-ordered files could deliver an average of 10% faster read times, with some scenarios showing improvements of up to 50%. You can configure V-order using Delta table properties. Here's an example:

```
CREATE TABLE person (id INT, name STRING, age INT)

USING parquet TBLPROPERTIES("delta.parquet.vorder.enabled" = "true");
```

It's important to acknowledge that for V-ordering, there's a cost to performance optimization. Sorting can impact average write times by 15%.[1] However, this feature can be disabled if necessary. The reason for the negative impact on write times is the additional sorting step that occurs during the write process. This can become a concern when dealing with a large amount of data writes and limited data reads.

To ensure optimal performance, it is best to align V-ordering with the layers of your architecture or specific use case; for example, for use cases where the SQL endpoints are heavily used. This alignment will help you determine if the benefits of using V-order sorting outweigh the potential loss of performance.

Now that we have covered ordering, let's move on to data partitioning, which is another pattern that can improve performance.

Table Partitioning

Table partitioning is an effective strategy for managing large datasets, typically those that are several hundred gigabytes (GBs) up to many terabytes (TBs) in size. It involves dividing a large table into smaller, more manageable segments, significantly enhancing query performance by reducing the volume of data that needs to be read during each query. Within Delta Lake, you can partition a table by a specific column. The date column is a common choice for partitioning, but you can select any column frequently used in queries. For example, if you often retrieve data based on country, partitioning by the country column would be beneficial. To implement partitioning, you specify the desired column in the PARTITIONED BY clause. This tailored approach ensures quicker access to relevant data, streamlining query processes. Here, you can see an example:

```
CREATE TABLE events

USING DELTA

PARTITIONED BY (date)
```

Both partitioning and Z-ordering are most effective with very large tables. While you can combine Z-ordering and partitioning, it's important to remember that Z-order clustering can only occur within a partition and cannot use the same field for both techniques.

Liquid Clustering

Optimally implementing Z-order and partitioning comes with challenges and trade-offs. These optimizations require a fixed data layout, demanding careful up-front planning. To address these challenges, the Delta Lake project introduced a new feature called *liquid clustering* in mid-2024. This feature is set to replace traditional Z-ordering and partitioning. Liquid clustering simplifies the decision-making process for data layout and significantly boosts query performance by automatically optimizing the data layout. This means you no longer need to manually fine-tune the data arrangement for optimal performance.

Liquid clustering adapts the data layout dynamically based on the clustering keys. Unlike the static data layout seen in Hive-style partitioning, this flexible, or "liquid," layout changes in response to evolving query patterns. This dynamic adjustment tackles issues like suboptimal partitioning and column cardinality. Moreover, you can apply incremental optimizations and recluster columns without the need to rewrite the data. For more detailed information, check out the official Delta Lake documentation on liquid clustering.

Compaction and Optimized Writes

Delta Lake offers an operation named OPTIMIZE, specifically engineered to address the small files problem, a common issue that we discussed in "Hadoop's Distributed File System". As you update, delete, or add new data, Delta Lake stores these changes in smaller files. Over time, these smaller files accumulate and can slow down data retrieval processes. To tackle this issue, you can run an optimize job, which compacts the smaller files into larger, more manageable files, enhancing query performance. You can run the optimize job using the OPTIMIZE statement, as shown here:

```
OPTIMIZE delta_table_name;
```

Another way to increase performance is by optimizing writes. By default, this feature is turned off, but you can activate it in Delta Lake version 3.1 and later. Simply set the delta.autoOptimize.optimizeWrite table property to true. With this setting enabled,[2] Delta Lake automatically optimizes the data file layout during writes, leading to enhanced query performance. You can find more information on these features in the Delta Lake documentation.

Now that we have explored several table optimization techniques, let's delve into the transaction log, as it plays a crucial role in safeguarding data integrity and facilitating data recovery.

DeltaLog

The Delta transaction log, also known as the DeltaLog, is a key feature of Delta Lake frequently used within all the layers of the Medallion architecture. DeltaLog, as mentioned earlier in Chapter 1, is a transaction log that carefully records every change made to data within a Delta table. This includes additions, alter statements, optimize jobs, modifications, and deletions of data. The log maintains a comprehensive history of the table and is stored

as a series of JSON files in the *_delta_log* directory, located within the Delta table's directory.

Each transaction generates a new log file, capturing detailed information about the performed actions, the files added, the files removed, and other relevant transaction details. By default, the retention threshold for these files is set to seven days. However, you can modify this setting using the ALTER TABLE SET TBLPROPERTIES statement, allowing for greater flexibility in managing how long transaction logs are retained. This feature ensures robust data management and enhances the ability to audit and roll back changes if necessary.

Note

The VACUUM statement cleans up a table directory by removing any files not managed by Delta and deleting data files that are not part of the table's latest transaction log state and are older than a specified retention threshold. This helps maintain an efficient and streamlined data storage environment, while also ensuring cost savings and compliance.

For the Medallion architecture, DeltaLog provides a robust safety measure. Should an error arise or data integrity be compromised, particularly in the Silver or Gold layers, you have the ability to quickly revert to a prior version of your Delta table. This rollback is facilitated through the use of the RESTORE command, enabling access to earlier versions of a table by time or version number. This feature supports crucial functions such as data auditing, rollbacks, and the replication of experiments or reports.

However, it is essential to note that DeltaLog's main purpose is geared toward data recovery and auditing rather than serving as a conventional historical database. For example, attempting to identify records changed over the last month through the transaction log would require extensive data processing, as all files must be read and compared for changes. A more efficient way to track these historical changes is to manage them directly within a table, using a method like a slowly changing dimension. This approach not only optimizes performance but also helps maintain data integrity over time. We'll come back to this topic in "Slowly changing dimensions".

Table Management Takeaway

For optimal data management performance, align *-ordering methods like V-ordering and liquid clustering with your architecture's specific layers, focusing on the queries and access patterns of each layer. This alignment ensures efficient query processing and data retrieval. Additionally, setting thoughtful retention thresholds for Delta tables is critical. Choose thresholds that support your operational needs and data recovery scenarios, balancing data freshness with storage costs.

With this comprehensive understanding of the roles and considerations of the various ingestion and table management patterns, we are well-equipped to make informed decisions about the structure and methodologies of the Medallion architecture. Now, let's summarize the key insights gathered and consider how they inform the next stages of our architectural journey.

Conclusion

Laying the groundwork for the Medallion architecture is a meticulous process that begins with a comprehensive understanding of the initial preconditions, such as identifying source systems and determining the most efficient methods for data export and ingestion. Each source system may require a customized approach to effectively manage data extraction and loading, which underscores the importance of not underestimating this foundational phase. During this phase, it is crucial to engage in detailed planning and collaboration with the owners of the source systems. This collaboration will help you make informed decisions about whether to apply mediation, employ additional landing zones, and use integration runtime components, as well as how to choose between batch and real-time processing. These choices come with their own sets of nuances and can significantly impact the amount of work required for collecting data.

Moreover, the selection of ETL and orchestration tools is a critical factor that can dictate the design of the initial layers and influence the overall architecture. Careful consideration and standardization are required to ensure that these tools not only meet the current needs but also align with the broader organizational requirements. In the context of tools and services, standardization is key for effective governance, metadata management, lineage collection, consistent data quality reporting, and other aspects of data management. This decision will affect the flexibility and scalability of the Medallion architecture.

In addition, the management of tables and the approach to data modeling are closely linked and vital for optimizing performance. Effective table management strategies must be considered early in the design process to ensure that the architecture can handle the intended data loads efficiently.

With these fundamental aspects covered, we have set the stage for building a Medallion architecture. In Chapter 3, we will delve deeper into the specifics of each layer, discussing its detailed design, considerations, and established best practices.

1 Miles Cole shares specific numbers on the performance gains for different workloads in his blog post: "To V-Order or Not: Making the Case for Selective Use of V-Order in Fabric Spark".

2 Similar to V-ordering, using the delta.autoOptimize.optimizeWrite feature introduces a bit of extra overhead. For moderate-sized jobs, handling between 2 and 5 million rows, you can expect an additional delay of about 5 to 10 seconds.

Chapter 3. Demystifying the Medallion Architecture

In Chapter 1, we explored the evolution of Spark and Delta Lake, and introduced you to the Medallion architecture. This design pattern helps organize data logically within modern lakehouse architectures. It utilizes three layers (Bronze, Silver, and Gold) to progressively refine datasets throughout the lifecycles of data ingestion, data transformation, and data loading into various destinations.

Explaining the Medallion architecture to organizations often feels like opening a can of worms, as each layer, meant to address different concerns, lacks clear definitions and descriptive guidelines. This ambiguity leads to more questions than answers, creating a cycle of confusion and inefficiency.

Despite the popularity of this three-layered design, there's significant debate about the scope, purpose, and best practices for each layer. Moreover, the gap between theory and practical application is substantial. In this chapter, I will share insights from my practical experiences on designing each layer of the Medallion architecture by using a theoretical viewpoint. In Part II, the focus shifts from theory to practice. The insights from this chapter are carried forward when engaging in a hands-on exercise where you'll be taught to build a real solution architecture.

The Three-Layered Design

Before discussing the specifics of each layer in the Medallion architecture, it's crucial to understand the high-level purposes and functions of the three primary layers: Bronze, Silver, and Gold. Figure 3-1 illustrates the flow of data from the Bronze layer through the Silver and into the Gold layer, highlighting key processes such as ingestion, processing, and usage for analytics.

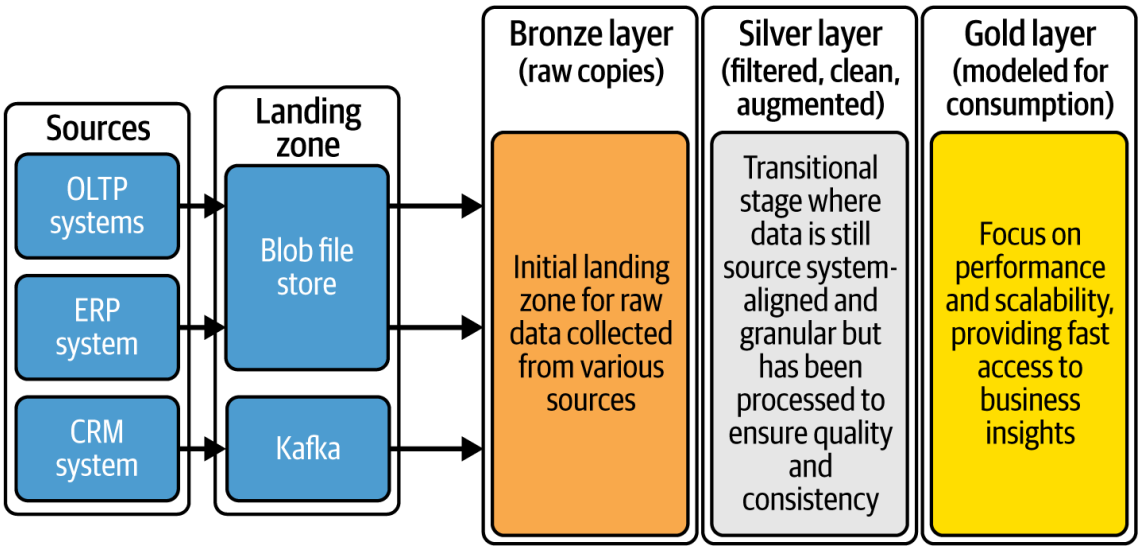

Figure 3-1. The Medallion architecture and its three layers: Bronze, Silver, and Gold

In Figure 3-1, each layer plays a distinct role in the transformation and refinement of data, facilitating a structured progression from raw data collection to data that can be used for data-driven value creation. I introduced the layers in Chapter 1, but this section will provide you with more details with key characteristics of each layer, setting the stage for a deeper exploration of the technologies and processes involved:

Bronze layer

Acts as the zone for raw data collected from various sources. Data in the Bronze layer is stored in its original structure without any transformation, serving as a historical record and a single source of truth. It ensures data is reliably captured and stored, making it available for further processing.

