Bitcoin: A Peer-to-Peer Electronic Cash System

Satoshi Nakamoto
satoshin@gmx.com
www.bitcoin.org

Abstract. A purely peer-to-peer version of electronic cash would allow online payments to be sent directly from one party to another without going through a financial institution. Digital signatures provide part of the solution, but the main benefits are lost if a trusted third party is still required to prevent double-spending. We propose a solution to the double-spending problem using a peer-to-peer network. The network timestamps transactions by hashing them into an ongoing chain of hash-based proof-of-work, forming a record that cannot be changed without redoing the proof-of-work. The longest chain not only serves as proof of the sequence of events witnessed, but proof that it came from the largest pool of CPU power. As long as a majority of CPU power is controlled by nodes that are not cooperating to attack the network, they'll generate the longest chain and outpace attackers. The network itself requires minimal structure. Messages are broadcast on a best effort basis, and nodes can leave and rejoin the network at will, accepting the longest proof-of-work chain as proof of what happened while they were gone.

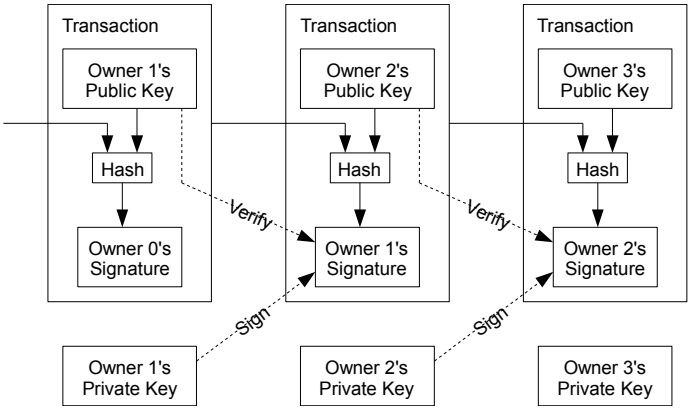
1. Introduction

Commerce on the Internet has come to rely almost exclusively on financial institutions serving as trusted third parties to process electronic payments. While the system works well enough for most transactions, it still suffers from the inherent weaknesses of the trust based model. Completely non-reversible transactions are not really possible, since financial institutions cannot avoid mediating disputes. The cost of mediation increases transaction costs, limiting the minimum practical transaction size and cutting off the possibility for small casual transactions, and there is a broader cost in the loss of ability to make non-reversible payments for non-reversible services. With the possibility of reversal, the need for trust spreads. Merchants must be wary of their customers, hassling them for more information than they would otherwise need. A certain percentage of fraud is accepted as unavoidable. These costs and payment uncertainties can be avoided in person by using physical currency, but no mechanism exists to make payments over a communications channel without a trusted party.

What is needed is an electronic payment system based on cryptographic proof instead of trust, allowing any two willing parties to transact directly with each other without the need for a trusted third party. Transactions that are computationally impractical to reverse would protect sellers from fraud, and routine escrow mechanisms could easily be implemented to protect buyers. In this paper, we propose a solution to the double-spending problem using a peer-to-peer distributed timestamp server to generate computational proof of the chronological order of transactions. The system is secure as long as honest nodes collectively control more CPU power than any cooperating group of attacker nodes.

2. Transactions

We define an electronic coin as a chain of digital signatures. Each owner transfers the coin to the next by digitally signing a hash of the previous transaction and the public key of the next owner and adding these to the end of the coin. A payee can verify the signatures to verify the chain of ownership.

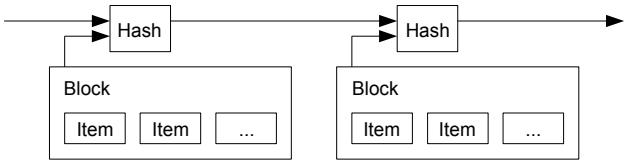


The problem of course is the payee can't verify that one of the owners did not double-spend the coin. A common solution is to introduce a trusted central authority, or mint, that checks every transaction for double spending. After each transaction, the coin must be returned to the mint to issue a new coin, and only coins issued directly from the mint are trusted not to be double-spent. The problem with this solution is that the fate of the entire money system depends on the company running the mint, with every transaction having to go through them, just like a bank.

We need a way for the payee to know that the previous owners did not sign any earlier transactions. For our purposes, the earliest transaction is the one that counts, so we don't care about later attempts to double-spend. The only way to confirm the absence of a transaction is to be aware of all transactions. In the mint based model, the mint was aware of all transactions and decided which arrived first. To accomplish this without a trusted party, transactions must be publicly announced [1], and we need a system for participants to agree on a single history of the order in which they were received. The payee needs proof that at the time of each transaction, the majority of nodes agreed it was the first received.

3. Timestamp Server

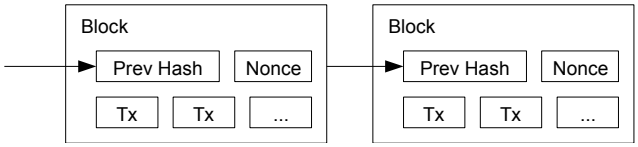
The solution we propose begins with a timestamp server. A timestamp server works by taking a hash of a block of items to be timestamped and widely publishing the hash, such as in a newspaper or Usenet post [2-5]. The timestamp proves that the data must have existed at the time, obviously, in order to get into the hash. Each timestamp includes the previous timestamp in its hash, forming a chain, with each additional timestamp reinforcing the ones before it.



4. Proof-of-Work

To implement a distributed timestamp server on a peer-to-peer basis, we will need to use a proof-of-work system similar to Adam Back's Hashcash [6], rather than newspaper or Usenet posts. The proof-of-work involves scanning for a value that when hashed, such as with SHA-256, the hash begins with a number of zero bits. The average work required is exponential in the number of zero bits required and can be verified by executing a single hash.

For our timestamp network, we implement the proof-of-work by incrementing a nonce in the block until a value is found that gives the block's hash the required zero bits. Once the CPU effort has been expended to make it satisfy the proof-of-work, the block cannot be changed without redoing the work. As later blocks are chained after it, the work to change the block would include redoing all the blocks after it.



The proof-of-work also solves the problem of determining representation in majority decision making. If the majority were based on one-IP-address-one-vote, it could be subverted by anyone able to allocate many IPs. Proof-of-work is essentially one-CPU-one-vote. The majority decision is represented by the longest chain, which has the greatest proof-of-work effort invested in it. If a majority of CPU power is controlled by honest nodes, the honest chain will grow the fastest and outpace any competing chains. To modify a past block, an attacker would have to redo the proof-of-work of the block and all blocks after it and then catch up with and surpass the work of the honest nodes. We will show later that the probability of a slower attacker catching up diminishes exponentially as subsequent blocks are added.

To compensate for increasing hardware speed and varying interest in running nodes over time, the proof-of-work difficulty is determined by a moving average targeting an average number of blocks per hour. If they're generated too fast, the difficulty increases.

5. Network

The steps to run the network are as follows:

1) New transactions are broadcast to all nodes.
2) Each node collects new transactions into a block.
3) Each node works on finding a difficult proof-of-work for its block.
4) When a node finds a proof-of-work, it broadcasts the block to all nodes.
5) Nodes accept the block only if all transactions in it are valid and not already spent.
6) Nodes express their acceptance of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash.

Nodes always consider the longest chain to be the correct one and will keep working on extending it. If two nodes broadcast different versions of the next block simultaneously, some nodes may receive one or the other first. In that case, they work on the first one they received, but save the other branch in case it becomes longer. The tie will be broken when the next proof-of-work is found and one branch becomes longer; the nodes that were working on the other branch will then switch to the longer one.

New transaction broadcasts do not necessarily need to reach all nodes. As long as they reach many nodes, they will get into a block before long. Block broadcasts are also tolerant of dropped messages. If a node does not receive a block, it will request it when it receives the next block and realizes it missed one.

