O'REILLY®

Delta Lake: Up & Running

Modern Data Lakehouse Architectures with Delta Lake

Early Release
Raw & Unedited
Compliments of
databricks

Bennie Haelen

Delta Lake: Up and Running

Modern Data Lakehouse Architectures with Delta Lake

Bennie Haelen

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Delta Lake: Up and Running

by Bennie Haelen

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Gary O'Brien and Aaron Black
Production Editor: Beth Kelly
Copyeditor: FILL IN COPYEDITOR
Proofreader: FILL IN PROOFREADER
Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

November 2023: First Edition

Revision History for the Early Release

2022-11-10: First Release
2022-11-21: Second Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098139728* for release details.

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly and Databricks. See our statement of editorial independence.

978-1-098-13966-7

[FILL IN]

Table of Contents

CHAPTER 1

Introduction

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at gobrien@oreilly.com.

As data engineers, we want to build large-scale data, machine learning, data science, and AI solutions that offer state-of-the-art performance. We build these solutions by ingesting large amounts of source data. We cleanse, normalize and combine the data, and ultimately present it to the downstream applications through an easy to consume data model.

As the amount of data that we need to ingest and process is ever-increasing, we need the ability to horizontally scale our storage. Additionally, we need the ability to dynamically scale our compute resources to address processing and consumption spikes. Since we are combining our data sources together into one data model, we not only need to append data to tables, but we often need to insert, update, or delete (i.e., MERGE or UPSERT) records, based upon complex business logic. We want to be able to perform these operations with transactional guarantees, and without having to rewrite large data files.

In the past, the above set of requirements had been addressed by two distinct toolsets. The horizontal scalability and de-coupling of storage and compute was offered by cloud-based data lakes, while transactional guarantees were offered by relational data warehouses. However, traditional data warehouses coupled storage and compute into an on-premises appliance, and do not have the degree of horizontal scalability associated with data lakes.

Delta Lake brings capabilities such as transactional reliability and support for UPSERTs and MERGEs to data lakes while maintaining the dynamic horizontal scalability and separation of storage and compute of data lakes. Delta Lake is one the enablers for building *Data lakehouses*, an open data architecture that combines the best of data warehouses and data lakes.

The goal of this book is to provide experienced data practitioners with practical instructions on how to set up Delta Lake and start using its unique features. First, we'll discuss why Delta Lake is an important tool for building modern enterprise data platforms and data science and AI solutions, followed by instructions on how to set up Delta Lake with Spark. Each of the subsequent chapters will walk you through the fundamental functions and operations of Delta Lake using step-by-step instructions and real-world examples.

In this introduction, we will take a brief look at relational databases and how they evolved into data warehouses. Next, we will look at the key drivers behind the emergence of data lakes. We will address the benefits and drawbacks of each architecture, and finally show how the Delta Lake storage layer combines the benefits of each architecture, enabling the creation of data lakehouse solutions.

Introduction of Relational Databases

In his historic 1970 paper[1] EF Codd introduced the concept of looking at data as logical *relations*, independent of the physical data storage. This logical relation between data entities became known as a *database model* or *schema*. Codd's writings led to the birth of the relational database. The first relational database systems were introduced in the mid-1970s by IBM and UBC.

Relational databases and their underlying SQL language became the standard storage technology for enterprise applications throughout the 1980s and 1990s. One of the main reasons behind this popularity was that relational databases offered a concept called transactions. A database transaction is a sequence of operations on a database that satisfies the *ACID* properties, ACID is an acronym which stands for four properties: atomicity, consistency, isolation, and durability.

[1] Codd, E. F. (1970). Relational Database: A Practical Foundation for Productivity. San Jose: San Jose Research Laboratory.

Atomicity ensures that all changes made to the database are executed as a single operation. This means that the transaction succeeds only when all changes have been performed successfully. For example, when I use my online banking system to transfer money from savings to checking, the consistency property will guarantee that the operation will only succeed when the money is deducted from my savings account and added to my checking account. The complete operation will either succeed or fail as a complete unit.

The **Consistency** property guarantees that database transitions from one consistent state at the beginning of the transaction to another consistent state at the end of the transaction. In our earlier example, the transfer of the money would only happen if my savings account had sufficient funds. If not, the transaction would fail, and the balances would stay in their original, consistent state.

Isolation ensures that concurrent operations that are happening within the database are not affecting each other. This property ensures that when multiple transactions are executed concurrently, their operations do not interfere with each other.

Durability refers to the persistence of committed transactions. It guarantees that once a transaction is completed successfully, it will result in a permanent state even in the event of a system failure. In our money transfer example, durability will ensure that updates made to both my savings and checking account are persistent and can survive a potential system failure.

Database systems continued to mature throughout the 1990s, and the advent of the Internet in the mid-1990s led to an explosive growth of data and the need to store this data.

Enterprise applications were using the RDBMS technology very effectively. Flagship products such as SAP and Salesforce would collect and maintain massive amounts of data.

However, this development was not without its drawbacks. Enterprise applications would store the data in their own, proprietary formats, leading to the rise of *data silos*. These data silos were owned and controlled by one department or business unit. Over time, organizations recognized the need to develop an Enterprise view across these different data silos, leading to the rise of *data warehouses*.

Data Warehouses

While each enterprise application has some type of reporting built-in, business opportunities were missed because of the lack of a comprehensive view across the organization. At the same time, organizations recognized the value of analyzing data over longer periods of time. Additionally, they wanted to be able to slice and dice the data over several cross-cutting subject matters such as customers, products, and other business entities.

This led to the introduction of the data warehouse, a central relational repository of integrated, historical data from multiple data sources that presents a single integrated, historical view of the business with a unified schema, covering all perspectives of the enterprise.

Data Warehouse Architecture

A simple representation of a typical data warehouse architecture is shown in Figure 1-1.

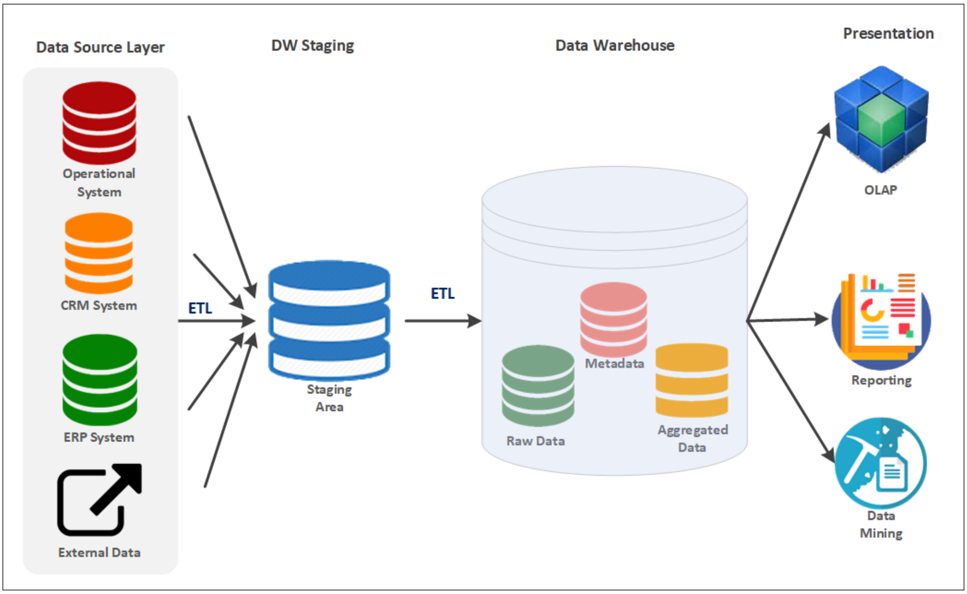


Figure 1-1. Data Warehouse Architecture

When we look at the diagram in figure x.x, we start with the data source layer on the left. Organizations need to ingest data from a set of heterogeneous data sources. While the data from the organization's ERP system(s) forms the backbone of the organizational model, we need to augment this data with the data from the operational systems that are running the day-to-day operations such as human relationship (HR) systems and workflow management software. Additionally, organizations might want to leverage the customer interaction data covered by their CRM- and POS systems. In addition to the core data sources listed above, there is a need to ingest data from a wide array of external data sources, in a variety of formats such as spreadsheets, CSV files etc.

These different source systems each have their own data format. Therefore, the data warehouse contains a *staging area* where the data from the different sources can be brought together into one common format. To do this the system must ingest the data from the original data sources. The actual ingestion process varies by data source types. Some systems allow direct database access, others allow data to be ingested through an API, while many data sources still rely on file extracts.

Next, the data warehouse needs to transform the data into a standardized format, allowing the downstream processes to easily access the data. Finally, the transformed data is loaded into the staging area. In relational data warehouses, this staging area is typically a set of flat relational staging tables without any primary- of foreign keys, and simple data types.

This process of extracting data, transforming it to a standard format and loading it into the data warehouse is commonly referred to as **E**xtract, **T**ransform and **L**oad (ETL). ETL tools can perform several other tasks on the ingested data before finally loading the data into the data warehouse. These tasks include elimination of duplicate records. Since a data warehouse will be the one source of truth, we do not want it to contain multiple copies of the same data. Additionally, duplicate records also prevent the generation of a unique primary key for each record.

ETL tools also allow us to combine data from multiple data sources. For example, one view of our customers might be captured in CRM systems while other attributes are found in an ERP system. The organization needs to combine these different aspects into one comprehensive view of a customer. This is where we start to introduce a schema to the data warehouse. In our example of a customer, the schema will define the different columns for the customer table, which columns are required, the data type and constraints of each column etc.

Having canonical, standardized representations of columns such as date and time is important, ETL tools can ensure that all these types of columns are formatted using the same standard throughout the data warehouse.

Finally, organizations want to perform quality checks on the data in keeping with their data governance standards. This might include dropping low quality data rows which do not meet this minimal standard.

Data warehouses are physically implemented on a monolithic physical architecture, made up out of a single large node, combining memory, compute and storage. This monolithic architecture forces organizations to scale their infrastructure vertically, resulting in expensive, often over-dimensioned infrastructure, which was provisioned for peak user load, while being near idle at other times

A data warehouse typically contains data that can be classified as follows:

Metadata
Contextual information about the data. This data is often stored in a data catalog. It enables the data analysts to describe, classify and easily locate the data stored in the data warehouse.

Summary data
Automatically created by the underlying data management system. The summary data will automatically be updated as new data is loaded into the warehouse. Summary data contains aggregations across several conformed dimensions. The main purpose of the summary data is to accelerate query performance.

Raw data
Maintained in its original format without any processing. Having access to the raw data enables the data warehouse system to re-process data in the case of load failures.

The data in the warehouse is consumed in the *presentation layer*. This is where the consumers can interact with the data stored in the warehouse. We can broadly identify two large groups of consumers:

Human consumers
These are the people within the organization who have a need to consume the data in the warehouse. These consumers can vary from knowledge workers, who need access to the data as an essential part of their job, to executives who typically consume highly summarized data, often in the form of dashboards and KPI's.

Internal or external Systems
The data in a data warehouse can be consumed by a variety of internal or external systems. This can include machine learning and AI toolsets, or internal applications which need to consume data in the warehouse. Some systems might directly access the data, others might work with data extracts, while still others might directly consume the data in a pub-sub model.

Human Consumers will leverage various analytical tools and technologies to create actionable insights into the data, including:

Reporting tools
These tools enable the user to develop insights into the data through visualizations such as tabular reports and a wide array of graphical representations.

Online Analytical Processing (OLAP) tools
Consumers need to slice and dice the data in a variety of ways. Online Analytical Processing (OLAP) tools present the data in a multidimensional format, allowing it to be queried from multiple perspectives. They leverage pre-stored aggregations, often stored in memory, to serve up the data with fast performance.

Data Mining

These tools allow a data analyst to find patterns in the data through mathematical correlations and classifications. They assist the analysts in recognizing previously hidden relationships between different data sources. In a way, data mining tools can be seen as a precursor to modern data science tools.

Dimensional Modeling

Data warehouses introduced the need for a comprehensive data model that spans the different subject areas in a corporate enterprise. The technique used to create these models became known as *dimensional modeling*.

Driven by the writings and ideas of visionaries such as Bill Inmon and Ralph Kimball, dimensional modeling was first introduced in Kimball's seminal book *The Data Warehouse toolkit: The Complete Guide to Dimensional Modeling*.[2] Kimball defines a methodology that focuses on a bottoms-up approach, ensuring that the team delivers real value with the data warehouse as soon as possible.

A Dimensional model is described by a *star schema.* A star schema is a way of organizing the data for a given business process (e.g., sales) into a structure which facilitates easy analytics. It consists of two types of tables:

- A *fact table*, which is the primary, or central table for the schema. The fact table captures the primary measurements, metrics, or "facts" of the business process Staying with our sales business process example, a Sales Fact Table would include:
- Units sold
- Sales amount

Fact tables have a well-defined grain. The grain of a fact table would be the level of granularity at which we store the data in our fact table. For our Sales Fact Table, the grain would be at the sales detail level.

- Multiple *dimension* tables which are related to the fact table. A Dimension provides the context surrounding the selected business process. In a Sales scenario example, the list of dimensions could include:
- Product
- Customer
- Salesperson

[2] Kimball, R. (1996). The Data Warehouse toolkit: The Complete Guide to Dimensional Modeling. The Kimball Group.

- Store

The dimension tables “surround” the fact table, which is why these types of schemas are referred to as “star schemas”.

A star schema consists of fact tables, linked to their associated dimensional tables through primary and foreign key relationships. A star schema for our sales subject area is shown in Figure 1-2.

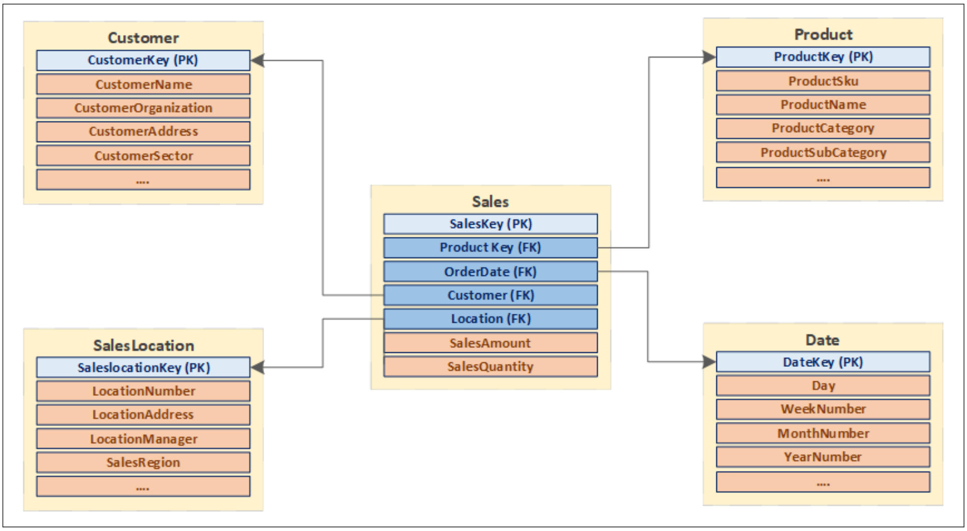


Figure 1-2. Sales Dimensional Model

Data Warehouse Benefits & Challenges

Data warehouses have inherent strengths that have served the business community well.