Its key characteristics are high volume, variety, and veracity. The data is immutable to maintain the integrity of its original state.

Silver layer

Refines, cleanses, and standardizes the raw data, preparing it for more complex operational and analytical tasks. In this layer, data undergoes quality checks, standardization, deduplication, and other enhancements that improve its reliability and usability. The Silver layer acts as a transitional stage where data is still granular but has been processed to ensure quality and consistency.

Its key characteristics are that data in the Silver layer is more structured and query-friendly, making it easier for analysts and data scientists to work with.

Gold layer

Delivers refined data optimized for specific business insights and decision making. The Gold layer involves aggregating, summarizing, and enriching data to support high-level reporting and analytics. This layer focuses on performance, usability, and scalability, providing fast access to key metrics and insights.

Its key characteristics are that it's highly curated and optimized for consumption, so the data in the Gold layer supports strategic business operations and decisions.

Note

While the shift from on-premises RDBMSs to decoupled and distributed architectures has transformed technology, the fundamental methodologies of data warehousing and data modeling are still relevant. If you're not confident with these skills, I recommend brushing up on the basics. Great resources include Kimball Group's books on data warehousing and dimensional modeling, as well as *Star Schema: The Complete Reference* by Christopher Adamson (McGraw Hill Osborne Media). These great resources provide a solid foundation for understanding the principles of data modeling.

Before we dive into the specifics of each layer, let's start by getting our mindset right about the Medallion layers. It's crucial to think of these layers as logical, not physical. So, when talking about, for example, the Bronze layer, don't frame it as just one physical layer. Instead, view it as a logical layer that could span across several physical layers. This approach will help us stay flexible and make more sense of the structure. Having said that, let's get started.

Bronze Layer

The Medallion architecture starts with the Bronze layer. The main goal of this layer is to store data from various sources in its original state without any modifications. It ensures that the data is easily accessible for further processing and enables users to explore and analyze it instantly. Essentially, it serves as a queryable reservoir for raw data.

In the next couple of sections, we'll first discuss the processing hierarchy and how this influences the potential Bronze layer's overall design. After that, we'll focus on the archiving and control objectives by discussing processing full and incremental loads, historization, schema management, and technical validation checks. I'll conclude with usage and a summarization the key points about the Bronze layer before moving on to the Silver layer.

Processing Hierarchy

Whether the Bronze layer consists of a single physical layer or multiple sublayers depends on the complexity of the data sources and the organization's needs. In earlier discussions in Chapter 2, we touched on the possibility of having extra landing zones and whether these zones are part of the Bronze layer. Let's revisit this topic with an example scenario and draw some conclusions.

Traditional source systems often dictate the format in which data must be exported, typically using formats like CSV or JSON. Consider a legacy application that exports data only to a local file storage location. In this scenario, you would integrate the data into your Medallion architecture through batch processing.

Here's a typical process flow: the process begins by staging the data in a landing zone, where you perform minimal processing tasks such as decompressing zipped files, comparing checksums, and generating metadata. Following this, you copy the data to the Bronze layer, converting it into a format like Delta Lake to enhance performance. At this stage, you also carry out basic validations to ensure data integrity. Optionally, you might also classify or encrypt the data to protect sensitive information. Finally, you compare and integrate the data with your existing Bronze datasets to maintain consistency and completeness. We'll revisit these details when implementing the Bronze layer in Chapter 5 using external tables or Delta copies.

Note

Encrypting the raw data is essential to protect personally identifiable information (PII) from unauthorized access. Therefore, it's not uncommon to encrypt the data before it lands in the Bronze layer. This encryption can be done using open source encryption frameworks, such as Fernet, or other encryption tools.

In the data processing hierarchy, the Bronze layer often includes multiple sublayers: an internal staging area, a processing and refinement area, and eventually storage in Delta format. Thus, whether the Bronze layer is a single physical layer or multiple sublayers depends on data source complexity, organizational requirements, and the decision on whether to include the landing zone as part of the Bronze layer.

With the processing hierarchy established, it's crucial to focus on how data is processed and managed over time because each data source is unique and the way data is delivered can vary significantly between source systems. Some systems provide full extracts, others deliver delta increments, and still others stream data continuously. Therefore, deciding on the appropriate processing pattern is essential. Let's explore these strategies in detail, starting with processing full data loads, followed by processing incremental loads.

Processing Full Data Loads

Full data loads involve transferring the entire dataset from the source system, rather than just changes, ensuring all data is available for downstream processing. When processing full data loads, large batches of data are typically handled at fixed intervals, often with the optional use of landing zones. After validation, the data is stored in its raw form and accumulated in folders alongside all previous data deliveries. At this stage, minimal transformations are usually performed; sometimes data is filtered, or sensitive columns are encrypted. After accumulating the data, the most recent delivery is typically either converted to the Delta Lake format or typically exposed as external tables for processing in the next layer. We will revisit these patterns in Chapter 5, where we explore the processing of full data loads in greater detail and demonstrate concrete examples.

Processing Incremental Data Loads

The pattern of loading increments, or "delta loads,"[1] is often employed in scenarios where only changes or updates to the data need to be processed, allowing for more efficient data handling and reduced processing time compared to full data loads. In this approach, you typically receive only the data that has changed, often in the form of events, for example. This data is usually staged in a preliminary layer, where you might first apply some lightweight transformations. These could include changing the file format (e.g., from JSON or Parquet) and converting the data to the Delta Lake format or making simple data type corrections. After these initial steps, you can utilize append mode or merge operations to integrate this incoming new data with the existing data already present in the Bronze layer. Let's explore these methods further:

Append mode

In the context of Delta Lake, "append mode" means that new data is added to an existing Delta table without modifying or deleting the existing data. This is useful for scenarios where you want to continuously add new records to a dataset, such as logging or streaming data, without affecting the existing data.

Here is an example where df represents a DataFrame, a structured collection of data with named columns. Using Spark, you can read data into the DataFrame and subsequently append it to a Delta table:

```
df = spark.read.json(f"{filePath}/events/*.json")

df.write.format("delta").mode("append").saveAsTable("events")
```

Append mode does not address updates or deletions—it simply appends new data to the end of the dataset. Merge operations are more sophisticated than a simple append because they allow for both the insertion of new records and the updating of existing records based on some key.

Merge mode

In Delta Lake, the "merge mode" provides a way to perform upserts (a combination of updates and inserts) to a Delta table. This is particularly useful when you need to update existing records and insert new ones in a single operation, based on a specified condition.

Suppose you manage a table containing customer information, including a status flag labeled is_current for customers present in the source system within the last 180 days. You

can use a MERGE operation on the Delta table, which efficiently allows for multiple changes simultaneously:

- Insert new customers.
- Update the status of customers who have recently returned.
- Modify the status of existing customers in the Delta table who are now inactive.

Delta Lake efficiently handles these operations behind the scenes, ensuring data integrity and streamlined processing. We'll connect back to this merge approach in Chapter 5.

Processing incremental loads is an efficient method to manage large datasets by processing only the changes since the last load instead of reprocessing the entire datasets. In scenarios with large source system tables, such as those containing millions of rows, this approach could be particularly useful. Furthermore, incremental loading is also essential for efficiently updating data in any layers, such as the Silver and Gold layers, without reprocessing the entire dataset. A consideration for incremental loading is to combine it with a change data feed to push incremental changes to your next layers.

Tip

By implementing incremental processing with Delta Lake and Spark, and utilizing configurations like trigger(AvailableNow=True), organizations can manage their data more effectively and economically.

For incremental loading to work effectively, your (source system) tables need to have unique incremental identifiers or updated_at columns. These markers allow you to detect any new or updated records since the last load. Furthermore, it's crucial to ensure that the source system does not update records that are older than the last fetched increment. If updates on older records are possible, standard incremental loading might miss these changes. In such cases, using CDC or similar tools might be necessary. These tools are designed to capture all data changes—inserts, updates, deletions—from the transaction logs and can be more reliable for ensuring complete data synchronization.

For streaming, you would typically specify the initial position's startingVersion by using the Delta (or Iceberg) transaction log. This allows you to pick up exactly where you left off without needing to run any additional commands.

Alternatively, you could use a max function on the identifier or *updated_at* column from your Bronze layer to determine the most recent entry processed. Once you have the highest value (max(updated_at)) from the Bronze layer, you can then query the source data for records with identifiers or timestamps greater than this value. This strategy ensures that only new or updated records since the last load are processed.

Maintaining a metadata control table that tracks processed records is another recommendation. This helps manage data flow and ensures consistency, particularly in complex systems.

For a deeper understanding of incremental delta loads and their practical usage, consider watching "Incremental Refresh with Your Warehouse Without a Date in Microsoft Fabric". This video on incremental delta loads provides comprehensive insights and examples to help you implement this technique effectively.

By capturing incoming data in its incremental form, you can integrate it with the complete historical dataset. This process of historization is crucial not only for building a comprehensive archive but also for ensuring that current data is readily accessible for further downstream processing. We'll explore this topic next.

Data Historization Within the Bronze Layer

The Bronze layer is designed primarily for storing complete snapshots or copies of data, similar to what you'd find in a staging environment in traditional data warehouse environments. This layer is especially useful if your method involves fully extracting and overwriting data, as it allows you to maintain a comprehensive history of all data deliveries. The Bronze layer usually also serves as a robust archive, preserving data for many years, which is vital for audit purposes. However, there may be instances where you need to clean up data that has been processed and is no longer required.

In a Medallion architecture, archiving data is straightforward when you use an efficient storage format like Parquet or Delta and organize it in folders.[2] By structuring your data in interval-partitioned tables and sorting them into folders labeled with a YYYY/MM/DD or datetime format, you keep your data well-organized and manageable. This systematic organization not only keeps your data tidy but also simplifies access and auditing. If you ever need to look back at data from a specific day, you can easily recreate its state, ensuring you can retrieve historical data accurately whenever needed.

However, it's important to note that the Bronze layer is not intended for historization of data in the sense of representing a fully processed SCD2 table. Data in the Bronze layer is often considered immutable, meaning it is read-only. While you may append or merge data to incrementally grow the dataset over time, updates or processing of the data are not typical practices at this stage.

