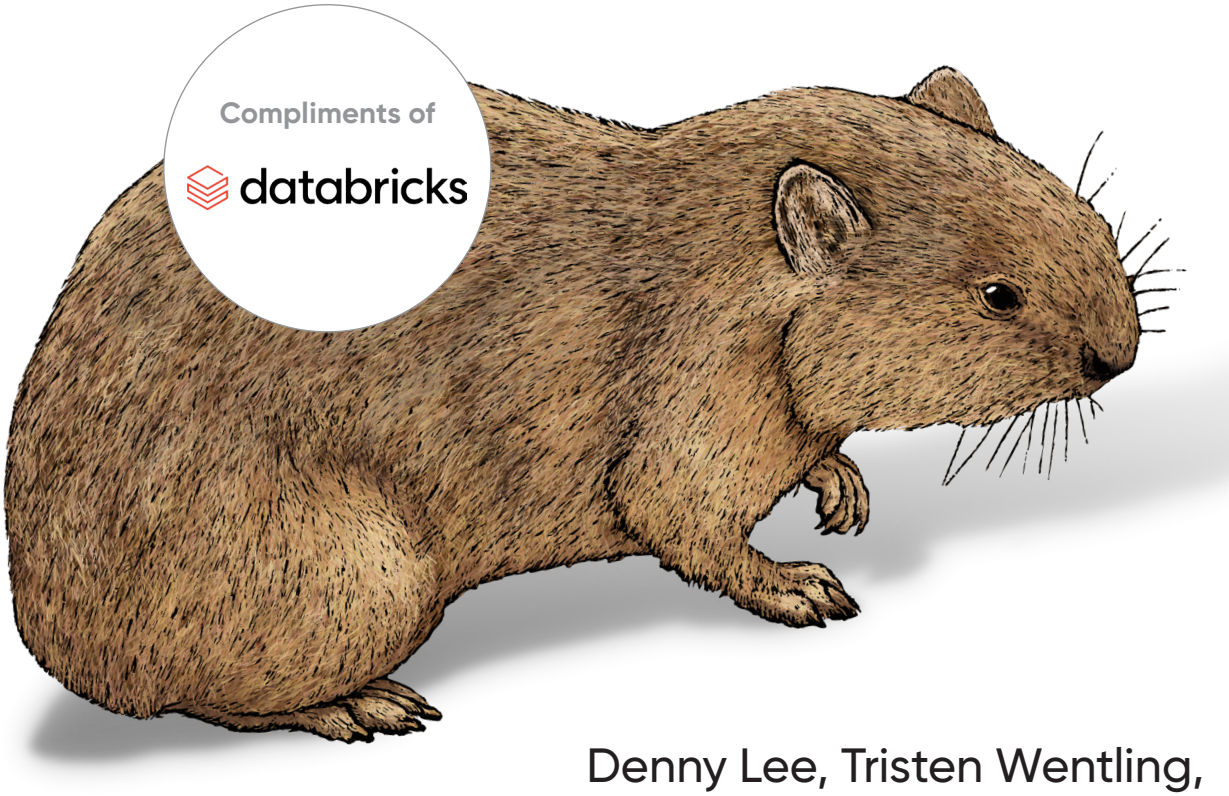
O'REILLY®

Delta Lake

The Definitive Guide

Modern Data Lakehouse Architectures with Data Lakes



Denny Lee, Tristen Wentling,
Scott Haines & Prashanth Babu

Forewords by Michael Armbrust
& Dominique Brezinski

databricks

One unified platform for data and AI

Combine data warehouse performance with data lake flexibility

The complexity of combining both data warehouses and data lakes creates data silos, higher costs and slower decision-making.

The Databricks Data Intelligence Platform is built on lakehouse architecture, which combines the best elements of data lakes and data warehouses to help you reduce costs and deliver on your data and AI initiatives faster.

Built on open source and open standards, a lakehouse simplifies your data estate by eliminating the silos that historically complicate data and AI.

Learn more at databricks.com/lakehouse

Praise for *Delta Lake: The Definitive Guide*

Delta Lake has revolutionized data architectures by combining the best of data lakes and warehouses into the lakehouse architecture. This definitive guide by O'Reilly is an essential resource for anyone looking to harness the full potential of Delta Lake. It offers deep insights into building scalable, reliable, high-performance data architectures. Whether you're a data engineer, scientist, or practitioner, this book will empower you to tackle your toughest data challenges with confidence and precision.

—Matei Zaharia, associate professor of computer science at UC Berkeley and cofounder and chief technologist at Databricks

This book not only provides excellent code examples for Delta Lake but also explains what happens behind the scenes. It's a resource I'll continue to rely on as a practical reference for Delta Lake APIs. Furthermore, it covers the latest exciting innovations within the Delta Lake ecosystem.

—Ryan Zhu, founding developer of Delta Lake, cocreator of Delta Sharing, Apache Spark PMC member, Delta Lake maintainer

The authors of this book fuse deep technical knowledge with pragmatism and clear exposition to allow readers to bring their Spark data lakehouse aspirations to life with the Delta Lake framework.

—Matt Housley, CTO and coauthor of Fundamentals of Data Engineering

Open table formats are the future. If you are invested in Delta Lake, this book will take you from zero to 100, including use cases, integrations, and how to overcome hiccups.

—Adi Polak, author of Scaling Machine Learning with Spark

There are two types of people in data: those who believe they understand what Delta Lake is and those who read this book.

—Andy Petrella, part of the second group, author of Fundamentals of Data Observability*, and founder of Kensu*

Look no further if you want to master all things Delta Lake. Denny, Tristen, Scott, and Prashanth have gone above and beyond to give you more experience than you could ever imagine.

—Jacek Laskowski, freelance Data(bricks) engineer

Delta Lake is much more than Apache Parquet with a commit log. *Delta Lake: The Definitive Guide* takes the mystery out of streaming, data governance, and design patterns.

—Bartosz Konieczny, waitingforcode.com

Delta Lake: The Definitive Guide

Modern Data Lakehouse Architectures with Data Lakes

Denny Lee, Tristen Wentling, Scott Haines, and Prashanth Babu

Forewords by Michael Armbrust and Dominique Brezinski

O'REILLY®

Delta Lake: The Definitive Guide
by Denny Lee, Tristen Wentling, Scott Haines, and Prashanth Babu

Copyright © 2025 O'Reilly Media, Inc. All rights reserved.

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 (*https://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Aaron Black
Development Editor: Gary O'Brien
Production Editor: Gregory Hyman
Copyeditor: Arthur Johnson
Proofreader: Emily Wydeven
Indexer: BIM Creatives, LLC
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

November 2024: First Edition

Revision History for the First Edition
2024-10-29: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098151942* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Delta Lake: The Definitive Guide*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors 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.

978-1-098-15195-9

[LSI]

Table of Contents

Foreword by Michael Armbrust

The Delta protocol was first conceived when I met Dominique Brezinski at Spark Summit 2017. As he described to me the scale of data processing that he was envisioning, I knew that, through our collaborative approach to running Apache Spark, Databricks had already laid down the building blocks of the cloud-scale computing environment necessary to make him successful. Yet I also knew that these fundamentals would inevitably prove to be insufficient without us introducing a novel system to manage the complexities of transactional access to the ever-growing lake of data that Dom had been collecting in his private cloud. Recognizing that Apache Spark itself could serve as the engine of scalable transaction consistency enforcement was the key insight that underpins the ongoing success of Delta Lake. That is, to simplify and scale, we treated the metadata like how we processed and queried the data.

Translating this single insight and the resulting protocol into Delta Lake, a comprehensive toolset for developers to use in any streaming data management solution, has been a long road, with many collaborations along the way. Becoming an open source project allowed Delta Lake to evolve through community input and contributions. The robust ecosystem that has resulted now includes multiple implementations of the Delta protocol, in multiple frameworks, such as Flink, Trino, Presto, and Pulsar, and in multiple languages, including Rust, Go, Java, Scala, Hive, and Python.

To celebrate and further build on this vibrant open source community, I'm now excited to present *Delta Lake: The Definitive Guide*. This guide details Delta Lake's architecture, use cases, and best practices, catering to data engineers, scientists, and analysts alike. It encapsulates years of innovation in data management, offering a comprehensive resource for unlocking Delta Lake's full potential. As you explore this book, you'll gain the knowledge to leverage Delta Lake's capabilities in your projects. I'm eager to see how you'll use it to drive innovation and achieve your data goals.

Welcome to the shore of the Delta Lake. The water is great—let's take a swim!

— Michael Armbrust
Creator of Delta Lake, Spark PMC Member,
Delta Lake TSC and Maintainer

Foreword by Dominique Brezinski

Delta Lake emerged from Michael and my discussions about the challenges I encountered when building a high-scale streaming ETL system using Apache Spark, EC2, and S3. We faced the same challenges at Apple in processing vast amounts of data for intrusion monitoring and threat response. We needed to build a system that could do not only streaming ingestion but also streaming detection and support performant queries over a long retention window of large datasets. From these requirements Delta Lake was created to support ACID transactions and seamless integration of batch and streaming processes, allowing us to handle petabytes of daily data efficiently.

This guide reveals Delta Lake's architectural fundamentals, practical applications, and best practices. Whether you're a data engineer, scientist, or business leader, you'll find valuable insights to leverage Delta Lake effectively.

I'm excited for you to explore this guide and witness how Delta Lake can propel your own innovations. Together, we're shaping the future of data management, enabling the construction of reliable and performant data lakehouses.

— Dominique Brezinski
Distinguished Engineer, Apple
Delta Lake Technical Steering Committee Member

Preface

Welcome to *Delta Lake: The Definitive Guide*! Since it became an open source project in 2019, Delta Lake has revolutionized how organizations manage and process their data. Designed to bring reliability, performance, and scalability to data lakes, Delta Lake addresses many of the inherent challenges traditional data lake architectures face.

Over the past five years, Delta Lake has undergone significant transformation. Originally focused on enhancing Apache Spark, Delta Lake now boasts a rich ecosystem with integrations across various platforms, including Apache Flink, Trino, and many more. This evolution has enabled Delta Lake to become a versatile and integral component of modern data engineering and data science workflows.

Who This Book Is For

As a team of production users and maintainers of the Delta Lake project, we're thrilled to share our collective knowledge and experience with you. Our journey with Delta Lake spans from small-scale implementations to internet-scale production lakehouses, giving us a unique perspective on its capabilities and how to work around any complexities.

The primary goal of this book is to provide a comprehensive resource for both newcomers and experts in data lakehouse architectures. For those just starting with Delta Lake, we aim to elucidate its core principles and help you avoid the common mistakes we encountered in our early days. If you're already well versed in Delta Lake, you'll find valuable insights into the underlying codebase, advanced features, and optimization techniques to enhance your lakehouse environment.

Throughout these pages, we celebrate the vibrant Delta Lake community and its collaborative spirit! We're particularly proud to highlight the development of the Delta Rust API and its widely adopted Python bindings, which exemplify the community's innovative approach to expanding Delta Lake's capabilities. Delta Lake has evolved

significantly since its inception, growing beyond its initial focus on Apache Spark to embrace a wide array of integrations with multiple languages and frameworks. To reflect this diversity, we've included code examples featuring Flink, Kafka, Python, Rust, Spark, Trino, and more. This broad coverage ensures that you'll find relevant examples regardless of your preferred tools and languages.

While we cover the fundamental concepts, we've also included our personal experiences and lessons learned. More importantly, we go beyond theory to offer practical guidance on running a production lakehouse successfully. We've included best practices, optimization techniques, and real-world scenarios to help you navigate the challenges of implementing and maintaining a Delta Lake–based system at scale.

Whether you're a data engineer, architect, or scientist, our goal is to equip you with the knowledge and tools to leverage Delta Lake effectively in your data projects. We hope this guide serves as your companion in building robust, efficient, and scalable lakehouse architectures.

How This Book Is Organized

We organized the book so that you can move from chapter to chapter—introducing concepts, demonstrating key concepts via example code snippets, and providing full code examples or notebooks in the book's GitHub repository. The earlier chapters provide the fundamentals on how to install Delta Lake, its essential operations, understanding its ecosystem, building native Delta Lake applications, and maintaining your Delta Lake; the later chapters expand on these fundamentals and dive deeper into the features before coming back up to review how you can architect this all together for your production workloads:

Chapter 1, "Introduction to the Delta Lake Lakehouse Format"
: We explain Delta Lake's origins, what it is and what it does, its anatomy, and the transaction protocol. We impress upon you that the Delta transaction log is the single source of truth and is subsequently the single source of the relationship between its metadata and data.

Chapter 2, "Installing Delta Lake"
: We discuss the various ways to install Delta Lake, whether through pip or through Docker implementations for Rust, Python, and Apache Spark.

Chapter 3, "Essential Delta Lake Operations"
: In this chapter we look at CRUD operations, merge operations, conversion from Parquet to Delta, and management of Delta Lake metadata.

Chapter 4, "Diving into the Delta Lake Ecosystem"

We delve into the Delta Lake ecosystem, discussing the many frameworks, services, and community projects that support Delta Lake. This chapter includes code samples for the Flink DataStream Connector, Kafka Delta Ingest, and Trino.

Chapter 5, "Maintaining Your Delta Lake"

While Delta Lake provides optimal reading and writing out of the box, developers reading this book will want to further tweak Delta Lake configuration and settings to get even more performance. This chapter looks at using table properties, optimizing your table with Z-Ordering, table tuning and management, and repairing/restoring your table.

Chapter 6, "Building Native Applications with Delta Lake"

The delta-rs project was built from scratch by the community starting in 2020. Together, we built a Delta Rust API using native code, thus allowing developers to take advantage of Delta Lake's reliability without needing to install or maintain the JVM (Java virtual machine). In this chapter, we will dive into this project and its popular Python bindings.

We'd like to give a shout-out to R. Tyler Croy, who not only contributed to and helped with this entire book but also is the author of Chapter 6.

Chapter 7, "Streaming In and Out of Your Delta Lake"

We discuss the importance of streaming and Delta Lake and dive deeper into streaming with Apache Flink, Apache Spark, and delta-rs. We also discuss streaming options, advanced usage with Apache Spark, and Change Data Feed.

Chapter 8, "Advanced Features"

Delta Lake contains advanced features such as generated columns and deletion vectors, which support a novel approach for Merge-on-Read (MoR).

Chapter 9, "Architecting Your Lakehouse"

Taking a 10,000-meter view, how should you architect your lakehouse with Delta Lake? Answering that question involves understanding the lakehouse architecture, transaction support, the medallion architecture, and the streaming medallion architecture.

Chapter 10, "Performance Tuning: Optimizing Your Data Pipelines with Delta Lake"

This is probably our most fun chapter! In it, we further discuss Z-Ordering, liquid clustering, table statistics, and performance considerations.

Chapter 11, "Successful Design Patterns"
: To help you build a successful production environment, we look at slashing compute costs, efficient streaming ingestion, and coordinating complex systems.

Chapter 12, "Foundations of Lakehouse Governance and Security", and Chapter 13, "Metadata Management, Data Flow, and Lineage"
: Next, we have detailed chapters on lakehouse governance! From access control and the data asset model to unifying data warehousing and lake governance, data security, metadata management, and data flow and lineage, these two chapters set the foundation for your governance story.

Chapter 14, "Data Sharing with the Delta Sharing Protocol"
: Delta Sharing is an open protocol for secure, real-time data sharing across organizations and computing platforms. It allows data providers to share live data directly from their Delta Lake tables without the need for data replication or copying to another system. In this chapter, we explore these topics further.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Used to call attention to code snippets of particular interest, within the context of the discussion.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://oreil.ly/dldg_code*.

If you have a technical question or a problem using the code examples, please send email to *support@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Delta Lake: The Definitive Guide* by Denny Lee, Tristen Wentling, Scott Haines, and Prashanth Babu (O'Reilly). Copyright 2025 O'Reilly Media, Inc., 978-1-098-15194-2."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-889-8969 (in the United States or Canada)
707-829-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
https://www.oreilly.com/about/contact.html

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/DeltaLakeDefGuide*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Acknowledgments

This book has truly been a team effort and a labor of love. As authors, we were driven by a strong desire to share our lessons learned and best practices with the community. The journey of bringing this book to life has been immensely rewarding, and we are deeply grateful to everyone who contributed along the way.

First and foremost, we would like to extend our heartfelt thanks to some of the early contributors who played a pivotal role in making Delta Lake a reality. Our sincere gratitude goes out to Ali Ghodsi, Allison Portis, Burak Yavuz, Christian Williams, Dominique Brezinski, Florian Valeye, Gerhard Brueckl, Matei Zaharia, Michael Armbrust, Mykhailo Osypov, QP Hou, Reynold Xin, Robert Pack, Ryan Zhu, Scott Sandre, Tathagata Das, Thomas Vollmer, Venki Korukanti, and Will Jones; your vision and dedication laid the foundation for this project, and without your efforts, this book would not have been possible.

We are also incredibly thankful to the numerous reviewers who provided us with invaluable guidance. Their diligent and constructive feedback, informed by their technical expertise and perspectives, has shaped this book into a valuable resource for learning about Delta Lake. Special thanks to Adi Polak, Aditya Chaturvedi, Andrew Bauman, Andy Petrella, Bartosz Konieczny, Holden Karau, Jacek Laskowski, Jobinesh Purushothaman, Matt Housley, and Matt Powers; your insights have been instrumental in refining our work.

A massive shout-out goes to R. Tyler Croy, who started as a reviewer of this book and eventually joined the author team. His contributions have been invaluable, and his work on Chapter 6, "Building Native Applications with Delta Lake", is a testament to his dedication and expertise. Thank you, Tyler, for your unwavering support and for being an integral part of this journey.

Last but certainly not least, we want to thank the Delta Lake community. As of this book's release, it has been a little more than five years since Delta Lake became open source. Throughout this time, we have experienced many ups and downs, but we have grown together to create an amazing project and community. Your enthusiasm, collaboration, and support have been the driving force behind our success.

Thank you all for being a part of this incredible journey!

Denny

On a personal note, I would like to express my deepest gratitude to my wonderful family and friends. Your unwavering support and encouragement have been my anchor throughout this journey. A special thank-you to my amazing children, Katherine, Samantha, and Isabella, for your patience and love. And to my partner and wonderful wife, Hua-Ping, I could not have done this without you or your constant support and patience.

Tristen

I could not have made it through the immense effort (and many hours) required to pour myself into this book without the many people who helped me become who I am today. I want to thank my wife, Jessyca, for her loving and patient endurance, and my children, Jake, Zek, and Ada, for always being a motivation and a source of inspiration to keep going the distance. I would also like to thank my good friend Steven Yu for helping to guide and encourage me over the years we've known each other; my parents, Kirk and Patricia, for always being encouraging; and the numerous colleagues with whom I have shared many experiences and conversations.