They serve up high quality, cleansed and normalized data from different data sources in a common format. Since data from the different departments is presented in a common format, each department will review results in line with the other departments. Having timely, accurate data is the basis for strong business decisions.

- Since they store large amounts of historical data, they enable historical insights, allowing the user to analyze different time periods and trends.
- Data warehouses tend to be very reliable, based upon the underlying relational database technology, which executes ACID transactions.
- Warehouses are modeled with standard star-schema modeling techniques, creating fact tables and dimensions. More and more pre-built template models

became available for various subject areas such as sales and CRM, further accelerating the development of such models.

- Data warehouses are ideally suited for business intelligence and reporting, basically addressing the "what happened" question of the data maturity curve. Data warehouse, combined with BI tools can generate actionable insights for marketing, finance, operations, and sales.

The fast rise of the internet, social media and the availability of multi-media devices such as smart phones disrupted the traditional data landscape, giving rise to the term *big data*. Big data is defined as data that arrives in ever higher **volumes**, with more **velocity,** and a greater **variety** of formats and has higher **veracity.** These are known as the four Vs of data:

Volume. The volume of data created, captured, copied, and consumed globally is increasing rapidly. As described in Statistica[3], over the next five years, global data creation is projected to grow to more than 200 zettabytes (a zettabyte is a 2 to the power 70 number of bytes).

Velocity. In today's modern business climate, timely decisions are critical. To make these decisions, organizations need their information to flow quickly, ideally as close to real-time as possible. For example, stock trading applications need to have access to near-real-time data so advanced trading algorithms can make millisecond decisions, and communicate these decisions to their stakeholders. Access to timely data can give organizations a competitive advantage.

Variety. Variety refers to the number of different "types" of data that are now available. The traditional data types were all structured, and typically offered as relational databases, or extracts thereof. With the rise of big data, data now arrives in new unstructured types. Unstructured and semi-structured data types, such as IoT device messages, text, audio, and video require additional preprocessing to derive business meaning.

Veracity. Veracity defines the trustworthiness of the data. Here, we want to make sure that the data is accurate and of high quality. Data can be ingested from several sources. It is important to understand the chain of custody of the data, ensure we have rich metadata, and understand the context under which the data was collected. Additionally, we want to ensure that our view of the data is complete, with no missing components or late-arriving facts.

Data Warehouses have a hard time addressing these four Vs.

[3] https://www.statista.com/statistics/871513/worldwide-data-created/

Traditional data warehouse architectures struggle to facilitate exponentially increasing data **volumes**. They suffer from both storage and scalability issues. With volumes reaching petabytes, it becomes challenging to scale storage capabilities without spending large amounts of money. Traditional data warehouse architectures do not use in-memory and parallel processing techniques, preventing them from vertically scaling the data warehouse.

Data Warehouse architectures are also not a good fit to address the **velocity** of big data. Data warehouses do not support the types of streaming architecture required to support near real-time data. ETL Data load windows can only be shortened so much until the infrastructure starts to buckle.

While data warehouses are very good at storing structured data, they are not well suited to store and query the **variety** of semi-structured or unstructured data.

Data warehouses have no built-in support for tracking the trustworthiness of the data. Data warehouse metadata is mainly focused on schema, and less on lineage, data quality and other **veracity** variables.

Further, data warehouses are based upon a closed, proprietary format and typically only support SQL-based query tools. Because of their proprietary format, data warehouses do not offer good support for data science and machine learning tools.

Because of these limitations, data warehouses are expensive to build, as a result projects often fail before going live, and those that do go live have a hard time keeping up with the ever-changing requirements of the modern business climate and the four Vs.

The limitations of the traditional data warehouse architecture gave rise to a more modern architecture, based upon the concept of a *data lake*.

Introducing Data Lakes

A Data Lake is a cost-effective central repository to store structured, semi-structured or unstructured data at any scale, in the form of files and blobs. The term “data lake” came from the analogy of a real river or lake, holding the water, or in this case data, with several tributaries which are flowing the water (aka “data”) into the lake in real-time. A canonical representation of a typical data lake is shown in Figure 1-3.

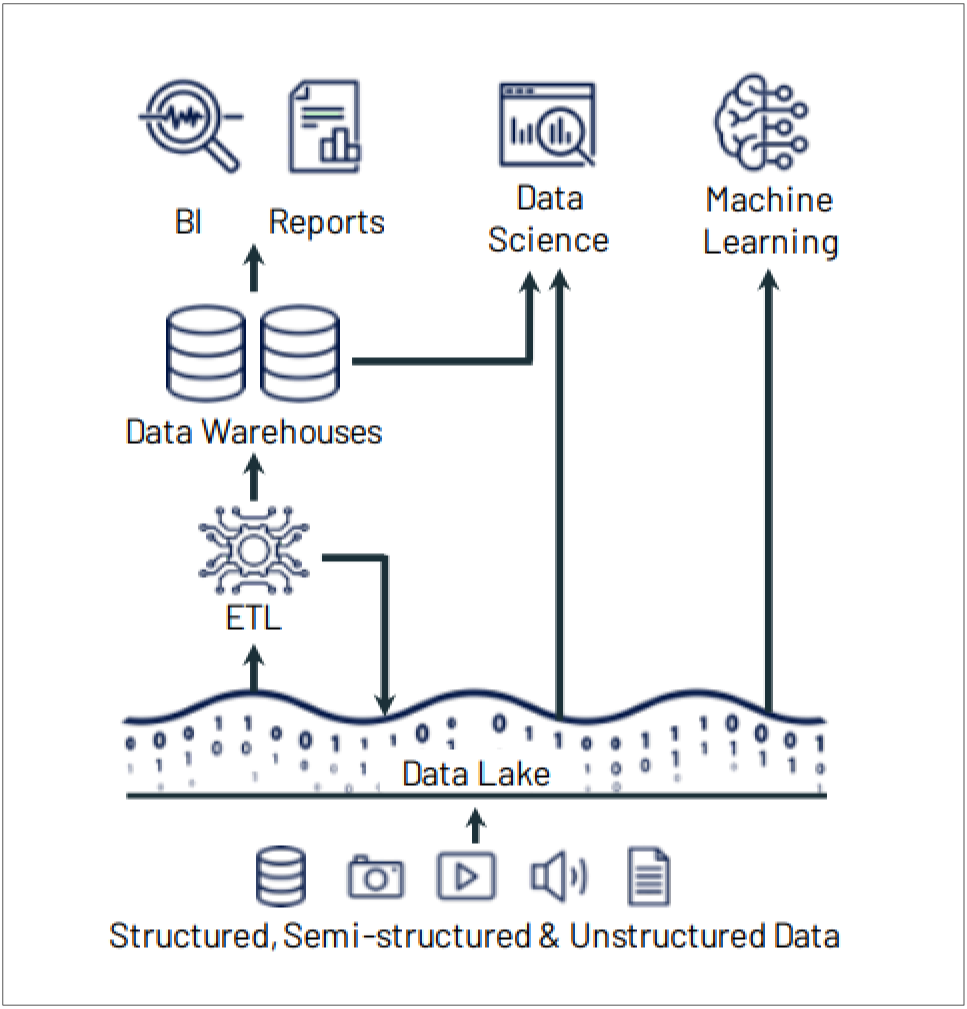


Figure 1-3. Canonical Data Lake

The initial data lakes and big data solutions were built with on-premises clusters, based upon the Apache Hadoop open-source set of frameworks. Hadoop was used to efficiently store and process large datasets ranging in size from gigabytes to petabytes of data. Instead of using one large computer to store and process the data, Hadoop leverages clustering of multiple commodity compute nodes to analyze large volumes of datasets in parallel more quickly.

Hadoop would leverage the MapReduce framework to parallelize compute tasks over multiple compute nodes. The Hadoop Distributed File System (HDFS) was a file system that was designed to run on standard or low-end hardware. HDFS was very fault-tolerant and supported large datasets.

Starting in 2015, cloud data lakes such as S3, ADLS, and GCS started replacing HDFS. These cloud-based storage systems have superior Service Level Agreements (SLAs) (often greater than 10 nines), offer geo-replication and most importantly, offer extremely low cost with the option to utilize even lower-cost cold storage for archival purposes.

At the lowest level, the unit of storage in a data lake is a blob of data. Blobs are by nature unstructured enabling the storage of semi-structured and unstructured data, such as large audio and video files. At a higher level, the cloud storage systems provide file semantics and file-level security on top of the blob storage, enabling the storage of highly structured data. Because of their high bandwidth ingress and egress channels, data lakes also enable streaming use cases, such as the continuous ingestion of large volumes of IoT data or streaming media.

Compute engines enable large volumes of data to be processed in an ETL-like fashion and delivered to consumers such as traditional data warehouses and machine learning and AI toolsets. Streaming data can be stored in real-time databases, and reports can be created with traditional BI and reporting tools.

Data lakes are enabled through a variety of technologies, as listed below:

Storage

Data lakes require very large, scalable storage systems, like the ones typically offered in cloud environments. The storage needs to durable, scalable and should offer interoperability with a variety of third-party tools, libraries, and drivers. Note that data lakes separate out the concepts of storage and compute, allowing both to scale independently. Independent scaling of storage and compute allows for on-demand, elastic fine-tuning of resources, allowing our solution architectures to be more flexible. The ingress and egress channels to the storage systems should support high bandwidths, enabling the ingestion or consumption of large batch volumes, or the continuous flow of large volumes of streaming data, such as IoT and streaming media.

Compute

High amounts of compute power are required to process the large amounts of data stored in the storage layer. Several compute engines are available on the different cloud platforms. The go-to compute engine for data lakes is Apache Spark. Spark is an open-source unified analytics engine, which can be deployed through various environments, such as Databricks, Microsoft Synapse or Snowflake. Big Data compute engines will leverage compute clusters. Compute clusters pool compute nodes to tackle complete data collection and processing tasks.

Formats

The shape of the data on disk defines the formats. A wide array of storage formats is available. Most data lakes use standardized formats such as Parquet or Avro.

Metadata

Modern, cloud-based storage systems maintain metadata (data about the data). This includes various timestamps that describe when data was written or accessed, data schemas and a variety of tags, which contain information about the usage and owner of the data.

Benefits and Drawbacks of Data Lakes

A data lake architecture enables the consolidation of an organization's data assets into one central location.

Data lakes have some very strong benefits. They use open formats, such as Parquet and Avro. These formats are well-understood by a variety of tools, drivers, and libraries which enables easy interoperability, especially in our open-source ecosystem.

Data lakes are deployed on mature cloud storage sub-systems, allowing them to benefit from the scalability, monitoring ease of deployment, and the low storage costs associated with these systems. Automated DevOps tools such as Terraform have well-established drivers, enabling automated deployments and maintenance.

Unlike data warehouses, data lakes support all data types, including semi-structured and unstructured data, enabling workloads such as media processing.

Because of their high throughput ingress channels, they are very well suited for streaming use cases, such as the ingestion of IoT sensor data, media streaming, or Web clickstreams.

However, as data lakes become more popular and widely used, organizations started to recognize some challenges with traditional data lakes. While the underlying cloud storage is relatively inexpensive, building and maintaining an effective data lake requires expert skills, resulting in high-end staffing or increased consulting services costs.

While it is easy to ingest data into the data in its raw form, transforming the data into a form that can deliver business values can be very expensive. Traditional data lakes have low latency query performance, so they cannot be used for interactive queries. As a result, the organization's data teams must still transform and load the data into something like a data warehouse, resulting in an extended time to value. This resulted in a data lake + warehouse architecture. This architecture continues to be dominant in the industry f(I have personally implemented dozens of those types of these systems), but is now declining because of the rise of data lakehouses.

Data lakes have typically used a "schema on read" strategy, where data can be ingested in any format without schema enforcement. Only when the data is read, can some type of schema be applied. This lack of schema enforcement can result in data quality issues, allowing the pristine data lake to become a "data swamp".

Data lakes do not offer any kind of transactional guarantees. Data files can only be appended to, leading to expensive re-writes of previously written data to make a simple update. This leads to an issue referred to as the "small file problem", where multiple small files are created for a single entity. If this issue is not managed well, these small files slow the read performance of the overall data lake, lead to stale data and wasted storage. Data lake administrators need to run repeated operations to consolidate these smaller files into larger files which are optimized for efficient read operations.

Now that we have discussed the strengths and weaknesses of both data warehouses and data lakes, we will introduce the data lakehouse, which combines the strengths, and addresses the weaknesses of both technologies.

Data Lakehouse

The concept of the data lakehouse was first introduced by Armbrust, Ghodsi, Xin and Zaharia.[4] The authors define a lakehouse as "*a data management system based upon low-cost and directly-accessible storage that provides analytics DBMS management and performance features such as ACID transactions, data versioning, auditing, indexing, caching and query optimization*".

When we unpack the above, we can define a lakehouse as a system which merges the flexibility, low cost, and scale of a data lake with the data management and ACID transactions of data warehouses, addressing the limitations of both. Like *data lakes*, the lakehouse architecture leverages the low-cost cloud storage systems, with the inherent flexibility and horizontal scalability that comes with those systems. It also continues to leverage the data formats like Parquet and AVRO, which enable the integration with machine learning and AI systems. Like *data warehouses*, a data lakehouse will enable ACID transactions on data assets stored on these low-cost storage systems, enabling the reliability that used to be the exclusive domain of relational database systems.

Data lakehouses allows the implementation of similar data structures and data management features to those in a data warehouse, directly on the same storage resources used for traditional data lakes. data lakehouses are a good bit for BOTH business intelligence AND machine learning/AI use cases.

[4] https://www.cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf

Data lakehouses are an especially good match for most, if not all, cloud environments with separate compute and storage resources. Different computing applications can run on-demand on completely separate computing nodes, such as a Spark cluster while directly accessing the same storage data. It is, however, conceivable that one could implement a data lakehouse over an on-premises storage system such as HDFS.

Data Lakehouse benefits

An overview of the data lakehouse architecture is shown in Figure 1-4.