Note

You can apply data versioning by using the Delta Lake time travel feature. It is primarily designed for data recovery, auditing, and reproducing experiments or reports, rather than for comprehensive long-term data archiving.

Historization is crucial, especially when dealing with Bronze-level datasets. This process enables the archiving of data iterations over time, which can be essential for auditing or debugging purposes. Furthermore, historization supports the ability to perform reloads and recoveries. This feature is vital for handling instances of poor-quality data or disruptive changes in schemas, allowing you to revert to a previous version of the dataset and reprocess it.

While delving into the complexities of processing historical data, you encounter a significant challenge: schema evolution. As businesses grow and data sources are frequently updated, the database structure (schema) can change. This evolution can lead to inconsistencies in data processing, making it essential to manage schema changes effectively. We'll explore this topic in the next section.

Schema Evolution and Management

The Bronze layer is crucial for managing widely varying and ever-changing data schemas as data enters the system. Effectively handling schema evolution at this stage sets the

foundation for all the data processing and analytics that follow. You can tackle incoming schema changes either in the Bronze layer or wait until the data moves to the Silver layer. The timing of addressing these changes depends largely on your specific needs and the characteristics of your data.

When managing schemas, organizations face design choices, each with its own approach to schema evolution:

Schema-on-read

This method applies the schema dynamically as you read the data. With this, data is stored without a strict schema in place. The schema is only inferred or applied when the data is accessed during processing (reading). This flexible approach requires the system to recognize and possibly adapt to various data schemas right after the point of ingestion. This approach is particularly useful for semi-structured or unstructured data or structured data without any built-in schema enforcement, such as CSV or Parquet files.

Schema-on-write

This approach involves defining the schema—like the table and column names, the data types, and the primary keys—as you write the data into storage. With this method, you must define the schema up front, making sure that the data conforms to this schema right from the start, before writing. It's a more rigid approach compared to schema-on-read, as it enforces schema consistency from the beginning. When applied, it's generally used for structured data.

When using Delta Lake, if there is no predefined schema, the system will establish an initial schema by using the StructType from the DataFrame that is being converted to Delta format. In Apache Spark, StructType refers to a container that defines the structure and types of columns in a DataFrame.

In the Bronze layer, schema-on-read is a common approach where data is stored in its original format without enforcing a schema during ingestion. This method offers flexibility since it does not demand an up-front schema definition and can gracefully handle changes in data structure. For instance, imagine you partition your data daily using a YYYY/MM/DD time partitioning format, storing each day's data as Parquet files within corresponding folders labeled by date. With schema-on-read, you can immediately access and read this data across different days without needing to predefine how the data structure should look. This approach allows for easy integration and analysis of data collected over time. However, there are several important factors to consider.

Firstly, relying solely on schema-on-read does not allow for blind data processing. You may need to detect and log schema changes, handle failures, restore data using Delta Lake's time travel feature, and add schema-related metadata for more refined processing in later stages. We'll delve into these aspects and introduce tools like Auto Loader in the detailed discussion in "Databricks Auto Loader".

Secondly, defining a data update method is essential, especially for historization and integration of data. Options include continuously appending to or merging with existing datasets, completely overwriting them, or applying time-based partitioning. The choice largely depends on the nature of the data, its usage, the specific system requirements, and performance considerations. For instance, appending is commonly used for transactional

data because this type of data grows incrementally with each new transaction. Overwriting might be chosen for datasets that receive regular source corrections, or where only the most recent data snapshot is relevant. Time-based partitioning can be implemented to enhance query performance and manage large datasets effectively, especially when data access is time-sensitive.[3]

Therefore, the Bronze layer often combines schema-on-read and schema-on-write techniques. Initially, schema-on-read is applied in the landing or pre-Bronze layer. Hence, you first read data without explicitly declaring the data schema. Then, as you move the data to a layer where it becomes "queryable," you switch to schema-on-write. For this critical layer, it's wise to choose a storage format like Delta Lake that supports schema evolution. Delta Lake allows the data schema to adapt as it evolves, eliminating the need to overhaul or reload the entire dataset. This capability keeps the data current and usable, even as changes occur. We'll revisit this feature shortly.

However, be mindful that keeping schemas consistent can pose challenges, especially if the source systems are prone to frequent changes. This requires vigilance and possibly more sophisticated schema management strategies to ensure data integrity and system reliability. For this reason, to keep things running smoothly, it's essential to maintain a schema that consistently matches the source system as data enters this layer. You usually set up the schema during the data ingestion process to comply with specific standards, either through data definition languages (DDLs) or programmatically using tools like Spark SQL or Databricks Auto Loader. We'll revisit this subject in more detail in Part II.

MergeSchema and Schema Enforcement

Delta Lake offers a set of features for managing data schemas, such as handling schema evolution and schema enforcement. The features you apply depend on your system's specific needs and design, and you can see them across all layers of the Medallion architecture.

For example, as data evolves, there might be a need to modify the table schema to accommodate new types of data (e.g., adding new columns). mergeSchema simplifies this process. It's a feature designed to help manage and evolve the schema of Delta Lake tables over time. To enable this, add the following option in a write operation in Apache Spark: .option("mergeSchema", "true"). Delta Lake then takes care of adjusting the table schema in the following ways:

- If a column exists in the source DataFrame but not in the Delta table, Delta adds the new column. All existing rows will have a null value in this new column.
- If a column is in the Delta table but not in the source DataFrame, it remains unchanged. New records will have null values for these missing columns.
- Adding a NullType column sets all existing rows to null for that column.
- If a column with the same name but a different data type exists, Delta Lake tries to convert the data to the new type.[4] If the conversion fails, Delta Lake throws an error.

In case of errors or failures, you can roll back the table to a previous version using Delta Lake's table history. This rollback feature allows you to recover from schema evolution issues. After reverting, you can reprocess the corrected data using the appropriate schema.

For more detailed information on how Delta Lake supports schema evolution, check out the blog posts at Delta Lake Schema Enforcement and Delta Lake Schema Evolution.

To handle schema changes more strictly, you can use the Delta constraints feature, which is closely aligned with the schema-on-write approach. This method ensures that the data written to a Delta table strictly matches the table's schema, and it rejects any writes that do not conform. As data enters the system, Delta Lake verifies its adherence to the predefined schema. Any deviation results in an error, thereby maintaining the integrity and consistency of your data. However, this approach can make it challenging to handle persistent and disruptive changes in the source system's schema.

If you encounter changes that affect compatibility, for example, changes that cannot be handled via the mergeSchema option, you'll need to reconcile the schemas. When adjusting a Delta table's schema, you have several options:

Using SQL ALTER COLUMN statements

Updating schemas with SQL ALTER COLUMN statements involves making direct changes, which requires careful coordination between the source system owners and the data platform managers. One effective way to manage these modifications is by checking the operations or scripts into a repository. This method is very precise but can be labor-intensive and prone to errors due to the intricate coordination required.

Automated schema evolution

You can implement automated tools or scripts that can detect schema changes in the source system and automatically generate and apply the necessary ALTER statements to the target system. We'll discuss this approach in more detail in "Handling Schema Evolution".

Using a metadata-driven framework

This recommended approach not only helps maintain the original and new target schemas but also provides a robust way to manage changes systematically. By maintaining mappings between schemas, you create a scalable method that can handle complex transformations and schema evolution without extensive manual intervention. This ensures the metadata repository is always up-to-date and accurately reflects the schemas in use. It might be helpful to integrate this with your continuous integration and deployment (CI/CD) pipeline to automate updates and deployments. We'll connect back to this approach in Chapter 5.

Restricting disruptive changes

Sometimes, you might consider requiring source system teams to avoid making disruptive changes. Thus, you'd only allow backward-compatible changes to be made. While this places constraints on the source systems, it can significantly reduce the complexity and frequency of schema updates.

Alternatively, for very disruptive source system schema changes that cannot be reconciled with the previous version, you might consider creating a new version of the data pipeline. In this scenario, the new data would land in a new target location, allowing you to manage significant schema changes more effectively. This allows you to maintain multiple versions of the schema concurrently, and applications or processes can specify which version of the schema they need to interact with.

As we explore schema evolution and the strategies to adapt data structures to evolving business needs, it becomes evident that one of the essential aspects of managing these schema changes is the necessity to implement robust technical validation checks to detect errors and unknown changes. These processes are closely interlinked and, together, they significantly improve data management.

Technical Validation Checks

Technical validation checks are crucial, as issues with data integrity, accuracy, completeness, and consistency can have significant operational and strategic impacts on any organization. Within the lakehouse architecture, the Bronze layer acts as the initial repository for raw, typically unstructured data. Here, data is stored in its native format, which may include inconsistencies or errors.

Implementing validations, such as format, schema, and completeness checks, at this stage is essential for several reasons. First, it ensures that the data conforms to specified standards, which helps maintain consistency across the dataset. This is particularly important as data from various sources may not adhere to a stable structure. Moreover, early validation helps in identifying and addressing data integrity issues. In particular, it's useful when applying a schema-on-write approach. By catching errors at the onset, it prevents the propagation of inaccuracies to subsequent layers—Silver and Gold—where data is further processed and refined for analytical purposes.

The design of the Bronze layer centers on strong data validation controls because it acts as the primary shield for technical validation checks and observability. You have different options: perform these checks in one go, break them into segments, or integrate them during data transfer to the Silver layer. To validate data effectively, many companies use scripts and keep a metadata repository. This repository holds technical schema details from their source systems, enabling automatic enforcement of data validation rules, both declaratively and dynamically.

The design of your Bronze layer also hinges on your tolerance for data integrity issues. If you anticipated a failure in downstream processes or if data consumers cannot handle these issues, you should halt the pipeline. This approach, known as *intrusive data integrity processing*, involves storing validated and incorrect data in a separate folder or table. By doing this, you can easily identify and rectify the issues, ensuring these datasets do not enter your actual Bronze layer.

Conversely, if you can tolerate data integrity or completeness issues, you may opt to continue processing the data, a method referred to as *nonintrusive data quality processing*. In this scenario, both validated and incorrect data are stored directly in the target Bronze-layer folder or table. This allows you to proceed with data processing, but be mindful that you may need to address these issues in another layer (typically Silver) later on.

Most of the checks validate the technical aspects of the data source to prevent data from advancing to the next zone unless it meets the set of predefined standards. If data doesn't meet the next zone's standards, it's the responsibility of the data owners to rectify this in close collaboration with application teams and data engineers. These team members, who manage and source the data, must ensure that the data fulfills the integrity requirements stipulated by the consumers or users of the lakehouse.

In this way, the Bronze layer not only serves as a repository for raw data but also as a critical checkpoint for data integrity and accuracy, ensuring only data that meets predefined quality standards progresses further into the lakehouse architecture. This proactive approach to data management helps safeguard the integrity and usability of the data throughout its lifecycle in the lakehouse.