Scott

Getting to the end of a book as an author is a fascinating journey. It requires patience and dedication, but even more, you leave part of yourself behind in the pages you write, and in a very real sense you leave the world behind as you write. Finding the time to write is a balancing act that tries the patience of your friends and family. To my wife, Lacey: thanks for putting up with another book. To my dogs, Willow and Clover: I'm sorry I missed walks and couch time. To my family: thanks for always being there, and for pretending to get excited as I talk about distributed data (your glassy eyes give you away every time). To my friends: I owe all of you more personal time now and promise to drive up to the Bay Area more often. Last, I lost my little sister Meredith while writing this book, and as a means of memorializing her, I've hidden inside jokes and things that would have made her laugh throughout the book and in the examples and data. I love you, Meredith.

Prashanth

I extend my deepest gratitude to my wife, Kavyasudha, for her unwavering support, patience, and love throughout the journey. Your belief in me, even during the most challenging times, has been my anchor. To our curious and joyful child, Advaith, thank you for your infectious laughter and understanding, which have provided endless motivation and joy. Your curiosity and energy remind me daily of the importance of perseverance and passion. To both of you, I extend all my love and appreciation.

CHAPTER 1

Introduction to the Delta Lake Lakehouse Format

This chapter explains Delta Lake's origins and how it was initially designed to address data integrity issues around petabyte-scale systems. If you are familiar with Delta Lake's history and instead want to dive into what Delta Lake is, its anatomy, and the Delta transaction protocol, feel free to jump ahead to the section "What Is Delta Lake?" on page 6 later in this chapter.

The Genesis of Delta Lake

In this section, we'll chart the course of Delta Lake's short evolutionary history: its genesis and inspiration, and its adoption in the community as a lakehouse format, ensuring the integrity of every enterprise's most important asset: its data. The Delta Lake lakehouse format was developed to address the limitations of traditional data lakes and data warehouses. It provides ACID (atomicity, consistency, isolation, and durability) transactions and scalable metadata handling and unifies various data analytics tasks, such as batch and streaming workloads, machine learning, and SQL, on a single platform.

Data Warehousing, Data Lakes, and Data Lakehouses

There have been many technological advancements in data systems (high-performance computing [HPC] and object databases, for example); a simplified overview of the advancements in querying and aggregating large amounts of business data systems over the last few decades would cover data warehousing, data lakes, and lakehouses. Overall, these systems address online analytics processing (OLAP) workloads.

Data warehousing

Data warehouses are purpose-built to aggregate and process large amounts of structured data quickly (Figure 1-1). To protect this data, they typically use relational databases to provide ACID transactions, a step that is crucial for ensuring data integrity for business applications.

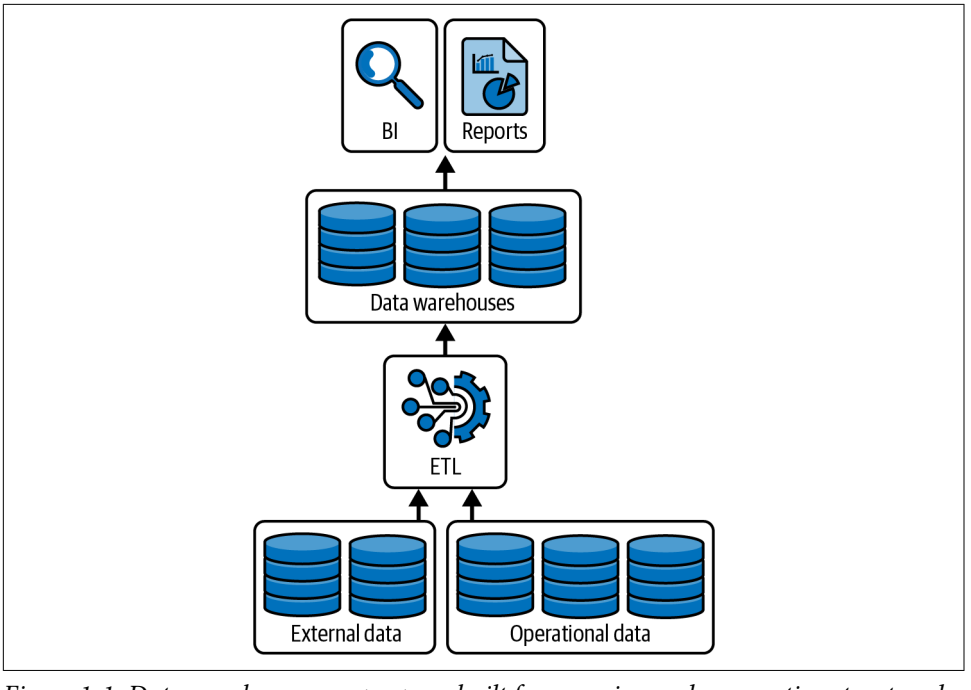


Figure 1-1. Data warehouses are purpose-built for querying and aggregating structured data

Building on the foundation of ACID transactions, data warehouses include management features (backup and recovery controls, gated controls, etc.) to simplify the database operations as well as performance optimizations (indexes, partitioning, etc.) to provide reliable results to the end user more quickly. While robust, data warehouses are often hard to scale to handle the large volumes, variety of analytics (including event processing and data sciences), and data velocity typical in big data scenarios. This limitation is a critical factor that often necessitates using more scalable solutions such as data lakes or distributed processing frameworks like Apache Spark.

Data lakes

Data lakes are scalable storage repositories (HDFS, cloud object stores such as Amazon S3, ADLS Gen2, and GCS, and so on) that hold vast amounts of raw data in their native format until needed (see Figure 1-2). Unlike traditional databases,

data lakes are designed to handle an internet-scale volume, velocity, and variety of data (e.g., structured, semistructured, and unstructured data). These attributes are commonly associated with big data. Data lakes changed how we store and query large amounts of data because they are designed to scale out the workload across multiple machines or nodes. They are file-based systems that work on clusters of commodity hardware. Traditionally, data warehouses were scaled up on a single machine; note that massively parallel processing data warehouses have existed for quite some time but were more expensive and complex to maintain. Also, while data warehouses were designed for structured (or tabular) data, data lakes can hold data in the format of one's choosing, providing developers with flexibility for their data storage.

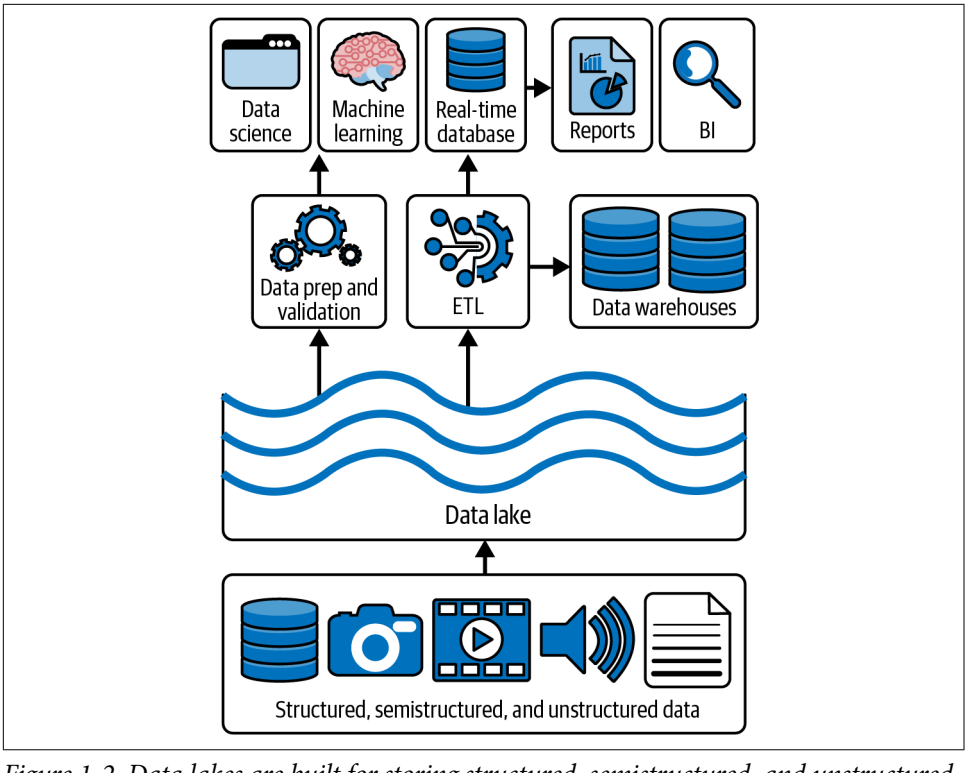


Figure 1-2. Data lakes are built for storing structured, semistructured, and unstructured data on scalable storage infrastructure (e.g., HDFS or cloud object stores)

While data lakes could handle all your data for data science and machine learning, they are an inherently unreliable form of data storage. Instead of providing ACID protections, these systems follow the BASE model—basically available, soft-state, and eventually consistent. The lack of ACID guarantees means the storage system processing failures leave your storage in an inconsistent state with orphaned files.

Subsequent queries to the storage system include files that should not result in duplicate counts (i.e., wrong answers).

Together, these shortcomings can lead to an infrastructure poorly suited for BI queries, inconsistent and slow performance, and quite complex setups. Often, the creation of data lakes leads to unreliable data swamps instead of clean data repositories due to the lack of transaction protections, schema management, and so on.

Lakehouses (or data lakehouses)

The lakehouse combines the best elements of data lakes and data warehouses for OLAP workloads. It merges the scalability and flexibility of data lakes with the management features and performance optimization of data warehouses (see Figure 1-3). There were previous attempts to allow data warehouses and data lakes to coexist side by side. But such an approach was expensive, introducing management complexities, duplication of data, and the reconciliation of reporting/analytics/data science between separate systems. As the practice of data engineering evolved, the concept of the *lakehouse* was born. A lakehouse eliminates the need for disjointed systems and provides a single, coherent platform for all forms of data analysis. Lakehouses enhance the performance of data queries and simplify data management, making it easier for organizations to derive insights from their data.

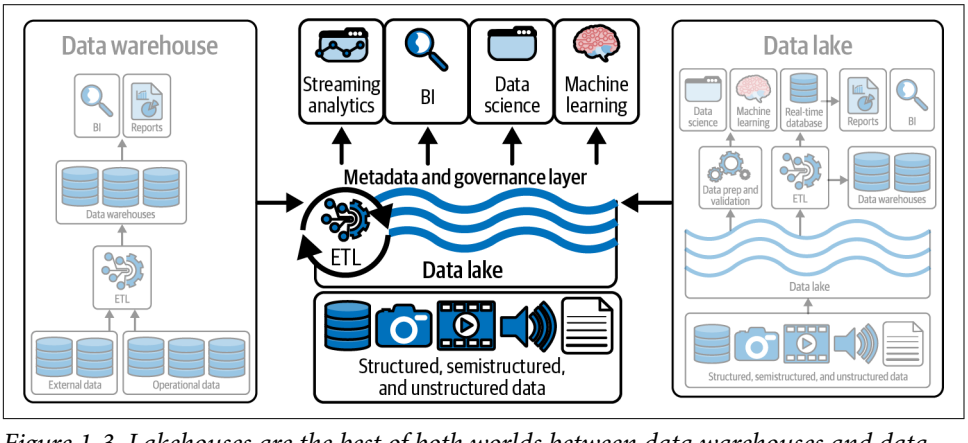


Figure 1-3. Lakehouses are the best of both worlds between data warehouses and data lakes

Delta Lake, Apache Iceberg, and Apache Hudi are the most popular open source lakehouse formats. As you can guess, this book will focus on Delta Lake.[1]

1 To learn more about lakehouses, see the 2021 CIDR whitepaper "Lakehouse: A New Generation of Open Platforms That Unify Data Warehousing and Advanced Analytics".

Project Tahoe to Delta Lake: The Early ~~Years~~ Months

The 2021 online meetup From Tahoe to Delta Lake provided a nostalgic look back at how Delta Lake was created. The panel featured "old school" developers and Delta Lake maintainers Burak Yavuz, Denny Lee, Ryan Zhu, and Tathagata Das, as well as the creator of Delta Lake, Michael Armbrust. It also included the "new school" Delta Lake maintainers who created the delta-rs project, QP Hou and R. Tyler Croy.

The original project name for Delta Lake was "Project Tahoe," as Michael Armbrust had the initial idea of providing transactional reliability for data lakes while skiing at Tahoe in 2017. Lake Tahoe is an iconic and massive lake in California, symbolizing the large-scale data lake the project aimed to create. Michael is a committer/PMC member of Apache Spark™; a Delta Lake maintainer; one of the original creators of Spark SQL, Structured Streaming, and Delta Lake; and a distinguished software engineer at Databricks. The transition from "Tahoe" to "Delta Lake" occurred around New Year's 2018 and came from Jules Damji. The rationale behind changing the name was to invoke the natural process in which rivers flow into deltas, depositing sediments that eventually build up and create fertile ground for crops. This metaphor was fitting for the project, as it represented the convergence of data streams into a managed data lake, where data practitioners could cultivate valuable insights. The Delta name also resonated with the project's architecture, which was designed to handle massive and high-velocity data streams, allowing the data to be processed and split into different streams or views.

But why did Armbrust create Delta Lake? He created it to address the limitations of Apache Spark's file synchronization. Specifically, he wanted to handle large-scale data operations and needed robust transactional support. Thus, his motivation for developing Delta Lake stemmed from the need for a scalable transaction log that could handle massive data volumes and complex operations.

Early in the creation of Delta Lake are two notable use cases that emphasize its efficiency and scalability. Comcast utilized Delta Lake to enhance its data analytics and machine learning platforms and manage its petabytes of data. This transition reduced its compute utilization from 640VMs to 64VMs and simplified job maintenance from 84 to 3 jobs. By streamlining its processing with Delta Lake, Comcast reduced its compute utilization by 10x, with 28x fewer jobs. Apple's information security team employed Delta Lake for real-time threat detection and response, handling over 300 billion events per day and writing hundreds of terabytes of data daily. Both cases illustrate Delta Lake's superior performance and cost-effectiveness compared to traditional data management methods. We will look at additional use cases in Chapter 11.

What Is Delta Lake?

Delta Lake is an open source storage layer that supports ACID transactions, scalable metadata handling, and unification of streaming and batch data processing. It was initially designed to work with Apache Spark and large-scale data lake workloads.

With Delta Lake, you can build a single data platform with your choice of high-performance query engine to address a diverse range of workloads, including (but not limited to) business intelligence (BI), streaming analytics/complex event processing, data science, and machine learning, as noted in Figure 1-4.

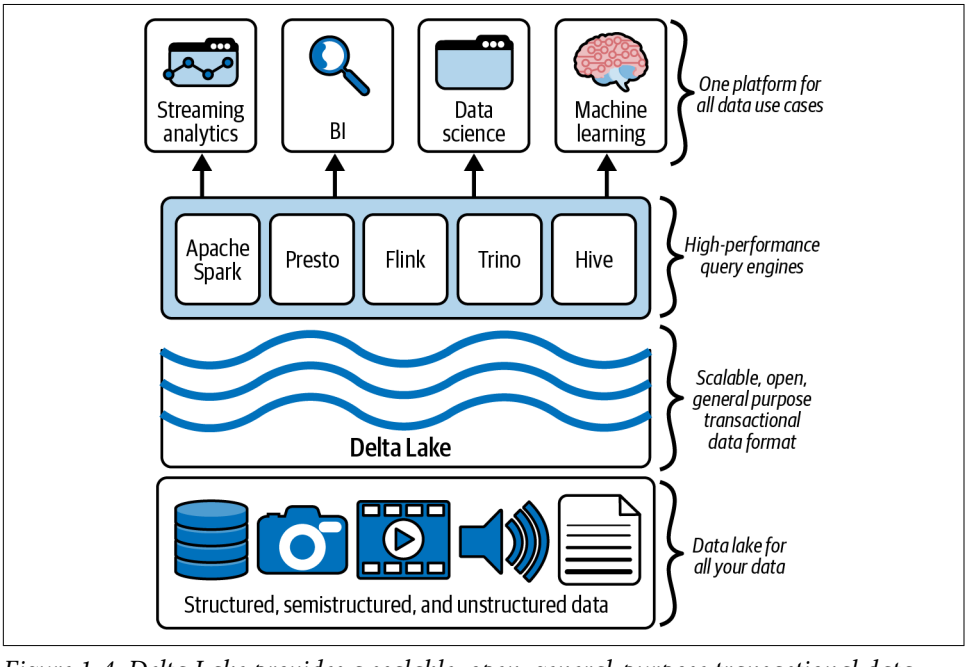


Figure 1-4. Delta Lake provides a scalable, open, general-purpose transactional data format for your lakehouse

However, as it has evolved, Delta Lake has been optimally designed to work with numerous workloads (small data, medium data, big data, etc.). It has also been designed to work with multiple frameworks (e.g., Apache Spark, Apache Flink, Trino, Presto, Apache Hive, and Apache Druid), services (e.g., Athena, Big Query, Databricks, EMR, Fabric, Glue, Starburst, and Snowflake), and languages (.NET, Java, Python, Rust, Scala, SQL, etc.).

Common Use Cases

Developers in all types of organizations, from startups to large enterprises, use Delta Lake to manage their big data and AI workloads. Common use cases include:

Modernizing data lakes
: Delta Lake helps organizations modernize their data lakes by providing ACID transactions, scalable metadata handling, and schema enforcement, thereby ensuring data reliability and performance improvements.

Data warehousing
: There are both data warehousing technologies and *techniques*. The Delta Lake lakehouse format allows you to apply data warehousing techniques to provide fast query performance for various analytics workloads while also providing data reliability.

Machine learning/data science
: Delta Lake provides a reliable data foundation for machine learning and data science teams to access and process data, enabling them to build and deploy models faster.

Streaming data processing
: Delta Lake unifies streaming and batch data processing. This allows developers to process real-time data and perform complex transformations on the fly.

Data engineering
: Delta Lake provides a reliable and performant platform for data engineering teams to build and manage data pipelines, ensuring data quality and accuracy.

Business intelligence
: Delta Lake supports SQL queries, making it easy for business users to access and analyze data and thus enabling them to make data-driven decisions.

Overall, Delta Lake is used by various teams, including data engineers, data scientists, and business users, to manage and analyze big data and AI workloads, ensuring data reliability, performance, and scalability.

Key Features

Delta Lake comprises the following key features that are fundamental to an open lakehouse format (please see the VLDB research article "Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores" for a deeper dive into these features):

ACID transactions
: Delta Lake ensures that data modifications are performed atomically, consistently, in isolation, and durably, i.e., with ACID transaction protections. This means that when multiple concurrent clients or tasks access the data, the system maintains data integrity. For instance, if a process fails during a data modification, Delta Lake will roll back the changes, ensuring that the data remains consistent.

Scalable metadata
: The metadata of a Delta Lake table is the transaction log, which provides transactional consistency per the aforementioned ACID transactions. With a petabyte-scale table, the table's metadata can itself be exceedingly complicated to maintain. Delta Lake's scalable metadata handling feature is designed to manage metadata efficiently for large-scale datasets without its operations impacting query or processing performance.

Time travel
: The Delta Lake time travel feature allows you to query previous versions of a table to access historical data. Made possible by the Delta transaction log, it enables you to specify a version or timestamp to query a specific version of the data. This is very useful for a variety of use cases, such as data audits, regulatory compliance, and data recovery.