6. Incentive

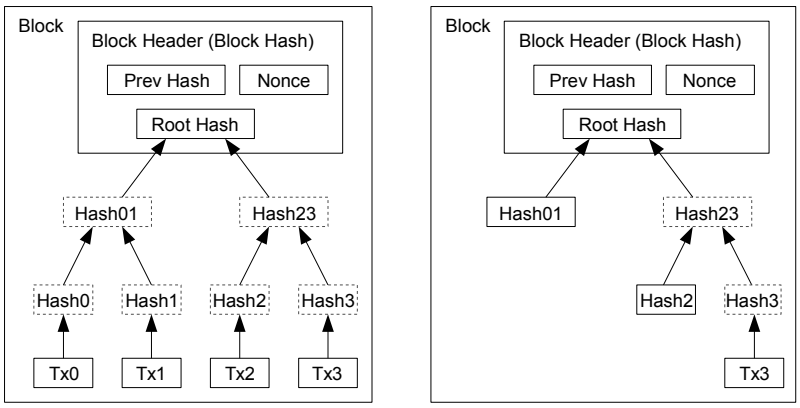
By convention, the first transaction in a block is a special transaction that starts a new coin owned by the creator of the block. This adds an incentive for nodes to support the network, and provides a way to initially distribute coins into circulation, since there is no central authority to issue them. The steady addition of a constant of amount of new coins is analogous to gold miners expending resources to add gold to circulation. In our case, it is CPU time and electricity that is expended.

The incentive can also be funded with transaction fees. If the output value of a transaction is less than its input value, the difference is a transaction fee that is added to the incentive value of the block containing the transaction. Once a predetermined number of coins have entered circulation, the incentive can transition entirely to transaction fees and be completely inflation free.

The incentive may help encourage nodes to stay honest. If a greedy attacker is able to assemble more CPU power than all the honest nodes, he would have to choose between using it to defraud people by stealing back his payments, or using it to generate new coins. He ought to find it more profitable to play by the rules, such rules that favour him with more new coins than everyone else combined, than to undermine the system and the validity of his own wealth.

7. Reclaiming Disk Space

Once the latest transaction in a coin is buried under enough blocks, the spent transactions before it can be discarded to save disk space. To facilitate this without breaking the block's hash, transactions are hashed in a Merkle Tree [7][2][5], with only the root included in the block's hash. Old blocks can then be compacted by stubbing off branches of the tree. The interior hashes do not need to be stored.



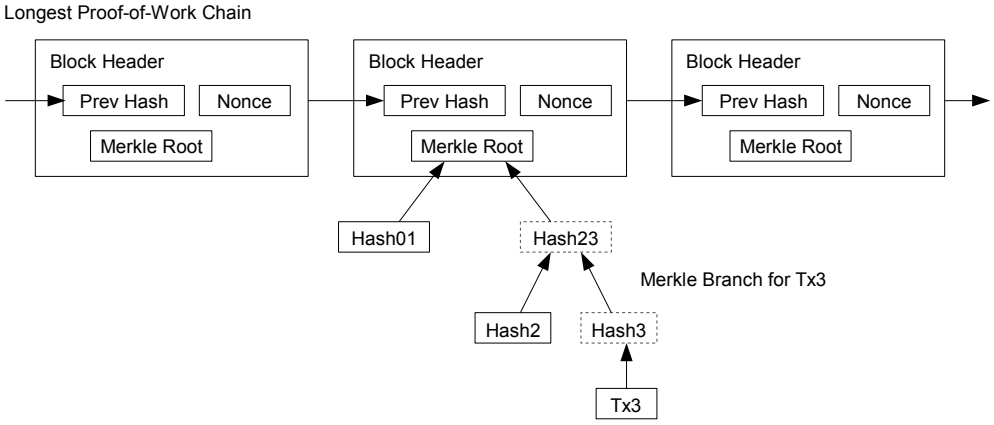
Transactions Hashed in a Merkle Tree

After Pruning Tx0-2 from the Block

A block header with no transactions would be about 80 bytes. If we suppose blocks are generated every 10 minutes, 80 bytes * 6 * 24 * 365 = 4.2MB per year. With computer systems typically selling with 2GB of RAM as of 2008, and Moore's Law predicting current growth of 1.2GB per year, storage should not be a problem even if the block headers must be kept in memory.

8. Simplified Payment Verification

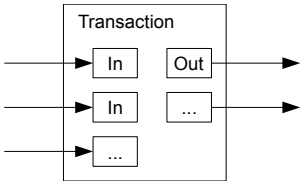
It is possible to verify payments without running a full network node. A user only needs to keep a copy of the block headers of the longest proof-of-work chain, which he can get by querying network nodes until he's convinced he has the longest chain, and obtain the Merkle branch linking the transaction to the block it's timestamped in. He can't check the transaction for himself, but by linking it to a place in the chain, he can see that a network node has accepted it, and blocks added after it further confirm the network has accepted it.



As such, the verification is reliable as long as honest nodes control the network, but is more vulnerable if the network is overpowered by an attacker. While network nodes can verify transactions for themselves, the simplified method can be fooled by an attacker's fabricated transactions for as long as the attacker can continue to overpower the network. One strategy to protect against this would be to accept alerts from network nodes when they detect an invalid block, prompting the user's software to download the full block and alerted transactions to confirm the inconsistency. Businesses that receive frequent payments will probably still want to run their own nodes for more independent security and quicker verification.

9. Combining and Splitting Value

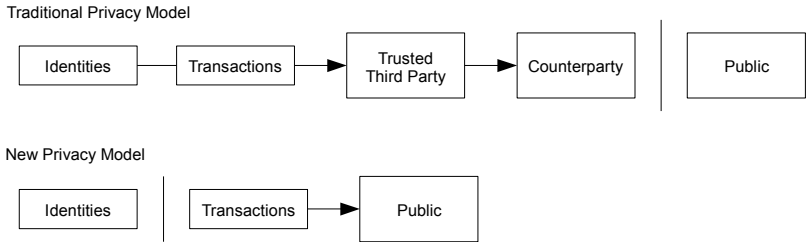
Although it would be possible to handle coins individually, it would be unwieldy to make a separate transaction for every cent in a transfer. To allow value to be split and combined, transactions contain multiple inputs and outputs. Normally there will be either a single input from a larger previous transaction or multiple inputs combining smaller amounts, and at most two outputs: one for the payment, and one returning the change, if any, back to the sender.



It should be noted that fan-out, where a transaction depends on several transactions, and those transactions depend on many more, is not a problem here. There is never the need to extract a complete standalone copy of a transaction's history.

10. Privacy

The traditional banking model achieves a level of privacy by limiting access to information to the parties involved and the trusted third party. The necessity to announce all transactions publicly precludes this method, but privacy can still be maintained by breaking the flow of information in another place: by keeping public keys anonymous. The public can see that someone is sending an amount to someone else, but without information linking the transaction to anyone. This is similar to the level of information released by stock exchanges, where the time and size of individual trades, the "tape", is made public, but without telling who the parties were.



As an additional firewall, a new key pair should be used for each transaction to keep them from being linked to a common owner. Some linking is still unavoidable with multi-input transactions, which necessarily reveal that their inputs were owned by the same owner. The risk is that if the owner of a key is revealed, linking could reveal other transactions that belonged to the same owner.

11. Calculations

We consider the scenario of an attacker trying to generate an alternate chain faster than the honest chain. Even if this is accomplished, it does not throw the system open to arbitrary changes, such as creating value out of thin air or taking money that never belonged to the attacker. Nodes are not going to accept an invalid transaction as payment, and honest nodes will never accept a block containing them. An attacker can only try to change one of his own transactions to take back money he recently spent.

The race between the honest chain and an attacker chain can be characterized as a Binomial Random Walk. The success event is the honest chain being extended by one block, increasing its lead by +1, and the failure event is the attacker's chain being extended by one block, reducing the gap by -1.

The probability of an attacker catching up from a given deficit is analogous to a Gambler's Ruin problem. Suppose a gambler with unlimited credit starts at a deficit and plays potentially an infinite number of trials to try to reach breakeven. We can calculate the probability he ever reaches breakeven, or that an attacker ever catches up with the honest chain, as follows [8]:

p = probability an honest node finds the next block
q = probability the attacker finds the next block
q_z = probability the attacker will ever catch up from z blocks behind

$$q_z = \begin{cases} 1 & \text{if } p \leq q \\ (q/p)^z & \text{if } p > q \end{cases}$$

Given our assumption that $p > q$, the probability drops exponentially as the number of blocks the attacker has to catch up with increases. With the odds against him, if he doesn't make a lucky lunge forward early on, his chances become vanishingly small as he falls further behind.