A variety of tools are available for managing data ingestion, validation, and orchestration processes, and ensuring that your data's integrity is intact:

- You can use Delta Lake's schema validation features for addressing compatibility and consistency issues. For instance, schema enforcement ensures that developers adhere to predefined standards, keeping the tables clean. When data is written to a table, Delta Lake performs schema validation to check if the data's schema matches the table's schema. If the schema is compatible, the validation passes, and the write succeeds. However, if the schema is incompatible, Delta Lake cancels the transaction, preventing any data from being written.
- You can use data quality frameworks or tools, such as Delta Live Tables, Great Expectations, data build tool (dbt), Ataccama, or Monte Carlo Data. We'll come back the differences between these tools in Chapter 6.
- You can use custom solutions with scripts, code, and schema metadata to validate data integrity. This approach is often used when organizations have specific requirements that are not met by existing tools or frameworks. Custom solutions can be tailored to the organization's unique needs, providing a more flexible and customizable way of working.
- Azure Data Factory supports schema drift detection, which makes you less vulnerable to changes in upstream data sources.

Now that we've covered some design considerations, let's explore how the Bronze layer is used and its role in the broader context of data utilization. This will help us understand its importance and integration within the larger data management framework.

Usage and Governance

Some data practitioners argue that business users can benefit from querying or conducting ad hoc analyses on Bronze data. However, raw data poses significant challenges because it demands a deep understanding of the source system's design and the intricate business logic embedded within. The frequent presence of numerous small tables complicates usability. Being tightly coupled to the original source system presents significant challenges, especially when the source system undergoes changes. These complexities often discourage direct use of Bronze-layer data for business analysis.

To ensure proper governance, implement strict access controls to prevent unauthorized data access, manipulation, or deletion. The data in the Bronze layer is immutable, meaning it remains unaltered from its original state, although exceptions exist when handling overly technical data. This immutability requires maintaining detailed logs that record data ingestion times, sources, and any system interactions to enable accurate traceability.

It's crucial to set up alerts to monitor any anomalies in data size, format, or arrival times to maintain the integrity of data pipelines. Developing and regularly updating an incident

response plan is vital for addressing issues related to incorrect data deliveries effectively. Implementing best practices, such as thoroughly documenting changes and maintaining a clear reference catalog, is also recommended.

In essence, the Bronze layer serves as a foundational staging area in the Medallion architecture. It collects raw data from diverse sources and processes it by validating and converting it into the preferred format, such as Parquet or Delta. Although data in this layer is typically immutable, there are exceptions, particularly when it involves enriching the data with metadata or applying filters and transformations to manage sensitive information effectively. Data in this layer may either be completely raw or slightly augmented.

The Bronze Layer in Practice

The Bronze layer serves as the foundation for capturing and storing raw data in its most authentic form. It functions as the base input layer that reliably ingests and preserves all source data in an unaltered state, safeguarding against data loss and corruption. This makes it a reliable foundation for (re)loading and further processing.

The Bronze data layer should not be viewed as rigid. It should be tailored to meet the specific needs of your organization. This could involve adding extra landing areas, preprocessing layers to handle incremental data deliveries, and data validation parking zones, as illustrated in Figure 3-2. Or, you can treat (parts of) the Bronze layer as a virtual layer, where data is not physically stored but is instead represented by a logical representation of the source systems. By adapting the design to your organization's needs, you can maximize the effectiveness of the Bronze layer.

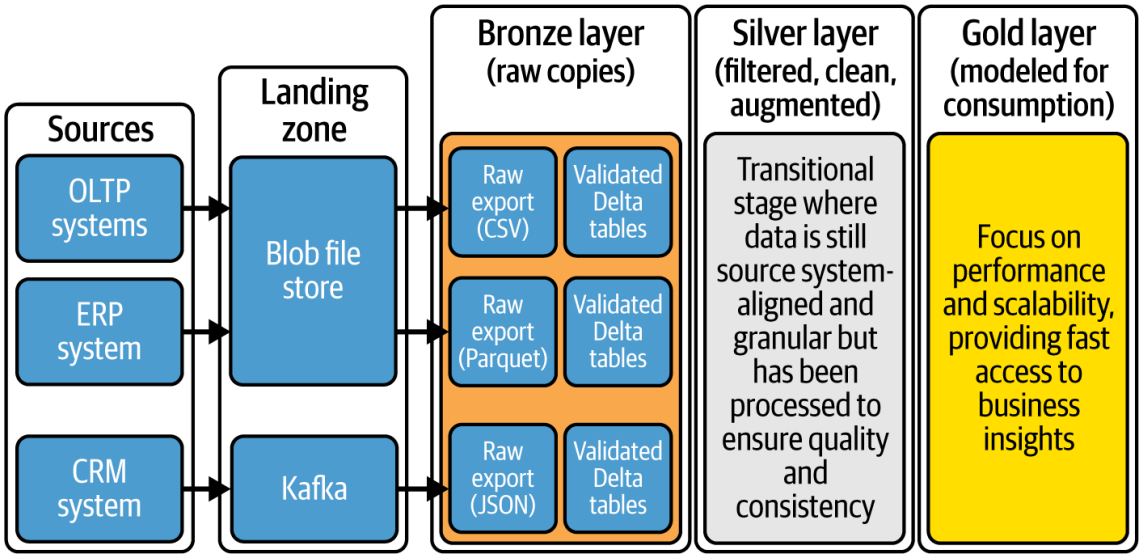

Figure 3-2. How the Bronze layer could look in practice

While the Bronze layer is indispensable for data integrity and historical accuracy, the rawness of the data limits its direct applicability for analytical purposes. Data in this state often contains inconsistencies, redundancies, and anomalies that can skew insights if not addressed. Furthermore, the Bronze layer is tightly coupled with source systems, which introduces risks and dependencies. Thus, transitioning to the Silver layer becomes essential.

Silver Layer

Having validated the data in our Bronze layer, where it now resides in a queryable state, we can progress to the Silver layer. This layer focuses on cleansing and enhancing the data. Here, formats are standardized for elements such as date and time, reference data is enforced, naming standards are conformed, duplicates are removed, and a series of functional data quality checks are conducted. Additionally, low-quality data rows are discarded and irrelevant data is filtered out. Most importantly, at this stage, data generally isn't yet merged or integrated with data from other sources.

Designing a Silver layer, similar to the Bronze layer, requires making complex decisions. Let's dive into these nuances and explore some common trade-offs. We'll start with data cleaning activities, then tackle a design consideration. Next, we'll discuss its usage, which includes operational querying and machine learning applications.

Cleaning Data Activities

Frequently, organizations encounter poor-quality source data, necessitating thorough cleansing. Many data cleansing tasks can be automated within the loading and processing phases, although some are more effectively addressed at the operational source. You want to avoid that these data quality issues pop up elsewhere. Source systems, for example, could also have interfaces to other applications. Therefore, it's recommended to fix data quality issues at the source of origination.

The data cleansing process can be governed by specific ETL rules or, in some cases, through mapping tables. It's also important to recognize that developing a cleaning process is complex and often requires multiple revisions. After loading data into the Gold layer, you may discover issues that necessitate a return to the Silver layer, and sometimes even to the Bronze layer, to resolve these issues effectively.

The following activities are examples of aspects of data cleaning:

Reducing noise and removing inauthentic data

Eliminate irrelevant data, such as unnecessary columns or rows, to enhance data quality and reduce storage requirements. This step also involves removing inauthentic data, which does not accurately reflect the true (golden) source.

Handling missing values

Assess missing data within your dataset and decide the approach for addressing it—options include removal, substitution with a default value, or employing imputation techniques to estimate missing values based on surrounding data.

Removing duplicates

Verify that your dataset is free of duplicate records unless specifically required (e.g., for retention compliance). Duplicates can distort analysis and lead to inaccurate models.

Trimming spaces

Remove unnecessary spaces in data entries, particularly in string data types where leading and trailing spaces could impact sorting, searching, and other string manipulation tasks.

Error corrections

Identify and rectify errors in data entry, such as typos, incorrect capitalization, or erroneous units. This category also covers the detection and correction of outliers—data points that deviate markedly from the norm.

Consistency checks

Confirm data consistency throughout your dataset. This includes standardizing abbreviations, terminologies, and units of measure (e.g., consistently using *NL* instead of alternating with *NETHERLANDS*).

Standardizing formats

Ensure date formats are consistent (e.g., using [YYYY]-[MM]-[DD]), recognizing that different locales may prefer alternative formats (like [DD]-[MM]-[YYYY]).

Correcting types

Ensure that columns are assigned correct data types, such as numeric, date, or floats, as appropriate.

Fixing ranges

Check that data in each column adheres to specified value ranges.

Fixing uniqueness

Ensure that data in each column maintains uniqueness, where required.

Fixing constraints

Validate that no column is orphaned; each child record must correspond to an existing parent record in the parent table.

Masking sensitive data

Conceal any PII appearing in clear text prior to data usage to protect privacy and comply with regulations.

Anomaly detection

Detect and fix anomalies that could indicate data quality issues. For instance, a sudden spike in sales data could be a sign of a data quality issue.

Applying master data management

Process data to ensure the accuracy, uniformity, and consistency of an organization's shared critical data.

Standardizing data

Process data like addresses, phone numbers, locations, and reference codes to ensure consistency and accuracy across systems.

Conforming data

Conform data using a common data model to standardize and harmonize information across different systems.

This list is not meant to be exhaustive. Writing cleaning rules is a complex task that requires a deep understanding of the data and the business processes that generated the data. Remember, after cleaning, the table structures are generally the same as in the Bronze layer. It's important to note that incorrect or rejected data isn't typically deleted. Instead, it's flagged or filtered out and then stored in a sibling quarantine table within the Silver layer. We'll revisit this topic in more detail with code snippets and examples in Chapter 6.

Now that you have cleaned the data, what should you do next? Should you retain it in its original structure, or should you remodel it? Let's explore these questions in the following sections.

Designing the Silver Layer's Data Model

The design of the Silver layer's data model is a critical aspect of the Medallion architecture. This is often a hot topic, as there are many possibilities. It is significantly influenced by the number of data sources collected, the extent to which these sources share the same key elements or objects that can be matched, related, and combined across different source systems, and the need to harmonize these overlapping key elements for an integrated perspective.

We'll start with conforming and renaming columns, followed by denormalization. Next, we'll discuss SCDs, the use of surrogate keys, and harmonization with other sources. Finally, we'll discuss 3NF and data vault modeling techniques and close with a discussion on the Silver layer's usage and governance.

Conforming and renaming columns

In a Medallion architecture, it's common for the Silver and Bronze layers to align (straight one-to-one) in terms of their tables, though the way data is presented in these layers might slightly differ.