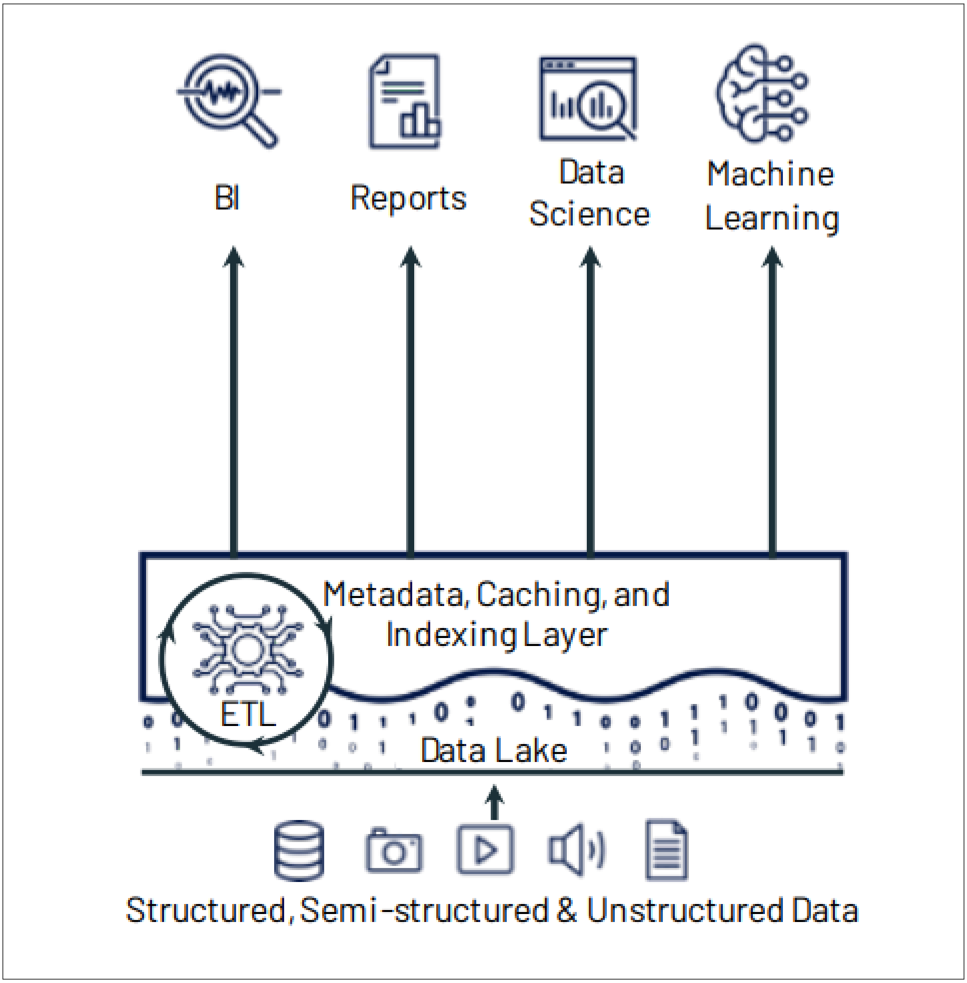

BI
Reports
Data Science
Machine Learning
Metadata, Caching, and Indexing Layer
ETL
Data Lake
Structured, Semi-structured & Unstructured Data

Figure 1-4. Data Lakehouse Architecture Overview

With the data lakehouse architecture, we no longer need to have a copy of our data in the data lake, and another copy in some type of data warehouse storage. Indeed, we can serve up our data directly from the Data Lake with comparable performance to a classical data warehouse.

Since we can continue to leverage the low-cost cloud-based storage technologies, and we no longer need to copy data from the data lake to a data warehouse, we can realize significant cost savings, both in infrastructure and staff and consulting overhead.

Since less data movement takes place and our ETL is simplified, opportunities for data quality issues are significantly reduced, and finally, because the data lakehouse combines the ability to store large data volumes and refined dimensional models, development cycles are reduced, and the time-to-value is significantly reduced.

The evolution from data warehouses to data lakes to a data lakehouse architecture is shown in Figure 1-5.

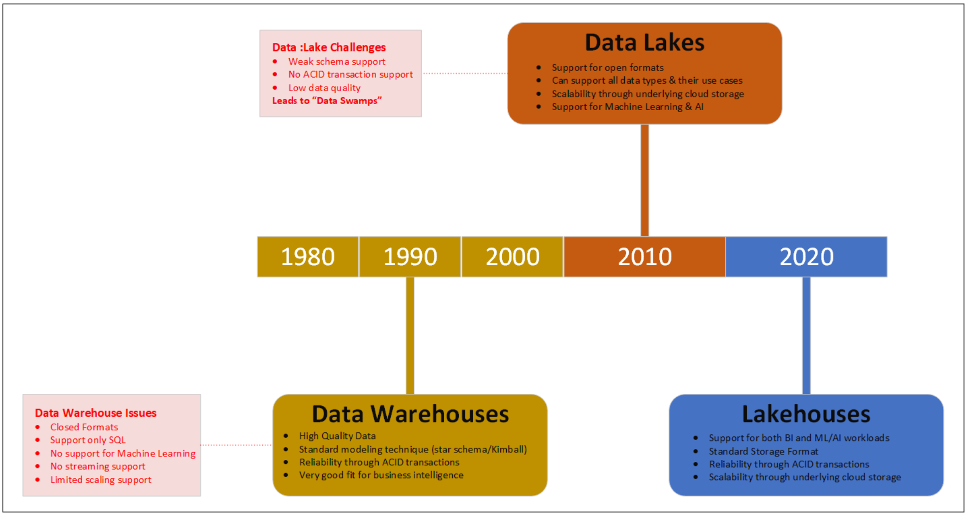

Data :Lake Challenges
• Weak schema support
• No ACID transaction support
• Low data quality
Leads to "Data Swamps"
Data Lakes
• Support for open formats
• Can support all data types & their use cases
• Scalability through underlying cloud storage
• Support for Machine Learning & AI
1980
1990
2000
2010
2020
Data Warehouse Issues
• Closed Formats
• Support only SQL
• No support for Machine Learning
• No streaming support
• Limited scaling support
Data Warehouses
• High Quality Data
• Standard modeling technique (star schema/Kimball)
• Reliability through ACID transactions
• Very good fit for business intelligence
Lakehouses
• Support for both BI and ML/AI workloads
• Standard Storage Format
• Reliability through ACID transactions
• Scalability through underlying cloud storage

Figure 1-5. Evolution of Data Architectures

Implementing a Data Lakehouse

As we mentioned earlier, data lakehouses will leverage low-cost object stores like Amazon S3 or Azure ADLS, storing the data in an open-source format such as Apache Parquet. However, since data lakehouse implementations run ACID transactions against this data, we need a transactional metadata layer on top of the cloud storage, which defines which objects are part of which table version.

This will allow a data lakehouse to implement features such as ACID transactions and versioning within that metadata layer, while keeping the bulk of the data in the low-cost object storage. The data lakehouse client is able to keep using data in an open-source file format that they are already familiar with.

Next, we need to address system performance. As we mentioned earlier, to be effective, data lakehouse implementations need to achieve great SQL performance. Data warehouses were very good at optimizing performance because they worked with a closed storage format. This allowed them to maintain statistics, build clustered indexes, move hot data on fast SSD devices etc.

In a data lakehouse, which is based upon open-source standard file formats, we do not have that luxury, since we are unable to change the storage format. However, data lakehouses can leverage a plethora of other optimizations, which leave the data files unchanged. This included caching, auxiliary data structures such as indexes and statistics and data layout optimizations.

The final tool that can speed up analytic workloads is the development of a standard DataFrame API. Most of the popular ML tools out there, such as TensorFlow and Spark MLlib already have support for DataFrames. DataFrames were first introduced by R and pandas and provide a simple table abstraction of the data with a multitude of transformation operations, most of which originate from relational algebra.

In Spark, the Dataframe API is declarative, and lazy evaluation is used to build an execution DAG (Directed Acyclic Graph). This graph can then be optimized before any action consumes that underlying data in the Dataframe. This gives the data lakehouse several new optimization features such as caching and auxiliary data.

Figure X.X shows how these requirements fit into an overall data lakehouse system design:

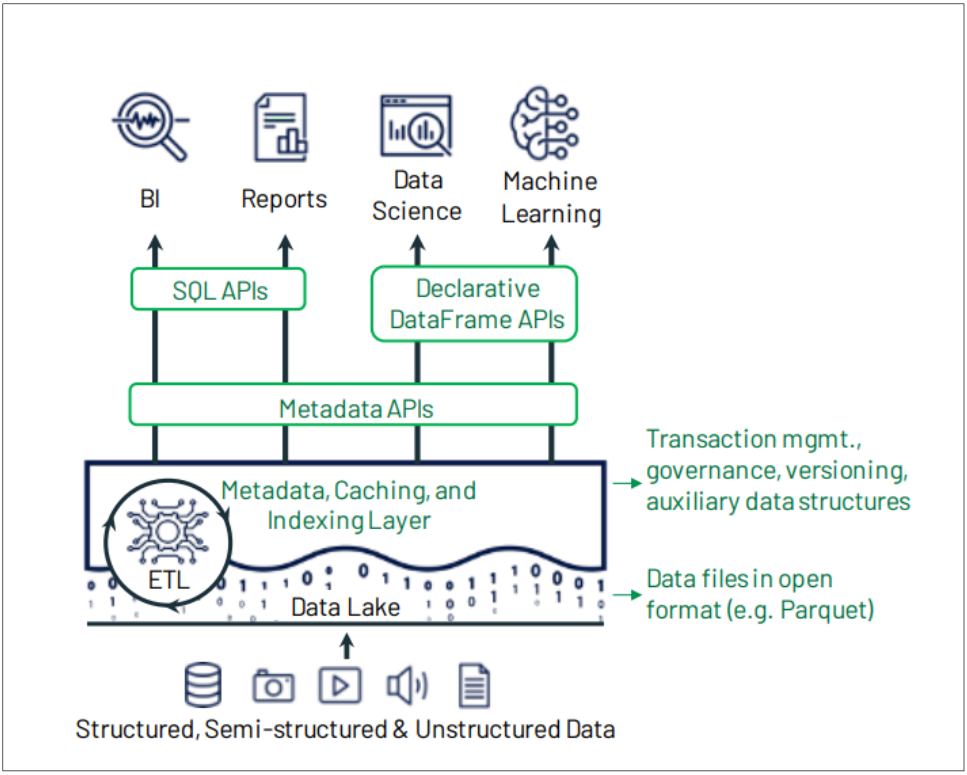

BI
Reports
Data Science
Machine Learning
SQL APIs
Declarative DataFrame APIs
Metadata APIs
Metadata, Caching, and Indexing Layer
ETL
Data Lake
Transaction mgmt., governance, versioning, auxiliary data structures
Data files in open format (e.g. Parquet)
Structured, Semi-structured & Unstructured Data

Figure 1-6. Data lakehouse implementation

A number of data lakehouse implementations are available such as Apache Iceberg and Delta Lake. Since Delta Lake is the focus of this book, we will illustrate how Delta lake facilitates the requirements for implementing a data lakehouse.

Delta Lake

As was mentioned in the previous section, a possible data lakehouse solution can be built on top of *Delta Lake*. Delta Lake is the metadata, caching and indexing layer on top of a data lake storage that provides an abstraction level to serve ACID transactions and other management features.

Delta Lake is a file-based open-source metadata layer that enables data lakehouse implementations. Delta Lake provides ACID transactions, scalable meta data handling, a unified process model that spans batch and streaming, full audit history, and support for SQL DML statements. It can run on existing data lakes and is fully compatible with several processing engines, including Apache Spark.

Delta Lake is an open-source framework, the specification of which can be found at https://delta.io. A detailed description of how Delta Lake provides ACID transaction can be found in the work of Armbrust et al[5]

Delta lake provides the following features:

Transactional ACID guarantees

Delta Lake will make sure that all data lake transactions using Spark, or any other processing engine are committed for durability and exposed to other readers in an atomic fashion. This is made possible through the Delta transaction log. In the next chapter we will cover the transaction log in detail.

Full DML support

Traditional data lakes do not support transactional, atomic updates to the data. Delta Lake fully supports all DML operations, including deletes and updates, but also complex data merge, or upsert scenarios. This support greatly simplifies the creation of dimensions and fact tables when building a modern data warehouse (MDW).

Audit History

The Delta Lake transaction log records every change made to the data, in the order that these changes were made. Therefore, the transaction log becomes the full audit trail of any changes made to the data. This enables admins and developers to roll back to earlier versions of data after accidental deletions and edits. This feature is referred to as Time Travel and is covered in detail in chapter X: Time Travel.

Unification of batch and streaming into one processing model

Delta Lake can work with batch and streaming sinks or sources. It can perform MERGEs on a data stream, which is a common requirement when merging IoT data with device reference data. It also enables use cases where we receive CDC data from external data sources. We will cover streaming in detail in chapter xx: Streaming.

Schema enforcement and evolution

Delta lake will enforce a schema when writing or reading data from the lake. However, when explicitly enabled for a data entity, it allows for a safe evolution of the schema, enabling use cases where the data needs to evolve. Schema enforcement and evolution is covered in chapter XX.

Rich metadata support and scaling

Having the ability to support large volumes of data is great, but if the metadata cannot scale accordingly, the solution will fall short. Delta Lake will scale out all metadata

[5] https://www.databricks.com/wp-content/uploads/2020/08/p975-armbrust.pdf

processing operations by leverage Spark or other compute engines, allowing it to efficiently process the metadata for petabytes of data.

Data Lakehouse-based Architectures

A data lakehouse architecture is made up out of three layers as shown in Figure 1-6. The data lakehouse storage layer is built on standard cloud-storage technology such as Azure ADLS Gen 2 storage, or AWS s3 storage. This provides the data lakehouse with a highly scalable, low-cost storage layer.

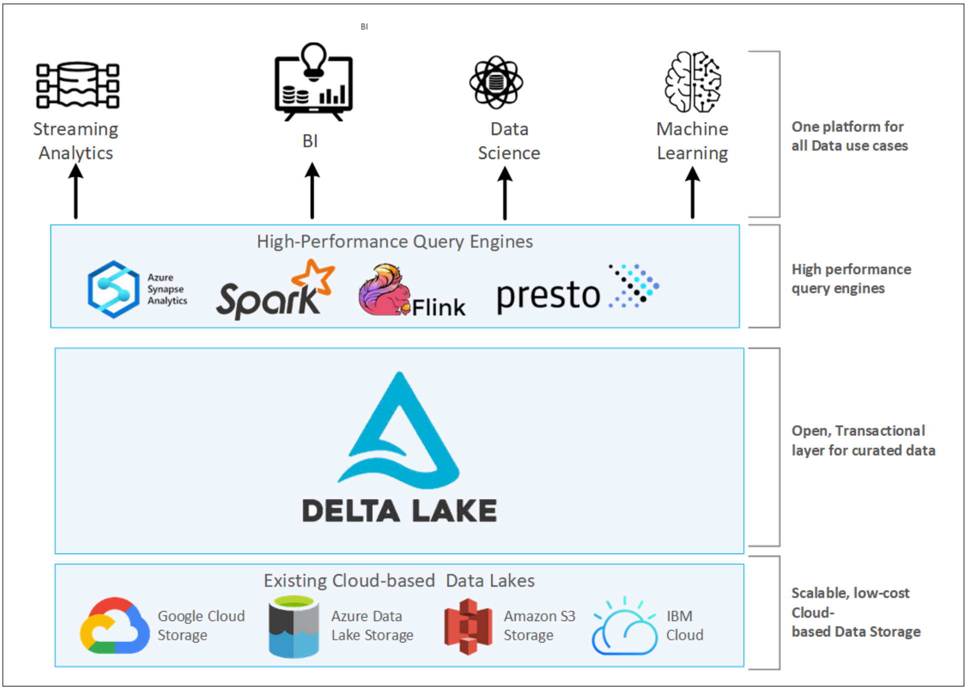

BI
Streaming Analytics
BI
Data Science
Machine Learning
One platform for all Data use cases
High-Performance Query Engines
Azure Synapse Analytics
Spark
Flink
presto
High performance query engines
DELTA LAKE
Open, Transactional layer for curated data
Existing Cloud-based Data Lakes
Google Cloud Storage
Azure Data Lake Storage
Amazon S3 Storage
IBM Cloud
Scalable, low-cost Cloud-based Data Storage

Figure 1-7. Data lakehouse Layered Architecture

The transactional layer of the data lakehouse is provided by Delta Lake. This brings ACID guarantees to the data lakehouse, enabling an efficient transformation of raw data into curated, actionable data. Besides the ACID support, Delta lake offers a rich set of additional features, such as DML support, scalable metadata processing, streaming support and a rich audit log. The top layer of the data lakehouse stack is made up out of high-performance query and processing engines, which leverage to underlaying cloud compute resources. Supported query engines include

- Apache Spark
- Apache Hive

- Presto
- Trino

Please consult the Delta Lake website (https://delta.io) for a complete list of supported compute engines.