We now consider how long the recipient of a new transaction needs to wait before being sufficiently certain the sender can't change the transaction. We assume the sender is an attacker who wants to make the recipient believe he paid him for a while, then switch it to pay back to himself after some time has passed. The receiver will be alerted when that happens, but the sender hopes it will be too late.

The receiver generates a new key pair and gives the public key to the sender shortly before signing. This prevents the sender from preparing a chain of blocks ahead of time by working on it continuously until he is lucky enough to get far enough ahead, then executing the transaction at that moment. Once the transaction is sent, the dishonest sender starts working in secret on a parallel chain containing an alternate version of his transaction.

The recipient waits until the transaction has been added to a block and z blocks have been linked after it. He doesn't know the exact amount of progress the attacker has made, but assuming the honest blocks took the average expected time per block, the attacker's potential progress will be a Poisson distribution with expected value:

$$\lambda = z\frac{q}{p}$$

To get the probability the attacker could still catch up now, we multiply the Poisson density for each amount of progress he could have made by the probability he could catch up from that point:

$$\sum_{k=0}^{\infty} \frac{\lambda^k e^{-\lambda}}{k!} \cdot \begin{cases} (q/p)^{(z-k)} & \text{if } k \le z \\ 1 & \text{if } k > z \end{cases}$$

Rearranging to avoid summing the infinite tail of the distribution...

$$1 - \sum_{k=0}^{z} \frac{\lambda^k e^{-\lambda}}{k!} \left(1 - (q/p)^{(z-k)}\right)$$

Converting to C code...

```
#include <math.h>
double AttackerSuccessProbability(double q, int z)
{
    double p = 1.0 - q;
    double lambda = z * (q / p);
    double sum = 1.0;
    int i, k;
    for (k = 0; k <= z; k++)
    {
        double poisson = exp(-lambda);
        for (i = 1; i <= k; i++)
            poisson *= lambda / i;
        sum -= poisson * (1 - pow(q / p, z - k));
    }
    return sum;
}
```

Running some results, we can see the probability drop off exponentially with z.

```
q=0.1
z=0     P=1.0000000
z=1     P=0.2045873
z=2     P=0.0509779
z=3     P=0.0131722
z=4     P=0.0034552
z=5     P=0.0009137
z=6     P=0.0002428
z=7     P=0.0000647
z=8     P=0.0000173
z=9     P=0.0000046
z=10    P=0.0000012

q=0.3
z=0     P=1.0000000
z=5     P=0.1773523
z=10    P=0.0416605
z=15    P=0.0101008
z=20    P=0.0024804
z=25    P=0.0006132
z=30    P=0.0001522
z=35    P=0.0000379
z=40    P=0.0000095
z=45    P=0.0000024
z=50    P=0.0000006
```

Solving for P less than 0.1%...

```
P < 0.001
q=0.10   z=5
q=0.15   z=8
q=0.20   z=11
q=0.25   z=15
q=0.30   z=24
q=0.35   z=41
q=0.40   z=89
q=0.45   z=340
```

12. Conclusion

We have proposed a system for electronic transactions without relying on trust. We started with the usual framework of coins made from digital signatures, which provides strong control of ownership, but is incomplete without a way to prevent double-spending. To solve this, we proposed a peer-to-peer network using proof-of-work to record a public history of transactions that quickly becomes computationally impractical for an attacker to change if honest nodes control a majority of CPU power. The network is robust in its unstructured simplicity. Nodes work all at once with little coordination. They do not need to be identified, since messages are not routed to any particular place and only need to be delivered on a best effort basis. Nodes can leave and rejoin the network at will, accepting the proof-of-work chain as proof of what happened while they were gone. They vote with their CPU power, expressing their acceptance of valid blocks by working on extending them and rejecting invalid blocks by refusing to work on them. Any needed rules and incentives can be enforced with this consensus mechanism.

References

[1] W. Dai, "b-money," http://www.weidai.com/bmoney.txt, 1998.

[2] H. Massias, X.S. Avila, and J.-J. Quisquater, "Design of a secure timestamping service with minimal trust requirements," In *20th Symposium on Information Theory in the Benelux*, May 1999.

[3] S. Haber, W.S. Stornetta, "How to time-stamp a digital document," In *Journal of Cryptology*, vol 3, no 2, pages 99-111, 1991.

[4] D. Bayer, S. Haber, W.S. Stornetta, "Improving the efficiency and reliability of digital time-stamping," In *Sequences II: Methods in Communication, Security and Computer Science*, pages 329-334, 1993.

[5] S. Haber, W.S. Stornetta, "Secure names for bit-strings," In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pages 28-35, April 1997.

[6] A. Back, "Hashcash - a denial of service counter-measure," http://www.hashcash.org/papers/hashcash.pdf, 2002.

[7] R.C. Merkle, "Protocols for public key cryptosystems," In *Proc. 1980 Symposium on Security and Privacy*, IEEE Computer Society, pages 122-133, April 1980.

[8] W. Feller, "An introduction to probability theory and its applications," 1957.

O'REILLY®

Delta Lake The Definitive Guide

Modern Data Lakehouse Architectures with Data Lakes

Early Release
Raw & Unedited
Sponsored by
databricks

Denny Lee,
Prashanth Babu,
Tristen Wentling & Scott Haines

Delta Lake: The Definitive Guide

Modern Data Lakehouse Architectures with Data Lakes

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

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

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

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

Printed in the United States of America.

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

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

Acquisitions Editor: Aaron Black
Development Editor: Gary O'Brien
Production Editor: Gregory Hyman
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

May 2024: First Edition

Revision History for the Early Release
2023-06-22: First Release
2023-10-16: Second Release

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

978-1-098-15194-2

Table of Contents

CHAPTER 1

Installing Delta Lake

A Note for Early Release Readers

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

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

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

In this chapter, we will get you set up with Delta Lake and walk you through simple steps to get started 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 (https://go.delta.io/dockerhub) image is the best option. To skip the hassle of a local installation, try Databricks Community Edition for free, which includes the latest version of Delta Lake. Other options for using Delta Lake discussed in this chapter include the Delta Rust Python bindings, Delta Lake Rust API, and Apache Spark™.

Delta Lake Docker Image

The Delta Lake Docker is a self-contained image with 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; please follow the steps at Get Docker. Afterwards, you can either download the latest version of the Delta Lake docker from DockerHub (https://go.delta.io/dockerhub) or you can build the docker yourself by following the instructions from the Delta Lake Docker GitHub repository (https://go.delta.io/docker).

This 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: DataFusion created in 2017 and donated to the Apache Arrow project in 2019 is a very fast, extensible query engine for building high-quality data-centric systems written in Rust and uses the Apache Arrow in-memory format.
- ROAPI: ROAPI (read-only APIs) is a tool that builds on top of Apache Arrow and DataFusion, and is a no-code solution to automatically spin up read-only APIs for Delta Lake and other sources.
- 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 where control over system resources is crucial.

In this book we're using macOS. If you're running Windows you can use git bash, WSL, or any shell configured for bash commands.

Choose an Interface

We will discuss each of the following interfaces in detail and how to create and read Delta Lake tables with these interfaces.

- Python
- Jupyter Lab Notebook
- PySpark Shell

- Scala Shell
- Delta Rust API
- ROAPI

[Run Docker Container]

The bash entrypoint for all docker commands starts with the following command.

- Open a bash shell
- Run the container from the build image with a bash entrypoint using the following command

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

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

df = pd.DataFrame(range(5))                # Create Pandas DataFrame
write_deltalake("/tmp/deltars_table", df)  # Write Delta Lake table
df = pd.DataFrame(range(6, 11))            # 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 of the above code snippet should look similar to the following output:

```
## 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 file system that makes up this table. To do this, 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.

```
$ 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 are the files that contain the data you see in your Delta Lake table, while the `_delta_log` contains Delta's transaction log; we will discuss this more in a later Chapter.

JupyterLab Notebook

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

```
docker run --name delta_quickstart --rm -it -p 8888-8889:8888-8889 delta_quick-
start
```

The command will output a JupyterLab notebook URL, copy that URL and launch a browser to follow along the notebook and run each cell.

PySpark Shell

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

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

Next, launch a PySpark interactive shell session.

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

Let's run some basic commands in the shell.

```
# 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 a Delta Lake table, you can list the contents within the folder of your Delta Lake table. For example, in the previous 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 get something similar to below.