While it may not be a standard practice everywhere, many organizations I've worked with consider renaming columns to apply consistent naming conventions a best practice in the Silver layer, and I agree with this approach. By renaming and commenting on columns, data teams ensure these names are descriptive and truly reflective of the data they represent. This simplifies both the navigation and manipulation of datasets. Additionally, it enhances communication among team members who rely on the same data, thereby minimizing confusion and reducing errors during data processing. Here is an example of how to use SQL to create a table and rename columns with technical names to more user-friendly names in a query:

```
/** Create table with user-friendly comments **/

CREATE TABLE silver.customer (

  CustomerID BIGINT COMMENT 'Customer Identifier',

  FullName STRING COMMENT 'Customer Full Name',

  Region STRING COMMENT 'Region Code',

  SignupDate DATE COMMENT 'First Sign Up Date',

  LastLogin DATE  COMMENT 'Last Login Date'
```

```
);


/** Table and field names are easy to read **/
INSERT INTO silver.customer
(CustomerID, FullName, Region, SignupDate, LastLogin)
SELECT
  custID AS CustomerID,
  CONCAT(f_name, ' ', l_name) AS FullName,
  rgcd AS Region,
  sigdt AS SignupDate,
  lldt AS LastLogin
FROM
  bronze.cust_data;
```

Renaming columns often includes standardizing and conforming data within tables to ensure consistency and reliability. For example, you can adjust data to fit within a specific range or standardizing categorical data by using consistent codes or labels. These type of standardizations in the Silver layer support more seamless transformation and integration of data for the the Gold layer, where data is refined for specific business insights.

Using appropriate data types prevents implicit type conversions, which can slow down queries by consuming extra computational resources. By selecting the smallest data type that can accommodate your data, you can also achieve benefits in terms of data storage efficiency.

Note

Applying specific ranges or standardizing categorical data within Silver tables is an activity that can be closely related to master data management (MDM). MDM focuses on ensuring that an organization's shared data—often called master data—is consistent and accurate. Standardizing data within the Silver tables aligns with these objectives, as it enhances data uniformity and reliability across the organization. We'll revisit MDM in more detail in "Master Data Management".

While it's is closely related to the Bronze layer, the practices of renaming columns, establishing uniform data types, and ensuring consistent data that take place in the Silver layer might be particularly effective for later transformation.

We've just seen how the Silver layer follows the structures in the Bronze layer. Sometimes the alignment between the Silver and Bronze layer tables isn't as tight. Deciding how strictly to align these requires considering some trade-offs. In the next section, we'll explore denormalization, which illustrates how the alignment between the Bronze and Silver layers can become less precise.

Denormalization

To optimize query performance in data modeling, you can consolidate data into fewer tables, and, by reducing the need for complex joins, queries can run faster. This process is also referred to as *denormalization*. It sometimes occurs in the Silver layer and is even more common in the Gold layer. If you expect to frequently reload and intensively read data in your Silver layer, it's best to use a more denormalized data model as it generally offers better performance.

With a denormalized model, data is organized around commonly queried subject areas, eliminating the need for extensive joins and aligning well with distributed and column-based storage architectures. This approach simplifies the data structure, making it easier to access the required information quickly and improving query efficiency. Here is an example of how to denormalize data using a SQL statement:

```
INSERT INTO silver.customer_return

SELECT

 cus.CustomerID,

 cus.FullName,

 c.Region,

 c.SignupDate,

 c.LastLogin,

 st.OrderID,

 st.OrderDate,

 st.OrderAmount

FROM

 silver.customers cus

INNER JOIN

 silver.orders st

 ON cus.OrderID = st.OrderID

INNER JOIN

 silver.customer_details c

 ON cus.CustomerID = c.CustomerID
```

When applying denormalization, table volumes start to increase because redundant data is intentionally added to improve query performance. This redundancy results in more data being stored in the table, leading to a increased table volumes, which might impact performance depending on the scenario. In light of this, many organizations manage their tables via maintenance and optimization jobs.

Slowly changing dimensions

To build a comprehensive historical record of all changes over time, it's necessary to reprocess data into what are known as slowly changing dimensions type 2 (SCD2). This involves changing tables by adding additional columns such as start_date, end_date, and is_current to track changes more effectively. However, this process is complex and challenging, as defining a "change" can vary significantly between the source and target tables. Using business keys, surrogate keys, or generating hashes is crucial in this process, as they help in making comparisons that determine how to handle the source data.

A business key

In data modeling and database design, a business key refers to an attribute or set of attributes that can be used to uniquely identify a business concept or entity. Business keys are also often referred to as natural keys. Examples include email address as a primary key in a customer relationship management (CRM) system or a Social Security number.

A surrogate key

A surrogate key is a system-generated unique identifier used to uniquely identify records within a database table. Unlike business keys, surrogate keys do not have any inherent meaning in the business context and are primarily used for technical purposes. For example, CustomerID can serve as a surrogate key in a customer database, allowing for efficient record identification and management.

A hash

In data handling, a hash refers to a fixed-size result generated from input data of arbitrary size, using a hashing algorithm. This unique result, or "hash value," serves as a digital fingerprint for data.

Keep in mind that the structure of the target tables impacts this process; any columns not included in the target should not be part of the comparison.

Within the data engineering community, there is a debate about whether building SCDs should take place in the Silver layer or be reserved for the Gold layer. Most engineers argue that the Gold layer is more appropriate for SCD2,[5] as these tables are expensive to create and store and are generally underused. Thus, Silver tables should mainly contain current records.

However, this perspective has its nuances. If your Gold layer involves consolidating and merging various data sources, there might be value in creating a historical perspective in the Silver layer, especially when the authentic context from the source is crucial. This is often the case when building machine learning models that depend on historical data in its original context. Additionally, for operational reporting, maintaining historical data in its original form could eliminate the need for an operational data store close to the source. This is increasingly applicable where organizations have decreasing access to authentic data in operational systems, such as SaaS, outsourced services, and NoSQL solutions, and so on. Ultimately, deciding whether or not to implement SCDs in the Silver layer involves nuances and depends on specific project requirements.

Surrogate keys

The debate about SCDs relates to the one about whether or not to include surrogate keys in the Silver layer. A surrogate key is a unique identifier assigned to records, which has no

inherent business meaning but serves to uniquely identify a record within a table. Surrogate keys are typically generated using the auto increment feature, or alternatively, by hashing or concatenating multiple columns to create a unique identifier. Their primary advantage is their stability and permanence—they never change. A surrogate key is helpful for tracking changes in dimension attributes over time because its value doesn't change even if the business key changes.

My perspective is that surrogate keys do not typically belong in the Silver layer. The subsequent stage, where data from different sources is combined and merged to build dimensional and fact tables, is where surrogate keys should first be created. At this stage, tables are joined using natural/business keys to look up and add surrogate keys. Therefore, the Silver layer should focus on cleaning and better representing the dataset, rather than creating surrogate keys.

However, if there is a strong preference for using surrogate keys in all SCDs, it's possible to implement them in both the Silver and Gold layers. In this scenario, the surrogate key generated in the Silver layer could be used as a lookup key to find the corresponding surrogate key in the Gold layer. This approach can work, but it requires careful implementation to ensure data consistency and integrity across layers.

We have approached the end of discussing the Silver's design. In the Medallion architecture, as you learned, the Silver and Bronze layers share a close alignment. However, there are slight differences in how data is presented within these layers. For instance, columns may be renamed, and data may undergo standardization. Additionally, the underlying storage is optimized to enhance performance. Despite these modifications, the data typically maintains its source-oriented nature. At this stage, it is not (yet) integrated with data from other sources. Let's discuss this aspect, as well the need for enrichments.

Harmonization with Other Sources

I often address questions about whether sources within the Silver layer should already be integrated across sources so that an integrated or enterprise view can be created. This type of activity is nuanced.

Generally, my advice is to keep things separate for easier management and clearer isolation of concerns. To facilitate this, I recommend you don't merge or integrate data from different source systems prematurely. Doing so can create unnecessary coupled connections between applications. For example, a user interested in data from a single source might inadvertently be linked to other systems if the data is combined immediately after being loaded into the Bronze layer. As a result, these users could potentially experience impacts from other systems.

Thus, if your goal is to maintain an isolated design, it's better to move the integration or combination of data from different sources to the Gold layer. This strategy aligns with maintaining clear boundaries and minimizing dependencies between different systems. This philosophy also applies to aligning your lakehouses with the source-system side of your architecture. Therefore, if you are building (source system–aligned) data products and are keen on aligning data ownership, I caution engineers against prematurely cross-joining data from applications across different domains.

However, I've also seen organizations that prefer to integrate data from different sources in the Silver layer. In this scenario, the Silver layer acts more as traditional data integration layer, enabling the data to be combined and harmonized across entities before moving it to the Gold layer for final consumption. In this approach, I generally see more traditional data modeling techniques being used, such as the 3NF or data vault. Let's explore these models in more detail.

3NF and Data Vault

The concept of the third normal form (3NF), as mentioned in "Inmon Methodology", is a data modeling technique that is often used in operational or transactional database normalization to reduce redundancy and dependency. Some practitioners favor using the 3NF or other normalized forms for lakehouses because of the reduced data redundancy, adaptability, and improved consistency of data it offers.

Expanding on the principles of the 3NF, data vault introduces another normalized data modeling technique. It builds on the concepts of the 3NF by incorporating unique features such as hubs (unique business keys), links (data connections), and satellite tables (detailed descriptive information about the data).

The practical application of data vault modeling concepts generally takes place in the Silver layer of the Medallion architecture. This layer hosts both the raw data vault and the business vault. The raw vault presents structured, normalized representations of raw data, mapped to a conceptual business model. It integrates various sources using business keys, ensuring resilience against schema drift and tracking historical changes. The Silver layer also includes business vault elements like harmonization and intermediate transformations, which enrich the data and align it with enterprise-wide definitions. Point-in-time (PIT) and bridge tables further improve performance and querying, preparing data for use in the Gold layer. The roles of the Bronze and Gold layers, in this setup, remain largely unchanged, with the Bronze layer serving as the raw data staging area and the Gold layer transforming data for business intelligence and analytics.

The data vault structure is known for its adaptability, excelling in environments with frequent changes in data structures and business rules. This makes it particularly suitable for organizations with complex data needs. Several reasons might lead enterprises to prefer to use the 3NF or data vault structure in their Medallion architecture:

High integration needs

Enterprises with multiple, disparate source systems benefit from the 3NF and data vault's ability to create a unified, integrated, and consistent data model.

Complex enterprise environments

Organizations operating across multiple domains or with distributed teams require the flexibility and modularity offered by the 3NF and data vault.

Rapidly changing requirements and schema drift

The data vault's resilience to change and ability to adapt to evolving schemas make it ideal for dynamic environments where business and technical needs evolve frequently. By addressing these challenges, the data vault provides a future-proof framework for managing and delivering high-quality data.

Effective management of complex time dimensions

A data vault model can effectively manage multiple active timelines within the same records, such as creation time, functional processing time, and loading time. This capability enhances the ability to track and understand data evolution, which is crucial for audits, compliance, and detailed historical analysis.

Despite these advantages, using the 3NF or data vault for the Silver layer has drawbacks. While these modeling techniques offer improved flexibility and aim to save on storage costs, they are generally less favored in cloud-based data architectures due to scalability concerns. Practitioners often opt for wide, nested, denormalized tables because this setup maximizes cloud infrastructure efficiency. Denormalized tables avoid computationally expensive joins and data shuffling between Spark compute nodes, which can significantly slow down query performance.

Building a 3NF or data vault model can also be complex and usually requires more time and resources. Both models demand a detailed understanding of data relationships and dependencies, necessitating thorough planning and analysis. They also must adhere to stringent rules and structures in order to maintain their integrity and effectiveness. To address these challenges, organizations might consider frequently reviewing and implementing automation frameworks like VaultSpeed.