Unified batch/streaming
: Delta Lake was designed hand in hand with Apache Spark Structured Streaming to simplify the logic around streaming. Instead of having different APIs for batch and streaming, Structured Streaming uses the same in-memory Datasets/DataFrame API for both scenarios. This allows developers to use the same business logic and APIs, the only difference being latency. Delta Lake provides the ACID guarantees of the storage system to support this unification.

Schema evolution/enforcement
: Delta Lake's schema evolution and schema enforcement ensure data consistency and quality by enforcing a schema on write operations and allowing users to modify the schema without breaking existing queries. They also prevent developers from inadvertently inserting data with incorrect columns or types, which is crucial for maintaining data quality and consistency.

Audit history
: This feature provides detailed logs of all changes made to the data, including information about who made each change, what the change was, and when it was made. This is crucial for compliance and regulatory requirements, as it allows users to track changes to the data over time and ensure that data modifications are performed correctly. The Delta transaction log makes all of this possible.

DML operations
: Delta Lake was one of the first lakehouse formats to provide data manipulation language (DML) operations. This initially extended Apache Spark to support various operations such as insert, update, delete, and merge (or CRUD operations). Today, users can effectively modify the data using multiple frameworks, services, and languages.

Open source
: The roots of Delta Lake were built within the foundation of Databricks, which has extensive experience in open source (the founders of Databricks were the original creators of Apache Spark). Shortly after its inception, Delta Lake was donated to the Linux Foundation to ensure developers have the ability to use, modify, and distribute the software freely while also promoting collaboration and innovation within the data engineering community.

Performance
: While Delta Lake is a lakehouse storage format, it is optimally designed to improve the speed of your queries and processing for both ingestion and querying using the default configuration. While you can continually tweak the performance of Delta Lake, most of the time the defaults will work for your scenarios.

Ease of use
: Delta Lake was built with simplicity in mind right from the beginning. For example, to write a table using Apache Spark in Parquet file format, you would execute:

```
data.write.format("parquet").save("/tmp/parquet-table")
```

To do the same thing for Delta, you would execute:

```
data.write.format("delta").save("/tmp/delta-table")
```

Anatomy of a Delta Lake Table

A Delta Lake table or Delta table comprises several key components that work together to provide a robust, scalable, and efficient data storage solution. The main elements are as follows:

Data files
: Delta Lake tables store data in Parquet file format. These files contain the actual data and are stored in a distributed cloud or on-premises file storage system such as HDFS (Hadoop Distributed File System), Amazon S3, Azure Blob Storage (or Azure Data Lake Storage [ADLS] Gen2), GCS (Google Cloud Storage), or MinIO. Parquet was chosen for its efficiency in storing and querying large datasets.

Transaction log
: The transaction log, also known as the Delta log, is a critical component of Delta Lake. It is an ordered record of every transaction performed on a Delta Lake table. The transaction log ensures ACID properties by recording all changes to the table in a series of JSON files. Each transaction is recorded as a new JSON file in the *_delta_log* directory, which includes metadata about the transaction, such as the operation performed, the files added or removed, and the schema of the table at the time of the transaction.

Metadata
: Metadata in Delta Lake includes information about the table's schema, partitioning, and configuration settings. This metadata is stored in the transaction log and can be retrieved using SQL, Spark, Rust, and Python APIs. The metadata helps manage and optimize the table by providing information for schema enforcement and evolution, partitioning strategies, and data skipping.

Schema
: A Delta Lake table's schema defines the data's structure, including its columns, data types, and so on. The schema is enforced on write, ensuring that all data written to the table adheres to the defined structure. Delta Lake supports schema evolution (add new columns, rename columns, etc.), allowing the schema to be updated as the data changes over time.

Checkpoints
: Checkpoints are periodic snapshots of the transaction log that help speed up the recovery process. Delta Lake consolidates the state of the transaction log by default every 10 transactions. This allows client readers to quickly catch up from the most recent checkpoint rather than replaying the entire transaction log from the beginning. Checkpoints are stored as Parquet files and are created automatically by Delta Lake.

Figure 1-5 is a graphical representation of the structure of a Delta Lake table.

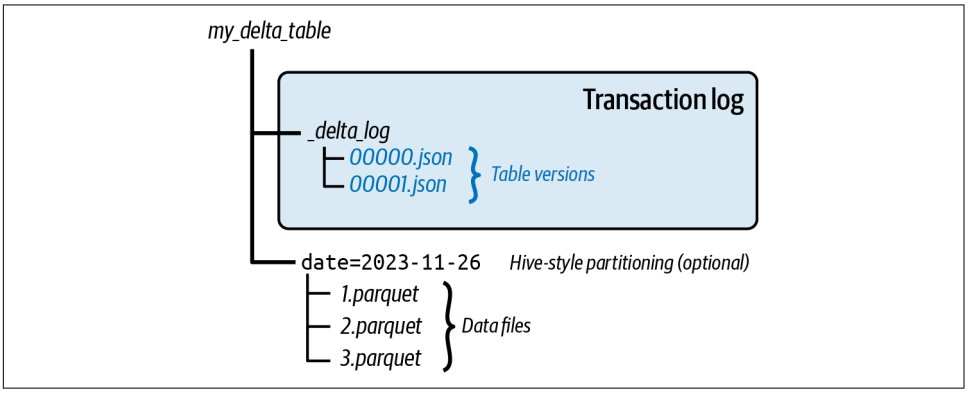


Figure 1-5. Delta Lake table layout for the transaction log and data files (adapted from an image by Denny Lee)[2]

Delta Transaction Protocol

In the previous section, we described the anatomy of a Delta Lake table. The Delta transaction log protocol is the specification defining how clients interact with the table in a consistent manner. At its core, all interactions with the Delta table must begin by reading the Delta transaction log to know what files to read. When a client modifies the data, the client initiates the creation of new data files (i.e., Parquet files) and then inserts new metadata into the transaction log to commit modifications to the table. In fact, many of the original Delta Lake integrations (delta-spark, Trino connector, delta-rust API, etc.) had codebases maintained by different communities. A Rust client could write, a Spark client could modify, and a Trino client could read from the same Delta table without conflict because they all independently followed the same protocol.

Implementing this specification brings ACID properties to large data collections stored as files in a distributed filesystem or object store. As defined in the specification, the protocol was designed with the following goals in mind:

Serializable ACID writes
: Multiple writers can modify a Delta table concurrently while maintaining ACID semantics.

2 Denny Lee, "Understanding the Delta Lake Transaction Log at the File Level", *Denny Lee* (blog), November 26, 2023.

Snapshot isolation for reads
: Readers can read a consistent snapshot of a Delta table, even in the face of concurrent writes.

Scalability to billions of partitions or files
: Queries against a Delta table can be planned on a single machine or in parallel.

Self-describing
: All metadata for a Delta table is stored alongside the data. This design eliminates the need to maintain a separate metastore to read the data and allows static tables to be copied or moved using standard filesystem tools.

Support for incremental processing
: Readers can tail the Delta log to determine what data has been added in a given period of time, allowing for efficient streaming.

Understanding the Delta Lake Transaction Log at the File Level

To better understand this in action, let's look at what happens at the file level when a Delta table is created. Initially, the table's transaction log is automatically created in the *_delta_log* subdirectory. As changes are made to the table, the operations are recorded as ordered *atomic commits* in the transaction log. Each commit is written out as a JSON file, starting with *000...00000.json*. Additional changes to the table generate subsequent JSON files in ascending numerical order, so that the next commits are written out as *000...00001.json*, *000...00002.json*, and so on. Each numeric JSON file increment represents a new version of the table, as described in Figure 1-5.

Note how the structure of the data files has not changed; they exist as separate Parquet files generated by the query engine or language writing to the Delta table. If your table utilizes Hive-style partitioning, you will retain the same structure.

The Single Source of Truth

Delta Lake allows multiple readers and writers of a given table to all work on the table at the same time. It is the central repository that tracks all user changes to the table. This concept is important because, over time, processing jobs will invariably fail in your data lake. The result is partial files that are not removed. Subsequent processing or queries will not be able to ascertain which files should or should not be included in their queries. To show users correct views of the data at all times, the Delta log is the *single source of truth*.

The Relationship Between Metadata and Data

As the Delta transaction log is the single source of truth, any client who wants to read or write to your Delta table *must* first query the transaction log. For example, when inserting data while creating our Delta table, we initially generate two Parquet files: *1.parquet* and *2.parquet*. This event would automatically be added to the transaction log and saved to disk as commit *000...00000.json* (see A in Figure 1-6).

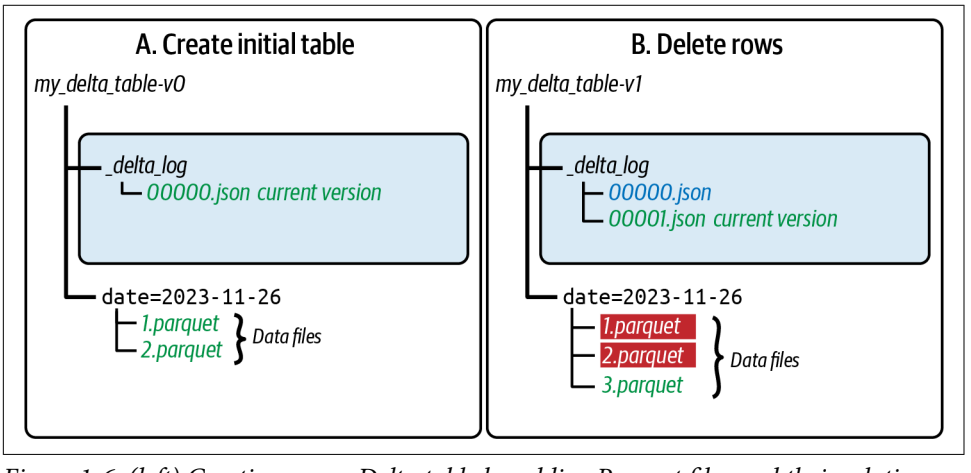


Figure 1-6. (left) Creating a new Delta table by adding Parquet files and their relationship with the Delta transaction log; (right) deleting rows from this Delta table by removing and adding files and their relationship with the Delta transaction log

In a subsequent command (B in Figure 1-6), we run a `DELETE` operation that results in the removal of rows from the table. Instead of modifying the existing Parquet files (*1.parquet*, *2.parquet*), Delta creates a third file (*3.parquet*).

Multiversion Concurrency Control (MVCC) File and Data Observations

For deletes on object stores, it is faster to create a new file or files comprising the unaffected rows rather than modifying the existing Parquet file(s). This approach also provides the advantage of multiversion concurrency control (MVCC). MVCC is a database optimization technique that creates copies of the data, thus allowing data to be safely read and updated concurrently. This technique also allows Delta Lake to provide time travel. Therefore, Delta Lake creates multiple files for these actions, providing atomicity, MVCC, and speed.

We can speed up this process by using deletion vectors, an approach we will describe in Chapter 8.

The removal/creation of the Parquet files shown in B in Figure 1-6 is wrapped in a single transaction recorded in the Delta transaction log in the file *000...00001.json*. Some important observations concerning atomicity are:

- If a user were to read the Parquet files without reading the Delta transaction log, they would read duplicates because of the replicated rows in all the files (*1.parquet*, *2.parquet*, *3.parquet*).
- The remove and add actions are wrapped in the single transaction log *000...00001.json*. When a client queries the Delta table at this time, it records both of these actions and the filepaths for that snapshot. For this transaction, the filepath would point only to *3.parquet*.
- Note that the remove operation is a soft delete or tombstone where the physical removal of the files (*1.parquet*, *2.parquet*) has yet to happen. The physical removal of files will happen when executing the `VACUUM` command.
- The previous transaction *000...00000.json* has the filepath pointing to the original files (*1.parquet*, *2.parquet*). Thus, when querying for an older version of the Delta table via time travel, the transaction log points to the files that make up that older snapshot.

Observing the Interaction Between the Metadata and Data

While we now have a better understanding of what happens at the *individual* data file and metadata file level, how does this all work together? Let's look at this problem by following the flow of Figure 1-7, which represents a common data processing failure scenario. The table is initially represented by two Parquet files (*1.parquet* and *2.parquet*) at t_0.

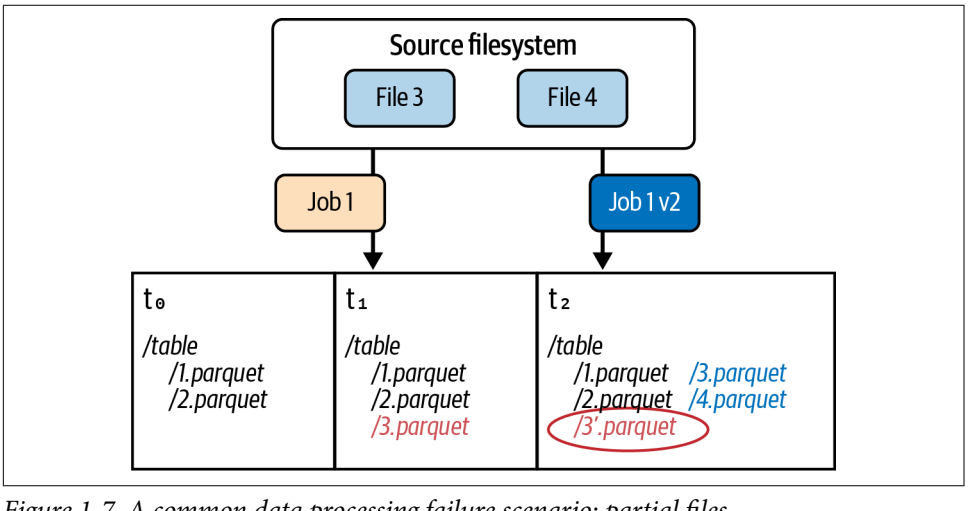


Figure 1-7. A common data processing failure scenario: partial files

At t_1, job 1 extracts file 3 and file 4 and writes them to storage. However, due to some error (network hiccup, storage temporarily offline, etc.), an incomplete portion of file 3 and none of file 4 are written into *3.parquet*. Thus, *3.parquet* is a partial file, and this incomplete data will be returned to any clients that subsequently query the files that make up this table.

To complicate matters, at t_2, a new version of the same processing job (job 1 v2) successfully completes its task. It generates a new version of *3.parquet* and *4.parquet*. But because the partial *3'.parquet* (circled) exists alongside *3.parquet*, any system querying these files will result in double counting.

However, because the Delta transaction log tracks which files are valid, we can avoid the preceding scenario. Thus, when a client reads a Delta Lake table, the engine (or API) initially verifies the transaction log to see what new transactions have been posted to the table. It then updates the client table with any new changes. This ensures that any client's version of a table is always synchronized. Clients cannot make divergent, conflicting changes to a table.

Let's repeat the same partial file example on a Delta Lake table. Figure 1-8 shows the same scenario in which the table is represented by two Parquet files (i.e., *1.parquet* and *2.parquet*) at t_0. The transaction log records that these two files make up the Delta table at t_0 (Version 0).

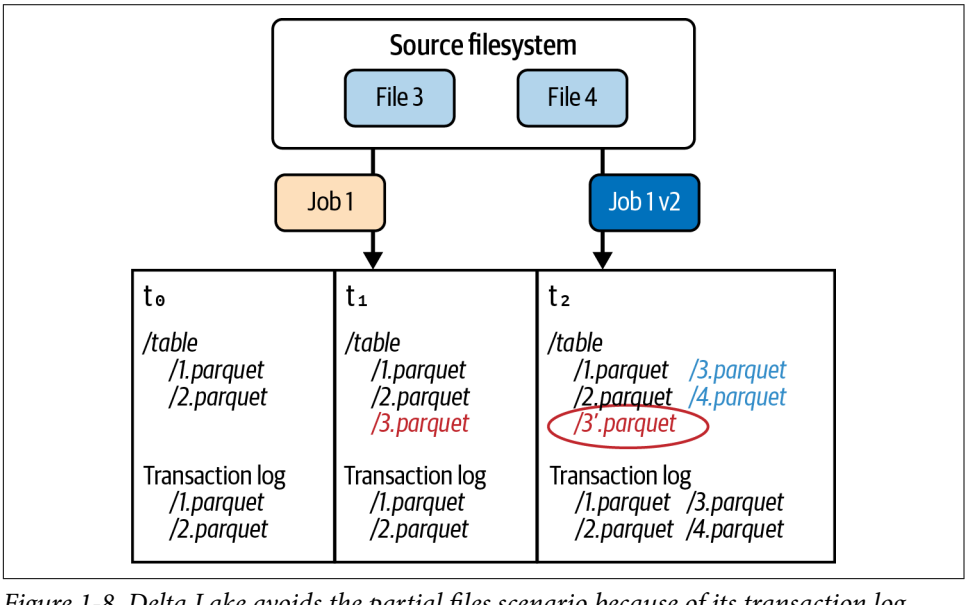


Figure 1-8. Delta Lake avoids the partial files scenario because of its transaction log

At t_1, job 1 fails with the creation of *3.parquet*. However, because the job failed, the transaction was *not* committed to the transaction log. No new files are recorded;

notice how the transaction log has only *1.parquet* and *2.parquet* listed. Any queries against the Delta table at t_1 will read only these two files, even if other files are in storage.

At t_2, job 1 v2 is completed, and its output is the files *3.parquet* and *4.parquet*. Because the job was successful, the Delta log includes entries only for the two successful files. That is, *3'.parquet* is *not* included in the log. Therefore, any clients querying the Delta table at t_2 will see only the correct files.

Table Features

Originally, Delta tables used protocol versions to map to a set of features to ensure user workloads did not break when new features in Delta were released. For example, if a client wanted to use Delta's Change Data Feed (CDF) option, users were required to upgrade their protocol versions and validate their workloads to access new features (Figure 1-9). This ensured that any readers or writers incompatible with a specific protocol version were blocked from reading or writing to that table to prevent data corruption.