```
$ 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
```

Scala Shell

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

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

Launch a Scala interactive shell session.

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

Next, run some basic commands in the shell.

```
// 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 to verify the Delta Lake table, please refer to the PySpark Shell section.

Delta Rust API

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

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

Next, execute `examples/read_delta_table.rs` to review the Delta Lake table 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.

```
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

```
cargo run --example read_delta_datafusion
```

Running the above command should list the schema and 5 rows of the data from `covid19_nyt` Delta Lake table.

ROAPI

The rich open ecosystem around Delta Lake enables many novel utilities; one such utility is included in the quickstart container: ROAPI (read-only APIs). With ROAPI, you can spin up read-only APIs for static Delta Lake data sets without requiring a single line of code. You can query your Delta Lake table with Apache Arrow and DataFusion using ROAPI which are also pre-installed in this docker.

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

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

Start the roapi API using the following nohup command. The API calls are pushed to the `nohup.out` file.

Please note if you haven't created the `deltars_table` in your container, create it via the `deltalake for Python` option above. Alternatively you may omit the following from the command: `--table 'deltars_table=/tmp/deltars_table/,format=delta'` as well as any steps that call the `deltars_table`.

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

Open another shell and connect to the same Docker image.

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

Run the below steps in the shell launched in the previous step.

Check the schema of the two Delta Lake tables

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

The output of the above command should be along the following lines

```
{
   "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.

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

The output of the above command should be along the following lines.

```
[{"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.

```
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 above command should be along the following lines.

```
[
    {"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

The Delta Lake implementation in Rust was originally developed by Scribd to build faster and cheaper streaming data ingestion pipelines. Scribd adopted Delta Lake because of its open protocol and ecosystem, but found that Apache Spark was too heavy-weight for simple streaming data ingestion from Apache Kafka. Workloads that required zero transformation or aggregation were well suited to implementation in Rust. The initial versions of the library were developed in the open, in tandem with the kafka-delta-ingest application primarily by QP Hou, Christian Williams, and Mykhailo Osypov. The choice of Rust was a precinct one, as it allowed the project to grow dramatically after the introduction of Python bindings which exposed the Delta Lake implementation to the Python ecosystem with minimal changes. Since its creation in the Spring of 2020, the delta-rs project has had almost a hundred different contributors from almost every continent, and helped bring Delta Lake into countless projects big and small.

Various bindings available

The Rust library provides a strong foundation for other non-JVM based libraries to build 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 "deltalake" Python package has been built and tested on Python versions 3.7 and later, and offers many pre-built "wheels" for easy installation on most major operating systems and architectures.

Multiple community bindings have been developed 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 the level of investment in data tooling like 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 in a so-called "Delta Kernel", which aims to provide a native Delta library interface for connectors that abstracts away

the Delta protocol into one place. This work is still early but is expected to help consolidate support for native (e.g. C/C++) and higher level engines (e.g. Python, Node) so that everybody can benefit from the more advanced features, such as Deletion Vectors, by simply upgrading their underlying Delta Kernel versions.

Installation

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

```
pip install deltalake
```

After installation, you can follow the exact same steps as in the Delta Lake for Python section and execute the code snippet from that section.

Apache Spark with Delta Lake

Apache Spark is a robust, open-source engine designed for the processing and analysis of large-scale data sets. It's architected to be both rapid and versatile, 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.

One of Spark's distinguishing features is its multi-language support, broadening its accessibility to a diverse range of users. It allows developers to construct applications in several languages including Java, Scala, Python, R, and SQL. Furthermore, Spark incorporates numerous libraries that enable a wide array of data analysis tasks, encompassing machine learning, stream processing, and graph analytics. These attributes position Apache Spark as a preferred solution for the efficient processing of voluminous data at high velocity.

Spark is predominantly written 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. Based on data published by Databricks, both SQL and Python use has grown dramatically over the past few years as they provide a high performance starting point for many different workloads.

For a more detailed introduction to Spark, please check *Learning Spark* or *Spark: The Definitive Guide*.

Setting up Delta Lake with Apache Spark

Please follow these instructions to set up Delta Lake with Apache Spark. Steps in this section could be executed on your local machine in either of the following two ways:

Interactive execution
: Start the Spark shell (for Scala language, with `spark-shell` or for Python, with `pyspark`) with Delta Lake and run the code snippets interactively in the shell.

Run as a project
: Instead of code snippets, if you have code in multiple files, you can setup 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 of the following instructions, make sure to install the correct 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 mentioned in the official Apache Spark installation instructions here, make sure you have a valid Java version installed (8, 11, or 17) and that Java is configured correctly on your system using either the system `PATH` or `JAVA_HOME` environmental variable.

Windows users should follow the instructions in this blog, making sure to use the correct version of Apache Spark™ that is compatible with Delta Lake 2.3.0 and above.

Set up an interactive shell

To use Delta Lake interactively within the Spark SQL, Scala, or Python shell, 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 Command Line Interface (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 from Downloading Spark, either using pip or by downloading and extracting the archive and running `spark-sql` in the extracted directory.

```
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.DeltaCata-
log"
```

In the Spark SQL shell prompt, please copy and paste the following:

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

The SQL query concludes the creation of your first Delta Lake table using Spark SQL.

The data written to the above table, could be simply read back with another simple SQL query as below:

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

PySpark Shell

The PySpark Shell, also known as the PySpark Command Line Interface, is an interactive environment that facilitates engagement with Spark's API using 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 REPL (Read Eval Print Loop), 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 on the command prompt:

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

Run PySpark with the Delta Lake package and additional configurations:

```
pyspark --packages io.delta:delta-core_2.12:2.3.0 --conf "spark.sql.exten-
sions=io.delta.sql.DeltaSparkSessionExtension" --conf "spark.sql.cata-
log.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog"
```

In the PySpark shell prompt, copy paste the following:

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

The code snippet concludes the creation of your first Delta Lake table using PySpark.

The data written to the above table, could be simply read back with a simple Pyspark code snippet as below:

```
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 Command Line Interface (CLI), is an interactive platform that allows users to interact with Spark's API utilizing 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 from Downloading Spark, either using pip or by downloading and extracting the archive and running spark-shell in the extracted directory.

```
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.Del-
taCatalog"
```

In the Scala shell prompt, please copy paste the following:

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

This code snippet concludes the creation of your first Delta Lake table using Scala shell. The data written to the table can be read back with a simple PySpark code snippet as below:

```
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 available too. This could be very useful for setting up a Python project and also more importantly for unit testing. Delta Lake can be installed using the following command:

```
pip install delta-spark
```

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

```
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

Databricks provides a platform for personal use with Databricks Community Edition, which gives us a cluster of 15 GB memory which might be just enough to learn Delta Lake with the help of Notebooks and bundled Spark version.

Start by signing up for Databricks Community Edition by going to databricks.com/try.

Fill in your details on the form and click on Continue. Choose Community Edition by clicking on the link: "Get started with Community Edition" on the second page of the registration form.

After successfully creating your account, you will receive an email to verify your email address. Please complete the verification. Once you login to the Databricks Community Edition, you will view the Databricks workspace similar to Figure 1-1.

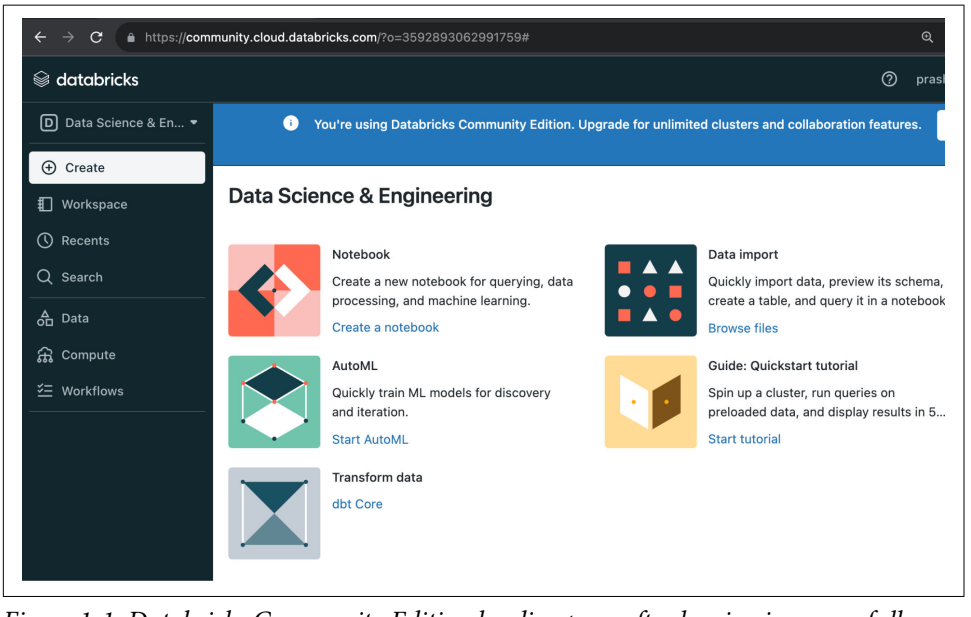


Figure 1-1. Databricks Community Edition landing page after logging in successfully

Create a Cluster with Databricks Runtime

Start by clicking on the Compute menu item on the left pane. All the clusters you create will be listed on this page. However, this is the first time you are logging into this account, so this page doesn't list any clusters yet as in Figure 1-2.