Furthermore, the flexibility of lakehouse architecture causes practitioners to question the necessity of adopting a heavily normalized model like the data vault, especially if significant schema changes are not a concern. The Medallion architecture simplifies data reloading from the Bronze layer through its queryable raw original tables, and Delta supports the time travel feature, enabling quick rollbacks of Silver-layer data to previous data versions.

Tip

For extensive insights on this topic, I highly recommend Simon Whiteley's video on data vault offers.

Given the complexities and demands of models like the 3NF and data vault, organizations often choose larger denormalized tables for practical reasons. These tables offer better performance, ease of use, and simplify overall design, making them particularly appealing in environments where performance and simplicity are prioritized over strict data normalization. I'll come back to this subject when discussing the Gold layer.

In closing, data modeling is a complex topic that requires a strategic and nuanced approach tailored to specific business needs and context. It demands a careful balance of technical, business, and operational considerations. The approach to data modeling isn't simply black or white. Depending on your organization's specific requirements and constraints, you can select different data modeling strategies tailored to various use cases.

Moreover, deciding whether to integrate source systems in the Silver layer doesn't restrict you from applying some level of enterprise standardization to the data. Conforming to defined standards such as data types, centrally managed reference data, and naming conventions is encouraged. Some customers even implement basic business rules, like calculating new values. However, such activities should be kept to a minimum. The Silver layer should focus primarily on ensuring the data is clean and standardized rather than on

heavily augmenting data and applying complex business rules. Let's address the need of data enrichments in detail by studying operational querying and machine learning.

Operational Querying and Machine Learning

Machine learning models perform best when the training data is closely aligned with the specific context or domain of the business problem being addressed. Therefore, many organizations utilize the Silver layer immediately for operational querying and machine learning workloads. However, this often leads to a debate on whether to enrich the data further, especially for machine learning applications. For example, transforming categorical data into a numerical format simplifies processing by machine learning algorithms. Many organizations, recognizing the need for an additional (sandbox or machine learning) layer for feature engineering and training models, go beyond the typical three layers in their data architecture.

These considerations play a crucial role in deciding where to position certain enrichment activities. If your goal is to enable operational reporting that necessitates data enrichment, I recommend beginning the enrichment process in the Silver layer. Although this approach may require extra adjustments during the merging process in the Gold stage, the increased flexibility is worth the effort.

Alternatively, if you value maintaining flexibility and prefer to separate concerns for easier management, consider delaying the enrichment of data until the Gold layer. This strategy isolates concerns and simplifies management, making it an effective approach for handling complex data structures.

Managing Overlapping Requirements

Let's continue discussing enrichments. There's an ongoing debate among engineers about where to apply business rules—should it be in the Silver or Gold layer? This discussion often revolves around themes of reusability and standardization. Generally, I suggest keeping transformations minimal in the Silver layer. The Gold layer, which is designed for end use, is where you should place business rules. This allows for customizations that meet specific use case needs and makes it easier to manage updates and maintenance.

However, this approach can lead to complications when the same integration needs to be reused often by other teams. If data in the Gold layer is made specific for one initial business unit, another team needing the same data might find themselves reconstructing the logic. This underscores the need for thoughtful planning in how the Silver and Gold layers are structured and where you choose to enrich and transform data. We'll explore the structuring again when discussing curated and semantic layers for the Gold layer.

Automation Tasks

The key to scalability lies in automating the majority of your tasks. However, it's crucial to recognize that not all processing steps across the various layers of the Medallion architecture can be easily standardized or automated to the same degree.

When it comes to automation, different tools and frameworks present unique considerations. As illustrated in Figure 3-3, transitioning data from the source to the Bronze layer is particularly challenging. This complexity arises from the diversity of technologies and vendor solutions at the source system side. For instance, some vendors may use

unique APIs or proprietary services for data extraction, which can result in a variety of data formats. For example, if a vendor supports only a CSV export format, then adapting your processes to accommodate this format becomes necessary.

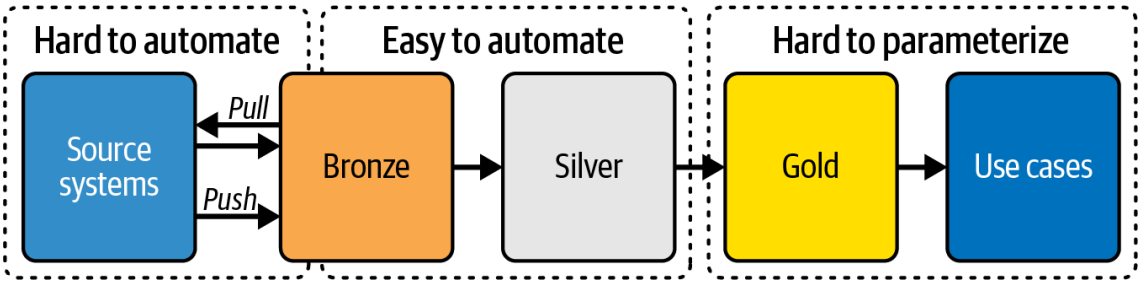

Figure 3-3. Highlighting the easily automated stages versus the complex processing steps within the Medallion architecture

Once you've overcome this initial hurdle and the data is available in a standardized (Delta Lake) format in the Bronze and Silver layers, you can proceed with further standardization and automation. At this stage, tools and metadata-driven frameworks become crucial, facilitating more streamlined and automated processing. This approach helps to maximize scalability and efficiency across the data processing architecture.

Returning to Figure 3-3, the transition from the Bronze to the Silver layer generally appears to be more straightforward compared to the one from the source systems to the Bronze layer. At this stage, transformations are usually predictable and relatively simple, involving tasks such as renaming columns, applying filters, fixing data quality issues, utilizing lookup tables, and defaulting data. Due to its predictable nature, this phase of data engineering is easier to parameterize, making it more efficient and manageable.[6] Consequently, many organizations rely on metadata-driven frameworks with common scripts and/or notebooks.

These frameworks enable you to define your data engineering tasks in a declarative manner. Essential elements like schema information, data quality rules, natural and business keys, and mapping rules are all stored within a metadata repository. This repository is then utilized to automatically generate the transformation code. By simply updating the metadata, you can effortlessly modify the transformations, which significantly automates the data engineering process.

Other frameworks to consider include dbt, the open source command-line tool previously mentioned, which excels by allowing transformations to be defined using templates, with a syntax similar to SELECT statements in SQL. Another declarative data engineering framework to consider, especially for those working within the Databricks environment, is Delta Live Tables (DLT). It not only facilitates transformations but also manages task orchestration, cluster management, monitoring, data quality, and error handling.

In Figure 3-3, the final stage involves delivering refined data to consumers based on their unique requirements. This step is challenging to parameterize, primarily due to the complex business logic often required for integration, which isn't easily captured using metadata alone. However, by employing templates and services, you can introduce a level of standardization to your workflows. This approach streamlines the process, enhancing efficiency and manageability.

Understanding and managing the variances in different stages of data engineering is crucial for creating a scalable framework. By effectively leveraging automation tools and frameworks, organizations can optimize their data processing pipelines, ensuring they are both robust and adaptable.

The Silver Layer in Practice

In the Medallion architecture, data's journey began in the Bronze layer, where raw data from diverse sources was ingested and stored without alteration to preserve its original structure and integrity. This ensured a reliable dataset that could be referenced back to its source, which is crucial for both compliance and traceability.

Moving into the Silver layer, you cleanse, standardize, and (slightly) enrich the raw data. The primary focus is on cleaning and enhancing the representation of the dataset. This involves applying minimal transformations and augmentations, conducting data quality checks, and ensuring that the data is clean, usable, and in a standard format. The Silver layer also ensures that data is queryable and primed for further processing in the Gold layer.

Tip

Refine and transform source-aligned data in the Silver layer for operational consistency. This approach allows you to maintain a single source of truth, effectively replacing traditional operational data stores with a more dynamic and scalable lakehouse solution.

The structure of the Silver layer itself, whether it constitutes a single physical layer or includes multiple stages, largely depends on specific organizational requirements. For instance, to enhance auditability, you might divide this layer into three distinct stages: one for cleansing, another for conforming to standards, and potentially a third for building SCDs.

If you want to both align data ownership and integrate data, consider setting up separate layers: one for source system-aligned cleaned data and another for harmonized data. Whatever approach you take, keeping each stage clear and consistent is crucial.

Figure 3-4 represents how the Silver layer could look in practice.

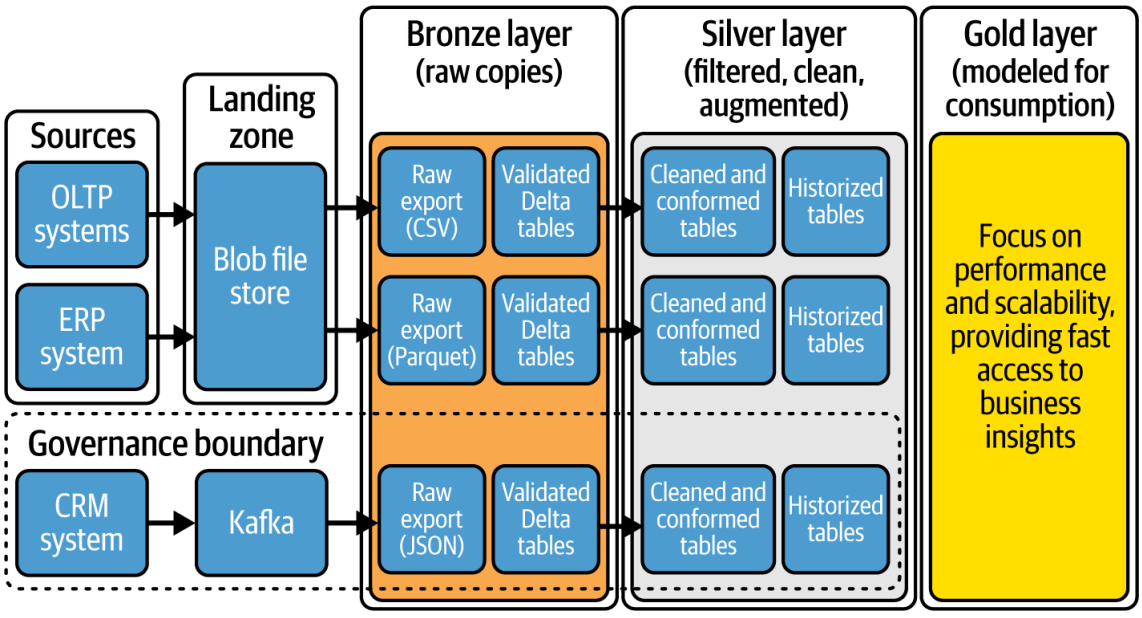

Figure 3-4. How the Silver layer could look in practice

The Silver layer typically mirrors the Bronze layer, presenting an exact representation but with crucial enhancements: tables are cleaned and data is standardized. This simple setup generally suits most organizational needs by ensuring data consistency and usability. However, the architecture of the Silver layer can be adjusted depending on the complexity of your data, the number of source systems involved, and the specific performance and flexibility requirements of your organization. This adaptability allows the Silver layer to not only address current data processing needs but also scale and evolve to meet future demands.