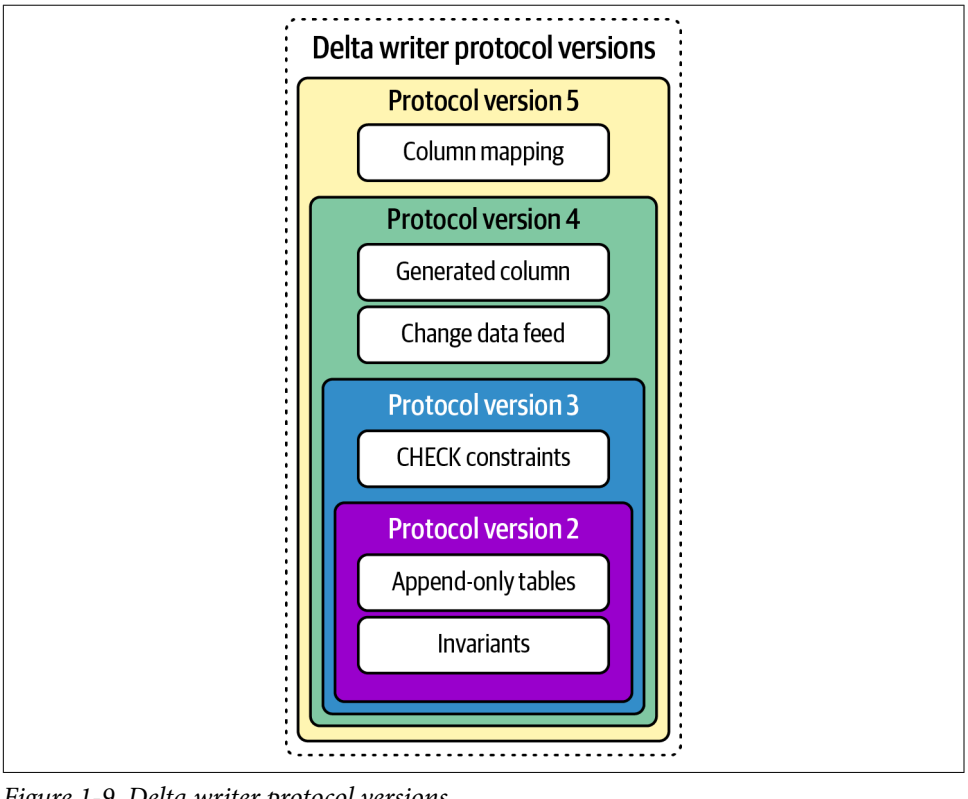


Figure 1-9. Delta writer protocol versions

But this process slows feature adoption because it requires the client and table to support *all* features in that protocol version. For example, with protocol version 4, your Delta table supports both generated columns and CDF. For your client to read this table, it must support both generated columns and Change Data Feed even if you only want to use CDF. In other words, Delta connectors have no choice but to implement all features just to support a single feature in the new version.

Introduced in Delta Lake 2.3.0, *Table Features* replaces table protocol versions to represent features a table uses so connectors can know which features are required to read or write a table (Figure 1-10).

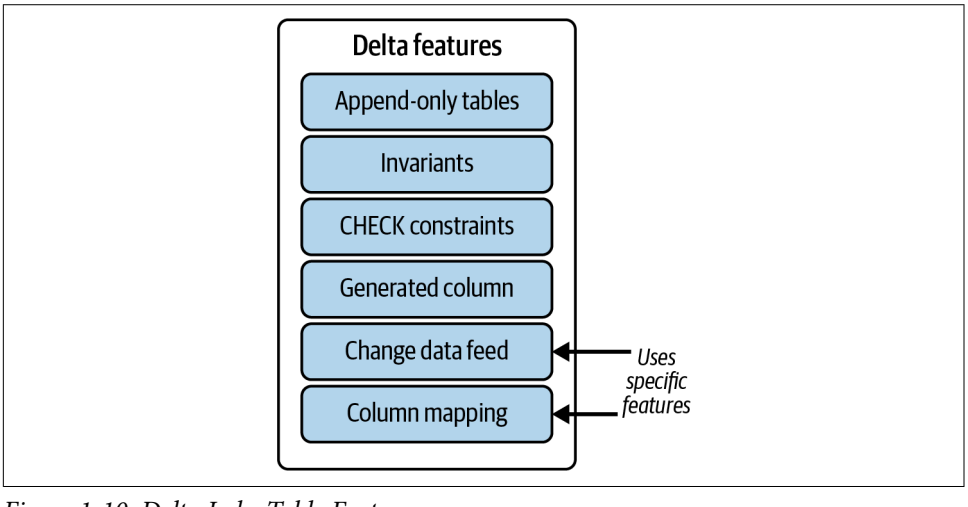


Figure 1-10. Delta Lake Table Features

The advantage of this approach is that any connectors (or integrations) can selectively implement certain features of their interest, instead of having to work on all of them. A quick way to view what table features are enabled is to run the query `SHOW TBLPROPERTIES`:

```
SHOW TBLPROPERTIES default.my_table;
```

The output would look similar to the following:

```
Key (String)                          Value (String)
delta.minReaderVersion                3
delta.minWriterVersion                7
delta.feature.deletionVectors         supported
delta.enableDeletionVectors           true
delta.checkpoint.writeStatsAsStruct   true
delta.checkpoint.writeStatsAsJson     false
```

To dive deeper, please refer to "Table Features" in the GitHub page for the Delta transaction protocol.

Delta Kernel

As previously noted, Delta Lake provides ACID guarantees and performance across many frameworks, services, and languages. As of this writing, every time new features are added to Delta Lake, the connector must be rewritten entirely, because there is a tight coupling between the metadata and data processing. Delta Kernel simplifies the development of connectors by abstracting out all the protocol details so the connectors do not need to understand them. Kernel itself implements the Delta transaction log specification (per the previous section). This allows the connectors to build only against the Kernel library, which provides the following advantages:

Modularity
: Creating Delta Kernel allows for more easily maintained parity between Delta Lake Rust and Scala/JVM, enabling both to be first-class citizens. All metadata (i.e., transaction log) logic is coordinated and executed through the Kernel library. This way, the connectors need only to focus on how to perform their respective frameworks/services/languages. For example, the Apache Flink/Delta Lake connector needs to focus only on reading or modifying the specific files provided by Delta Kernel. The end client does not need to understand the semantics of the transaction log.

Extensibility
: Delta Kernel decouples the *logic* for the metadata (i.e., transaction log) from the data. This allows Delta Lake to be modular, extensible, and highly portable (for example, you can copy the entire table with its transaction log to a new location for your AI workloads). This also extends (pun intended) to Delta Lake's extensibility, as a connector is now, for example, provided the list of files to read instead of needing to query the transaction log directly. Delta Lake already has many integrations, and by decoupling the logic around the metadata from the data, it will be easier for all of us to maintain our various connectors.

Delta Kernel achieves this level of abstraction through the following requirements:

It provides narrow, stable APIs for connectors.
: For a table scan query, a connector needs to specify only the query schema, so that the Kernel can read only the required columns, and the query filters for Kernel to skip data (files, rowgroups, etc.). APIs will be stable and backward compatible. Connectors should be able just to upgrade the Delta Kernel version without rewriting their client code—that is, they automatically get support for an updated Delta protocol via Table Features.

It internally implements the protocol-specific logic.
: Delta Kernel will implement all of the following operations:

  - Read JSON files
  - Read Parquet log files
  - Replay log with data skipping
  - Read Parquet data and DV files
  - Transform data (e.g., filter by DVs)

  While Kernel internally implements the protocol-specific logic, better engine-specific implementations can be added (e.g., Apache Spark or Trino may have better JSON and Parquet reading capabilities).

It provides APIs for plugging in better performance.
: These include *Table APIs* for connectors to perform table operations such as data scans and *Engine APIs* for plugging in connector-optimized implementations for performance-sensitive components.

As of this writing, Delta Kernel is still in the early stages, and building your own Kernel connector is outside the scope of this book. If you would like to dive deeper into how to build your own Kernel connector, please refer to the following resources:

- "[Umbrella Feature Request] Delta Kernel APIs to simplify building connectors for reading Delta tables"
- Delta Kernel—Java
- Delta Kernel—Rust
- "Delta Kernel: Simplifying Building Connectors for Delta"

Delta UniForm

As noted in the section "Lakehouses (or data lakehouses)" on page 4, there are multiple lakehouse formats. Delta Universal Format, or UniForm, is designed to simplify the interoperability among Delta Lake, Apache Iceberg, and Apache Hudi. Fundamentally, lakehouse formats are composed of metadata and data (typically in Parquet file format).

What makes these lakehouse formats different is how they create, manage, and maintain the *metadata* associated with this data. With Delta UniForm, the metadata of other lakehouse formats is generated concurrently with the Delta format. This way, whether you have a Delta, Iceberg, or Hudi client, it can read the data, because all of their APIs can understand the metadata. Delta UniForm includes the following support:

- Apache Iceberg support as part of Delta Lake 3.0.0 (October 2023)
- Apache Hudi support as part of Delta Lake 3.2.0 (May 2024)

For the latest information on how to enable these features, please refer to the Delta UniForm documentation.

Conclusion

In this chapter, we explained the origins of Delta Lake, what it is and what it does, its anatomy, and the transaction protocol. We emphasized that the Delta transaction log is the single source of truth and thus is the single source of the relationship between its metadata and data. While still early, this has led to the development of Delta Kernel as the foundation for simplifying the building of Delta connectors for Delta Lake's many frameworks, services, and community projects. The core difference between the different lakehouse formats is their metadata, so Delta UniForm unifies them by generating all formats' metadata.

CHAPTER 2

Installing Delta Lake

In this chapter, we will show you how to set up Delta Lake and walk you through the simple steps to start writing your first standalone application.

There are multiple ways you can install Delta Lake. If you are just starting, using a single machine with the Delta Lake Docker image is the best option. If you want to skip the hassle of a local installation, the Databricks Community Edition, which includes the latest version of Delta Lake, is free. Various free trials of Databricks, which natively provides Delta Lake, are also available; check your cloud provider's documentation for additional details. Other options discussed in this chapter include the Delta Rust Python bindings, the Delta Rust API, and Apache Spark. In this chapter, we also create and verify the Delta Lake tables for illustrative purposes. Delta Lake table creation and other CRUD operations are covered in depth in Chapter 3.

Delta Lake Docker Image

The Delta Lake Docker image contains all the necessary components to read and write with Delta Lake, including Python, Rust, PySpark, Apache Spark, and Jupyter Notebooks. The basic prerequisite is having Docker installed on your local machine (you can find installation instructions at Get Docker). Once you have Docker installed, you can either download the latest prebuilt version of the Delta Lake Docker image from DockerHub or build the Docker image yourself by following the instructions from the Delta Lake Docker GitHub repository. Once the image has been built or you have downloaded the correct image, you can then move on to running the quickstart in a notebook or shell. The Docker image is the preferred option to run all the code snippets in this book.

Please note this Docker image comes preinstalled with the following:

Apache Arrow
: Apache Arrow is a development platform for in-memory analytics and aims to provide a standardized, language-independent columnar memory format for flat and hierarchical data, as well as libraries and tools for working with this format. It enables fast data processing and movement across different systems and languages, such as C, C++, C#, Go, Java, JavaScript, Julia, MATLAB, Python, R, Ruby, and Rust.

DataFusion
: Created in 2017 and donated to the Apache Arrow project in 2019, DataFusion is a fast, extensible query engine for building high-quality data-centric systems written in Rust that uses the Apache Arrow in-memory format.

ROAPI
: ROAPI is a no-code solution to automatically spin up read-only APIs for Delta Lake and other sources; it builds on top of Apache Arrow and DataFusion.

Rust
: Rust is a statically typed, compiled language that offers performance akin to C and C++, but with a focus on safety and memory management. It's known for its unique ownership model that ensures memory safety without a garbage collector, making it ideal for systems programming in which control over system resources is crucial.

We're using Linux/macOS in this book. If you're running Windows, you can use Git Bash, WSL, or any shell configured for bash commands. Please refer to the implementation-specific instructions for using other software, such as Docker.

We will discuss each of the following interfaces in detail, including how to create and read Delta Lake tables with each one:

- Python
- PySpark Shell
- JupyterLab Notebook
- Scala Shell
- Delta Rust API
- ROAPI

Run Docker Container

To start a Docker container with a bash shell:

1. Open a bash shell.
2. Run the container from the build image with a bash entrypoint using the following command in bash:

```
docker run --name delta_quickstart --rm -it \
--entrypoint bash delta_quickstart
```

Delta Lake for Python

First, open a bash shell and run a container from the built image with a bash entrypoint.

Next, use the `python3` command to launch a Python interactive shell session. The following code snippet will create a Pandas DataFrame, create a Delta Lake table, generate new data, write by appending new data to this table, and finally read and then show the data from this Delta Lake table:

```
# Python
import pandas as pd
from deltalake.writer import write_deltalake
from deltalake import DeltaTable

df = pd.DataFrame(range(5), columns=["id"])      # Create Pandas DataFrame
write_deltalake("/tmp/deltars_table", df)        # Write Delta Lake table
df = pd.DataFrame(range(6, 11), columns=["id"]) # Generate new data
write_deltalake("/tmp/deltars_table", \
        df, mode="append")                       # Append new data
dt = DeltaTable("/tmp/deltars_table")            # Read Delta Lake table
dt.to_pandas()                                   # Show Delta Lake table
```

The output should look similar to the following:

```
# Output
    0
0   0
1   1
... ...
8   9
9  10
```

With these Python commands, you have created your first Delta Lake table. You can validate this by reviewing the underlying filesystem that makes up the table. To do that, you can list the contents within the folder of your Delta Lake table that you saved in */tmp/deltars-table* by running the following `ls` command after you close your Python process:

```
# Bash
$ ls -lsgA /tmp/deltars_table
total 12
4 -rw-r--r-- 1 NBuser 1610 Apr 13 05:48 0-...-f3c05c4277a2-0.parquet
4 -rw-r--r-- 1 NBuser 1612 Apr 13 05:48 1-...-674ccf40faae-0.parquet
4 drwxr-xr-x 2 NBuser 4096 Apr 13 05:48 _delta_log
```

The *.parquet* files contain the data you see in your Delta Lake table, while the *_delta_log* contains the Delta table's transaction log. We will discuss the transaction log in more detail in Chapter 3.

PySpark Shell

First, open a bash shell and run a container from the built image with a bash entrypoint.

Next, launch a PySpark interactive shell session:

```
# Bash
$SPARK_HOME/bin/pyspark --packages io.delta:${DELTA_PACKAGE_VERSION} \
--conf "spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension" \
--conf \
"spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog"
```

Let's run some basic commands in the shell:

```
# Python
# Create a Spark DataFrame
data = spark.range(0, 5)

# Write to a Delta Lake table
(data
   .write
   .format("delta")
   .save("/tmp/delta-table")
)

# Read from the Delta Lake table
df = (spark
        .read
        .format("delta")
        .load("/tmp/delta-table")
        .orderBy("id")
      )
```

```
# Show the Delta Lake table
df.show()
```

To verify that you have created a Delta Lake table, you can list the contents within your Delta Lake table folder. For example, in the preceding code, you saved the table in */tmp/delta-table*. Once you close your `pyspark` process, run a list command in your Docker shell, and you should see something similar to the following:

```
# Bash
$ ls -lsgA /tmp/delta-table
total 36
4 drwxr-xr-x 2 NBuser 4096 Apr 13 06:01 _delta_log
4 -rw-r--r-- 1 NBuser  478 Apr 13 06:01 part-00000-56a2c68a-f90e-4764-8bf7-
a29a21a04230-c000.snappy.parquet
4 -rw-r--r-- 1 NBuser   12 Apr 13 06:01 .part-00000-56a2c68a-f90e-4764-8bf7-
a29a21a04230-c000.snappy.parquet.crc
4 -rw-r--r-- 1 NBuser  478 Apr 13 06:01 part-00001-bcbb45ab-6317-4229-
a6e6-80889ee6b957-c000.snappy.parquet
4 -rw-r--r-- 1 NBuser   12 Apr 13 06:01 .part-00001-bcbb45ab-6317-4229-
a6e6-80889ee6b957-c000.snappy.parquet.crc
4 -rw-r--r-- 1 NBuser  478 Apr 13 06:01 part-00002-9e0efb76-
a0c9-45cf-90d6-0dba912b3c2f-c000.snappy.parquet
4 -rw-r--r-- 1 NBuser   12 Apr 13 06:01 .part-00002-9e0efb76-
a0c9-45cf-90d6-0dba912b3c2f-c000.snappy.parquet.crc
4 -rw-r--r-- 1 NBuser  486 Apr 13 06:01 part-00003-909fee02-574a-47ba-9a3b-
d531eec7f0d7-c000.snappy.parquet
4 -rw-r--r-- 1 NBuser   12 Apr 13 06:01 .part-00003-909fee02-574a-47ba-9a3b-
d531eec7f0d7-c000.snappy.parquet.crc
```

JupyterLab Notebook

Open a bash shell and run a container from the built image with a JupyterLab entrypoint:

```
# Bash
docker run --name delta_quickstart --rm -it \
-p 8888-8889:8888-8889 delta_quickstart
```

The command will output a JupyterLab notebook URL. Copy the URL and launch a browser to follow along in the notebook and run each cell.

Scala Shell

First, open a bash shell and run a container from the built image with a bash entrypoint. Next, launch a Scala interactive shell session:

```
# Bash
$SPARK_HOME/bin/spark-shell --packages io.delta:${DELTA_PACKAGE_VERSION} \
--conf "spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension" \
--conf \
"spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog"
```

Let's run some basic commands in the shell:

```
// Scala
// Create a Spark DataFrame
val data = spark.range(0, 5)

// Write to a Delta Lake table
(data
   .write
   .format("delta")
   .save("/tmp/delta-table")
)

// Read from the Delta Lake table
val df = (spark
            .read
            .format("delta")
            .load("/tmp/delta-table")
            .orderBy("id")
         )

// Show the Delta Lake table
df.show()
```

For instructions on verifying the Delta Lake table, please refer to "PySpark Shell" on page 24.

Delta Rust API

First, open a bash shell and run a container from the built image with a bash entrypoint.

Next, execute *examples/read_delta_table.rs* to review the metadata and files of the `covid19_nyt` Delta Lake table; this command will list useful output, including the number of files written and their absolute paths, among other information:

```
# Bash
cd rs
cargo run --example read_delta_table
```

Finally, execute *examples/read_delta_datafusion.rs* to query the `covid19_nyt` Delta Lake table using DataFusion:

```
# Bash
cargo run --example read_delta_datafusion
```

Running the above command should list the schema and five rows of the data from the `covid19_nyt` Delta Lake table.

ROAPI

The rich open ecosystem around Delta Lake enables many novel utilities, such as ROAPI, which is included in the quickstart container. With ROAPI, you can spin up read-only APIs for static Delta Lake datasets without a single line of code. You can query your Delta Lake table with Apache Arrow and DataFusion using ROAPI, which is also preinstalled in the Docker image.

Open a bash shell and run a container from the built image with a bash entrypoint:

```
# Bash
docker run --name delta_quickstart --rm -it \
-p 8080:8080 --entrypoint bash delta_quickstart
```

The API calls are pushed to the *nohup.out* file. If you haven't created the `deltars_table` in your container, create it via the option described in the section "Delta Lake for Python" on page 23. Alternatively, you may omit `--table 'deltars_table=/tmp/deltars_table/,format=delta'` from the command, as well as any steps that call the `deltars_table`.

Start the ROAPI utility using the following `nohup` command:

```
# Bash
nohup roapi --addr-http 0.0.0.0:8080 \
--table 'deltars_table=/tmp/deltars_table/,format=delta' \
--table 'covid19_nyt=/opt/spark/work-dir/rs/data/COVID-19_NYT,format=delta' &
```

Next, open another shell and connect to the same Docker image:

```
# Bash
docker exec -it delta_quickstart /bin/bash
```

Run the next three steps in the bash shell you launched in the previous step.