The Medallion Architecture

An example of a Delta Lake based data lakehouse solution architecture is provided Figure 1-7. This architectural pattern with bronze, silver and gold layers is often referred to as the *Medallion Architecture*. While it is only one of many data lakehouse architecture patterns, it is a great fit for Modern Data Warehouses, Data Marts, and other analytical solutions.

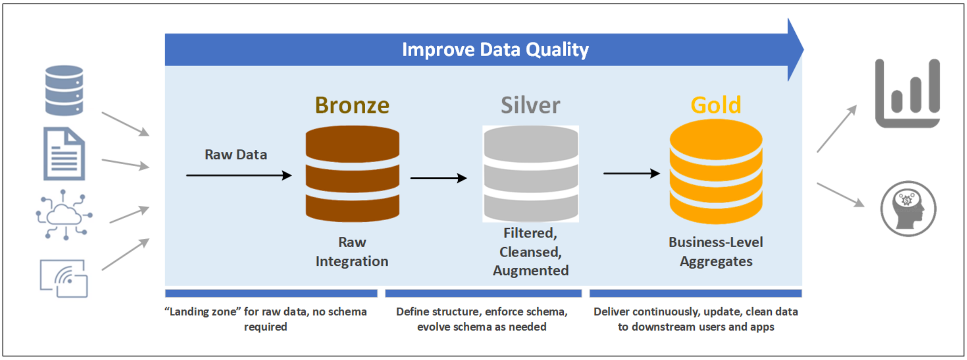

Improve Data Quality
Bronze
Silver
Gold
Raw Data
Raw Integration
Filtered, Cleansed, Augmented
Business-Level Aggregates
"Landing zone" for raw data, no schema required
Define structure, enforce schema, evolve schema as needed
Deliver continuously, update, clean data to downstream users and apps

Figure 1-8. Data Lakehouse Solution Architecture

At the highest level, we have three components in this solution. To the left we have the different data sources. A data source can take on many forms, some examples are provided below:

- A set of CSV or TXT files on an external data lake.
- An on-premises database, such as Oracle or SQL Server.
- Streaming data sources such as Kafka or Azure Event Hubs.
- REST APIs from a SAS service such as Salesforce or ADP.

The central component implements what is called the "medallion architecture". A medallion architecture is a data design pattern used to logically organize data in a data lakehouse, through a bronze, silver, and gold layer. The bronze layer is where we land the data ingested from our source systems on the left. Data in the bronze zone is typically landed "as-is", but can be augmented with additional metadata such as the loading date and time, processing identifiers etc.

In the silver layer, the data from the bronze layer is cleansed, normalized, merged and conformed. This is where the Enterprise view of the data across the different subject areas is gradually coming together.

The data in the gold layer is "consumption-ready" data. This data can be in the format of a classic star schema, containing dimensions and fact tables, or it could be in any data model that is befitting to the consuming use case.

The goal of the medallion architecture is to improve the structure and quality of the data incrementally and progressively as it flows through each layer of the architecture.

On the right we have the different consumers of the data lakehouse. This includes reporting and BI consumers, machine learning tools and AI solutions.

A summary of the Bronze, Silver and Gold layers is provided below. For each layer, we explore its business value and properties, and include implementation details (such as "how it's done):

	Bronze	Silver	Gold
Business Value	• Audit on exactly what was received from the source • Ability to reprocess without going back to the sources	• First layer that is useful to the business • Enables data discover, self-service, ad-hoc reporting, advanced analytics and ML	• The data is in a format that is easy for the business users to navigate. • Highly performant
Properties	• No business rules or transformations of any kind • Should be fast and easy to get new data to this layer	• Prioritize speed to market and write performance- just enough transformations • Quality data expected	• Prioritize business use cases and user experience • Precalculated, business-specific transformations • Can have separate views of the data for different consumption use cases
How it's done	• Must include a copy of what was received • Typically, data is stored in folders based upon the date received	• Delta Merge • Can include light modeling (3nf, vaulting) • Data quality checks should be included	• Prioritize denormalized, read-optimized data models • Full Transformed • Aggregated

In the next section, we will take a more detailed look at the different layers that make up the Medallion architecture.

The Bronze Layer (raw data)

Raw data from the data sources is ingested into the Bronze layer without any transformations or business rule enforcement. This layer is the "landing zone" for our raw data, so all table structures in this layer correspond exactly to the source system structure. The format of the data source is maintained, so when the data source is a CSV file, it is stored in Bronze as a CSV file, JSON data is written as JSON etc. Data extracted from a database table is typically landed in Bronze as a Parquet or AVRO file.

At this point, no schema is required. As data is ingested, a detailed audit record is maintained, which includes the data source, whether a full or incremental load was performed, and detailed watermarks to support the incremental loads where needed. The Bronze layer includes an archival mechanism, so that data can be retained for long periods of time. This archive, together with the detailed audit records can be used to re-process data in case of a failure somewhere downstream in the medallion architecture.

The ingested data is landed in the Bronze zone "as-is", maintaining the structure and data types of the source system format, although it is often augmented with additional metadata, such as the date and time of the load and ETL process system identifiers. The goal of the ingestion process is to land the source data quickly and easily in the bronze layer with just enough auditing and metadata to enable data lineage and re-processing.

The bronze layer is often used as a source for a Change Data Capture (CDC) process, allowing newly arriving data to be immediately processed downstream through the silver and gold layers.

The Silver Layer

In the Silver layer we first cleanse and normalize the data. We ensure that standard formats are used for constructs such as date and time, we enforce the company's column naming standard, de-duplicate the data, and perform a series of additional data quality checks, dropping low-quality data rows when needed.

Next, related data is combined and merged together. The Delta Lake MERGE capabilities work very well for this purpose. For example, customer data from various sources (Sales, CRM, POS systems etc..) is combined into a single entity. Conformed data, which are those data entities that are re-used across different subject areas is identified and normalized across the views. In our previous example, the combined customer entity would be an example of such conformed data.

At this point, the combined "Enterprise View" of the data starts to emerge. Note that we apply a "just-enough" philosophy here, where we provide just enough detail with the least amount of effort possible, making sure that we maintain our agile approach to building our medallion architecture.

At this point, we start enforcing schema, and allow the schema to evolve downstream. The Silver layer is also the layer where we can apply GDPR and/or PII/PHI enforcement rules.

Because this is the first layer where data quality is enforced, and the Enterprise view is created, it serves as a useful data source for the business, especially for purposes such as self-service analytics and ad-hoc reporting. The silver layer proves to be a great data source for machine learning and AI use cases. Indeed, these types of algorithms work best with the "less polished" data in the silver layer instead of the consumption formats in the gold layer.

The Gold Layer

In the Gold layer, we create business level aggregates. This can be done through a standard Kimball Star Schema, an Inmon Data Mart, or any other modeling technique that fits the consumer use case. The final layer of data transformations and data quality rules are applied here, resulting in high-quality, reliable data that can serve as the "single source of truth" in the organization.

The Gold layer continuously delivers high quality, clean data to the downstream users and applications. The data model in the gold layer often includes many different perspectives, or views of the data, depending on the consumption use cases.

The Gold layer will implement several Delta Lake optimization techniques, such as partitioning, data skipping and z-ordering to ensure that we deliver quality data in a performant way.

The Delta Ecosystem

As we mentioned earlier, Delta Lake enables us to build data lakehouse architectures, which enables both data warehousing and machine learning/AI applications to be hosted directly on a data lake. Today, Delta Lake is the most widely used data lakehouse format used by over 7,000 organizations, processing exabytes of data per day.

However, data warehouses and machine learning applications are not the only application target of Delta Lake. Beyond its core transactional ACID supports which brings reliability to data lakes, Delta Lake enables us to seamlessly ingest and consume both streaming and batch data with one solution architecture.

Another important component of Delta Lake is Delta Sharing, which enables companies to share data sets with each other in a secure manner.

Delta Lake 2.0 now also supports standalone readers and writers, enabling any client (Python, Ruby or Rust) to write data directly to Delta Lake without requiring any big data engine such as Spark or Flink.

Data Lake ships with an extended set of open-source connectors, including Presto, Flink and Trino.

The Delta Lake storage layer is now used extensively on many different platforms, include Azure Data Lake Storage Gen 2, Amazon s3 and Google's Cloud Storage. All components of Delta Lake 2.0 have been open sourced by Databricks.

The success of Delta Lake and data lakehouses has spawned a completely new ecosystem, build around the Delta technology. This ecosystem is made up of a variety of individual components including Delta Lake Storage, Delta Sharing, and Delta Connectors.

Delta lake storage

Delta lake storage is an open-source storage layer that runs on top of cloud-based data lakes. It adds transactional capabilities to data lake files and tables, thereby bringing data warehouse-like features to a standard data lake. Delta lake storage is the core component of the ecosystem because all other components depend on this layer.

Delta Sharing

Data sharing is a very common use case in the business world. For example, for preventative maintenance and diagnostic purposes, a mining company might want to securely share IoT information from their massive mining truck engines with the manufacturer. A thermostat manufacturer might want to securely share HVAC data with a public utility to optimize the power grid load on high-usage days. However, in the past, implementing a secure, reliable data sharing solution was very challenging, and required expensive, custom development.

Delta Sharing is an open-source protocol for securely sharing large datasets of Delta Lake data. It allows a user to securely share data stored in S3, ADLS or GCS. With Delta Sharing users can directly connect to the shared data, using their favorite toolsets like Spark, Rust, Power BI, without having to deploy any additional components. Notice that the data can be shared across cloud providers, without any custom development.

Delta sharing enables use cases such as:

- Data stored in Azure ADLS can be processed by a Spark Engine on AWS.
- Data stored in AWS S3 can be processed by Rust on GCP.

Please refer to Chapter X of this book for a detailed discussion of Delta Sharing.

Delta Connectors

The main goal of *Delta Connectors* was to bring Delta Lake to other big data engines outside of Apache Spark.

Delta connectors are open-source connectors that directly connect to Delta Lake. The framework includes *Delta Standalone* which is a Java native library which allows direct reading and writing the Delta Lake tables without requiring an Apache Spark cluster. Consuming Applications can use Delta Standalone to directly connect to Delta files written by their big data infrastructure. This eliminates the need for data duplication into another format before it can be consumed.

Other native libraries are available for:

- Hive. The Hive Connector reads Delta tables directly from Apache Hive.
- Flink. The Flink/Delta Connector reads and writes Delta tables from Apache Flink application. The connector includes a sink for writing to Delta tables from Apache Flink, and a source for reading Delta tables using Flink.
- Sql-delta-import. This connector allows for importing data from a JDBC data source directly into a Delta table.
- Power BI. The Power BI connector is a custom Power Query function which allows Power BI to read a Delta table from any file-based data source supported by Microsoft Power BI.

Delta connectors is a fast-growing ecosystem, with more connectors becoming available on a regular basis. The integrations page on the Delta Lake home page describes the currently available integrations.[6]

[6] You can go to the Delta Connectors Github page if you are interested in directly contributing to the fast-growing Delta Connectors ecosystem.

CHAPTER 2
Getting Started with Delta Lake

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at gobrien@oreilly.com.

In the previous chapter, we introduced Delta Lake, and saw how it adds transactional guarantees, DML support, auditing, a unified streaming and batch model, schema enforcement and a scalable metadata model to traditional data lakes.

In this chapter, we will go hands-on with Delta Lake. We will first set up Delta Lake on a local machine with **Spark** installed. We will run Delta Lake samples in two interactive shells:

1. First, we will run the *PySpark interactive* shell with the Delta Lake packages. This will allow us to type in and run a simple two-line Python program that creates a Delta file.
2. Next, we will run a similar program with the *Spark Scala shell*. Although we do not cover theScala language extensively in this book, we want to demonstrate that both the Spark Shell and Scala are options with Delta Lake.

Next, we will create a **helloDeltaLake** starter program in Python inside our favorite editor and run the program interactively in the **PySpark** shell. The environment we set up in this chapter and the **helloDeltaLake** program will be the basis for most other programs we create in this book.

Once our environment is up and running, we are ready to take a deeper look at the Delta file format. Since Delta Lake uses Parquet as the underlying storage medium, we first take a more detailed look at the Parquet format. Since partitions and partition files play an important role when we study the transaction log later, we will study the mechanism of both automatic and manual partitioning.

Next, we move on to Delta files and investigate how a Delta file adds a transaction log in the **_delta_log** directory.

The remainder of this chapter is dedicated to the *transaction log*. We will create and run several Python programs to investigate the details of transaction log entries, what kind of actions are recorded and what Parquet part files are written when and how they relate to the transaction log entries. We will look at more complex update examples and their impact on the transaction log. Finally, we will introduce the concept of checkpoint files and how they help Delta Lake to implement a scalable metadata system.

Getting a standard Spark Image

Setting up Spark on a local machine can be daunting. You have to adjust many different settings, update packages, and so on. Therefore, I chose to use a Docker container. If you do not have Docker installed, you can download it free from their website[1]. The specific container that I used was the standard Apache Spark image[2]. To download the image, you can use the following command:

```
docker pull apache/spark
```

Once you have pulled down the image, you can start the container with the following command:

```
docker run -it apache/spark /bin/sh
```

The spark installation is in the **/opt/spark** directory. You can find **pyspark**, **spark-sql** and all other tools in its /opt/spark/bin directory.