With data now cleaned and organized, the Gold layer takes on the role of refining this information to its most valuable form. This layer is where data is transformed into actionable insights that directly support decision-making processes. The operations in the Gold layer are complex because they involve advanced data modeling techniques, aggregation, and the application of business logic that aligns with strategic goals.

Gold Layer

Reaching the Gold layer, you encounter the most intricate part of your data architecture. This layer is the pinnacle of data refinement, designed specifically for decision making and reporting. As such, the data in the Gold layer is optimized for high-performance querying and analytics, ensuring it supports critical business functions effectively.

The complexity of the Gold layer cannot be overstated. It stems from the variety and intricacies of different source systems and the daunting task of merging all these into a single, harmonized view. This layer incorporates numerous complex business rules and engages in extensive post-processing activities such as calculations, enrichments, application-specific optimizations, corrections, transformations, aggregations, and many more.

Additionally, the work in the Gold layer is heavily influenced by business requirements, which can vary significantly. Some users may need simple flat structures, while others might require a more complex, well-modeled star schema that includes dimensions and facts. There may also be multiple user groups with potentially overlapping and contrasting requirements. This diversity of needs makes the Gold layer a challenging and complex part of the data architecture. Because the Gold layer is multifaceted, it's often broken down into multiple sublayers or stages to manage the complexity effectively.

Given this context, let's focus on the data model design by building a star schema for a straightforward Medallion architecture design. This approach and best practice will help you structure your data effectively, facilitating easier access and analysis. Once you have established a solid foundation with the star schema, you can explore the nuances of this design and further complexities that the Gold layer entails. This step-by-step approach ensures you address both the foundational and advanced aspects of data refinement in the Gold layer.

Star Schema

Star schemas will likely address most of your needs. They excel at conducting complex analyses on historical data and transforming data into entities such as an OLAP cube (a multidimensional data array based on online analytical processing). When you design star schemas, remember that their role extends beyond just boosting performance. They are

shaped based on how users interact with data. Think of your data model as a public interface, similar to an API or a function. It's essential to make this interface intuitive and logically structured, just like any tool designed for user interaction. This way, users can navigate and utilize the data more effectively.

Clarity and relevance to business users are paramount. Even if a data model boasts exceptional performance, it will fall short if it doesn't align with how users conceptualize and navigate their domain. If business users find the model unnatural for slicing, dicing, and analyzing data according to their everyday business processes, the model will not be effective.

When designing a data model, prioritize a structure that reflects the end user's intuitive mental model: the star schemas. The star schema gets its name from its shape, a central fact table with dimension tables connecting to it. This approach ensures that the data model is not only efficient but also accessible and valuable to those who rely on it for making informed business decisions.

To develop a star schema, you must first understand the business requirements by engaging with stakeholders to capture their needs and expectations. Following this, you declare the granularity of the schema, which dictates the level of detail in the fact and dimension tables, ensuring they can be properly joined. The level of granularity also determines the aggregations that will be required.

Star Schema

In Kimball methodology, there are two key types of tables used to organize and manage data:

Fact table

A fact table is the central table in a star schema. A fact table stores quantitative data for analysis and is designed to be compact, fast, and adaptable.

Dimension table

Dimension tables are used to describe dimensions of the facts; they provide the context for the data. In essence, dimension tables store attributes related to the measurements in the fact tables, which helps in making the data understandable and readable.

In a typical star schema, a single fact table is surrounded by multiple dimension tables. The dimension tables are usually less voluminous than the fact table but have more text fields. This design allows for efficient storage and fast retrieval of data, which is vital for slicing-and-dicing operations in business intelligence and data analysis.

Next, you identify the dimensions that will structure the schema. For example, an air transportation company might need dimensions for time, locations, customers, airplanes, and so on.[7] After establishing the dimensions, pinpoint the facts that will populate the schema, completing the basic framework.

Loading the star schema involves two primary tasks: loading the dimension tables and loading the fact tables. These steps are crucial for operationalizing the schema to support business analysis and decision-making processes. Let's take a closer look at each of these tasks.

Loading the dimension tables

Loading dimension tables in a star schema is complicated by the need to handle SCDs, an incremental process that involves comparing incoming data with the existing data in the dimension table. This comparison helps identify new or changed data, manage surrogate keys, and appropriately insert or update dimension records.

In this step of processing SCDs, the ETL process scans the existing dimension table for the corresponding business key. If it finds a match, the process updates the existing record and inserts a new record with updated information. If it doesn't find a match, the process inserts a new record and assigns a new surrogate key. It's crucial to note that if business keys from different sources overlap, you must make adjustments. Using source system identifiers is one effective way to manage these overlaps.

It's essential to harmonize the data before inserting records into the dimension tables. This process involves transforming data into a common format. Each source system is assessed to identify common attributes and values, a task that often requires significant business knowledge. During harmonization, records may need reorganization, cleaning, and correction. Codes might be decoded, multipart attributes could be split, and null field values might be replaced with mandatory values.

After harmonizing the records, they are ready for insertion into the dimension table. This process might involve several hops or stages. Typically, data is first loaded into a temporary table and then transferred into the dimension table. This multistage approach helps ensure data integrity and consistency in the dimension tables of the star schema.

Loading the fact tables

Loading the fact tables in a star schema, while seemingly simpler than loading dimensional tables, still presents its challenges. The primary task involves replacing the business keys, which describe business transactions, with surrogate keys linked to the dimension tables. Each row in the fact table includes foreign key references to rows in the dimension tables.

Warning

It's crucial to create the dimension tables before the fact tables because fact tables rely on dimension tables for their surrogate keys. Without the presence of business keys in the dimension tables, it becomes impossible to locate and assign the appropriate surrogate keys.

During this loading process, you might encounter situations where certain relationships between the dimensions and facts in the tables cannot be established. This is what practitioners call an *early arriving fact*. In such cases, the creation of placeholder records becomes necessary. These records represent the missing entries and help maintain the integrity of the relationships between the dimension and fact tables, ensuring the star schema functions correctly.

Optimizing loads

When loading a star schema, various bottlenecks can occur, such as excessively long lookup times that force users to endure lengthy waits. To address this issue, you can optimize the data processing by incorporating administrative columns into your tables. For

example, adding columns like type1_hash and type2_hash can streamline the detection of type 1 and type 2 changes during the ETL process.

Note

In a SCD1 model, you overwrite the old value with the new one, keeping no history. The SCD2 model preserves both current and historical records within the same file or table.

Additionally, including columns such as creation_date or update_date helps in identifying newly added data and modifications to existing data. These columns allow for a more efficient assessment of the data that needs processing, thereby optimizing the loading process and minimizing bottlenecks. This strategic approach not only enhances performance but also improves the overall efficiency of managing and updating the star schema.

Star Schema Design Nuances

Managing the ETL process and building a star schema constitute a deep and complex process; the previous sections have merely introduced a few basic concepts. As you build your own star schema, you'll encounter a variety of nuances and design considerations inherent in this approach to data modeling. You'll likely discover that the process needs to be divided into several substeps, and that what initially appears as a single Gold layer will evolve and increase in complexity over time.

The Kimball approach to developing star schemas is essentially a set of design principles. It provides guidelines and common conventions for you or your developers to follow. However, there is significant diversity in how different organizations manage their models. Let's explore some alternative approaches to give you a broader perspective on the possibilities.

Curated, Semantic, and Platinum Layers

While implementing a star schema with facts and dimensions is a popular strategy for the Gold layer, they can become increasingly complex as dimensional models expand and new data sources are incorporated. This growth often results in overlapping requirements that are similar but distinct enough to require different design approaches, potentially altering the design of the Gold layer and sometimes even the Silver layer.

For example, some organizations might add extra conformed or curation layers, with separate (semantic) layers for data marts that use star schemas. Occasionally, these additional layers are called *Platinum* layers, highlighting their specialized and highly refined nature. In such setups, it's common for organizations to follow some type of enterprise data modeling, like creating standard reference tables and conformed dimensions, which can be used across multiple data marts within a larger lakehouse architecture. Despite being considered complex and time-consuming, this approach remains popular because it ensures data reusability and standardization across different parts of the organization.

The approach of altering the design of the Gold layer comes with various nuances, influenced by factors like the size of the organization, the number of source systems intended for integration, semantic modeling requirements, and specific consumer requirements. Each of these factors can significantly impact how the data modeling strategy is implemented and sustained, making it crucial to tailor the approach to meet

specific organizational needs and capacities. Now, let's delve into another design approach that simplifies the data model.

One-Big-Table Design

While designing star schemas can be highly beneficial, the process is also labor-intensive. Consequently, some practitioners prefer to implement a one-big-table (OBT) approach due to its speed and simplicity, which can outweigh considerations of model flexibility and extensibility.

OBT involves storing all relevant data for analysis or operations in a single, large table. This method avoids distributing data across multiple tables or organizing it according to more complex schemas, such as star or snowflake schemas. This approach is favored for several reasons:

Easier to manage

OBT is often simpler to manage and understand, particularly for those who are not specialists in data warehousing. This simplicity can be advantageous for smaller teams or projects where complexity can add significant overhead. Some practitioners point out that maintaining a single big table can reduce the overhead associated with managing multiple tables and relationships in a star schema. This can lead to easier data management and lower costs in terms of both time and resources.

Better performance

For certain types of queries, especially those that do not require aggregating large volumes of data from various dimensions, a single big table can offer better performance. The elimination of joins that would be necessary in a star schema can lead to faster query execution times.

Flexible schema

A single big table provides a flexible schema that can be easier to modify and extend compared to a more rigid star schema. This can be particularly valuable in fast-paced environments where business requirements change frequently, necessitating quick adjustments to the data model.

Preferred by data scientists

Designs featuring one large table can also be preferred by data scientists who work with tools that expect data in one flat format. These OBT designs also offer convenience when transforming data into vector or graph-based datasets for modeling.

Great for long-term analysis

For datasets that inherently track changes over time (like sales or user activity data), a single, large table can make time-series analysis more straightforward. Analysts can observe historical trends and make future predictions based on a continuous stream of data.

Storing all of your data in a single table might seem to simplify the table structure, but it might make it more challenging to compose queries to extract meaningful insights. Let's break this down with a concrete example.

Consider a table called Orders where, instead of distributing data across multiple relational tables, all data is stored in a single table using multiple columns such as OrderID, OrderDate, CustomerID, and Products. Within this table, the Products column stores a nested array of data, allowing the storage of one or multiple products associated with each order. Here's an example:

OrderID: 5687

OrderDate: 2024-11-15

CustomerID: 112233

Products: [

```
 {
   "ProductID": 101,
   "ProductName": "Apple iPhone 15 Pro",
   "Quantity": 2,
   "Price": 1299.99
 },
 {
   "ProductID": 205,
   "ProductName": "Dell XPS 15",
   "Quantity": 1,
   "Price": 1899.99
 }
]
```

Although all data is stored in a single table, the nested structure of the Products column can complicate queries that require aggregating or filtering data. For instance, aggregating data based on a specific product can be challenging. This is because the data is stored in a nested format, which requires additional processing to extract the relevant information. Additionally, this strategy often leads to data duplication across multiple rows, which can increase memory demands for systems like Spark and, subsequently, degrade performance.