Check the schema of the two Delta Lake tables:

```
# Bash
curl localhost:8080/api/schema
```

The output of the preceding command should be along the following lines:

```
# Output
{
   "covid19_nyt":{"fields":[{"name":"date","data_type":"Utf8","nulla-
ble":true,"dict_id":0,"dict_is_ordered":false},
{"name":"county","data_type":"Utf8","nulla-
ble":true,"dict_id":0,"dict_is_ordered":false},
{"name":"state","data_type":"Utf8","nulla-
ble":true,"dict_id":0,"dict_is_ordered":false},
{"name":"fips","data_type":"Int32","nulla-
ble":true,"dict_id":0,"dict_is_ordered":false},
{"name":"cases","data_type":"Int32","nulla-
ble":true,"dict_id":0,"dict_is_ordered":false},
```

```
{"name":"deaths","data_type":"Int32","nulla-
ble":true,"dict_id":0,"dict_is_ordered":false}]},
   "deltars_table":{"fields":[{"name":"0","data_type":"Int64","nulla-
ble":true,"dict_id":0,"dict_is_ordered":false}]}
}
```

Query the `deltars_table`:

```
# Bash
curl -X POST -d "SELECT * FROM deltars_table"  localhost:8080/api/sql
```

The output of the preceding command should be along the following lines:

```
# Output
[{"0":0},{"0":1},{"0":2},{"0":3},{"0":4},{"0":6},{"0":7},{"0":8},{"0":9},
{"0":10}]
```

Query the `covid19_nyt` Delta Lake table:

```
# Bash
curl -X POST \
-d "SELECT cases, county, date FROM covid19_nyt ORDER BY cases DESC LIMIT 5" \
localhost:8080/api/sql
```

The output of the preceding command should be along the following lines:

```
# Output
[
{"cases":1208672,"county":"Los Angeles","date":"2021-03-11"},
{"cases":1207361,"county":"Los Angeles","date":"2021-03-10"},
{"cases":1205924,"county":"Los Angeles","date":"2021-03-09"},
{"cases":1204665,"county":"Los Angeles","date":"2021-03-08"},
{"cases":1203799,"county":"Los Angeles","date":"2021-03-07"}
]
```

Native Delta Lake Libraries

While Delta Lake's core functionality is deeply integrated with Apache Spark, the underlying data format and transaction log are designed to be language agnostic. This flexibility has spurred the development of native Delta Lake libraries in various programming languages, offering direct interaction with Delta Lake tables without the overhead of Spark.

These libraries provide lower-level access to Delta Lake's features, enabling developers to build highly optimized and specialized applications in a language-agnostic way. Developers can choose the language that best suits their needs and expertise. We discuss this in more detail in Chapter 6.

Multiple Bindings Available

The Rust library provides a strong foundation for other non-JVM-based libraries to build pipelines with Delta Lake. The most popular and prominent of those bindings are the Python bindings, which expose a `DeltaTable` class and optionally integrate seamlessly with Pandas or PyArrow. At the time of this writing, the Delta Lake Python package is compatible with Python versions 3.7 and later and offers many prebuilt wheels for easy installation on most major operating systems and architectures.

Multiple communities have developed bindings on top of the Rust library, exposing Delta Lake to Ruby, Node, or other C-based connectors. None have yet reached the maturity presently seen in the Python package, partly because none of the other language ecosystems have seen a level of investment in data tooling like that in the Python community. Pandas, Polars, PyArrow, Dask, and more provide a very rich set of tools for developers to read from and write to Delta tables.

More recently, there has been experimental work on a so-called Delta Kernel initiative, which aims to provide a native Delta Lake library interface for connectors that abstracts away the Delta protocol into one place. This work is still in an early phase but is expected to help consolidate support for native (C/C++, for example) and higher-level engines (e.g., Python or Node) so that everybody can benefit from the more advanced features, such as deletion vectors, by simply upgrading their underlying Delta Kernel versions.

Installing the Delta Lake Python Package

Delta Lake provides native Python bindings based on the delta-rs project with Pandas integration. This Python package can be easily installed with the following command:

```
# Bash
pip install delta lake
```

After installation, you can follow the same steps as outlined in "Delta Lake for Python" on page 23.

Apache Spark with Delta Lake

Apache Spark is an open source engine designed for the processing and analysis of large-scale datasets. It's architected to be both rapid and versatile and is capable of managing a variety of analytics, both batch and real-time. Spark provides an interface for programming comprehensive clusters, offering implicit data parallelism and fault tolerance. It leverages in-memory computations to enhance speed and data processing over MapReduce operations.

Spark offers multilingual support, which allows developers to construct applications in several languages, including Java, Scala, Python, R, and SQL. Spark also incorporates numerous libraries that enable a wide array of data analysis tasks encompassing machine learning, stream processing, and graph analytics.

Spark is written predominantly in Scala, but its APIs are available in Scala, Python, Java, and R. Spark SQL also allows users to write and execute SQL or HiveQL queries. For new users, we recommend exploring the Python API or SQL queries to get started with Apache Spark.

Check out *Learning Spark* (O'Reilly) or *Spark: The Definitive Guide* (O'Reilly) for a more detailed introduction to Spark.

Setting Up Delta Lake with Apache Spark

The steps in this section can be executed on your local machine in either of the following ways:

Interactive execution
: Start the Spark shell (with `spark-shell` for Scala language, or with `pyspark` for Python) with Delta Lake and run the code snippets interactively in the shell. In this chapter, we will focus on interactive execution.

Run them as a project
: If, instead of code snippets, you have code in multiple files, you can set up a Maven or sbt project (Scala or Java) with Delta Lake, with all the source files, and run the project. You could also use the examples provided in the GitHub repository.

For all the following instructions, make sure to install the version of Spark or PySpark that is compatible with Delta Lake 2.3.0. See the release compatibility matrix for details.

Prerequisite: Set Up Java

As noted in the official Apache Spark installation instructions, you must ensure that a valid Java version (8, 11, or 17) has been installed and configured correctly on your system using either the system `PATH` or the `JAVA_HOME` environmental variable.

Readers should make sure to use the Apache Spark version that is compatible with Delta Lake 2.3.0 and above.

Setting Up an Interactive Shell

To use Delta Lake interactively within the Spark SQL, Scala, or Python shells, you need a local installation of Apache Spark. Depending on whether you want to use SQL, Python, or Scala, you can set up either the SQL, PySpark, or Spark shell, respectively.

Spark SQL shell

The Spark SQL shell, also referred to as the Spark SQL CLI, is an interactive command-line tool designed to facilitate the execution of SQL queries directly from the command line.

Download the compatible version of Apache Spark by following instructions in the Spark documentation, either by using pip or by downloading and extracting the archive and running `spark-sql` in the extracted directory:

```
# Bash
bin/spark-sql --packages io.delta:delta-core_2.12:2.3.0 --conf \
"spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension" --conf \
"spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog"
```

To create your first Delta Lake table, run the following in the Spark SQL shell prompt:

```
-- SQL
CREATE TABLE delta.`/tmp/delta-table` USING DELTA AS
SELECT col1 AS id FROM VALUES 0, 1, 2, 3, 4;
```

You can read back the data written to the table with another simple SQL query:

```
-- SQL
SELECT * FROM delta.`/tmp/delta-table`;
```

PySpark shell

The PySpark shell, also known as the PySpark CLI, is an interactive environment that facilitates engagement with Spark's API using the Python programming language. It serves as a platform for learning, testing PySpark examples, and conducting data analysis directly from the command line. The PySpark shell operates as a Read-Eval-Print Loop (REPL), providing a convenient environment for swiftly testing PySpark statements.

Install the PySpark version that is compatible with the Delta Lake version by running the following in the command prompt:

```
# Bash
pip install pyspark==<compatible-spark-version>
```

Next, run PySpark with the Delta Lake package and additional configurations:

```
# Bash
pyspark --packages io.delta:delta-core_2.12:2.3.0 --conf \
"spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension" --conf \
"spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog"
```

Finally, to create your first Delta Lake table, run the following in the PySpark shell prompt:

```
# Python
data = spark.range(0, 5)
data.write.format("delta").save("/tmp/delta-table")
```

You can read back the data written to the table with a simple PySpark code snippet:

```
# Python
df = spark.read.format("delta").load("/tmp/delta-table")
df.show()
```

Spark Scala shell

The Spark Scala shell, also referred to as the Spark Scala CLI, is an interactive platform that allows users to interact with Spark's API using the Scala programming language. It is a potent tool for data analysis and serves as an accessible medium for learning the API.

Download the compatible version of Apache Spark by following instructions in the Spark documentation, either by using pip or by downloading and extracting the archive and running spark-shell in the extracted directory:

```
# Bash
bin/spark-shell --packages io.delta:delta-core_2.12:2.3.0 --conf \
"spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension" --conf \
"spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog"
```

To create your first Delta Lake table, run the following in the Scala shell prompt:

```
// Scala
val data = spark.range(0, 5)
data.write.format("delta").save("/tmp/delta-table")
```

You can read back the data written to the table with a simple PySpark code snippet:

```
// Scala
val df = spark.read.format("delta").load("/tmp/delta-table")
df.show()
```

PySpark Declarative API

A PyPi package containing the Python APIs for using Delta Lake with Apache Spark is also available. This could be very useful for setting up a Python project and, more importantly, for unit testing. Delta Lake can be installed using the following command:

```
# Bash
pip install delta-spark
```

And `SparkSession` can be configured with the `configure_spark_with_delta_pip` utility function in Delta Lake:

```
# Python
from delta import *
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"
  )
)
```

Databricks Community Edition

With the Databricks Community Edition, Databricks has provided a platform for personal use that gives you a cluster of 15 GB memory, which might be just enough to learn Delta Lake with the help of notebooks and the bundled Spark version.

To sign up for Databricks Community Edition, go to the Databricks sign-up page, fill in your details on the form, and click Continue. Choose Community Edition by clicking on the "Get started with Community Edition" link on the second page of the registration form.

After you have successfully created your account, you will be sent an email to verify your email address. After completing the verification, you can log in to Databricks Community Edition to view the Databricks workspace.

Create a Cluster with Databricks Runtime

Start by clicking on the Compute menu item in the left pane. All the clusters you create will be listed on this page. If this is the first time you are logging in to this account, the page won't list any clusters since you haven't yet created any.

Clicking on Create Compute will bring you to a new cluster page. Databricks Runtime 13.3 LTS is, at the time of this writing, selected by default. You can choose any of the latest (preferably LTS) Databricks Runtimes for running the code. For this example, we chose Databricks Runtime 13.3 LTS. For more information on Databricks Runtime releases and the compatibility matrix, please check the Databricks website.

Next, choose any name you'd like for your cluster; we chose "Delta_Lake_DLDG" (see Figure 2-1). Then hit the Create Cluster button at top to launch the cluster.

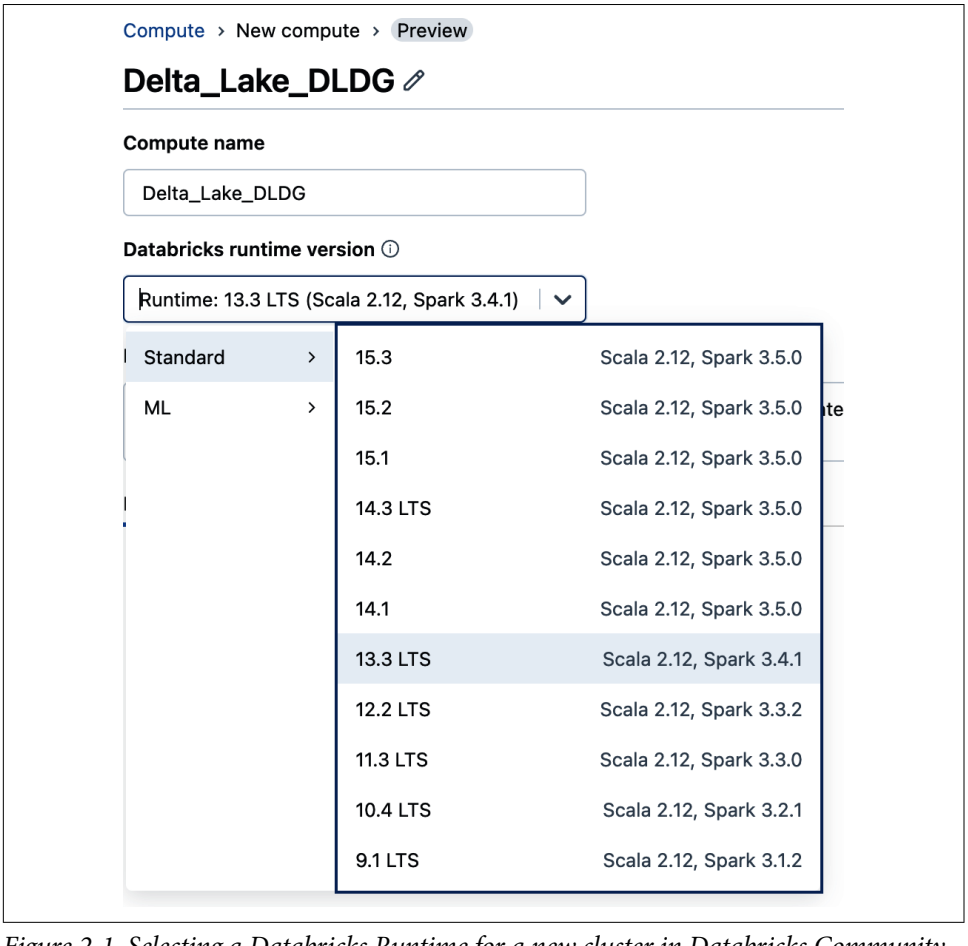


Figure 2-1. Selecting a Databricks Runtime for a new cluster in Databricks Community Edition

You can create only one cluster at a time with Databricks Community Edition. If a cluster already exists, you will need to either use it or delete it before you can create a new cluster.

Your cluster should be up and running within a few minutes, as shown in Figure 2-2.

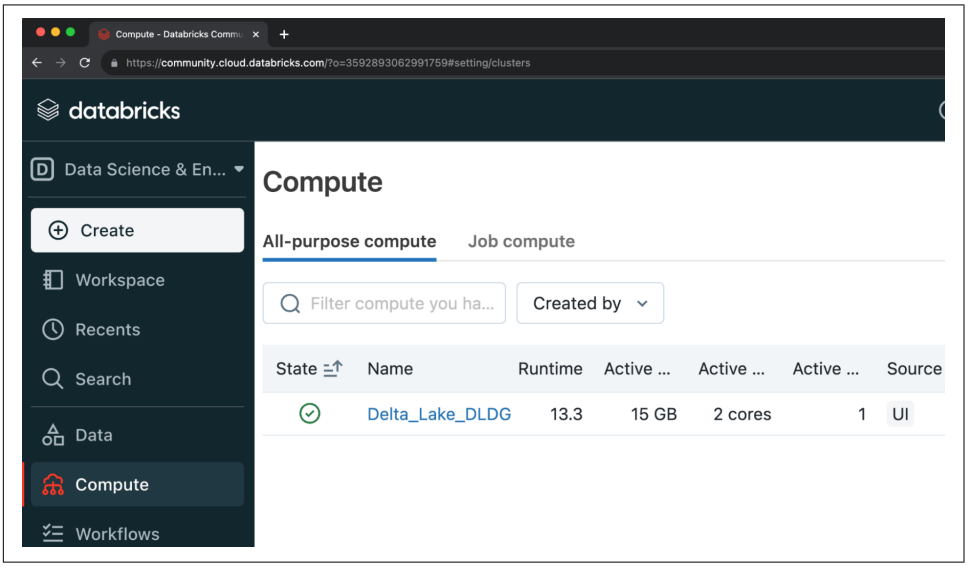


Figure 2-2. A new cluster up and running

Delta Lake is bundled in the Databricks Runtime, so you don't need to install Delta Lake explicitly either through pip or by using the Maven coordinates of the package to the cluster.

Importing Notebooks

For brevity and ease of understanding, we will use the Jupyter Notebook we saw in the section "JupyterLab Notebook" on page 25. This notebook is available in the delta-docs GitHub repository. Please copy the notebook link and keep it handy, as you will import the notebook in this step.

Go to Databricks Community Edition and click on Workspace, and then click on the three stacked dots at top right, as shown in Figure 2-3.

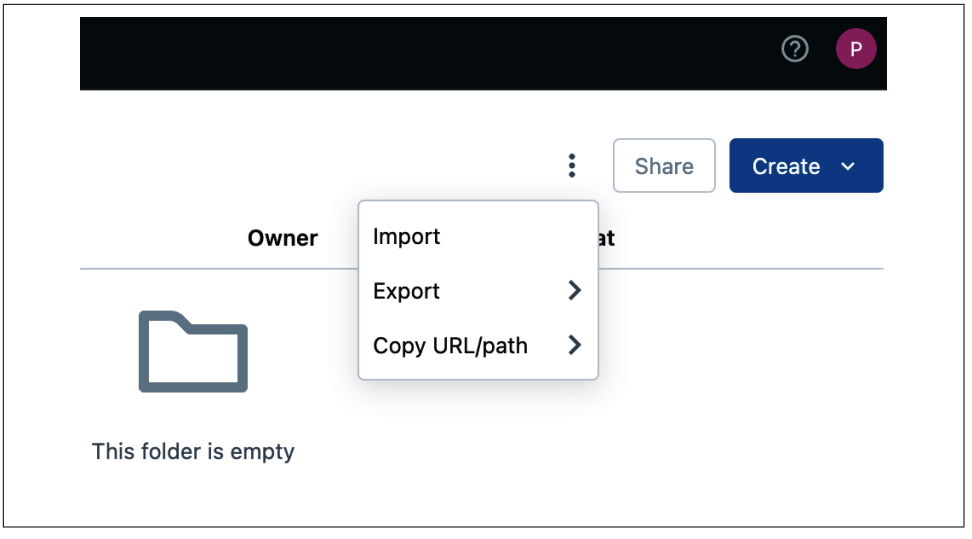


Figure 2-3. Importing a notebook in Databricks Community Edition

In the dialog box, click on the URL radio button, paste in the notebook URL, and click Import. This will render the Jupyter Notebook in Databricks Community Edition.

Attaching Notebooks

Now select the Delta_Lake_DLDG cluster you created earlier to run this notebook, as shown in Figure 2-4.

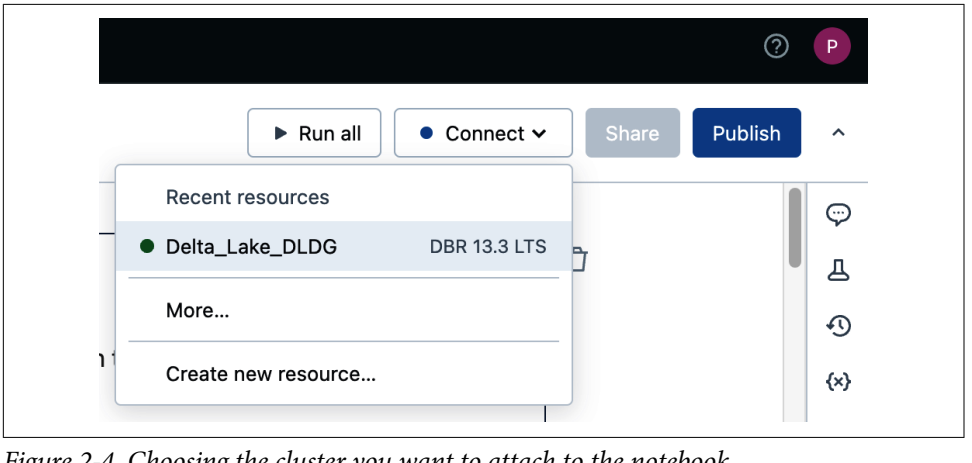


Figure 2-4. Choosing the cluster you want to attach to the notebook

You can now run each cell in the notebook and press Control + Enter on your keyboard to execute the cell. When a Spark Job is running, Databricks Community Edition shows finer details directly in the notebook. You can also navigate to the Spark UI from here.