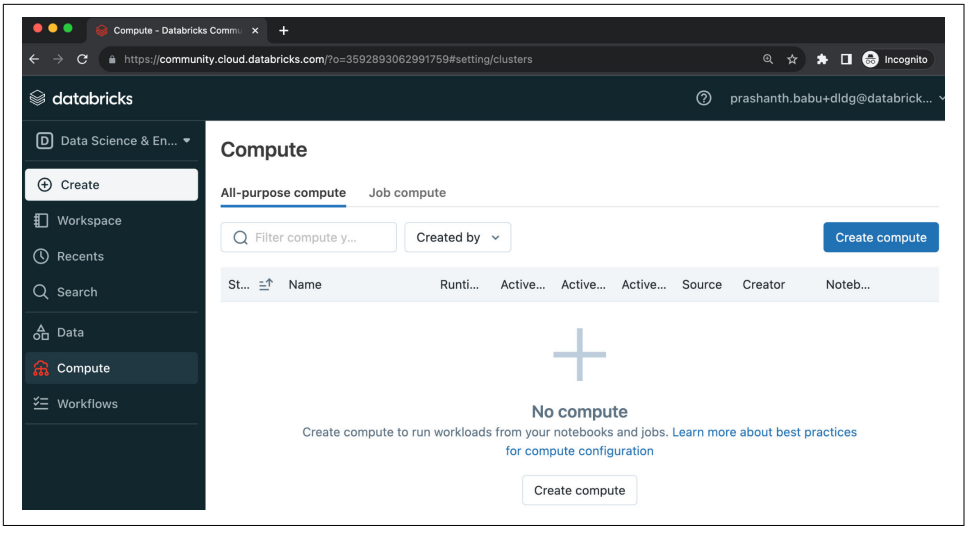


Figure 1-2. Databricks Community Edition Clusters page

On the next page, clicking on Create Compute will bring you to a New Cluster page. The Databricks Runtime 13.3 LTS is selected by default (at the time of writing). You can choose any of the latest (preferably LTS) Databricks Runtimes for running the code.

In this case, 13.3 Databricks Runtime has been chosen (Figure 1-3). For more info on Databricks Runtime releases and the compatibility matrix, please check the Databricks website. The cluster name chosen is "Delta_Lake_DLDG". Please choose any name you'd like and hit the Create Cluster button at the top to launch the cluster.

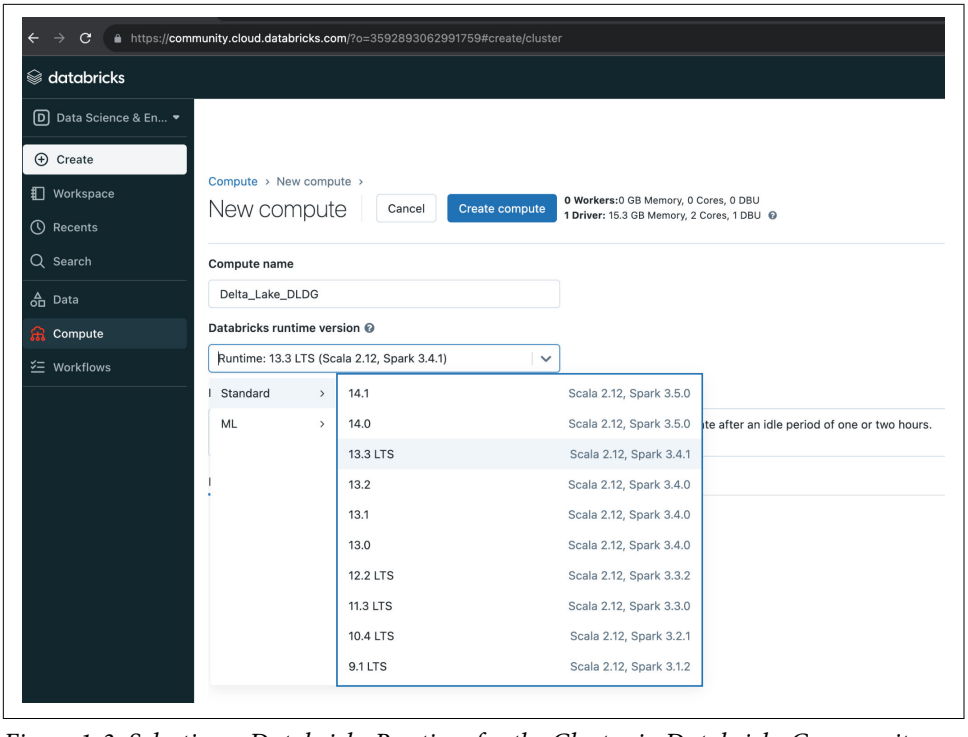


Figure 1-3. Selecting a Databricks Runtime for the Cluster in Databricks Community Edition

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

Your cluster should be up and running within a few minutes as shown in Figure 1-4.

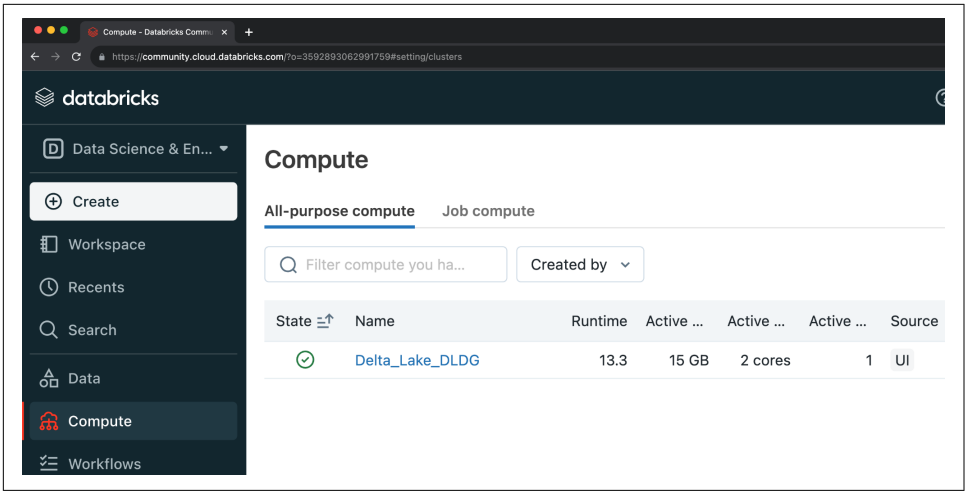


Figure 1-4. Cluster up and running

Databricks bundles Delta Lake in the Databricks Runtime, so there is no need to install Delta Lake explicitly either through pip or using Maven coordinates of the package to the cluster.

Importing notebooks

For brevity and ease of understanding, we will (re)use the Jupyter notebook we saw in the previous section on JupyterLab notebook. This notebook is available in the delta-docs GitHub repository here. Please copy the notebook link and keep it handy as we will be importing this notebook in this step.

Go to Databricks Community Edition and click on Workspace then Users and then on the downward arrow beside your email as shown in Figure 1-5.