Managing changes within a single large table can also become complex quickly. For instance, adding a new field that impacts numerous rows can turn update operations into a significant endeavor. Typically, such changes necessitate recreating the entire table, which can be both time-consuming and resource-intensive. This highlights the challenges of maintaining a large, single-table database structure. While it offers simplicity in some areas, it requires careful consideration of potential complexities and performance issues that might arise, especially as data scales and evolves.

Serving Layer

So far, we have discussed the design and data modeling for a lakehouse, including the various layers involved. In the realm of technology architecture, especially concerning the Gold layer, you might still find it necessary to replicate data across other types of databases, in addition to lakehouses, using Delta tables. In this scenario, data is transferred from the curated or presentation layer from the lakehouse to various other services like Azure Data Explorer, Azure SQL, or a graph database service, to name a few. This transfer makes it easier for end users to access the data and caters to the diverse needs of different business lines.

Take, for example, a business unit proficient in Azure SQL within an organization. This unit has developed applications and utilizes reporting tools that depend on data stored in Azure SQL databases. Rather than having the central team manage data in the lakehouse, this business unit manages the data within their own environment. They transfer data from the lakehouse to an Azure SQL database, essentially creating a "data mart." This approach saves the business unit time in data preparation, enabling them to focus more on extracting deeper insights directly from the data. For more information, visit James Serra's blog post, "Serving Layers with a Data Lake".

Something similar is often seen with reporting tools. For instance, Power BI, a widely used reporting tool, can directly connect to Delta tables in a lakehouse. Yet, many organizations opt for Power BI's Import mode to ensure consistent performance and fine-grained security. In Import mode, Power BI replicates data from the lakehouse into its own in-memory engine, called VertiPaq.[8] This approach enhances query performance and data retrieval efficiency, which is particularly beneficial for handling large datasets in a reporting model.

Lakehouse architectures often employ a diverse mix of technology services to satisfy various needs effectively. A typical lakehouse architecture includes serverless compute for ad hoc querying, Spark for big data processing, and Delta tables where the bulk of the data is stored. Relational databases might be used for handling more complex queries, time-series databases that cater to the Internet of Things (IoT) and streaming analysis, and reporting cubes like Power BI for facilitating analytics and visualization. The decision to complement the Gold layer with an additional layer using other database technologies often hinges on usability, compatibility with other services, flexibility, performance, and cost considerations.

This configuration of additional databases is a common setup for many organizations, demonstrating that lakehouse architecture is versatile but also can be tailored and integrated to specific requirements. It's crucial to understand that lakehouse architecture does not offer a one-size-fits-all solution but rather provides a flexible framework that can adapt to the diverse needs of different organizations.

The Gold Layer in Practice

The Gold layer of the Medallion architecture plays a crucial role in optimizing data for decision making and high-performance analytics. To achieve this, aligning closely with data governance is crucial to maintain compliance, integrity, and security. It's important to document and catalog all datasets, maintain transparency about how data is used, and segment data for specific use cases. Clearly defining roles and responsibilities within this framework also ensures accountability and adherence to best practices. We'll revisit some of these concepts in Chapter 11.

Furthermore, the data stored in the Gold layer needs a structure that is straightforward, self-explanatory, and optimized (see "Managing Delta Tables") for reading. This setup must cater to various use cases that interact with this data. To design an effective model in this layer, it's crucial to align technical strategies with business needs. This eventually will result in additional physical (sub-)layers, as depicted in Figure 3-5.

Organizations seeking a competitive edge in a data-driven landscape will benefit from the flexibility offered by the various modeling approaches. While it might be challenging to master the diverse formats on offer, your consumers will benefit from the ability to fully leverage their data assets offered by utilizing the most appropriate model.

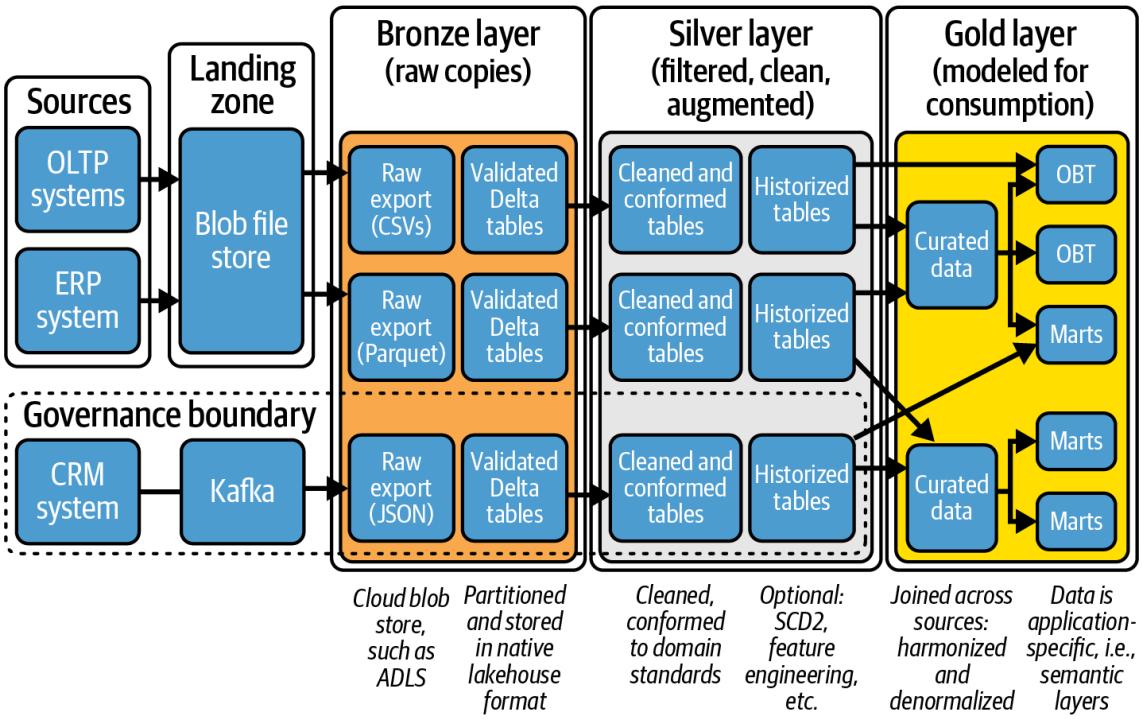

Figure 3-5. How the Gold layer could look in practice

Conclusion

The journey through the Medallion architecture and its layered approach—Bronze, Silver, and Gold—highlights that each layer of this architecture serves a unique purpose, tailored to refine data progressively from its raw form in the Bronze layer to a highly processed and decision-ready format in the Gold layer. Table 3-1 provides a high-level overview of the layers and their key characteristics.

Layer	Purpose	Data model	Applied Transformations	File and table format	ETL technology
Landing	Landing zone of raw data from	Raw, as-is	None, as-is	Delivery file formats, e.g.,	Azure Data Factory, Kafka,

Layer	Purpose	Data model	Applied Transformations	File and table format	ETL technology
	source systems	from source		CSV, Parquet, and JSON	Auto Loader, Azure Event Hubs, Databricks LakeFlow Connect
Bronze	Representation of validated raw data using standardized table formats	Source system schema	Minimal, such as applying filters and adding metadata	Native lakehouse format, e.g., Delta Lake or Iceberg	SQL, Python with frameworks, such as DLT
Silver	Clean, historized, and read-optimized version, although still source-oriented	Varies: mirrors the Bronze layer, subject-oriented, 3NF or data vault	Historized using SCD2, lightweight transformations, aligned with reference data, feature engineering, etc.	Native lakehouse format, e.g., Delta Lake or Iceberg	SQL, Python with frameworks, such as dbt or Great Expectations
Gold	Optimized for value creation	Kimball data modeling or OBT	Harmonized, aggregated and with complex business logic applied	Native lakehouse format, e.g., Delta	SQL, Python with frameworks, such as dbt,

Layer	Purpose	Data model	Applied Transformations	File and table format	ETL technology
				Lake or Iceberg	semantic models

Table 3-1. Medallion layers overview table

But what makes the Medallion architecture so crucial for your organization's data strategy? And what compelling conclusions can be drawn from this journey? The answer lies in its flexible, modular approach that allows organizations to tailor their data processes to specific needs. While the concept of three distinct layers offers a structured approach, it's not a one-size-fits-all solution. The key is understanding the strengths and limitations of each layer, which can be adapted to better align with operational realities and strategic goals.

In order to take advantage of this flexible layering, it's essential to recognize the importance of defining organization-wide standards. While flexibility is necessary, clear standards must guide engineering teams to ensure consistency and effectiveness. Given the ambiguity in layer roles and the lack of clear industry definitions, defining what is expected from each layer, how data is validated and signed off at every stage, and the exact roles and responsibilities within the architecture are critical to maintaining a robust data strategy. We'll connect back to these topics in Part IV.

Organizations that initially adopted meticulous models like the data vault sometimes faced performance issues, while those that prioritized performance with simplified models like one big table often encountered inflexibility. These experiences illustrate that effective data architecture requires a dynamic approach, balancing performance and flexibility. It is not just about choosing a model but about allowing for iterative refinements and adjustments to meet both current and future needs. To achieve this balance, organizations should consider setting clear standards for data modeling in relation to what is expected of each layer. By defining these expectations, organizations can ensure that their data is processed efficiently, but also adaptable and usable in relation to evolving business expectations.

Let's continue exploring the Medallion architecture by implementing a practical example, keeping in mind all the best practices and considerations we've learned so far. This hands-on experience will help us gain a deeper understanding of how to apply the Bronze, Silver, and Gold layers in a real-world scenario, providing valuable insights into data modeling and architecture complexities. In the next chapter, we will start by setting up the necessary infrastructure for implementing the Medallion architecture using services such as Microsoft Fabric.

1 Delta loads refer to the process of loading only the changes (newly added or modified data) since the last data load, rather than loading the entire dataset again.

2 Depending on the compression algorithm, you can expect a size reduction of nearly 75% for your data in Parquet files from other formats, such as CSV.

3 See "Table Partitioning" for more on partitioning.

4 TypeWidening allows changes within the same type category. However, it doesn't permit changes from one type category to another, like converting a string to a bool. To make such a change, you must perform a table-wide overwrite and set overwriteSchema: true.

5 I discuss the differences between SCD1, SCD2, and SCD3 in "Kimball Methodology".

6 *Parameterize* refers to the technique of defining and using parameters to control the processing steps of a data transformation pipeline. In this approach, parameters are variables or settings that can be adjusted externally without altering the underlying code of the scripts or notebooks.

7 I'll come back to this in Part III with more concrete examples.

8 VertiPaq is an in-memory columnar data storage engine used by Microsoft Power BI, as well as other Microsoft products like Analysis Services and Excel's Power Pivot.