You will be able to write to and read from the Delta Lake table within this notebook.

Conclusion

In this chapter, we explored the various approaches you can take to get started with Delta Lake, including Delta Docker, Delta Lake for Python, Apache Spark with Delta Lake, PySpark Declarative API, and finally Databricks Community Edition. We showed how easily you can run a simple notebook or a command shell to write to and read from Delta Lake tables. The next chapter will cover writing and reading operations in more detail.

Finally, we showed you how to use any of these approaches to install Delta Lake and the many different ways in which Delta Lake is available. You also learned how to use SQL, Python, Scala, Java, and Rust programming languages through the API to access Delta Lake tables. In the next chapter, we'll cover the essential operations you need to know to use Delta Lake.

CHAPTER 3
Essential Delta Lake Operations

This chapter explores the essential operations of using Delta Lake for your data management needs. Since Delta Lake functions as the storage layer and participates in the interaction layer of data applications, it makes perfect sense to begin with the foundational operations of persistent storage systems. You know that Delta Lake provides ACID guarantees already,[1] but focusing on CRUD operations (see Figure 3-1) will point us more toward the question "How do I use Delta Lake?"[2] This would be a woefully short story (and consequently this would be a short book) if that was all that you needed to know, however, so we will look at several additional things that are vital to interacting with Delta Lake tables: merge operations, conversion from so-called *vanilla* Parquet files, and table metadata.

Except where specified, *SQL* will refer to the Spark SQL syntax for simplicity's sake. If you are using Trino or some other SQL engine with Delta Lake, you can find additional details either in Chapter 4, which explores more of the Delta Lake ecosystem, or in the relevant documentation.[3] The Python examples will all use the Spark-based Delta Lake API for the same reason. Equivalent examples are presented for both throughout. It is also possible to leverage the equivalent operations using PySpark, and examples of that are shown where it makes sense to do so.

1 ACID transactions are discussed in Chapter 1.

2 For a comparison of variant approaches or different types of applications, start with the Wikipedia article on CRUD operations.

3 See the Trino SQL documentation, for example.

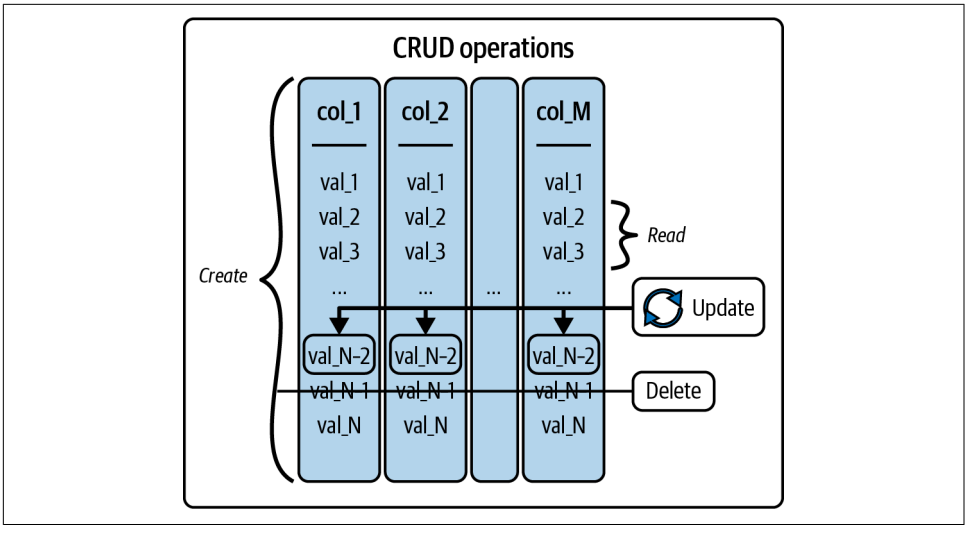


Figure 3-1. Create, read, update, and delete (CRUD) operations are among the most fundamental operations required for any persistent storage data system

We can perform operations with Delta Lake tables using the top-level directory path of a Delta Lake table or by accessing it via a catalog, like the Hive Metastore commonly used with Apache Spark, or the more advanced Unity Catalog.[4] You will see both methods used throughout this chapter; your choice of which method to use will depend primarily on personal preference and the features of the systems you are working with. Generally speaking, if you have a catalog available in the environment you use, it simplifies both the readability of your code and potential future transactions (imagine if you change a table's location). Note that if you use a catalog, you can set a location for the database object or individually for each table.

Create

Before much else can be done, you need to create a table so there's something to interact with. The actual creation operation can occur in different forms, as many engines will handle something like a nonexistent table simply by creating it as part of the processing during certain actions (such as an `append` operation in Spark SQL). "What gets created during this process?" you might ask. At its core, Delta Lake could not exist without Parquet, so one of the things you will see created is the Parquet file directory and data files, as if you had used Parquet to create the table. However, one of the new things you should notice is another file called *_delta_log*.

4 For a technical review of the Hive Metastore, including its design and the interaction operations that are fundamental to its operation, see the Apache Hive documentation.

Creating a Delta Lake Table

To create an empty Delta Lake table, you need to define the table's schema.[5] Using SQL, this will look just like any database table definition, except you will also specify that your table is based on Delta Lake by including the `USING DELTA` parameter:

```
-- SQL
CREATE TABLE exampleDB.countries (
  id LONG,
  country STRING,
  capital STRING
) USING DELTA;
```

All the examples and other supporting code for this chapter can be found in the book's GitHub repository.

In Python, you will start with the `TableBuilder` object yielded by the method `DeltaTable.create` and then add attributes like the table name and the definitions of the columns to be included. The `execute` command combines the definition into a query plan and puts it into action:

```
# Python
from pyspark.sql.types import *
from delta.tables import *

delta_table = (
    DeltaTable.create(spark)
    .tableName("exampleDB.countries")
    .addColumn("id", dataType=LongType(), nullable=False)
    .addColumn("country", dataType=StringType(), nullable=False)
    .addColumn("capital", dataType=StringType(), nullable=False)
    .execute()
    )
```

In either the Python or the SQL method of defining the table, the process itself is essentially just a matter of creating a named table object with a specification of the column names and types. One other element you might have noticed in the Python dialect is that we also have the option to specify nullability in Apache Spark. This setting will be ignored for Delta Lake tables, as it applies only to JDBC sources. An additional item you might commonly include during a create statement is the `IF NOT`

5 Type support does vary by engine to some degree, though most engines support most data types. For an example of the types supported by Azure Databricks, see the documentation page.

`EXISTS` qualifier in SQL or the alternative method `createIfNotExists` in Python. Their use is purely at your discretion.

Many of the examples throughout this chapter take the use of table objects accessed through a catalog for granted, but most of the essential operations here are well supported with direct file access methods. One of the key differences for Spark SQL is that it uses the path accessor `delta.`<TABLE>`` (note the backticks) in place of a table name. With the DeltaTable API, you will typically just swap out the `forPath` method in place of `forName`. In PySpark you'll have to turn to alternative methods at times as well, such as using `save` with a path `argument` in place of `saveAsTable` with a table name. Refer to Delta Lake's Python documentation for additional details that might need to be configured for path-based access in some cases (e.g., cloud provider–specific security configuration arguments).

Loading Data into a Delta Lake Table

Assuming you have a Delta table, the most common operations will consist of reading or writing, and naturally, before you read from a table you will probably want to write to it first so that there is something to read. This brings us to one of the most prominent differences between using SQL and using Python APIs with Delta Lake. With either method, you will first define a table and then put rows into that table, but between the two the syntax for doing this is a little different. With SQL you need to use an `INSERT` statement, whereas with Python you can use the similar `insertInto` method or use Spark `append` operations instead.

INSERT INTO

When you have an empty Delta Lake table, you can load data into it using the `INSERT INTO` command. The idea is to define where you are inserting data and then what you are inserting by providing the `VALUES` for each row with all the specific info of the columns:

```
-- SQL
INSERT INTO exampleDB.countries VALUES
(1, 'United Kingdom', 'London'),
(2, 'Canada', 'Toronto')
```

With PySpark DataFrame syntax, you just need to specify that inserting records into a specific table is the destination of a write operation with `insertInto` (note that columns are aligned positionally, so column names will be ignored with this method):

```
# Python
data = [
    (1, "United Kingdom", "London"),
    (2, "Canada", "Toronto")
    ]

schema = ["id", "country", "capital"]

df = spark.createDataFrame(data, schema=schema)

(
df
.write
.format("delta")
.insertInto("exampleDB.countries")
)
```

There might be cases in which you already have the required data (with the same schema and headers) in other formats such as CSV or Parquet. You can specify that the source is a file and select from it or even directly specify another table. This data, via a `SELECT` statement, can be swapped out with the `VALUES` argument in the `INSERT INTO` operation. You need to specify which columns you are selecting from the new data source or specify that you are selecting an entire table with `SELECT TABLE <table name>` instead:

```
-- SQL
INSERT INTO exampleDB.countries
SELECT * FROM parquet.`countries.parquet`;
```

This provides one way of appending preexisting data into a Delta Lake table. Another way will be through the `append` mode option for Spark DataFrame write operations.

Append

In addition to the `insertInto` method for a DataFrame, we can add new data to a Delta Lake table using append mode. In SQL, this just happens as part of the `INSERT INTO` operation, but for the `DataFrameWriter` you will explicitly set writing mode with the syntax `.mode(append)`, or with its longer specification `.option("mode", "append")`. This informs the `DataFrameWriter` that you are only adding additional records to the table. When a DataFrame is written with the mode set to `append` and the table already exists, data gets appended to it; however, if the table didn't exist before, it will be created:

```
# Python
# Sample data
data = [(3, 'United States', 'Washington, D.C.') ]

# Define the schema for the Delta table
schema = ["id", "country", "capital"]
```

```
# Create a DataFrame from the sample data and schema
df = spark.createDataFrame(data, schema=schema)

# Write the DataFrame to a Delta table in append mode
# (if the table doesn't exist, it will be created)
(df
.write
.format("delta")
.mode("append")
.saveAsTable("exampleDB.countries")
)
```

If the mode is not set, Delta Lake assumes you are creating a table by default, but if a table with that name already exists, you will receive the following error message:

```
AnalysisException: [TABLE_OR_VIEW_ALREADY_EXISTS] Can-
not create table or view `exampleDB`.`countries`
because it already exists.

Choose a different name, drop or replace the existing
object, add the IF NOT EXISTS clause to tolerate pre-
existing objects, or add the OR REFRESH clause to
refresh the existing streaming table.
```

For PySpark users, this is the most common method of appending data to a table because it provides a little more flexibility in the event you might like to specify different write modes at different points in development. It also uses the table specification to align column names from the incoming DataFrame, unlike the `insertInto` method.

CREATE TABLE AS SELECT

With `append` operations in PySpark, we noted that if you appended to a table that did not yet exist, then one would be created. In SQL with `INSERT INTO`, this is not the case. You must define the destination table to which you want to append the records before trying to insert records into it. One way to get behavior more like the `append` operation is to use a CTAS (Create Table As Select) statement to combine the creation of the table and the insertion of data into a single operation:

```
-- SQL
CREATE TABLE exampleDB.countries2 AS
SELECT * FROM exampleDB.countries
```

Using CTAS statements to create your tables gives you some additional simplicity for your table definitions. One of the biggest benefits is that you get to skip the step of defining the schema in cases where you don't need granular control over the type or over other column-level information. Whether you want to use this method or

standard `CREATE` and `INSERT` operations is up to you; for the most part, they will yield pretty much the same result. The main area in which the two methods differ is that they require a different number of transactions and will be represented separately in the transaction log you will see next.

The Transaction Log

When your table is created, you get a subdirectory within the Parquet structure called *_delta_log*. This is the transaction log that tracks all of the change history for a table.[6] If you inspect the file structure of the *_delta_log* directory, you will find that it contains JSON files:

```
# Bash
!tree countries.delta/_delta_log

countries.delta/_delta_log
└── 00000000000000000000.json
```

These files provide a record of all the operations that happen to the table and make some kind of change (i.e., not read operations). Each creation, insertion, or append action will add another JSON file to the transaction log and increment the version number of the table. The exact structure of the transaction log varies by implementation, but some of what you will commonly find within the transaction records is information about the creation of the table (such as what processing engine was used to create it, the number of records, or other metrics from write operations to the table), records of maintenance operations, and deletion information.[7]

While it may seem like a small thing, you should understand that the transaction log is the core component that makes Delta Lake work. Some might even go so far as to say that the transaction log *is* Delta Lake. The record of transactions and the way processing engines interact with it are what set Delta Lake apart from Parquet and provide ACID guarantees, the possibilities of exactly-once stream processing, and all the other magic Delta Lake provides to you. One example of the magic that comes from the transaction log is *time travel*, which is described in the next section.

The details may differ depending on where and how you are using Delta Lake, but the key takeaway is that you need to know that the transaction log exists and where to find it. Owing to the richness of the information often included in the transaction log, you may find it an invaluable tool for investigating processes, diagnosing errors, and monitoring the health of your data pipelines. Don't neglect the information available at your fingertips!

6 Chapter 1 includes a detailed review of the transaction log, but this is a critical concept.

7 Matthew Powers provides a handy reference to many implementations of the Delta Lake transaction log, if you want to compare what information might be available to each.

Read

Reading is such a fundamental operation in data processing that one could almost assume there is no need to look into it. However, there are several things concerning reading from Delta Lake tables that are worth focusing on, including a high-level understanding of how partition filtering works (which is explored much more deeply in Chapters 5 and 10) and how the transaction log allows querying views of the data from previous versions with time travel.

Querying Data from a Delta Lake Table

Just as with the rich database systems you might have encountered in the past, there are many kinds of SQL tricks you can use to manipulate data read from a Delta Lake table. What is central to getting to more advanced practices, however, is understanding that everything builds on top of basic read operations. In SQL this usually takes the form of a `SELECT` statement, whereas with the DeltaTable API, you will load an object and convert it to a Spark DataFrame:

```
-- SQL
SELECT * FROM exampleDB.countries

# Python
from delta.tables import DeltaTable

delta_table = DeltaTable.forName(spark, "exampleDB.countries")
delta_table.toDF()
```

Both methods will yield all table records, with the output limited only by the processing engine you're using (e.g., the default value of `show` in Spark gives 20 records). Often you will want to select just a subset of the data; this could be a single record, an entire file partition, or an arbitrary collection of records from disparate locations throughout the table.[8] To facilitate this, you just need to add a filtering action to the query or DataFrame definition:

```
-- SQL
SELECT * FROM exampleDB.countries
WHERE capital = "London"

# Python
delta_table_df.filter(delta_table_df.capital == 'London')
```

8 The concept of partitioning is a supported part of the Parquet file structure. For an in-depth exploration of partitioning, we suggest checking out the Spark documentation covering Parquet files.

This will give you all the results from your table that match the filtering condition. This is true even when the values don't exist. If a value is specified as a filtering condition but does not exist in the table, the result will be no records returned. In Spark, this will be an empty `DataFrame` object that still shares the originating table schema. You might also wish to select a subset of columns in conjunction with filters or just to perform other operations. To do so, just specify the columns required:

```
-- SQL
SELECT
    id,
    capital
FROM
    exampleDB.countries

# Python
delta_table_df.select("id", "capital")
```

While the operations themselves are simple, more is going on under the covers. Just like Parquet files, Delta Lake includes statistics on the underlying files to make these queries more efficient where possible. The amount of statistics can be controlled and varies by implementation (Chapter 12 focuses on these statistics and on how you can optimize performance). Parquet itself is a columnar file structure, so reading subsets of columns will often be more efficient overall. One area in which the transaction log comes into play and offers a distinct advantage over traditional Parquet files is the ability to read data from the past.

Reading with Time Travel

Courtesy of the transaction log in Delta Lake, you have additional table parameters that you can use to accomplish some otherwise difficult tasks. One thing made possible through the log is the ability to view or restore older versions of a table. This means you can check on previous versions or on what the data in a table looked like at a certain time, with less effort required to create backups of files, and without relying on native cloud service backup utilities.

The DeltaTable API used throughout this chapter does not directly support time travel. However, that feature is still available to Python users via PySpark. The API supports restoration actions, which are covered in "Repairing, Restoring, and Replacing Table Data" on page 108. You will also find some more advanced operations regarding deletion of data. In light of this limitation, equivalent expressions with PySpark are presented alongside the SQL expressions for time travel.

To view a previous version of a table in SQL, just add a qualifier to the query. There are two different options for specifying this. One is to specify the `VERSION AS OF` with a particular version number. For example, if you want to see which values of `id` existed as part of a specific version of the table, you might combine a `DISTINCT` query with time travel to version 1 of the table:

```
-- SQL
SELECT DISTINCT id FROM exampleDB.countries VERSION AS OF 1

# Python
(
spark
.read
.option("versionAsOf", "1")
.load("countries.delta")
.select("id")
.distinct()
)
```

Or if you want to see how many records existed before the current date without having to check the version number, you can use `TIMESTAMP AS OF` instead and specify the current date:[9]

```
-- SQL
SELECT count(1) FROM exampleDB.countries TIMESTAMP AS OF "2024-04-20"

# Python
(
spark
.read
.option("timestampAsOf", "2024-04-20")
.load("countries.delta")
.count()
)
```

While extremely useful as a feature, time travel is just a by-product of proper versioning on your table. It does exemplify the protections that you get for your data in terms of transaction guarantees and atomicity, though. So really, time travel is not the full benefit but a window into the protections provided by ACID transactions available in Delta Lake.[10]

9 This "same day" behavior does depend on the way Spark converts a date to a timestamp—i.e., in the examples, `2024-04-20 = 2024-04-20 00:00:00`.

10 To read more about retention timelines and versioning, see Chapter 5.

Update

Being able to create a table, add data to it, and read the data from it are all great capabilities. Sometimes, though, maybe while reading the data, an error in a name could be discovered. Or perhaps integration with a business system requires the abbreviation of a country name.

Suppose your sales team decides that it wants to use country abbreviations in place of the full names because they are shorter and display better in the graphs in sales reports. In this case, you would need to update the column value of "United Kingdom" to "U.K." in the table from the examples. To make these kinds of changes, all you need is an `UPDATE` statement that specifies what you want to change with `SET` and where you want to change it with a `WHERE` clause:

```
-- SQL
UPDATE exampleDB.countries
SET { country = 'U.K.' }
WHERE id = 1;

# Python
delta_table_df.update(
    condition = "id = 1",
    set = { "country": "'U.K.'" } )
```

Using `UPDATE` makes it easy to fix a specific value in a table. You can also use this to update many values in your table by using a less specific filtering clause. Omitting the `WHERE` clause completely would allow you to update values across the entire Delta Lake table. Each update action will increment the version of the table in the transaction log.