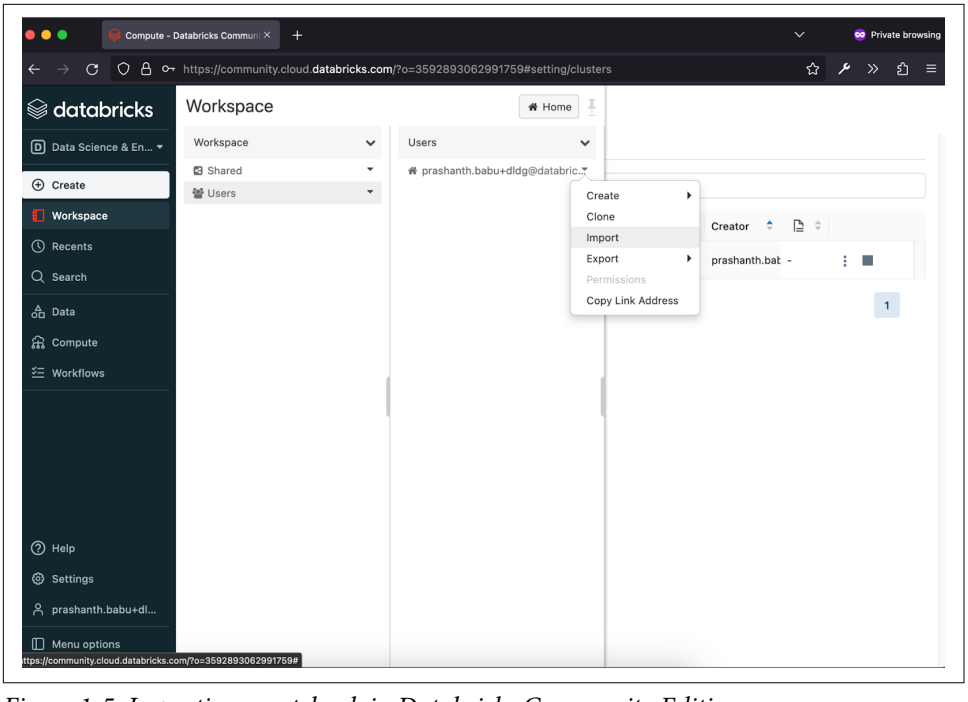


Figure 1-5. Importing a notebook in Databricks Community Edition

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

Attaching Notebooks

Now select the Cluster you created earlier to run this notebook. In this case, it is "Delta_Lake_Rocks" as shown in Figure 1-6.

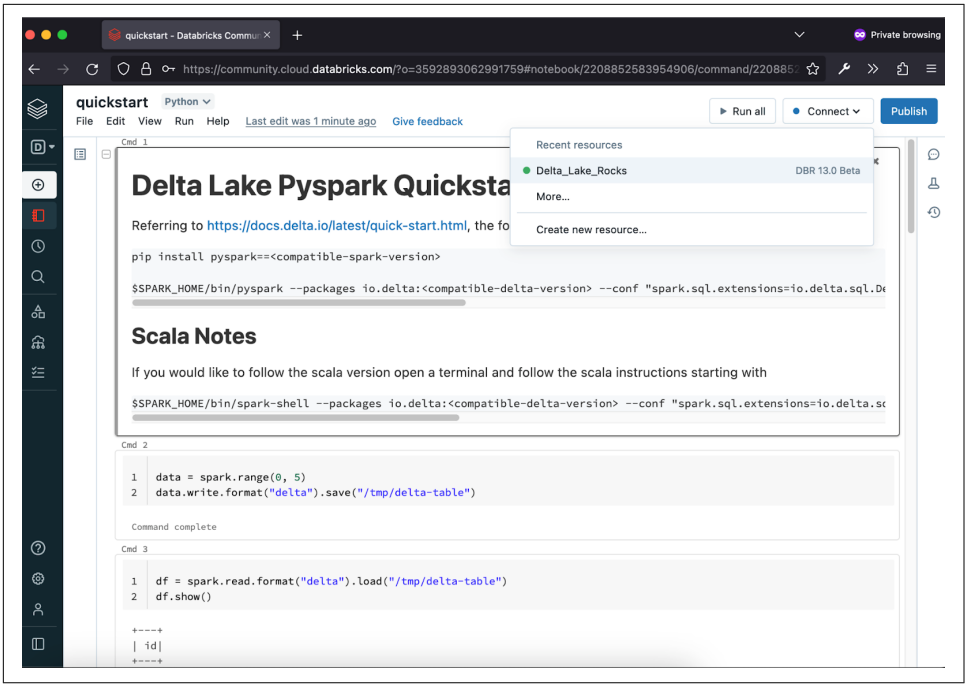


Figure 1-6. Choose the cluster you want to attach the notebook

Now you can run each cell in the notebook and press *Control* + *Enter* on your keyboard to execute the cell. When a Spark Job is running, Databricks 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.

Summary

In this chapter, we covered the various approaches you can take to get started with Delta Lake: Delta Docker, Delta Lake for Python, Apache Spark™ with Delta Lake, PySpark Declarative API and finally Databricks Community Edition. This would familiarize you with how a simple notebook or a command shell can be run easily to write to and read from Delta Lake tables.

Finally, through a very short example, we showed you how you can use any of the above approaches, how easy it is to install Delta Lake or how many different ways is Delta Lake available. We saw we could use SQL, Python, Scala, Java and Rust programming languages through the API for accessing the Delta Lake tables — which brings us to the next chapter: Using Delta Lake, where we examine various APIs in more detail on reading, writing and many other commands available.

CHAPTER 2

Maintaining your Delta Lake

> **A Note for Early Release Readers**
>
> With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the sixth chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *gobrien@oreilly.com*.

The process of keeping our Delta Lake tables running efficiently over time is akin to any kind of preventative maintenance for your vehicle or any other alternative mode of transportation (bikes, scooters, rollerblades). Like in life, we wouldn't wait for our tires to go flat before assessing the situation and finding a solution. We'd take action. In the tire use case, we'd start with simple observations, look for leaks, and ask ourselves "does the tire need to be patched?", could the problem be as simple as "adding some additional air", or is this situation more dire where we'll need to replace the whole tire. The process of assessing the situation, finding a remedy, and applying a solution can be applied to our Delta Lake tables as well and is all part of the general process of maintaining our Delta Lake tables. In essence, we just need to think in terms of *cleaning, tuning, repairing, and replacing*.

In the sections that follow, we'll learn to take advantage of the Delta Lake utility methods and learn about their associated configurations (aka table properties). We'll walk through some common methods for cleaning, tuning, repairing and replacing

our tables, in order to lend a helping hand while optimizing the performance and health of our tables, and ultimately building a firm understanding of the cause and effect relationships of the actions we take.

Using Delta Lake Table Properties

Delta Lake provides many utility functions to assist with the general maintenance (cleaning and tuning), repair, restoration, and even replacement for our critical tables; all of which are valuable skills for any data engineer. We'll begin this chapter with an introduction to some of the common maintenance-related Delta Lake table properties, and a simple exercise showcasing how to apply, modify, and remove table properties.

Table 2-1 will be referenced throughout the rest of this chapter, and whenever you need a handy reference. Each row provides the property name, internal data type, and the associated use case pertaining to cleaning, tuning, repairing, or replacing your Delta Lake tables.

The metadata stored alongside our table definitions include TBLPROPERTIES. With Delta Lake these properties are used to change the behavior of the utility methods. This makes it wickedly simple to add or remove properties, and control the behavior of your Delta Lake table.

Table 2-1. Delta Lake Table Properties Reference

Property	Data Type	Use With	Default
delta.logRetentionDuration	CalendarInterval	Cleaning	interval 30 days
delta.deletedFileRetentionDuration	CalendarInterval	Cleaning	interval 1 week
delta.setTransactionRetentionDuration	CalendarInterval	Cleaning, Repairing	(none)
delta.targetFileSize[a]	String	Tuning	(none)
delta.tuneFileSizesForRewrites[a]	Boolean	Tuning	(none)
delta.autoOptimize.optimizeWrite[a]	Boolean	Tuning	(none)
delta.autoOptimize.autoCompact[a]	Boolean	Tuning	(none)
delta.dataSkippingNumIndexedCols	Int	Tuning	32
delta.checkpoint.writeStatsAsStruct	Boolean	Tuning	(none)
delta.checkpoint.writeStatsAsJson	Boolean	Tuning	true

[a] *Properties exclusive to Databricks.*

The beauty behind using *tblproperties* is that they affect only the metadata of our tables, and in most cases don't require any changes to the physical table structure. Additionally, being able to opt-in, or opt-out, allows us to modify Delta Lake's behavior without the need to go back and change any existing pipeline code, and in most cases without needing to restart, or redeploy, our streaming applications.