[1] https://www.docker.com/

[2] https://hub.docker.com/r/apache/spark

I have included several instructions on how to work with the container in the readme of the book's GitHub [3]repository.

Using Delta Lake with PySpark

As we mentioned before, Delta Lake runs on top of your existing storage and is fully compatible with the existing Apache Spark APIs. This means that it is easy to get started with Delta Lake if you already have Spark installed or use a container as specified in the previous section.

With Spark in place, you can install the **delta-spark 2.1.0** package. You can find the delta-spark package in its pyspark directory.[4]

Enter the following command in a command shell:

```
pip-install delta-spark
```

Once you have **delta-spark** installed, you can run the python shell interactively like this:

```
pyspark --packages io.delta:delta-core_2.12:2.1.0 --conf "spark.sql.exten-
sions=io.delta.sql.DeltaSparkSessionExtension" --conf "spark.sql.catalog.spark_cata-
log=org.apache.spark.sql.delta.catalog.DeltaCatalog"
```

This will give you a **PySpark** shell from which you can interactively run commands:

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.2.2
      /_/
Using Python version 3.9.13 (tags/v3.9.13:6de2ca5, May 17 2022 16:36:42)
Spark context Web UI available at http://host.docker.internal:4040
Spark context available as 'sc' (master = local[*], app id = local-1665944381326).
SparkSession available as 'spark'.
```

Inside the shell, you can now run interactive PySpark commands. I always do a quick test by creating a **range()** with Spark, resulting in a **DataFrame** that I can then save in Delta Lake format (more details in the second section of this chapter).

The full code is provided below:

```
data = spark.range(0, 10)
data.write.format("delta").mode("overwrite").save("/book/testShell")
```

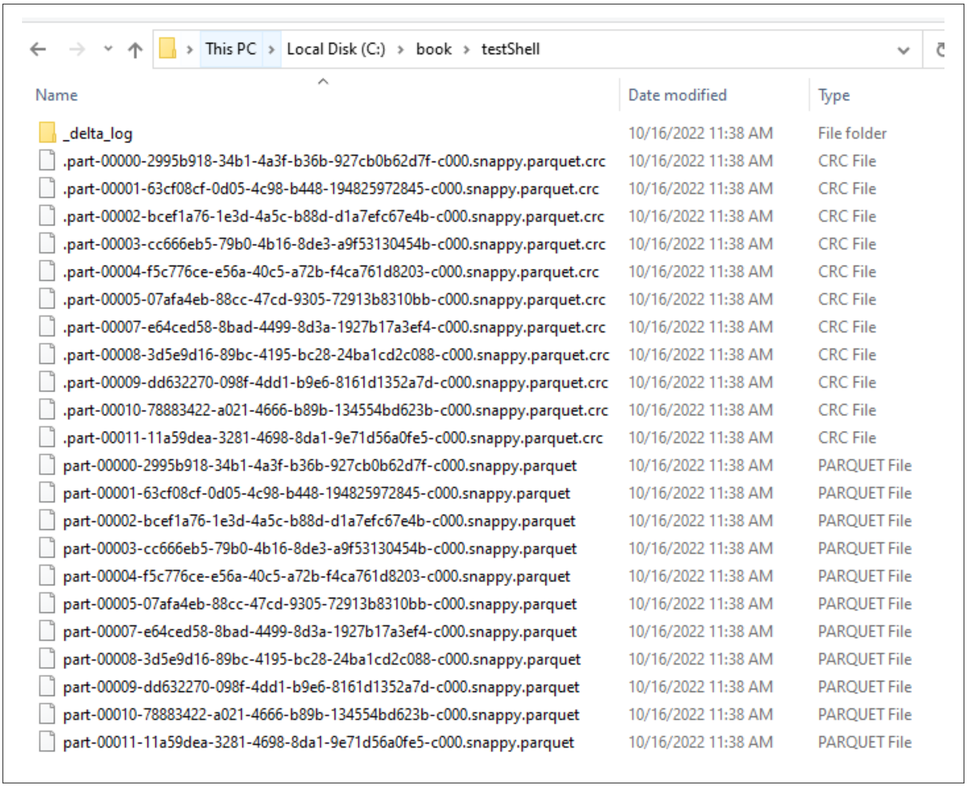
[3] https://github.com/benniehaelen/delta-lake-up-and-running

[4] https://pypi.org/project/delta-spark/2.1.0/

Below is a full run:

```
>>> data = spark.range(0, 10)
>>> data.write.format("delta").mode("overwrite").save("/book/testShell")
>>>
```

Here we see the statement to create the **range(),** followed by the **write** statement. We see that the Spark Executors do indeed run. When we open up the output directory, we can indeed find the generated Delta file as shown in figure x.x (again, more details on the Delta Lake format in the next section).



Name	Date modified	Type
_delta_log	10/16/2022 11:38 AM	File folder
.part-00000-2995b918-34b1-4a3f-b36b-927cb0b62d7f-c000.snappy.parquet.crc	10/16/2022 11:38 AM	CRC File
.part-00001-63cf08cf-0d05-4c98-b448-194825972845-c000.snappy.parquet.crc	10/16/2022 11:38 AM	CRC File
.part-00002-bcef1a76-1e3d-4a5c-b88d-d1a7efc67e4b-c000.snappy.parquet.crc	10/16/2022 11:38 AM	CRC File
.part-00003-cc666eb5-79b0-4b16-8de3-a9f53130454b-c000.snappy.parquet.crc	10/16/2022 11:38 AM	CRC File
.part-00004-f5c776ce-e56a-40c5-a72b-f4ca761d8203-c000.snappy.parquet.crc	10/16/2022 11:38 AM	CRC File
.part-00005-07afa4eb-88cc-47cd-9305-72913b8310bb-c000.snappy.parquet.crc	10/16/2022 11:38 AM	CRC File
.part-00007-e64ced58-8bad-4499-8d3a-1927b17a3ef4-c000.snappy.parquet.crc	10/16/2022 11:38 AM	CRC File
.part-00008-3d5e9d16-89bc-4195-bc28-24ba1cd2c088-c000.snappy.parquet.crc	10/16/2022 11:38 AM	CRC File
.part-00009-dd632270-098f-4dd1-b9e6-8161d1352a7d-c000.snappy.parquet.crc	10/16/2022 11:38 AM	CRC File
.part-00010-78883422-a021-4666-b89b-134554bd623b-c000.snappy.parquet.crc	10/16/2022 11:38 AM	CRC File
.part-00011-11a59dea-3281-4698-8da1-9e71d56a0fe5-c000.snappy.parquet.crc	10/16/2022 11:38 AM	CRC File
part-00000-2995b918-34b1-4a3f-b36b-927cb0b62d7f-c000.snappy.parquet	10/16/2022 11:38 AM	PARQUET File
part-00001-63cf08cf-0d05-4c98-b448-194825972845-c000.snappy.parquet	10/16/2022 11:38 AM	PARQUET File
part-00002-bcef1a76-1e3d-4a5c-b88d-d1a7efc67e4b-c000.snappy.parquet	10/16/2022 11:38 AM	PARQUET File
part-00003-cc666eb5-79b0-4b16-8de3-a9f53130454b-c000.snappy.parquet	10/16/2022 11:38 AM	PARQUET File
part-00004-f5c776ce-e56a-40c5-a72b-f4ca761d8203-c000.snappy.parquet	10/16/2022 11:38 AM	PARQUET File
part-00005-07afa4eb-88cc-47cd-9305-72913b8310bb-c000.snappy.parquet	10/16/2022 11:38 AM	PARQUET File
part-00007-e64ced58-8bad-4499-8d3a-1927b17a3ef4-c000.snappy.parquet	10/16/2022 11:38 AM	PARQUET File
part-00008-3d5e9d16-89bc-4195-bc28-24ba1cd2c088-c000.snappy.parquet	10/16/2022 11:38 AM	PARQUET File
part-00009-dd632270-098f-4dd1-b9e6-8161d1352a7d-c000.snappy.parquet	10/16/2022 11:38 AM	PARQUET File
part-00010-78883422-a021-4666-b89b-134554bd623b-c000.snappy.parquet	10/16/2022 11:38 AM	PARQUET File
part-00011-11a59dea-3281-4698-8da1-9e71d56a0fe5-c000.snappy.parquet	10/16/2022 11:38 AM	PARQUET File

Figure 2-1. Delta File generated from the Python Shell

Running Delta Lake in the Spark Scala shell

We can also run Delta Lake in the interactive Spark Scala shell. As specified in the Delta Lake Quickstart, we can start the Scala shell with the Delta Lake packages as follows:

```
spark-shell --packages io.delta:delta-core_2.12:2.1.0 --conf "spark.sql.exten-
sions=io.delta.sql.DeltaSparkSessionExtension" --conf "spark.sql.catalog.spark_cata-
log=org.apache.spark.sql.delta.catalog.DeltaCatalog"
```

This will start up the interactive Scala shell:

```
Spark context Web UI available at http://host.docker.internal:4040
Spark context available as 'sc' (master = local[*], app id = local-1665950762666).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.2.2
      /_/
Using Scala version 2.12.15 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_311)
Type in expressions to have them evaluated.
Type :help for more information.
scala>
```

Inside the shell, you can now run interactive Scala commands. Let's do a similar test on Scale as we did for the PySpark shell:

The full code is provided below:

```
val data = spark.range(0, 10)
data.write.format("delta").mode("overwrite").save("/book/testShell")
```

Below is a full run:

```
Scala> val data = spark.range(0, 10)
data: org.apache.spark.sql.Dataset[Long] = [id: bigint]
scala> data.write.format("delta").mode("overwrite").save("/book/testShell")
```

And, again when we check our output, we can find the generated Delta Lake table/data.

Running Delta Lake on Databricks

If you do not want to run Spark and Delta Lake on your local machine, you also have the option of running Delta Lake on Databricks on a cloud platform, like Azure, AWS, or Google cloud. These environments make it easy to get started with Delta Lake, since their installed runtimes already have a version of Delta Lake installed.

The additional benefit of the cloud is that you can create real Spark clusters, anything from a 4 node and 12 cores per node configuration with thousands of cores spanning hundreds of nodes to process terabytes or petabytes of data.

When using Databricks in the cloud, you have two options. You can either use their popular notebooks or connect your favorite development environment to a cloud-based Databricks cluster with dbx[5]. Dbx by Databricks labs is an open source tool that

[5] https://docs.databricks.com/dev-tools/dbx.html

allows you to connect to a Databricks cluster from an editing environment such as Visual Studio Code.

If you are not ready to sign up for a full-fledged cloud account, you can leverage the Databricks Community Edition[6]. The Databricks Community Edition is the free version of the Databricks cloud-hosted big data platform. Users can access a cluster with a complete notebook environment and an up-to-date runtime with Delta Lake installed on this platform.

Creating and Running a Spark Program: helloDeltaLake

Once we have the delta-spark packages installed creating your first PySpark program is very straightforward.

Follow these steps to create the PySpark program:

1. Create a new file (I named mine helloDeltaLake.py)
2. Add the necessary imports. At a minimum you need to import PySpark and Delta Lake:

   ```
   import pyspark
   from delta import *
   ```

3. Next, create a SparkSession builder which loads up the Delta Lake extensions, as follows:

   ```
   # Create a builder with the Delta extensions
   builder = pyspark.sql.SparkSession.builder.appName("MyApp")       \
     .config("spark.sql.extensions",                                  \
                   "io.delta.sql.DeltaSparkSessionExtension")         \
     .config("spark.sql.catalog.spark_catalog",                       \
                   "org.apache.spark.sql.delta.catalog.DeltaCatalog")
   ```

4. Next, we can create our SparkSession object itself. We will create the SparkSession object, and print out its version to ensure that the object is valid:

   ```
   # Create a Spark instance with the builder
   # As a result, we now can read and write Delta files
   spark = configure_spark_with_delta_pip(builder).getOrCreate()
   print(f"Hello, Spark version: {spark.version}")
   ```

5. To verify that our Delta Lake extensions are working correctly, we create a range and write it out in Delta Lake format:

   ```
   # Create a range, and save it in Delta Lake format to ensure
   # that our Delta Lake extensions are indeed working
   df = spark.range(0, 10)
   ```

[6] https://community.cloud.databricks.com/

```
    df.write                                   \
      .format("delta")                         \
      .mode("overwrite")                       \
      .save("/book/chapter02/helloDeltaLake")
```

That completes the code for our starter program. You can find the full code file in the /chapter02/helloDeltaLake.py location of the book's code repository. This code is a good place to start if you want to write your own code.

Running helloDeltaLake

To run our program, we can simply start a command prompt on Windows, or a terminal on MacOS, and navigate to the folder with our code.

We simply start pyspark with our program as input, as shown below:

```
pyspark < helloDeltaLake.py
```

When we run the program, we get our Spark version output:

```
Hello, Spark version: 3.2.2
```

And when we look at our output, we can see that we have written a valid Delta file. The details of the Delta Lake format are covered in the next section.

At this point, we have **PySpark** and Delta Lake installed successfully, and we were able to code and run a full-fledged **pySpark** program with Delta Lake extension.

I will be using the above approach to write and run most of the programs in this book. If I deviate from this standard approach, I will explicitly call it out in the book.

Now that we can run our own programs, we are ready to explore the Delta Lake format in detail in the next section.

The Delta Lake Format

In this section we will dive deeper into the Delta Lake format. When we save a file using the Delta Lake format, we are just writing a standard Parquet file with additional metadata.

Since Delta Lake writes out data as a Parquet file, we will take a more in-depth look at Parquet files. We first write out a simple Parquet file and take a detailed look at the artifacts written by Spark. This will give us a good understanding of files, which we will leverage throughout this book.

Next, we will write out a file in Delta Lake format, noticing how it triggers the creation of the **_delta_log** directory, containing the transaction log. We will take a detailed look at this transaction log and how it is used to generate a single source of truth. We will see how the transaction log implements the ACID *atomicity* property mentioned in chapter 1.

We will see how Delta Lake breaks down a transaction into individual, atomic commit actions, and how it records these actions in the transaction log as ordered, atomic units.

Finally, we will look at several use cases and investigate what Parquet part files and transaction log entries are written, and what is stored in these entries.

Since a transaction log entry is written for every transaction, we might end up with multiple small files. To ensure that this approach remains scalable, Delta Lake will create a checkpoint file every 10 transactions with the full transactional state. This way, a Delta Lake reader can simply process the checkpoint file, and the few transaction entries written afterwards. This results in a fast, scalable metadata system.

Parquet Files

The Apache Parquet file format has been one of the most popular big data formats for the last 20 years. Unlike row-based formats such as CSV or Avro, Parquet is column-oriented, meaning that the values of each table are stored next to each other, rather than in each record. Second, Parquet is open source, so it is free to use under the Apache Hadoop license and is compatible with most Hadoop data processing frameworks. Parquet is self-describing. In addition to the actual data, it contains metadata, including the schema and structure of the file.

Parquet files perform compression on a column-by-column basis and are built to support flexible compression options and extendable encoding schemas for each data type. For example, a different encoding can be used for compressing integer versus string data.

Writing a Parquet File

In the book repository, the **/chapter02/writeParquetFile** Python program creates a Spark DataFrame in memory, and writes it in Parquet format to the /parquetData folder using the standard PySpark API:

```
data = spark.range(0, 100)
data.write.format("parquet")     \
          .mode("overwrite")    \
          .save('/book/chapter02/parquetData')
```

When we look at what is written to disk in this case, we see the following:

```
 Directory of C:\book\chapter02\parquetData
10/17/2022  09:05 AM    <DIR>          .
10/17/2022  09:05 AM    <DIR>          ..
10/17/2022  09:05 AM               12 .part-00000-a3885270-f443-40dc-
```

```
bdf9-8ee9946cbb79-c000.snappy.parquet.crc
10/17/2022  09:05 AM                16 .part-00001-a3885270-f443-40dc-
bdf9-8ee9946cbb79-c000.snappy.parquet.crc
10/17/2022  09:05 AM                16 .part-00002-a3885270-f443-40dc-
bdf9-8ee9946cbb79-c000.snappy.parquet.crc
10/17/2022  09:05 AM                16 .part-00003-a3885270-f443-40dc-
bdf9-8ee9946cbb79-c000.snappy.parquet.crc
10/17/2022  09:05 AM                16 .part-00004-a3885270-f443-40dc-
bdf9-8ee9946cbb79-c000.snappy.parquet.crc
10/17/2022  09:05 AM                16 .part-00005-a3885270-f443-40dc-
bdf9-8ee9946cbb79-c000.snappy.parquet.crc
10/17/2022  09:05 AM                16 .part-00006-a3885270-f443-40dc-
bdf9-8ee9946cbb79-c000.snappy.parquet.crc
10/17/2022  09:05 AM                16 .part-00007-a3885270-f443-40dc-
bdf9-8ee9946cbb79-c000.snappy.parquet.crc
10/17/2022  09:05 AM                16 .part-00008-a3885270-f443-40dc-
bdf9-8ee9946cbb79-c000.snappy.parquet.crc
10/17/2022  09:05 AM                16 .part-00009-a3885270-f443-40dc-
bdf9-8ee9946cbb79-c000.snappy.parquet.crc
10/17/2022  09:05 AM                16 .part-00010-a3885270-f443-40dc-
bdf9-8ee9946cbb79-c000.snappy.parquet.crc
10/17/2022  09:05 AM                16 .part-00011-a3885270-f443-40dc-
bdf9-8ee9946cbb79-c000.snappy.parquet.crc
10/17/2022  09:05 AM                 8 ._SUCCESS.crc
10/17/2022  09:05 AM               511 part-00000-a3885270-f443-40dc-
bdf9-8ee9946cbb79-c000.snappy.parquet
10/17/2022  09:05 AM               513 part-00001-a3885270-f443-40dc-
bdf9-8ee9946cbb79-c000.snappy.parquet
10/17/2022  09:05 AM               517 part-00002-a3885270-f443-40dc-
bdf9-8ee9946cbb79-c000.snappy.parquet
10/17/2022  09:05 AM               513 part-00003-a3885270-f443-40dc-
bdf9-8ee9946cbb79-c000.snappy.parquet
10/17/2022  09:05 AM               513 part-00004-a3885270-f443-40dc-
bdf9-8ee9946cbb79-c000.snappy.parquet
10/17/2022  09:05 AM               517 part-00005-a3885270-f443-40dc-
bdf9-8ee9946cbb79-c000.snappy.parquet
10/17/2022  09:05 AM               513 part-00006-a3885270-f443-40dc-
bdf9-8ee9946cbb79-c000.snappy.parquet
10/17/2022  09:05 AM               513 part-00007-a3885270-f443-40dc-
bdf9-8ee9946cbb79-c000.snappy.parquet
10/17/2022  09:05 AM               517 part-00008-a3885270-f443-40dc-
bdf9-8ee9946cbb79-c000.snappy.parquet
10/17/2022  09:05 AM               513 part-00009-a3885270-f443-40dc-
bdf9-8ee9946cbb79-c000.snappy.parquet
10/17/2022  09:05 AM               513 part-00010-a3885270-f443-40dc-
bdf9-8ee9946cbb79-c000.snappy.parquet
10/17/2022  09:05 AM               517 part-00011-a3885270-f443-40dc-
bdf9-8ee9946cbb79-c000.snappy.parquet
10/17/2022  09:05 AM                 0 _SUCCESS
              26 File(s)          6,366 bytes
               2 Dir(s)  767,409,405,952 bytes free
```

A developer new to the big data world might be a bit shocked at this point. We only did a single write of 100 numbers, so how did we end up with 12 files? A bit of elaboration might be in order.

First, the file name you specify in the write is really the name of a *directory*, not really a file. As we can see the directory /parquetData is a directory containing 12 files.

When we look at the **.parquet** files, we see that we have 12 files. Spark is a highly parallel computational environment, where the system is attempting to keep each CPU core in your Spark cluster busy. In my case, I am running on my local machine, which means there is one machine in my cluster. When I look at the information for my system, I see that I have 12 cores.

When we look at the number of **.parquet** files that were written, we see that we have 12 files, which is equal to the number of cores on my cluster. And indeed, that is Spark's default behavior in this scenario. The number of files will be equal to the number of available cores. If we add the following statement to our code:

```
data = spark.range(0, 100)
data.write.format("parquet")     \
          .mode("overwrite")    \
          .save('/book/chapter02/parquetData')
print(f"The number of files is: {data.rdd.getNumPartitions()}")
```

We can see in our output that we indeed have 12 files:

```
The number of files is: 12
```

While this might look like overkill for this scenario where we are only writing 100 numbers, one can imagine a scenario where we are reading or writing very large datasets consisting of many files. Having the ability to split large datasets that can be processed in parallel can dramatically increase performance.

Writing a Delta File

So far, we have been working with Parquet files only. Now, let's take the first example from the previous section and save it in Delta Lake format instead of Parquet (code: **/chapter02/writeDeltaFile.py**) All we need to do is replace the **parquet** format with **delta**, as is shown in the code:

```
data = spark.range(0, 100)
data.write               \
    .format("delta")     \
     .mode("overwrite")   \
     .save('/book/chapter02/deltaData')
print(f"The number of files is: {data.rdd.getNumPartitions()}")
```

We get the same number of files:

```
The number of files is: 12
```

And when we look at the output, we see the addition of the **_delta_log** file, as shown below:

```
 Directory of C:\book\chapter02\deltaData
10/17/2022  09:36 AM    <DIR>          .
10/17/2022  09:36 AM    <DIR>          ..
10/17/2022  09:36 AM                16 .part-00000-a707e0b9-
b984-4642-976c-528a71a0b060-c000.snappy.parquet.crc
10/17/2022  09:36 AM                16 .part-00001-1f8ad5e6-836d-4ab7-a2f6-
dd73d22fb37b-c000.snappy.parquet.crc
10/17/2022  09:36 AM                16 .part-00002-
fd9f62af-024f-4097-9b1b-68e50858a871-c000.snappy.parquet.crc
10/17/2022  09:36 AM                16 .part-00003-23f8466e-a624-4987-
b786-5c8c1dd7d756-c000.snappy.parquet.crc
10/17/2022  09:36 AM                16 .part-00004-d4927a28-fa66-447b-
b9f9-067f951c9e08-c000.snappy.parquet.crc
10/17/2022  09:36 AM                16 .part-00005-fe003624-6ccf-49cc-a780-
bb16a601b2fd-c000.snappy.parquet.crc
10/17/2022  09:36 AM                16 .part-00006-
ee7f03c3-32a7-4c35-84d1-75d99ed658a8-c000.snappy.parquet.crc
10/17/2022  09:36 AM                16 .part-00007-32c9c8e1-e0a5-485e-baac-
f9d7a323c046-c000.snappy.parquet.crc
10/17/2022  09:36 AM                16 .part-00008-0f06006a-dea6-46d9-ba1f-
b5ba05c74ff7-c000.snappy.parquet.crc
10/17/2022  09:36 AM                16 .part-00009-9f1fbe69-d9ba-4adc-a846-
f2b5687c504e-c000.snappy.parquet.crc
10/17/2022  09:36 AM
16 .part-00010-19b88404-209e-422b-94aa-42e10c17bef0-c000.snappy.parquet.crc
10/17/2022  09:36 AM
16 .part-00011-0a84e9db-50bd-4eff-8b47-06be16983ad4-c000.snappy.parquet.crc
10/17/2022  09:36 AM               524 part-00000-a707e0b9-
b984-4642-976c-528a71a0b060-c000.snappy.parquet
10/17/2022  09:36 AM               519 part-00001-1f8ad5e6-836d-4ab7-a2f6-
dd73d22fb37b-c000.snappy.parquet
10/17/2022  09:36 AM               523 part-00002-
fd9f62af-024f-4097-9b1b-68e50858a871-c000.snappy.parquet
10/17/2022  09:36 AM               519 part-00003-23f8466e-a624-4987-
b786-5c8c1dd7d756-c000.snappy.parquet
10/17/2022  09:36 AM               519 part-00004-d4927a28-fa66-447b-
b9f9-067f951c9e08-c000.snappy.parquet
10/17/2022  09:36 AM               522 part-00005-fe003624-6ccf-49cc-a780-
bb16a601b2fd-c000.snappy.parquet
10/17/2022  09:36 AM               519 part-00006-
ee7f03c3-32a7-4c35-84d1-75d99ed658a8-c000.snappy.parquet
10/17/2022  09:36 AM               519 part-00007-32c9c8e1-e0a5-485e-baac-
f9d7a323c046-c000.snappy.parquet
10/17/2022  09:36 AM               523 part-00008-0f06006a-dea6-46d9-ba1f-
b5ba05c74ff7-c000.snappy.parquet
10/17/2022  09:36 AM               519 part-00009-9f1fbe69-d9ba-4adc-a846-
f2b5687c504e-c000.snappy.parquet
```

```
10/17/2022  09:36 AM             519
part-00010-19b88404-209e-422b-94aa-42e10c17bef0-c000.snappy.parquet
10/17/2022  09:36 AM             523
part-00011-0a84e9db-50bd-4eff-8b47-06be16983ad4-c000.snappy.parquet
10/17/2022  09:36 AM    <DIR>          _delta_log
              24 File(s)          6,440 bytes
```

The **_delta_log** file contains the transaction log with every single operation performed on our data.

The Delta Lake Transaction Log

The Delta Lake transaction log (also known as **DeltaLog**), is a sequential record of every transaction performed on a Delta Lake file since its creation. It is central to the Delta Lake functionality because it is at the core of many of its important features, including ACID transactions, scalable metadata handling, and time travel.

The main goal of the transaction log is to enable multiple readers and writers to operate on a given version of a dataset simultaneously, and provide additional information, like data skipping indexes, to the execution engine for more performant operations. The Delta Lake transaction log always shows the user a consistent view of the data and serves as a *single source of truth*. It is the central repository that tracks all changes the user makes to a Delta Lake dataset.

When a Delta Lake reader reads a Delta Lake dataset for the first time or runs a new query on an open file that has been modified since the last time it was read, *Spark looks at the transaction log to get the latest version of the table.* This ensures that a user's version of a file is always synchronized with the master record as of the most recent query and that users cannot make divergent, conflicting changes to a file.

How the Transaction Log Implements Atomicity

In chapter 1, we learned atomicity guarantees that all operations (albeit an INSERT, UPDATE, DELETE or MERGE) performed on your file will either succeed as a whole or not succeed at all. Without atomicity, any hardware failure or software bug can cause a data file to be written partially, resulting in corrupted or at a minimum invalid data.

The Delta Lake transaction log is the mechanism through which Delta Lake can offer the atomicity guarantee. The transaction log is also responsible for metadata, time travel and significantly faster metadata operations for large tabular datasets.

The transaction log is an ordered record of every transaction made against a Delta table since it was created. It acts as a single source of truth and tracks all changes made to the table. The transaction log allows users to reason about their data and trust its completeness and quality level.

The simple rule is if an operation is not recorded in the transaction log, it never happened.

In the next sections, we will illustrate these principles with several examples.

Breaking down Transactions into Atomic Commits

Whenever we perform a set of operations to modify a table or storage file (such as INSERTs, UPDATEs, DELETEs or MERGEs), Delta Lake will break down that operation into a series of atomic, discrete steps composed of one or more of the actions shown in table X-X.

Table 2-1. List of possible actions in a transaction log entry

Action	Description
Add file	Adds a file
Remove file	Removes a file
Update Metadata	Updates the table's metadata (e.g., changing the table or file's name, schema, or partitioning). The first transaction log entry for a table or file will always contain an Update Metadata action with the schema, the partition columns and other information.
Set transaction	Records that a structured streaming job has committed a micro-batch with the given stream ID. For more information, see Chapter X: Streaming
Change Protocol	Enables new features by switching the Delta Lake transaction log to the newest software protocol.
Commit Info	Contains information about the commit, which operation was made, from where, and at what time. Every transaction log entry will contain a Commit Info action.

These actions are recorded in the transaction log entries (***.json**) as ordered, atomic units known as commits. This is similar to how the git source control system tracks changes as atomic commits. This also implies that you can re-play each of the commits in the transaction log to get to the current state of the file.

For example, if a user creates a transaction to add a new column to a table and then adds data to it, Delta Lake would break this transaction down into its component action parts, and once the transaction completes, add them to the transaction log as the following commits:

1. **Update** metadata – change the schema to include the new column.
2. Add file – for each new file added

The Transaction Log at the File Level

When we write a Delta file, that file's transaction log is automatically created in the **_delta_log** subdirectory. As we continue to make changes to the Delta file, these changes will be automatically recorded as ordered, atomic commits in the transaction log. Each commit is written as a JSON file, starting with **00000000000000000000.json**. If we make additional changes to the file, Delta Lake will generate additional JSON files in ascending numerical order, so the next commit is written as **00000000000000000001.json**, the following one as **00000000000000000002.json** and so on.

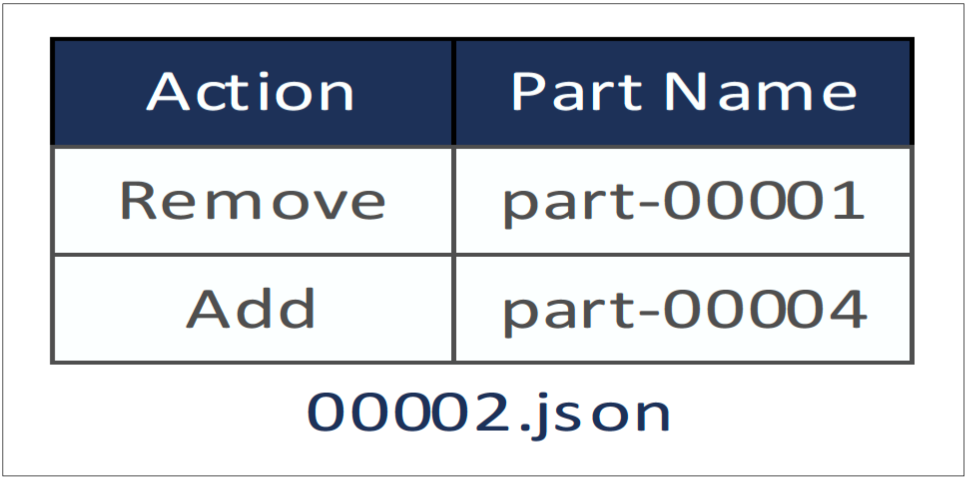
In the remainder of this chapter, we will use an abbreviated form for the transaction log entries for readability purposes. Instead of showing up to 19 digits, we will use an abbreviated form with up to 5 digits (so, we will use **00001.json** instead of the longer notation).

Additionally, we will be shortening the name of the parquet files. These names typically look as follows:

```
part-00007-71c70d7f-c7a8-4a8c-8c29-57300cfd929b-c000.snappy.parquet
```

We will abbreviate a name like this to **part-00007.parquet**, leaving off the GUID and the **snappy.parquet** portion.

In our example visualizations, we will visualize each transaction entry with the action name and the part file name affected, for example, in figure X.X, where we have a *Remove* (file) action, and another *Add* (file) action in a single transaction file.



Action	Part Name
Remove	part-00001
Add	part-00004

00002.json

Figure 2-2. Notation for a transaction log entry

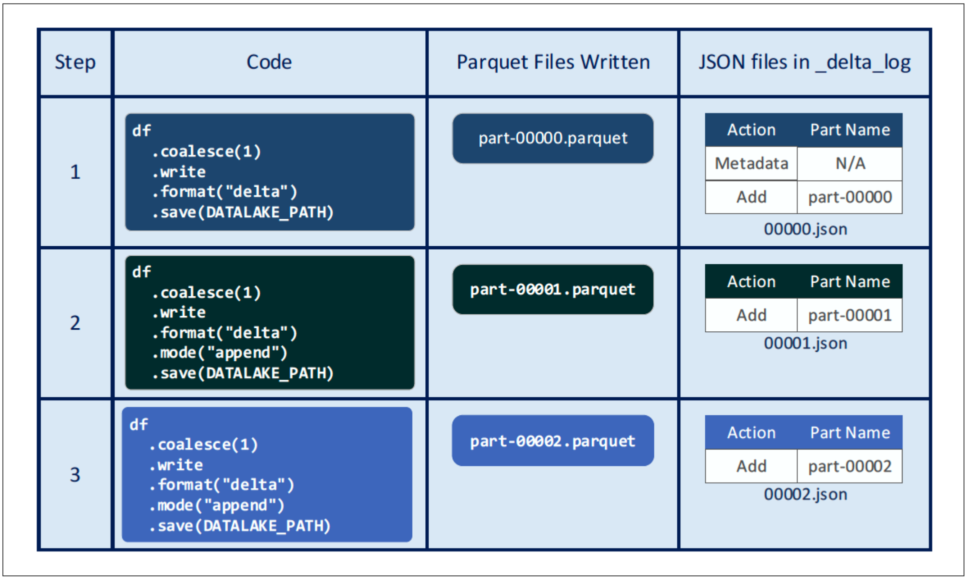
Write Multiple Writes to the Same File

Throughout this section, we will use a set of figures that describe each code step in detail. For each step, we show the following information:

- The actual code snippet is shown in the second column.

- Next to the code snippet we show the parquet part files written as a result of the code snippet execution.
- In the last column we show the transaction log entry's JSON files. We show the action and the affected parquet part file name for each transaction log entry.

For this first example we will use chapter02/MultipleWriteOperations.py from the book's repo to show multiple writes to the same file.



Step	Code	Parquet Files Written	JSON files in _delta_log

1	`df`
	`  .coalesce(1)`
	`  .write`
	`  .format("delta")`
	`  .save(DATALAKE_PATH)`
		part-00000.parquet	Action | Part Name
			Metadata | N/A
			Add | part-00000
			00000.json

2	`df`
	`  .coalesce(1)`
	`  .write`
	`  .format("delta")`
	`  .mode("append")`
	`  .save(DATALAKE_PATH)`
		`part-00001.parquet`	Action | Part Name
			Add | part-00001
			00001.json

3	`df`
	`  .coalesce(1)`
	`  .write`
	`  .format("delta")`
	`  .mode("append")`
	`  .save(DATALAKE_PATH)`
		`part-00002.parquet`	Action | Part Name
			Add | part-00002
			00002.json

Figure 2-3. Multiple writes to the same file

A step-by-step description of the different steps in figure x.x is shown below:

1. First, a new Delta Lake table is written to our path. One Parquet file was written to the output path (**part-00000.parquet**). The first transaction log entry (**00000.json**) has been created in the **_delta_log** directory. Since this is the first transaction log entry for the file, a *metadata* action is recorded, together with an *add file* action, indicating a single partition file was added.
2. Next we add data to the table. You can see a new parquet file (**part-00001.parquet**) has been written, and you created an additional entry (**00001.json)** in the transaction log. Like the first step, the entry contains an *add file* action, because you added a new file.
3. We add more data. Again, a new part file is written (**part-00002.parquet**), and a new transaction log file (**00002.json**) is added to the transaction log with an *add file* action.

Note that each transaction log entry will also have a *commitInfo* action, which contains the audit information for the transaction. We omitted the *commitInfo* log entries on the figures for readability purposes.

The sequence of operations for writes is very important. For each write operation, the part file is always written first, and only when that operation succeeds a transaction log file is added to the **_delta_log** folder. *The transaction is only considered complete when the transaction log entry is written successfully.*

Reading a Delta File

When the system reads a Delta Lake table, it will iterate through the transaction log to "compile" the current state of the table. The sequence of events when reading a file is as follows:

1. The transaction log files are read first.
2. The part files are read based on the log files.

Next, we will describe that sequence for the file written by the previous example (**multipleWriteOperations.py**). Delta Lake will read all our log files (**00000.json**, **00001.json** and **00002.json),** then it will read the three part files based upon the log information, as shown in figure x.x

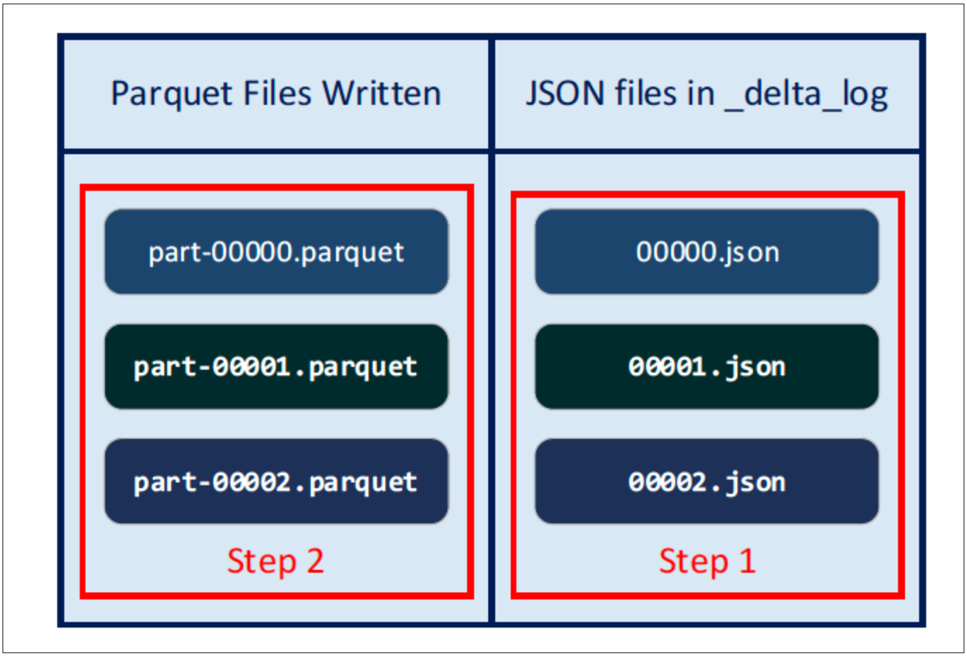
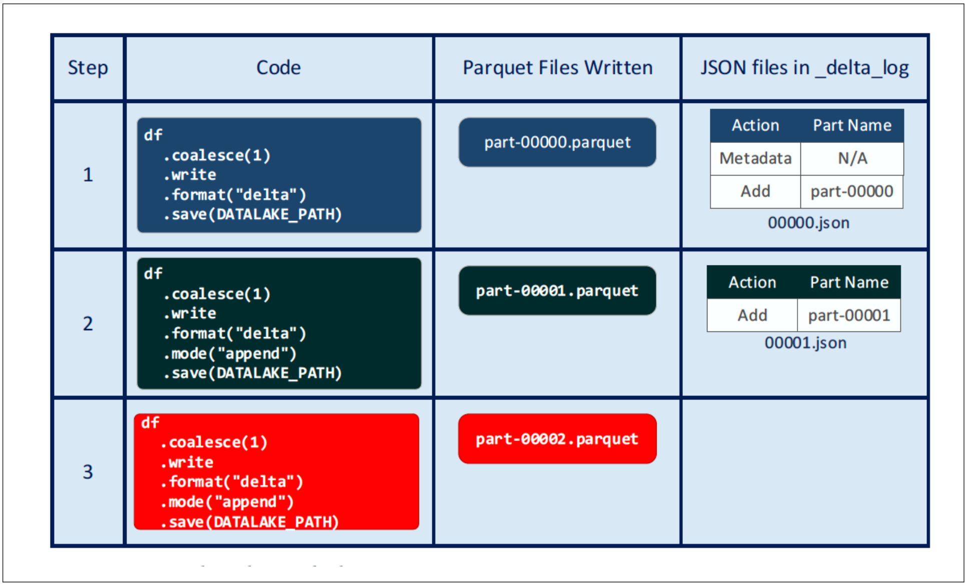


Figure 2-4. Read Operations

Note that the sequence of operations also implies that there could be part files that are no longer referenced in the transaction log. Indeed, this is a common occurrence in **UPDATE** or **DELETE** scenarios. Delta Lake does not delete these part files since they might be required again if the user uses the Time Travel feature of Delta Lake (covered in Chapter X). You can remove old, obsolete part files with the **VACUUM** command.

Failure scenario with a write operation

Next, let's see what happens if a write operation fails. In the previous write scenario, let's assume the write operation in step 3 fails halfway through. Part of the Parquet file might have been written, but the transaction log entry **00002.json** was not. We would have a scenario as shown in figure X.X.



Step	Code	Parquet Files Written	JSON files in _delta_log

1	df
	.coalesce(1)
	.write
	.format("delta")
	.save(DATALAKE_PATH)
		part-00000.parquet	Action | Part Name
			Metadata | N/A
			Add | part-00000
			00000.json

2	df
	.coalesce(1)
	.write
	.format("delta")
	.mode("append")
	.save(DATALAKE_PATH)
		part-00001.parquet	Action | Part Name
			Add | part-00001
			00001.json

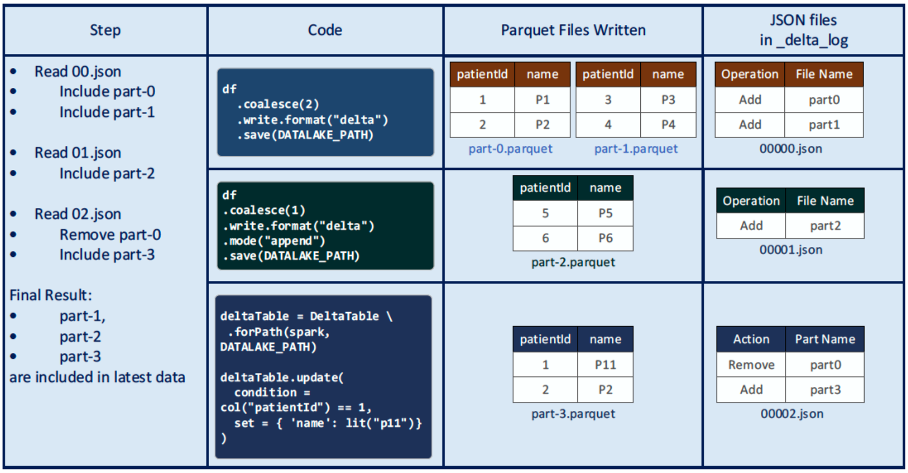
3	df
	.coalesce(1)
	.write
	.format("delta")
	.mode("append")
	.save(DATALAKE_PATH)
		part-00002.parquet	

Figure 2-5. Failure during the last write operation.

As you can see in figure X.X, the last transaction file is missing. According to the read sequence specified earlier, Delta Lake will read the first and second JSON transactions file, and its corresponding **part-00000** and **part-00001** parquet files. The Delta Lake reader will not read inconsistent data; it will read a consistent view through the first two transaction log files.

Update Scenario

The last scenario is contained in the chapter02/UpdateOperation.py code repo. To keep things simple, we have a small Delta Lake table with patient information. We are only tracking the **patientId** and the **name** of each patient. In this use case, we first create a Delta Lake table/dataset with four patients, two in each file. Next, we add data with two more patients. Finally, we update the name of our first patient. As we will see, this update has a bigger impact than we first expect. The full update scenario is shown in figure x.x.



Step

- Read 00.json
- Include part-0
- Include part-1
- Read 01.json
- Include part-2
- Read 02.json
- Remove part-0
- Include part-3

Final Result:
- part-1,
- part-2
- part-3

are included in latest data

Code

```
df
  .coalesce(2)
  .write.format("delta")
  .save(DATALAKE_PATH)
```

Parquet Files Written

patientId	name
1	P1
2	P2

part-0.parquet

patientId	name
3	P3
4	P4

part-1.parquet

JSON files in _delta_log

Operation	File Name
Add	part0
Add	part1

00000.json

```
df
.coalesce(1)
.write.format("delta")
.mode("append")
.save(DATALAKE_PATH)
```

patientId	name
5	P5
6	P6

part-2.parquet

Operation	File Name
Add	part2

00001.json