Delete

Deleting data from a table is the last of the CRUD operations to be explored here. Deletions can happen for many reasons, but a few of the most common ones are to remove specific records (e.g., right to be forgotten[11]), to replace erroneous or stale data (e.g., daily table refresh), or to trim a table time window (most often when the same data might be available elsewhere but you wish to keep a reporting table or similar to a trimmed length for performance or as part of the basis for calculations). For some of these, you would want to give explicit commands to remove values; in other cases, you might be able to let the system handle the deletion on your behalf.

11 The right to be forgotten or right to erasure is part of the EU's General Data Protection Regulation (GDPR). If this law applies to your data practices, we suggest you thoroughly review the EU's "Complete Guide to GDPR Compliance".

The two usual ways to achieve this are to use the `DELETE` command or to specify overwriting behavior.

Deleting Data from a Delta Lake Table

All that is required to delete records from a Delta Lake table is the `DELETE` statement. Functionally, it works very similarly to a `SELECT` statement in that you apply filtering with a `WHERE` clause to get the appropriate level of selectivity for determining the records to be deleted. This has the handy application of being able to first select and review the records that would be deleted from the table by simply switching between `SELECT` and `DELETE` in a SQL query. With the Python API, you do not have this same swappable parity (swapping `SELECT` for `DELETE`) to make reviewing records easier, but what *is* similar is supplying the condition on which to delete. It is still just an expression specifying matching criteria for the operation. One enhancement you gain with the Python API is that the expression itself can be a string containing an expression in SQL or can use functions out of the PySpark libraries. This can give you additional flexibility in the way you write your code:

```
-- SQL
DELETE FROM exampleDB.countries
WHERE id = 1;

# Python
from pyspark.sql.functions import col

delta_table.delete("id = 1")       # uses SQL expression
delta_table.delete(col("id") == 2) # uses PySpark expression
```

Adequate care should be taken to specify the `WHERE` clause so as to prevent unintentional deletion of additional data. Failure to include a `WHERE` clause will result in the deletion of all table records. Similar care should also be used when using overwrite mode with writes to a table. If this happens to you accidentally, you will need to restore a prior version of the table (see Chapter 5 for more details on how to do this).

Deleting many records from a table works similarly to deleting a single record. In cases in which the value in the expression matches multiple records, all those records will get deleted. You could also use inequality-based expressions to delete based on thresholds. An example of this kind of expression might look something like `"transaction_date <= date_sub(current_date(), 7)"`, which would trim the table values to have only values within the last week. Deleting large amounts of data from a table can often be associated with replacing the data in that table with a whole new set of records. Rather than doing this as a two-step operation, there may be cases in which you would like to *overwrite* the data instead.

Overwriting Data in a Delta Lake Table

Delta Lake makes it very easy to overwrite data in a table. This is both a feature and a warning for you. Overwrite mode allows you to replace the data in a table, no matter the size or number of files, with a new result set.[12] The exception is when you specify that you want to overwrite only a specific partition of the data, but even in this case, you should be deliberate about what you are doing, as partitions can also contain many files that might be overwritten during the process. With that in mind, overwriting tables is a fairly common process and can be used for updating records in a table when they are recomputed regularly, when an error occurs and you wish to replace some or all of a table as a result, or when you wish to change the structure of a table. Overwriting is included within this section's discussion since a core component of any of the outlined methods is implicitly deleting any preexisting data. Each approach to interacting with Delta Lake via Spark uses a different method to overwrite data. The DeltaTable API has a unique `replace` method, while PySpark and Spark SQL both have a way to specify overwriting as an operation mode.

The replace method

When using the DeltaTable Python API, there are distinct methods that allow for replacing the entire contents of a table. You can use either `replace` or `createOrReplace` to replace the contents of a table. Both methods are direct handlers that let you use the same `TableBuilder` object to define a new table structure over the top of the existing one:

```
# Python
delta_table2 = (
    DeltaTable.replace(spark)
    .tableName("countries.delta")
    .addColumns(data_df.schema)
    .execute()
    )
```

Working with the DeltaTable API allows you to overwrite the table schema with the column definitions coming from a DataFrame called `data_df`. Overall, if you are already working with Spark, you might find it easier to use the overwrite mode specification from the Spark `DataFrameWriter` instead.

Overwrite mode

Overwriting data by changing the output mode on a Spark `DataFrameWriter` can be a quick and efficient method for wholly replacing part or all of a dataset in Delta Lake. The overwrite mode parameter is a mirror of the append mode parameter used

12 More detailed discussion of overwrite semantics can be found in Chapter 5.

to add data to a Delta Lake table. In this case, instead of data being added to the table's preexisting data, the contents of the current DataFrame will just replace what is already in the table. All of the prior data will be removed and only the current data will be available going forward, unless you restore the table to the prior version:

```
# Python
(
spark
.createDataFrame(
    [
    (1, 'India', 'New Delhi'),
    (4, 'Australia', 'Canberra')
    ],
    schema=["id", "country", "capital"]
    )
.write
.format("delta")
.mode("overwrite") # specify the output mode
.saveAsTable("exampleDB.countries")
)
```

Using this method gives you the ability to switch between the two different output modes by changing just one word, which can be particularly useful during development and testing. You can do something similar in Spark SQL with `INSERT OVERWRITE`.

INSERT OVERWRITE

As a companion to `INSERT INTO`, `INSERT OVERWRITE` can be used in the same way as the overwrite mode with PySpark DataFrame syntax.[13] These two query-based commands function in the same way as append and overwrite modes in PySpark; that is, they allow you to switch between the `INTO` and `OVERWRITE` parameters without making other changes to your queries:

```
-- SQL
INSERT OVERWRITE exampleDB.countries
VALUES (3, 'U.S.', 'Washington, D.C.');
```

As with the overwrite mode or the `replace` method, using `INSERT OVERWRITE` will remove all previous data from the target table. This means you should exercise caution when using it and make sure you know what you are overwriting. As with the `INSERT INTO` command, you have a large amount of freedom with regard to the contents you want to insert into the target table. You can use specific values, other tables, or files as a source for writing over a target table.

13 Due to the way Trino interacts with files, it does not directly support `INSERT OVERWRITE`.

Merge

Combining inserts, updates, and/or deletes in processing data is common enough to warrant creating "shortcuts" for those actions. `MERGE` is a great example of such a shortcut, as it allows you to chain together multiple operations on a set of data with a unified set of matching conditions across the chain. It allows you to conditionally control actions based on the degree of matching or not matching as you specify. When the operations are limited only to combining inserts and updates, merging is also commonly called *upserting*.[14] This can be used to great advantage with practice, as many day-to-day data engineering patterns align with merge behavior.[15]

If you have many records to insert into a table but also need to update previously existing records, you may need to combine and perform several different queries. To accomplish this, you would need to identify which records are already in the table, update those records, and then take the additional records and insert those into the same target. `MERGE` lets you combine these actions into a single operation by conditionally qualifying the logic in your query based on how it matches against specified values in the table. In essence, you get to specify different actions based on whether or not particular values already exist in the key columns of the table.

The number of ways in which you can create combinations within a `MERGE` query is enormous, but generally speaking, you will define a set of matching criteria between a target table (the one you want to make changes to) and some source data (from a file or another table, for example). With matching criteria defined, you can take different actions depending on the matching status of each record coming from the source data:

`WHEN MATCHED`
: When the conditions are matched, you can either `DELETE` matching records or `UPDATE` with the entire new record or from specified columns.

`WHEN NOT MATCHED`
: When conditions are not matched, you can `INSERT` unmatched records either in their entirety or from specified columns.

`WHEN NOT MATCHED BY SOURCE`
: When no new records from the source match records in the target, you can `DELETE` those records or `UPDATE` with either the entire new record or from specified columns.

14 For a more in-depth exploration of its history and implementation comparisons across multiple SQL dialects, we recommend the "Merge (SQL)" Wikipedia article as a jumping-off point.

15 For a dedicated exploration of Delta Lake merge semantics, we suggest Nick Karpov's blog post.

With SQL, you simply combine the actions to build your entire `MERGE` query and execute it as a single statement. You start by specifying the target to merge into, the source to merge from, and the conditions on which you want to base your matching logic. Then, for an upsert, you will just define the update operation and insert operation details:

```
-- SQL
MERGE INTO exampleDB.countries A
USING (select * from parquet.`countries.parquet`) B
ON A.id = B.id
WHEN MATCHED THEN
  UPDATE SET
    id = A.id,
    country = B.country,
    capital = B.capital
WHEN NOT MATCHED
  THEN INSERT (
    id,
    country,
    capital
  )
  VALUES (
    B.id,
    B.country,
    B.capital
  )
```

With the DeltaTable API, you will use a new class called the `DeltaMergeBuilder` to specify these conditions and actions. Unlike in the SQL syntax, each combination of matching status and subsequent action to take has its own method to use. You can find the full list of supported combinations in the documentation. We recommend you combine multiple actions and just chain them together into a single transaction to help you break down the logical path of any particular record. Here is what it might look like if you wanted to do an upsert operation with a DataFrame containing new records; notice that, starting with the DeltaTable object, you first apply `MERGE` to specify the new record source and the matching conditions and then apply `whenMatchedUpdate` and `whenNotMatchedInsert` to cover both cases:

```
# Python
idf = (
    spark
    .createDataFrame([
        (1, 'India', 'New Delhi'),
        (4, 'Australia', 'Canberra')],
        schema=["id", "country", "capital"]
        )
    )

delta_table.alias("target").merge(
    source = idf.alias("source"),
```

```
condition = "source.id = target.id"
).whenMatchedUpdate(set =
{
"country": "source.country",
"capital": "source.capital"
}
).whenNotMatchedInsert(values =
{
"id": "source.id",
"country": "source.country",
"capital": "source.capital"
}
).execute()
```

Overall, using `MERGE` can help you simplify what otherwise would require several distinct queries with different kinds of join logic and associated actions.

Other Useful Actions

There are a couple more essential operations to be aware of with Delta Lake. One is a conversion that can simplify moving to Delta Lake from other file formats, and the other is a review of the functions you need to inspect several different kinds of metadata about your tables. Both can be highly valuable to you for many applications.

Parquet Conversions

Even in cases where you establish Delta Lake as the file format underlying all your data activities, you are still likely to encounter datasets coming from legacy systems, third-party providers, or other sources that use different formats. For a couple of file types, namely the Parquet and the Parquet-based Iceberg formats, there is a simple conversion method you can use to simplify some of your operations. The `CONVERT TO DELTA` command is the recommended approach for transforming an Iceberg or Parquet directory into a Delta table.

Regular Parquet conversion

Since a Delta Lake table is composed of Parquet files internally, the transaction log is the biggest difference when converting a Parquet table to a Delta Lake table. To create a log for an existing Parquet file, you just need to run `CONVERT TO DELTA` in SQL, or `convertToDelta` with the DeltaTable API, with the directory:

```
-- SQL
CONVERT TO DELTA parquet.`countries.parquet`

# Python
from delta.tables import DeltaTable
```

```
delta_table = (
    DeltaTable
    .convertToDelta(
        spark,
        "parquet.`countries.parquet`"
        )
    )
```

This command scans all Parquet files within the specified directory, infers the schema from the types stored in the Parquet files, and builds the Delta Lake transaction log *_delta_log*. If the Parquet directory is partitioned, you will also need to specify the partitioning columns using a `PARTITIONED BY` parameter in the SQL query or a SQL string as an additional argument for `convertToDelta`.

Iceberg conversion

Like Delta Lake, Apache Iceberg is composed of Parquet files internally. Is it possible to again use `CONVERT TO DELTA` in SQL, or `convertToDelta`, to convert Iceberg files? Partly yes and partly no. The DeltaTable API does not support the Iceberg conversion. Spark SQL, however, can support the conversion with `CONVERT TO DELTA`, but you will also need to install support for the Iceberg format in your Spark environment:

```
-- SQL
CONVERT TO DELTA iceberg.`countries.iceberg`
```

You should be able to accomplish this by installing an additional JAR file (*delta-iceberg*) to the cluster you are using.[16] Unlike with Parquet files, when converting Iceberg you will not need to specify the partitioning structure of the table, as it will infer this information from the source.[17]

There's one more thing you should know about this conversion process. An interesting side effect exists in converted Iceberg tables. Since both Iceberg and Delta Lake maintain distinctly separate transaction logs, none of the new files added through interactions via Delta Lake will be registered on the Iceberg side. However, since the Iceberg log is not removed, the new Delta Lake table will still be readable and accessible as an Iceberg table.[18]

16 For further exploration of using Apache Iceberg with Apache Spark, we suggest starting with the official quickstart guide.

17 There are some caveats to being able to convert Iceberg based on different feature usage. You should look at the documentation to check whether this might affect your specific situation.

18 This is different from Delta UniForm (Universal Format), which we discussed in Chapter 1.

Delta Lake Metadata and History

Often you will want to quickly check some information about the metadata related to one of your Delta Lake tables. It can be useful to review information such as the schema of a table, which reader or writer version is implemented for a table, or any other properties that might be set. To review this information, you only need to use `DESCRIBE DETAIL` in Spark SQL or `detail` with the DeltaTable API:

```
-- SQL
DESCRIBE DETAIL exampleDB.countries

# Python
delta_table.detail()
```

This will give you all the metadata details listed, as well as additional things such as the last time the table was modified or the number of files in the table. You can find a reference for the entire schema returned in the documentation.

Similarly, you might wish to check not just the most recent values of this metadata but also the metadata for each transaction. There are a couple of ways to easily access the information stored in the transaction log, which can provide a rich history of the changes that have taken place in the table over time. This has many potential applications, such as monitoring the frequency and size of `append` operations to a table or checking the source of a particular deletion.

In this case, instead of the detail, you will want the history of the table:

```
-- SQL
DESCRIBE HISTORY exampleDB.countries

# Python
delta_table.history()
```

Similar to the metadata, the table history contains a great deal of different transaction-level metadata and, depending on the type of transaction, many associated metrics. You can find an overview of the available information in the documentation as well.

Conclusion

The essential operations of Delta Lake provide a robust interaction layer for creating, reading, updating, and deleting data in tables, going well beyond traditional data lake capabilities. With ACID transactions, time travel, merge operations, and easy conversion from Parquet and Iceberg formats, Delta Lake offers a powerful storage and data management layer. By understanding the essential operations covered in this chapter—from basic CRUD actions to more advanced merge logic and transaction log introspection—you can effectively use Delta Lake to build reliable, high-performance data pipelines and applications.

CHAPTER 4
Diving into the Delta Lake Ecosystem

Over the last few chapters, we've explored Delta Lake from the comfort of the Spark ecosystem. The Delta protocol, however, offers rich interoperability not only across the underlying table format but within the computing environment as well. This opens the doors to an expansive universe of possibilities for powering our lakehouse applications, using a single source of table truth. It's time to break outside the box and look at the connector ecosystem.

The connector ecosystem is a set of ever-expanding frameworks, services, and community-driven integrations enabling Delta to be utilized from just about anywhere. The commitment to interoperability enables us to take full advantage of the hard work and effort the growing open source community provides without sacrificing the years we've collectively poured into technologies outside the Spark ecosystem.

In this chapter, we'll discover some of the more popular Delta connectors while learning to pilot our Delta-based data applications from outside the traditional Spark ecosystem. For those of you who haven't done much work with Apache Spark, you're in luck, since this chapter is a love song to Delta Lake without Apache Spark and a closer look at how the connector ecosystem works.

We will be covering the following integrations:[1]

- Flink DataStream Connector
- Kafka Delta Ingest
- Trino Connector

1 For the full list of evolving integrations, see "Delta Lake Integrations" on the Delta Lake website.

In addition to the four core connectors in this chapter, support for Apache Pulsar, ClickHouse, FINOS Legend, Hopsworks, Delta Rust, Presto, StarRocks, and general SQL import to Delta is also available at the time of writing.

What are connectors, you ask? We will learn all about them next.

Connectors

As people, we don't like to set limits for ourselves. Some of us are more adventurous and love to think about the unlimited possibilities of the future. Others take a more narrow, straight-ahead approach to life. Regardless of our respective attitudes, we are bound together by our pursuit of adventure, search for novelty, and desire to make decisions for ourselves. Nothing is worse than being locked in, trapped, with no way out. From the perspective of the data practitioner, it is also nice to know that what we rely on today can be used tomorrow without the dread of contract renegotiations! While Delta Lake is not a person, the open source community has responded to the various wants and needs of the community at large, and a healthy ecosystem has risen up to ensure that no one will have to be tied directly to the Apache Spark ecosystem, the JVM, or even the traditional set of data-focused programming languages like Python, Scala, and Java.

The *mission of the connector ecosystem* is to ensure frictionless interoperability with the Delta protocol. Over time, however, fragmentation across the current (`delta` < 3.0) connector ecosystem has led to multiple independent implementations of the Delta protocol and divergence across the current connectors. To streamline support for the future of the Delta ecosystem, Delta Kernel was introduced to provide a common interface and expectations that simplify true interoperability within the Delta ecosystem.

Kernel provides a seamless set of read- and write-level APIs that ensures correctness of operation and freedom of expression for the connector API implementation. This means that the behavior across all connectors will leverage the same set of operations, with the same inputs and outputs, while ensuring each connector can quickly implement new features without lengthy lead times or divergent handling of the underlying Delta protocol. Delta Kernel is introduced in Chapter 1.

There are a healthy number of connectors and integrations that enable interoperability with the Delta table format and protocols, no matter where we trigger operations from. Interoperability and unification are part of the core tenets of the Delta project and helped drive the push toward UniForm (introduced along with Delta 3.0), which provides cross-table support for Delta, Iceberg, and Hudi.

In the sections that follow, we'll take a look at the most popular connectors, including Apache Flink, Trino, and Kafka Delta Ingest. Learning to utilize Delta from your favorite framework is just a few steps away.

Apache Flink

Apache Flink is "a framework and distributed processing engine for stateful computations over unbounded and bounded data streams...[that] is designed to run in all common cluster environments [and] perform computations at in-memory speed and at any scale." In other words, Flink can scale massively and continue to perform efficiently while handling every increasing load in a distributed way, and while also adhering to exactly-once semantics (if specified in the `CheckpointingMode`) for stream processing, even in the case of failures or disruptions at runtime to a data application.

If you haven't worked with Flink before and would like to, there is an excellent book by Fabian Hueske and Vasiliki Kalavri called *Stream Processing with Apache Flink* (O'Reilly) that will get you up to speed in no time.

The assumption from here going forward is that we either (a) understand enough about Flink to compile an application or (b) are willing to follow along and learn as we go. With that said, let's look at how to add the `delta-flink` connector to our Flink applications.

Flink DataStream Connector

The Flink/Delta Connector is built on top of the Delta Standalone library and provides a seamless abstraction for reading and writing Delta tables using Flink primitives such as the `DataStream` and `Table` APIs. In fact, because Delta Lake uses Parquet as its common data format, there really are no special considerations for working with Delta tables aside from the capabilities introduced by the Delta Standalone library.