The general behavior when adding or removing table properties is no different than using common data manipulation language operators (DML), which consist of insert, delete, update, and in more advanced cases, upserts, which will insert, or update a row based on a match. Chapter 12 will cover more advanced DML patterns with Delta.

Any table changes will take effect, or become visible, during the next transaction (automatically) in the case of batch, and immediately with our streaming applications.

With streaming Delta Lake applications, changes to the table, including changes to the table metadata, are treated like any ALTER TABLE command. Other changes to the table that don't modify the physical table data, like with the utility functions `vacuum` and `optimize`, can be externally updated without breaking the flow of a given streaming application.

Changes to the physical table or table metadata are treated equally, and generate a versioned record in the Delta Log. The addition of a new transaction results in the local synchronization of the DeltaSnapshot, for any out of sync (stale) processes. This is all due to the fact that Delta Lake supports multiple concurrent writers, allowing changes to occur in a decentralized (distributed) way, with central synchronization at the tables Delta Log.

There are other use cases that fall under the maintenance umbrella that require intentional action by humans and the courtesy of a heads up to downstream consumers. As we close out this chapter, we'll look at using REPLACE TABLE to add partitions. This process can break active readers of our tables, as the operation rewrites the physical layout of the Delta table.

Regardless of the processes controlled by each table property, tables at the point of creation using CREATE TABLE, or after the point of creation via ALTER TABLE, which allows us to change the properties associated with a given table.

To follow along the rest of the chapter will be using the *covid_nyt* dataset (included in the book's GitHub repo) along with the companion docker environment. To get started, execute the following.

```
$ export DLDG_DATA_DIR=~/path/to/delta-lake-definitive-guide/datasets/
$ export DLDG_CHAPTER_DIR=~/path/to/delta-lake-definitive-guide/ch6
$ docker run --rm -it \
  --name delta_quickstart \
  -v $DLDG_DATA_DIR/:/opt/spark/data/datasets \
  -v $DLDG_CHAPTER_DIR/:/opt/spark/work-dir/ch6 \
  -p 8888-8889:8888-8889 \
  delta_quickstart
```

The command will spin up the JupyterLab environment locally. Using the url provided to you in the output, open up the jupyterlab environment, and click into `ch6/chp6_notebook.ipynb` to follow along.

Create an Empty Table with Properties

We've created tables many ways throughout this book, so let's simply generate an empty table with the SQL CREATE TABLE syntax. In Example 2-1 below, we create a new table with a single date column and one default table property *delta.logRetentionDuration*. We will cover how this property is used later in the chapter.

Example 2-1. Creating a Delta Table with default table properties

```
$ spark.sql("""
    CREATE TABLE IF NOT EXISTS default.covid_nyt (
      date DATE
    ) USING DELTA
     TBLPROPERTIES('delta.logRetentionDuration'='interval 7 days');
""")
```

It is worth pointing out that the *covid_nyt* dataset has 6 columns. In the preceding example we are purposefully being lazy since we can steal the schema of the full covid_nyt table while we import it in the next step. This will teach us how to evolve the schema of the current table by filling in missing columns in the table definition.

Populate the Table

At this point, we have an empty Delta Lake table. This is essentially a promise of a table, but at this time it only contains the */{tablename}/_delta_log* directory, and an initial log entry with the schema and metadata of our empty table. If you want to run a simple test to confirm, you can run the following command to show the backing files of the table.

```
$ spark.table("default.covid_nyt").inputFiles()
```

The inputFiles command will return an empty list. That is expected but also feels a little lonely. Let's go ahead and bring some joy to this table by adding some data. We'll execute a simple read-through operation of the *covid_nyt* Parquet data directly into our managed Delta Lake table (the empty table from before).

From your active session, execute the following block of code to merge the covid_nyt dataset into the empty *default.covid_nyt* table.

The COVID-19 dataset has the `date` column represented as a `STRING`. For this exercise, we have set the `date` column to a `DATE` type, and use the `withColumn("date", to_date("date", "yyyy-MM-dd"))` in order to respect the existing data type of the table.

```
$ from pyspark.sql.functions import to_date
    (spark.read
      .format("parquet")
      .load("/opt/spark/work-dir/rs/data/COVID-19_NYT/*.parquet")
      .withColumn("date", to_date("date", "yyyy-MM-dd"))
      .write
      .format("delta")
      .saveAsTable("default.covid_nyt")
    )
```

You'll notice the operation fails to execute.

```
$ pyspark.sql.utils.AnalysisException: Table default.covid_nyt already exists
```

We just encountered an *AnalysisException*. Luckily for us, this exception is blocking us for the right reasons. In the prior code block the exception that is thrown is due to the default behavior of the DataFrameWriter in Spark which defaults to errorIfExists. This just means if the table exists, then raise an exception rather than trying to do anything that could damage the existing table.

In order to get past this speed bump, we'll need to change the write mode of the operation to `append`. This changes the behavior of our operation stating that we are intentionally adding records to an existing table.

Let's go ahead and configure the write mode as `append`.

```
(spark.read
  ...
  .write
  .format("delta")
  .mode("append")
  ...
)
```

Okay. We made it past one hurdle and are no longer being blocked by the "*table already exists*" exception, however, we were met with yet another *AnalysisException*.

```
$ pyspark.sql.utils.AnalysisException: A schema mismatch detected when writing
to the Delta table (Table ID: xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx)
```

This time the AnalysisException is thrown due to a schema mismatch. This is how the Delta protocol protects us (the operator) from blindly making changes when there is a mismatch between the expected (committed) table schema (that currently has 1 column), and our local schema (from reading the *covid_nyt* parquet) that is currently uncommitted and has 6 columns. This exception is another guardrail that

is in place to block the accidental pollution of our table schema, a process known as *schema enforcement*.

Schema Enforcement and Evolution

Delta Lake utilizes a technique from traditional data warehouses called schema-on-write. This simply means that there is a process in place to check the schema of the writer against the existing table prior to a write operation being executed. This provides a single source of truth for a table schema based on prior transactions.

Schema Enforcement
: Is the controlling process that checks an existing schema before allowing a write transaction to occur, and results in throwing an exception in the case of a mismatch.

Schema Evolution
: Is the process of intentionally modifying an existing schema in a way that enables backwards compatibility. This is traditionally accomplished using ALTER TABLE {t} ADD COLUMN(S), which is also supported in Delta Lake, along with the ability to enable the mergeSchema option on write.

Evolve the Table Schema

The last step required to add the *covid_nyt* data to our existing table, is for us to explicitly state that yes, we approve of the schema changes we are bringing to the table, and intend to commit both the actual table data and the modifications to the table schema.

```
$ (spark.read
    .format("parquet")
    .load("/opt/spark/work-dir/rs/data/COVID-19_NYT/*.parquet")
    .withColumn("date", to_date("date", "yyyy-MM-dd"))
    .write
    .format("delta")
    .mode("append")
    .option("mergeSchema", "true")
    .saveAsTable("default.covid_nyt")
  )
```

Success. We now have a table to work with, the result of executing the preceding code. As a short summary, we needed to add two modifiers to our write operation for the following reasons:

1. We updated the write mode to an *append* operation. This was necessary given we created the table in a separate transaction, and the default write mode (errorIfExists) short circuits the operation when the Delta Lake table already exists.

2. We updated the write operation to include the mergeSchema option enabling us to modify the *covid_nyt* table schema, adding the 5 additional columns required by the dataset, within the same transaction where we physically also added the nyc_taxi data.

With everything said and done, we now have actual data in our table, and we evolved the schema from the parquet-based *covid_nyt* dataset in the process.

You can take a look at the complete table metadata by executing the following `DESCRIBE` command.

```
$ spark.sql("describe extended default.covid_nyt").show(truncate=False)
```

You'll see the complete table metadata after executing the `DESCRIBE` including the columns (and comments), partitioning (in our case none), as well as all available tblproperties. Using describe is a simple way of getting to know our table, or frankly any table you'll need to work with in the future.