```
deltaTable = DeltaTable \
 .forPath(spark,
DATALAKE_PATH)

deltaTable.update(
  condition =
col("patientId") == 1,
  set = { 'name': lit("p11")}
)
```

patientId	name
1	P11
2	P2

part-3.parquet

Action	Part Name
Remove	part0
Add	part3

00002.json

Figure 2-6. Updates and the transaction log

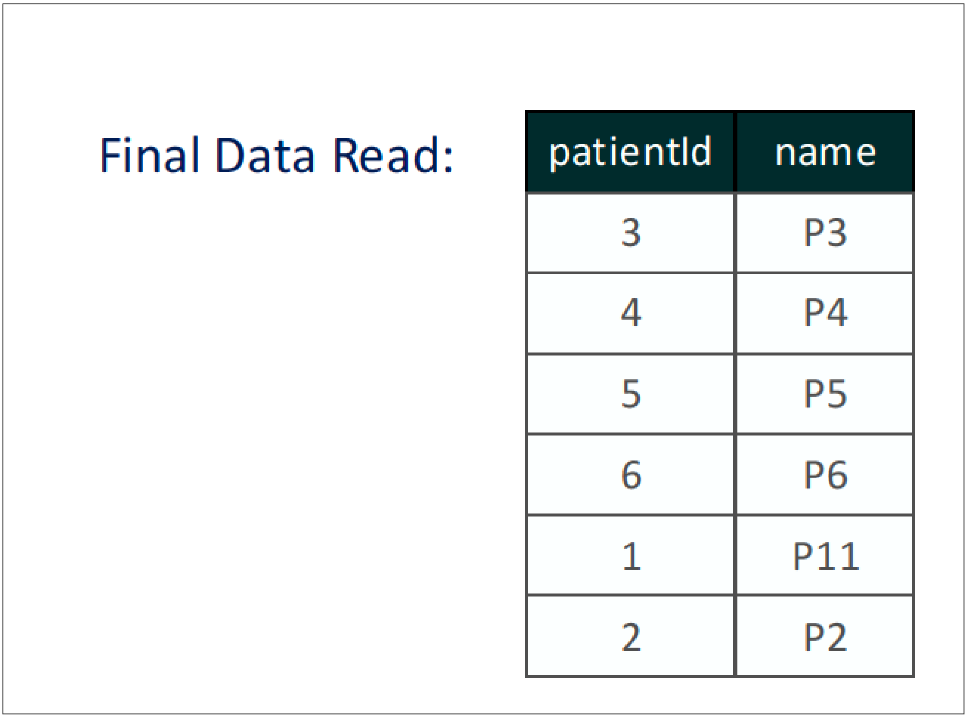
In this example, we execute the following steps:

1. The first code snippet creates a Spark DataFrame, with the **patientId** and **name** of four patients. We write the DataFrame to a Delta Lake table, forcing the data into two files with the **.coalesce(2).** As a result, we write two files. Once the **part-00000.parquet** and **part-00001.parquet** files are written, a transaction log entry is created (**00000.json**). Notice that the transaction log entry contains two *add file* actions indicating the **part-00000.parquet** and the **part-000001.parquet** files were added.
2. The next code snippet adds the data for two more patients (P5 and P6). This results in the creation of the **part-00002.parquet** file. Again, once the file is written, the transaction log entry is written (**00001.json**), and the transaction is complete. Again, the transaction log file has one *add file* action, indicating that a file (**part-2.parquet)** was added.
3. The code performs an update. In this case, we want to update the patient's name with **patientId** 1 from **P1** to **P11**. Currently, the record for **patientId** 1 is present in **part-0**. To perform an update, part-0 is read, and any record matching the patientId of 1 is updated from P1 to P11 and then written out to a new file, which in this case is part-3. Delta Lake writes the transaction log entry (**00002.json**).

Notice that it writes a *Remove File* action, saying that the **part-0** file is removed, and an *Add File* action, saying that the **part-3** file has been added. This is because the data from **part-0** was rewritten into **part-3**, and all modified rows (along with the unmodified rows) have been added to **part-3**, rendering the **part-0** file obsolete.

Notice that the **part-0** file is not deleted by Delta Lake, since a user might want to go back in time with Time Travel (see Chapter X), in which case the file is required. The VACUUM command can clean up unused files like this (see Chapter X: Time Travel).

Now that we have seen how the data is written during an update, let's look at how a read would determine what to read, as is shown in figure X.X.



Final Data Read:

patientId	name
3	P3
4	P4
5	P5
6	P6
1	P11
2	P2

Figure 2-7. Reading after an update

The read would proceed as follows:

1. The first transaction log entry is read (**00000.json**). This entry tells Delta Lake to include the **part-0** and **part-1** files.
2. The next entry (**00001.json**) is read, which tells Delta Lake to include the **part-2** file.

3. The last entry (**00002.json**) is read, which informs the reader to remove the **part-0** file and include **part-3.**

As a result, the reader ends up reading **part-1, part-2** and **part-3**, resulting in the correct data shown in figure x.x.

Scaling Massive Metadata

Checkpoint files

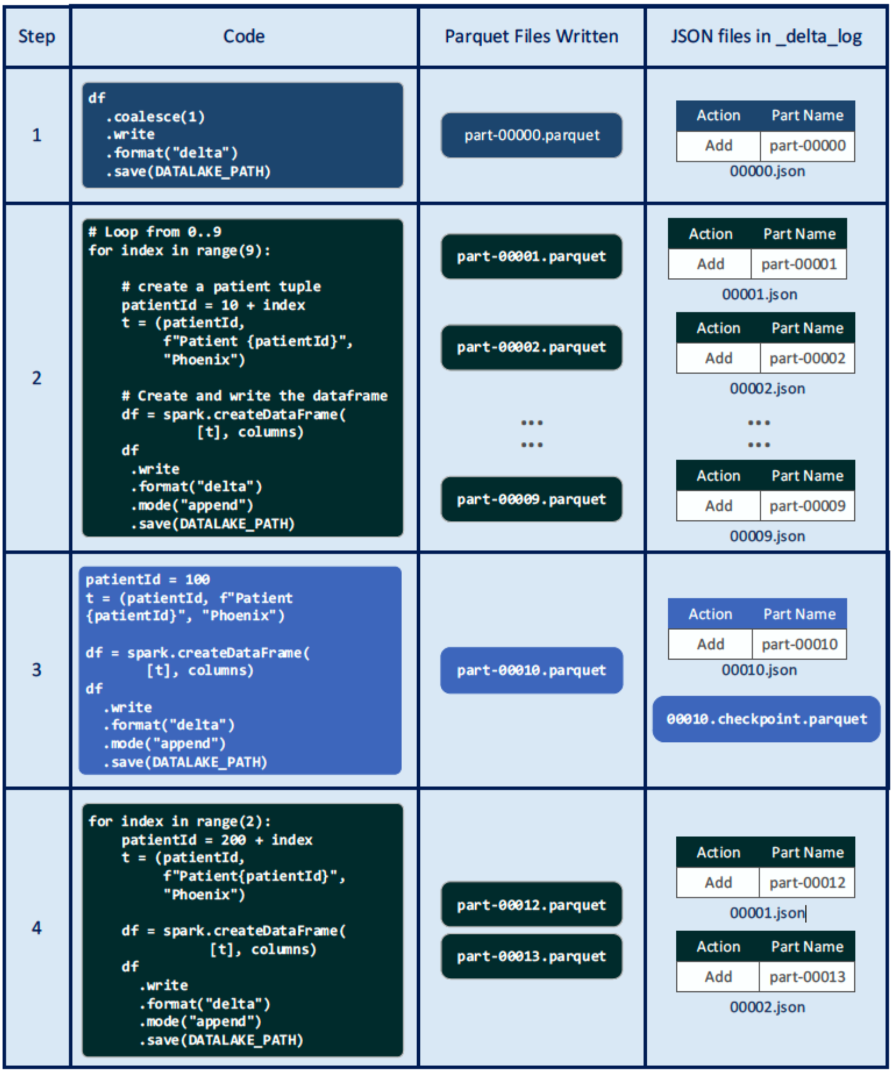
Now that we have seen how the transaction log records each operation, it is conceivable we can have many very large files with thousands of transaction log entries for a single Parquet file. How does Delta Lake scale its metadata handling without needing to read thousands of small files, which would negatively impact Spark's reading performance? Spark tends to be most effective when reading larger files, so how do we resolve this?

Once the Delta Lake writer has made the commits to the transaction log, it will save a *checkpoint file* in Parquet format in the **_delta_log** folder. The Delta Lake writer will continue to generate a new checkpoint every 10 commits.

A checkpoint file saves the entire state of the table at a given point in time. Note that "state" refers to the different actions, not the file's actual content. So, it will contain the *add file*, *remove file*, *update metadata*, *commit info* etc. actions, with all the context information. It will save this list in native Parquet format. This will allow Spark to read the checkpoint quickly. This gives the Spark reader a "shortcut" to fully reproduce a table's state and avoid reprocessing thousands of small JSON files, which could be inefficient.

Checkpoint File Example

Following is an example where we do multiple commits, and a checkpoint file is generated as a result. This example uses the code file chap02/ TransactionLogCheckPointExample.py from the book's repository.



Step	Code	Parquet Files Written	JSON files in _delta_log

1	`df`
	`  .coalesce(1)`
	`  .write`
	`  .format("delta")`
	`  .save(DATALAKE_PATH)`
		part-00000.parquet	Action | Part Name
			Add | part-00000
			00000.json

2	`# Loop from 0..9`
	`for index in range(9):`
	
	`    # create a patient tuple`
	`    patientId = 10 + index`
	`    t = (patientId,`
	`         f"Patient {patientId}",`
	`         "Phoenix")`
	
	`    # Create and write the dataframe`
	`    df = spark.createDataFrame(`
	`             [t], columns)`
	`    df`
	`     .write`
	`     .format("delta")`
	`     .mode("append")`
	`     .save(DATALAKE_PATH)`
		part-00001.parquet
		part-00002.parquet
		...
		...
		part-00009.parquet
			Action | Part Name
			Add | part-00001
			00001.json
			Action | Part Name
			Add | part-00002
			00002.json
			...
			...
			Action | Part Name
			Add | part-00009
			00009.json

3	`patientId = 100`
	`t = (patientId, f"Patient`
	`{patientId}", "Phoenix")`
	
	`df = spark.createDataFrame(`
	`      [t], columns)`
	`df`
	`  .write`
	`  .format("delta")`
	`  .mode("append")`
	`  .save(DATALAKE_PATH)`
		part-00010.parquet	Action | Part Name
			Add | part-00010
			00010.json
			00010.checkpoint.parquet

4	`for index in range(2):`
	`    patientId = 200 + index`
	`    t = (patientId,`
	`         f"Patient{patientId}",`
	`         "Phoenix")`
	
	`    df = spark.createDataFrame(`
	`              [t], columns)`
	`    df`
	`      .write`
	`      .format("delta")`
	`      .mode("append")`
	`      .save(DATALAKE_PATH)`
		part-00012.parquet
		part-00013.parquet
			Action | Part Name
			Add | part-00012
			00001.json
			Action | Part Name
			Add | part-00013
			00002.json

This example has the following steps:

1. The first code snippet creates a standard Spark DataFrame with several patients. Note that we apply a **coalesce(1)** transaction to the Dataframe, which forces that data into one single partition. Next, we write the DataFrame in Delta Lake format to a storage file. We verify that a single **part-00001.parquet** file was written. We also see that a single transaction log entry (**00000.json**) has been created in the

_delta_log directory. This directory entry contains an *add file* action for the **part-0001.parquet** file.

2. In the next step, we set up a loop over a **range(0, 9)**, which will loop nine times, creating a new patient tuple, then creating a DataFrame from that tuple, and writing the DataFrame to our storage file. Since we loop nine times, we create nine additional parquet files, from **part-00001.parquet** through **part-00009.parquet**. We also see nine additional transaction log entries, from **00001.json** through **00009.json**.
3. In step 3, we create one more patient tuple, convert it to a DataFrame and write it to our Delta Lake table. This creates one additional file (**part-00010.parquet**). Our transaction log has another commit (**00010.json**) with the *add file* action for our **part-0010.parquet** file. But the interesting fact is that it also creates a **000010.checkpoint.parquet** file. This is the checkpoint that we mentioned earlier. A checkpoint is generated every 10 commits. This parquet file contains the entire state of our table at the time of the commit in native Parquet format.
4. In the last step, our code adds data two more times, creating **part-00011.parquet** and **part-00012.parquet**, and two new log entries with *add file* entries pointing to these files.

If Delta Lake needs to recreate the state of the table, it will simply read the checkpoint file (**000010.checkpoint.parquet**), and the two additional log entries (**00011.json** and **00012.json**).

Displaying the checkpoint file

Now that we have generated out checkpoint.parquet file, let's have a look at its content using the **/chapter02/readCheckPointFile.py** python file:

```
# Set our output path for our Delta files
DATALAKE_PATH = "/book/chapter02/transactionLogCheckPointExample"
CHECKPOINT_PATH = "/_delta_log/00000000000000000010.checkpoint.parquet"
# Read the checkpoint.parquet file
checkpoint_df =                                \
  spark                                        \
  .read                                      \
  .format("parquet")                         \
  .load(f"{DATALAKE_PATH}{CHECKPOINT_PATH}")
# Display the checkpoint dataframe
checkpoint_df.show()
```

Notice how we do a parquet format read here, because the checkpoint file is indeed stored in Parquet format, not Delta Lake format.

The contents of the **checkpoint_df** dataframe is shown below:

```
+----+--------------------+------+--------------------+--------+
| txn|                 add|remove|            metaData|protocol|
+----+--------------------+------+--------------------+--------+
|null|{part-00000-f7d9f...|  null|                null|    null|
|null|{part-00000-a65e0...|  null|                null|    null|
|null|{part-00000-4c3ea...|  null|                null|    null|
|null|{part-00000-8eb1f...|  null|                null|    null|
|null|{part-00000-2e143...|  null|                null|    null|
|null|{part-00000-d1d13...|  null|                null|    null|
|null|{part-00000-650bf...|  null|                null|    null|
|null|{part-00000-ea06e...|  null|                null|    null|
|null|{part-00000-79258...|  null|                null|    null|
|null|{part-00000-23558...|  null|                null|    null|
|null|                null|  null|                null|  {1, 2}|
|null|                null|  null|{376ce2d6-11b1-46...|    null|
|null|{part-00000-eb29a...|  null|                null|    null|
+----+--------------------+------+--------------------+--------+
```

As we can see, the checkpoint file contains columns for the different actions (add, remove, metadata and protocol). We see our *add file* actions for the different parquet part files, we also see our *update metadata* action from when our Delta file was created, and the *change protocol* action which results from the initial Delta file write.

Note that DataFrame.show() will not show the DataFrame's record in order. The *change protocol* and *update metadata* records are always the first records in the checkpoint file, followed by the different add-file actions.

About the Author

Bennie Haelen is a principal architect with Insight Digital Innovation-a Microsoft and Databricks partner. As Principal architect with Insight, Bennie's primary focus areas are Modern Data Warehousing, Machine learning, AI, and IoT on various commercial cloud platforms. Bennie has overseen many Data + AI projects in different application domains such as health care, the public sector, oil and gas, and financial applications. Bennie has architected and delivered real time streaming Data Lakehouse applications with Databricks, Spark Structured Streaming, Delta Lake, and Microsoft Power BI for various application domains. Bennie brings a wealth of practical experience in implementing secure, enterprise-scale Data Lakehouse-based solutions to support business intelligence, data science and machine learning. Bennie has also been a frequent speaker at Databricks events at Microsoft Technology Centers around the country, and was a speaker at the Data+AI 2021 summit.