The standalone library provides the essential Java APIs for reading the Delta table metadata using the `DeltaLog` object. This allows us to read the full current version of a given table, or to begin reading from a specific version, or to find the approximate version of the table based on a provided ISO-8601 timestamp. We will cover the basic capabilities of the standalone library as we learn to use `DeltaSource` and `DeltaSink` in the following sections.

The full Java application referenced in the following sections is located in the book's Git repository under */ch04/flink/dldg-flink-delta-app/*.

As a follow-up for the curious reader, unit tests for the application provide a glimpse at how to use the Delta standalone APIs. You can walk through these under */src/test/* within the Java application.

Installing the Connector

Everything starts with the connector. Simply add the `delta-flink` connector to your data application using Maven, Gradle, or sbt. The following example shows how to include the `delta-flink` connector dependency in a Maven project:

```
<dependency>
  <groupId>io.delta</groupId>
  <artifactId>delta-flink</artifactId>
  <version>${delta-connectors-version}</version>
</dependency>
```

It is worth noting that Apache Flink is officially dropping support for the Scala programming language. The content for this chapter is written using Flink 1.17.1, which officially no longer has published Scala APIs. While you can still use Scala with Flink, Java and Python will be the only supported variants as we move toward the Flink 2.0 release. All of the examples, as well as the application code in the book's GitHub repository, are therefore written in Java.

The connector ships with classes for reading and writing to Delta Lake. Reading is handled by the `DeltaSource` API, and writing is handled by the `DeltaSink` API. We'll start with the `DeltaSource` API, move on to the `DeltaSink` API, and then look at an end-to-end application.

The value of the `delta-connectors-version` property will change as new versions are released. For simplicity, all supported connectors are officially included in the main Delta repository. This change was made at the time of the Delta 3.0 release.

DeltaSource API

The `DeltaSource` API provides static builders to easily construct sources for bounded or unbounded (continuous) data flows. The big difference between the two variants is specific to the bounded (batch) or unbounded (streaming) operations on the source Delta table. This is analogous to the batch or microbatch (unbounded) processing with Apache Spark. While the behavior of these two processing modes differs, the

configuration parameters differ only slightly. We'll begin by looking at the bounded source and conclude with the continuous source, as there are more configuration options to cover in the latter.

Bounded mode

To create the `DeltaSource` object, we'll be using the static `forBoundedRowData` method from the `DeltaSource` class. This builder takes the path to the Delta table and an instance of the application's Hadoop configuration, as shown in Example 4-1.

Example 4-1. Creating the `DeltaSource` bounded builder

```
% Path sourceTable = new Path("s3://bucket/delta/table_name")
  Configuration hadoopConf = new Configuration()
  var builder: RowDataBoundedDeltaSourceBuilder = DeltaSource.forBoundedRowData(
    sourceTable
    hadoopConf);
```

The object returned in Example 4-1 is a builder. Using the various options on the builder, we specify how we'd like to read from the Delta table, including options to slow down the read rates, filter the set of columns read, and more.

Builder options. The following options can be applied directly to the builder:

`columnNames` *(string ...)*
: This option provides us with the ability to specify the column names on a table we'd like to read while ignoring the rest. This functionality is especially useful on wide tables with many columns and can help alleviate unnecessary memory pressure for columns that will go unused anyway:

```
% builder.columnNames("event_time", "event_type", "brand", "price");
  builder.columnNames(
      Arrays.asList("event_time", "event_type", "brand", "price"));
```

`startingVersion` *(long)*
: This option provides us with the ability to specify the exact version of the Delta table's transaction to start reading from (in the form of a numeric `Long`). This option and the `startingTimestamp` option are mutually exclusive, as both provide a means of supplying a cursor (or transactional starting point) on the Delta table:

```
% builder.startingVersion(100L);
```

`startingTimestamp` *(string)*
: This option provides the ability to specify an approximate timestamp to begin reading from in the form of an ISO-8601 `string`. This option will trigger a scan of the Delta transaction history looking for a matching version of the table that was generated at or after the given timestamp. In the case where the entire table is newer than the timestamp provided, the table will be fully read:

```
% builder.startingTimestamp("2023-09-10T09:55:00.001Z");
```

The timestamp string can represent time with low precision—for example, as a simple date like `"2023-09-10"`—or with millisecond precision, as in the previous example. In either case, the operation will result in the Delta table being read from a specific point in table time.

`parquetBatchSize` *(int)*
: This option takes an integer controlling how many rows to return per internal batch, or generated split within the Flink engine:

```
% builder.option("parquetBatchSize", 5000);
```

Generating the bounded source. Once we finish supplying the options to the builder, we generate the `DeltaSource` instance by calling `build`:

```
% final DeltaSource<RowData> source = builder.build();
```

With the bounded source built, we can now read batches of our Delta Lake records off our tables—but what if we wanted to continuously process new records as they arrived? In that case, we can just use the continuous mode builder!

Continuous mode

To create this variation of the DeltaSource object, we'll use the static `forContinuousRowData` method on the `DeltaSource` class. The builder is shown in Example 4-2, and we provide the same base parameters as were provided to the `forBoundedRowData` builder, which makes switching from batch to streaming super simple.

Example 4-2. Creating the `DeltaSource` continuous builder

```
% var builder = DeltaSource.forContinuousRowData(
    sourceTable,
    hadoopConf);
```

The object returned in Example 4-2 is an instance of the `RowDataContinuousDeltaSourceBuilder`, and just like the bounded variant, it enables us to provide options for controlling the initial read position within the Delta table based on the `startingVersion` or `startingTimestamp`, as well as some additional options that control the frequency with which Flink will check the table for new entries.

Builder options. The following options can be applied directly to the continuous builder; additionally, all the options of the bounded builder (`columnNames`, `startingVersion`, `parquetBatchSize`, and `startingTimestamp`) apply to the continuous builder as well:

`updateCheckIntervalMillis` *(long)*
: This option takes a numeric `Long` value representing the frequency to check for updates to the Delta table, with a default value of 5,000 milliseconds:

```
% builder.updateCheckIntervalMillis(60000L);
```

If we know the table we are reading from is updated only periodically, then we can essentially reduce unnecessary I/O by using this setting. For example, if we know that new data will only ever be written on a one-minute cadence, then we can take a breather and set the frequency to check every minute. We can always modify this setting if there is a need to process faster, or slower, based on the behavior of the upstream Delta table.

`ignoreDeletes` *(boolean)*
: Setting this option allows us to ignore deleted rows. It is possible that your streaming application will never need to know that data from the past has been removed. If we are processing data in real time and considering the feed of data from our tables as append-only, then we are focused on the head of the table and can safely ignore the tail changes as data ages out.

`ignoreChanges` *(boolean)*
: Setting this option allows us to ignore changes to the table that occur upstream, including deleted rows, and other modifications to physical table data or logical table metadata. Unless the table is overwritten with a new schema, then we can continue to process while ignoring modifications to the table structure.

Generating the continuous source. Once we finish configuring the builder, we generate the `DeltaSource` instance by calling build:

```
% final DeltaSource<RowData> source = builder.build();
```

We have looked at how to build the `DeltaSource` object and have seen the connector configuration options, but what about table schema or partition column discovery? Luckily, there is no need to go into too much detail about those, since both are automatically discovered using the table metadata.

Table schema discovery

The Flink connector uses the Delta table metadata to resolve all columns and their types. For example, if we don't specify any columns in our source definition, all columns from the underlying Delta table will be read. However, If we specify a

collection of column names using the `DeltaSource` builder method (`columnNames`), then only that subset of columns will be read from the underlying Delta table. In both cases, the `DeltaSource` connector will discover the Delta table column types and convert them to the corresponding Flink types. This process of conversion from the internal Delta table data (Parquet rows) to the external data representation (Java types) provides us with a seamless way to work with our datasets.

Using the DeltaSource

After building the `DeltaSource` object (bounded or unbounded), we can now add the source into the streaming graph of our `DataStream` using an instance of the `StreamingExecutionEnvironment`.

Example 4-3 creates a simple execution environment instance and adds the source of our stream (`DeltaSource`) using `fromSource`. When we build the `StreamExecutionEnvironment` instance, we provide a `WatermarkStrategy`. Watermarks in Flink are similar in concept to watermarks for Spark Structured Streaming: they enable late-arriving data to be honored for a specific amount of time before they are considered too late to process and therefore dropped (ignored) for a given application.

Example 4-3. Creating the `StreamExecutionEnvironment` for our DeltaSource

```
% final StreamExecutionEnvironment env =
      StreamExecutionEnvironment.getExecutionEnvironment();
  env.setRuntimeMode(RuntimeExecutionMode.AUTOMATIC);
  env.enableCheckpointing(2000, CheckpointingMode.EXACTLY_ONCE);

  DeltaSource<RowData> source = ...
  env.fromSource(source, WatermarkStrategy.noWatermarks(), "delta table source")
```

We now have a live data source for our Flink job supporting Delta. We can choose to add additional sources, join and transform our data, and even write the results of our transforms back to Delta using the `DeltaSink`, or anywhere else our application requires us to go.

Next, we'll look at using the `DeltaSink` and then connect the dots with a full end-to-end example.

DeltaSink API

The `DeltaSink` API provides a static builder to egress to Delta Lake easily. Following the same pattern as the `DeltaSource` API, the `DeltaSink` API provides a builder class. Construction of the builder is shown in Example 4-4.

Example 4-4. Creating the `DeltaSink` builder

```
% Path deltaTable = new Path("s3://bucket/delta/table_name")
  Configuration hadoopConf = new Configuration()
  RowType rowType = …

  RowDataDeltaSinkBuilder sinkBuilder = DeltaSink.forRowData(
    sourceTable,
    hadoopConf,
    rowType);
```

The builder pattern for the `delta-flink` connector should already feel familiar at this point. The only difference with crafting this builder is the addition of the `RowType` reference.

RowType

Similar to the `StructType` from Spark, the `RowType` stores the logical type information for the fields within a given logical row. At a higher level, we can think about this in terms of a simple DataFrame. It is an abstraction that makes working with dynamic data simpler.

More practically, if we have a reference to the source, or transformation, that occurred prior to the `DeltaSink` in our `DataStream`, then we can dynamically provide the `RowType` using a simple trick. Through some casting tricks, we can apply a conversion between `TypeInformation<T>` and `RowData<T>`, as seen in Example 4-5.

Example 4-5. Extracting the `RowType` via `TypeInformation`

```
% public RowType getRowType(TypeInformation<RowData> typeInfo) {
  InternalTypeInfo<RowData> sourceType = (InternalTypeInfo<RowData>) typeInfo;
  return (RowType) sourceType.toLogicalType();
}
```

The `getRowType` method converts the provided `typeInfo` object into `InternalTypeInfo` and uses `toLogicalType`, which can be cast back to a `RowType`. In Example 4-6 we see how to use this method to gain an understanding of the power of Flink's `RowData`.

Example 4-6. Extracting the `RowType` from our `DeltaSource`

```
% DeltaSource<RowData> source = …
                TypeInformation<RowData> typeInfo = source.getProducedType();
                RowType rowTypeForSink = getRowType(typeInfo);
```

If we have a simple streaming application, chances are we've managed to get along nicely for a while without spending a lot of time manually crafting plain old Java objects (POJOs) and working with serializers and deserializers; or maybe we've

decided to use alternative mechanisms for creating our data objects, such as Avro or Protocol Buffers. It's also possible that we've never had to work with data outside of traditional database tables. No matter what the use case, working with columnar data means we have the luxury of simply reading the columns we want in the same way that we would with a SQL query.

Take the following SQL statement:

```
% select name, age, country from users;
```

While we could read all columns on a table using `select *`, it is always better to take only what we need from a table. This is the beauty of columnar-oriented data. Given the high likelihood that our data application won't need everything, we save compute cycles and memory overhead and provide a clean interface between the data sources we read from.

The ability to dynamically read and select specific columns—known as *SQL projection*—via our Delta Lake table means we can trust in the table's schema, which is not something we could always say of just any data living in the data lake. While a table schema can and will change over time, we won't need to maintain a separate POJO to represent our source table. This might not seem like a large lift, but the lower the number of moving parts, the simpler it is to write, release, and maintain our data applications. We only need to express the columns we expect to have, which speeds up our ability to create flexible data processing applications, as long as we can trust that the Delta tables we read from use backward compatible schema evolution. See Chapter 5 for more information on schema evolution.

Builder options

The following options can be applied directly to the builder:

`withPartitionColumns` *(string ...)*
: This builder option takes an array of strings that represent the subset of columns. The columns must exist physically in the stream.

`withMergeSchema` *(boolean)*
: This builder option must be set to *true* in order to opt into automatic schema evolution. The default value is false.

In addition to discussing the builder options, it is worth covering the semantics of exactly-once writes using the `delta-flink` connector.

Exactly-once guarantees

The `DeltaSink` does not immediately write to the Delta table. Rather, rows are appended to `flink.streaming.sink.filesystem.DeltaPendingFile`—not to be confused with Delta Lake—as these files provide a mechanism to buffer writes

(deltas) to the filesystem as a series of accumulated changes that can be committed together. The pending files remain open for writing until the checkpoint interval is met (Example 4-7 shows how we set the checkpoint interval for our Flink applications), and the pending files are rolled over, which is the point at which the buffered records will be committed to the Delta log. We specify the write frequency to Delta Lake using the interval supplied when we enable checkpointing on our `DataStream` object.

Example 4-7. Setting the checkpoint interval and mode

```
% StreamExecutionEnvironment
  .getExecutionEnvironment()
  .enableCheckpointing(2000, CheckpointingMode.EXACTLY_ONCE);
```

Using the checkpoint config above, we'd create a new transaction every two seconds at most, at which point the `DeltaSink` would use our Flink application `appId` and the `checkpointId` associated with the pending files. This is similar to the use of `txnAppId` and `txnVersion` for idempotent writes and will likely be unified in the future.

End-to-End Example

Now we'll look at an end-to-end example that uses the Flink DataStream API to read from Kafka and write to Delta. The application source code and Docker-compatible environment are provided in the book's GitHub repository under */ch04/flink/*, including steps to initialize the `ecomm.v1.clickstream` Kafka topic, write (produce) records to be consumed by the Flink application, and ultimately write those records into Delta. The results of running the application can be seen in Figure 4-1, which shows the Flink UI and represents the end state of the application.

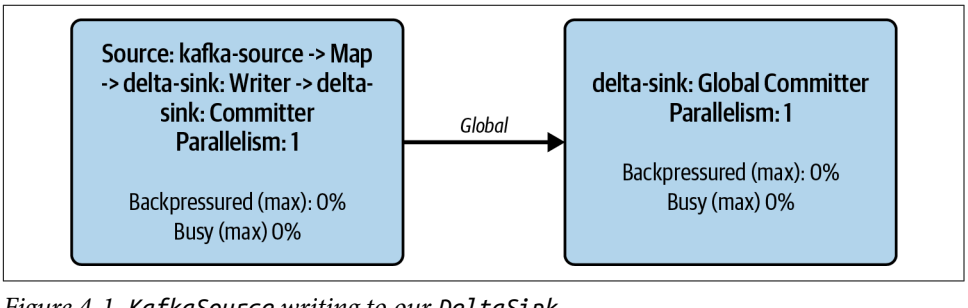


Figure 4-1. `KafkaSource` writing to our `DeltaSink`

Let's define our `DataStream` using the `KafkaSource` connector and the `DeltaSink` from earlier in this section within the scope of Example 4-8.

Example 4-8. `KafkaSource` to `DeltaSink DataStream`

```
% public DataStreamSink<RowData> createDataStream(
    StreamExecutionEnvironment env) throws IOException {

    final KafkaSource<Ecommerce> source = this.getKafkaSource();
    final DeltaSink<RowData> sink =
       this.getDeltaSink(Ecommerce.ECOMMERCE_ROW_TYPE);

    final DataStreamSource<Ecommerce> stream = env
       .fromSource(source, WatermarkStrategy.noWatermarks(), "kafka-source");

    return stream
           .map((MapFunction<Ecommerce, RowData>) Ecommerce::convertToRowData)
           .setParallelism(1)
           .sinkTo(sink)
           .name("delta-sink")
           .setDescription("writes to Delta Lake")
           .setParallelism(1);
  }
```

The example takes binary data from Kafka representing ecommerce transactions in JSON format. Behind the scenes, we deserialize the JSON data into ecommerce rows and then transform from the JVM object into the internal `RowData` representation required for writing to our Delta table. Then we simply use an instance of the `DeltaSink` to provide a terminal point for our `DataStream`.

Next, we call `execute` after adding some additional descriptive metadata to the resulting `DataStreamSink`, as we'll see in Example 4-9.

Example 4-9. Running the end-to-end example

```
% public void run() throws Exception {
    StreamExecutionEnvironment env = this.getExecutionEnvironment();
    DataStreamSink<RowData> sink = createDataStream(env);
    sink
      .name("delta-sink")
      .setParallelism(NUM_SINKS)
      .setDescription("writes to Delta Lake");

    env.execute("kafka-to-delta-sink-job");
  }
```

We've just scratched the surface on how to use the Flink connector for Delta Lake, and it is already time to take a look at another connector.

To run the full end-to-end application, just follow the step-by-step overview in the book's GitHub repository under *ch04/flink/README.md*.

In a similar vein as our end-to-end example with Flink, we'll next be exploring how to ingest the same ecommerce data from Kafka; however, this time we'll be using the Rust-based *kafka-delta-ingest* library.

Kafka Delta Ingest

The connector name sums up exactly what this little powerful library does. It reads a stream of records from a Kafka topic, optionally transforms each record (the data stream)—for example, from raw bytes to the deserialized JSON or Avro payload—and then writes the data into a Delta table. Behind the scenes, a minimal amount of user-provided configuration helps mold the connector to fulfill each specific use case. Due to the simplicity of the `kafka-delta-ingest` client, we reduce the level of effort required for one of the most critical phases of the data engineering life cycle—initial data ingestion into the *lakehouse* via Delta Lake.

Apache Kafka in a Nutshell

While Kafka has been around in the open source community since 2011, it is worth mentioning the basics before diving into the ingestion library. Feel free to skip this sidebar if you already are familiar with the basic Kafka components and architecture and just want to understand how to get the connector to work for you.

Kafka is a distributed event store and stream processing framework that provides a unified, high-throughput, low-latency platform for handling real-time data feeds.

Rather than being composed of *tables*, the Kafka architecture is built upon the notion of *topics*. In a similar fashion to our Delta tables, each topic has the ability to scale in an unbounded way (at the cost of storage space and cluster utilization). Each Kafka topic is partitioned between multiple *brokers* within a *cluster*, and each cluster can scale to meet the needs of the constituent topics contained within.

The real icing on the distributed cake is that Kafka is ultra reliable through simple configurations enabling high-availability and fault-tolerant topics through the use of what are called *in-sync replicas* (ISRs). Each replica stores a complete copy of one or more partitions within each unique Kafka topic, so if the broker is wiped out (e.g., it goes offline or becomes unavailable via network partitioning), the Kafka topic can delegate another broker to take over as the lead in the cluster, and a new broker can step up to receive an additional copy of the entire topic or a select number of partitions (ISRs). In this way, we can guarantee that the data flowing through a given