Alternatives to Automatic Schema Evolution

In the previous case, we used .option("mergeSchema", "true") to modify the behavior of the Delta Lake writer. While this option simplifies how we evolve our Delta Lake table schemas, it comes at the price of not being fully aware of the changes to our table schema. In the case where there are unknown columns being introduced from an upstream source, you'll want to know which columns are intended to be brought forward, and which columns can be safely ignored.

Intentionally Adding Columns with Alter Table

If we knew that we had 5 missing columns on our `default.covid_nyt` table, we could run an ALTER TABLE to add the missing columns.

```
$ spark.sql("""
  ALTER TABLE default.covid_nyt
  ADD COLUMNS (
    county STRING,
    state STRING,
    fips INT,
    cases INT,
    deaths INT
  );
  """)
```

This process may seem cumbersome given we learned how to automatically merge modifications to our table schema, but it is ultimately more expensive to rewind and undo surprise changes. With a little up front work, it isn't difficult to explicitly opt-out of automatic schema changes.

```
(spark.read
      .format("parquet")
```

```
        .load("/opt/spark/work-dir/rs/data/COVID-19_NYT/*.parquet")
        .withColumn("date", to_date("date", "yyyy-MM-dd"))
        .write
        .format("delta")
        .option("mergeSchema", "false")
        .mode("append")
        .saveAsTable("default.covid_nyt"))
    )
```

And voila. We get all the expected changes to our table intentionally, with zero surprises, which helps keep our tables clean and tidy.

Add or Modify Table Properties

The process of adding or modifying existing table properties is simple. If a property already exists, then any changes will blindly overwrite the existing property. Newly added properties will be appended to the set of table properties.

To showcase this behavior, execute the following ALTER TABLE statement in your active session.

```
$ spark.sql("""
  ALTER TABLE default.covid_nyc
  SET TBLPROPERTIES (
    'engineering.team_name'='dldg_authors',
    'engineering.slack'='delta-users.slack.com'
  )
  """)
```

This operation adds two properties to our table metadata, a pointer to the team name (`dldg_authors`) and the slack organization (`delta-users.slack.com`) for the authors of this book. Anytime we modify a table's metadata, the changes are recorded in the table history. To view the changes made to the table, including the change we just made to the table properties, we can call the history method on the DeltaTable python interface.

```
$ from delta.tables import DeltaTable
  dt = DeltaTable.forName(spark, 'default.covid_nyt')
  dt.history(10).select("version", "timestamp", "operation").show()
```

Which will output the changes made to the table.

```
+-------+-------------------+-----------------+
|version|          timestamp|        operation|
+-------+-------------------+-----------------+
|      2|2023-06-07 04:38:...|SET TBLPROPERTIES|
|      1|2023-06-07 04:14:...|            WRITE|
|      0|2023-06-07 04:13:...|     CREATE TABLE|
+-------+-------------------+-----------------+
```

To view (or confirm) the changes from the prior transaction you can call `SHOW TBLPROPERTIES` on the *covid_nyt* table.

```
$ spark.sql("show tblproperties default.covid_nyt").show(truncate=False)
```

Or you can execute the `detail()` function on the DeltaTable instance from earlier.

```
$ dt.detail().select("properties").show(truncate=False)
```

To round everything out, we'll now learn to remove unwanted table properties, then our journey can continue by learning to clean and optimize our Delta Lake tables.

Remove Table Properties

There would be no point in only being able to add table properties, so to close out the beginning of this chapter, let's look at how to use `ALTER TABLE table_name UNSET TBLPROPERTIES`.

Let's say we accidentally misspelled a property name, for example, `delta.loRgetentionDuratio`, rather than the actual property `delta.logRetentionDuration`, while this mistake isn't the end of the world, there would be no reason to keep it around.

To remove the unwanted (or misspelled) properties, we can execute `UNSET TBLPROPERTIES` on our `ALTER TABLE` command.

```
$ spark.sql("""
    ALTER TABLE default.covid_nyt
    UNSET TBLPROPERTIES('delta.loRgetentionDuratio')
  """)
```

And just like that, the unwanted property is no longer taking up space in the table properties.

We just learned to create delta lake tables using default table properties at the point of initial creation, relearned the rules of schema enforcement and how to intentionally evolve our table schemas, as well as how to add, modify, and remove properties. Next we'll explore keeping our Delta Lake tables clean and tidy.

(Spark Only) Default Table Properties

Once you become more familiar with the nuances of the various Delta Lake table properties, you can provide your own default set of properties to the SparkSession using the following spark config prefix:

```
spark.databricks.delta.properties.defaults.<conf>
```

While this only works for Spark workloads, you can probably imagine many scenarios where the ability to automatically inject properties into your pipelines could be useful.

```
spark...delta.defaults.logRetentionDuration=interval 2 weeks

spark...delta.defaults.deletedFileRetentionDuration=interval 28 days
```

Speaking of useful. Table properties can be used for storing metadata about a table owner, engineering team, communication channels (slack and email), and essentially anything else that helps to extend the utility of the descriptive table metadata, and lead to simplified data discovery and capture the owners and humans accountable for dataset ownership. As we saw earlier, the table metadata can store a wealth of information extending well beyond simple configurations.

Table 2-2 lists some example table properties that can be used to augment any Delta table. The properties are broken down into prefixes, and provide additional data catalog style information alongside your existing table properties.

Table 2-2. Using Table Properties for Data Cataloging

Property	Description
catalog.team_name	Provide the team name and answer the question "Who is accountable for the table?"
catalog.engineering.comms.slack	Provide the slack channel for the engineering team: use a permalink like https://delta-users.slack.com/archives/CG9LR6LN4 since channel names can change over time.
catalog.engineering.comms.email	*dldg_authors@gmail.com* : note this isn't a real email, but you get the point.
catalog.table.classification	Can be used to declare the type of table. Examples: pii, sensitive-pii, general, all-access, etc. These values can be used for role-based access as well. (integrations are outside of the scope of this book)

Delta Table Optimization

Remember the quote *"each action has an equal and opposite reaction"*[1]? Much like the laws of physics, changes can be felts as new data is inserted (appended), modified (updated), merged (upserted), or removed (deleted) from our Delta Lake tables (the action), the reaction in the system is to record each operation as an atomic transaction (version, timestamp, operations, and more), ensuring the table continues to serve not only its current use cases, but also ensuring it also retains enough history to allow us to rewind (time-travel) back to earlier state (point in the table's time), allowing us to fix (overwrite), or recover (replace) the table in the case that larger problems are introduced to the table.

However, before getting into the more complicated maintenance operations, let's first look at common problems that can sneak into a table over time, one of the best

1 Newton's Third Law

known of these is called the small file problem. Let's walk through the problem and solution now.

The Problem with Big Tables and Small Files

When we talk about the small file problem, we are actually talking about an issue that isn't unique to Delta Lake, but rather an issue with network IO, and a high (open-cost) for unoptimized tables consisting of way too many small files. Small files can be classified as any file under 64kb.

How can too many small files hurt us? The answer is in many different ways, but the common thread between all problems is that they sneak up over time, and require modifications to the layout of the physical files encapsulating our tables. Not recognizing when your tables begin to slow down and suffer under the weight of themselves can lead to potentially costly increases to distributed compute in order to efficiently open, and execute a query.

There is a true cost in terms of the number of operational steps required before the table is physically loaded into memory, which tends to increase over time until the point where a table can no longer be efficiently loaded.

This is felt much more in traditional Hadoop style ecosystems, like MapReduce and Spark, where the unit of distribution is bound to a task, and a file consists of "blocks" and each block takes 1 task. If we have 1 million files in a table that are 1 GB each, and a block size of 64MB, then we will need to distribute a whopping 15.65 million tasks to read the entire table. It is ideal to optimize the target file size of the physical files in our tables to reduce file system IO and network IO. When we encounter unoptimized files (the small files problem), then the performance of our tables suffer greatly because of it. For a solid example, say we had the same large table (~1 TB) but the files making up the table were evenly split at around 5kb each. This means we'd have 200k files per 1 GB, and around 200 million files to open before loading our table. In most cases the table would never open.

For fun, we are going to recreate a very real small files problem, and then figure out how to *optimize* the table. To follow along, head back to the session from earlier in the chapter, as we'll continue to use the *covid_nyt* dataset in the following examples.

Creating the Small File Problem

The covid_nyt dataset has over a million records. The total size of the table is less than 7mb split across 8 partitions which is a small dataset.

```
$ ls -lh /opt/spark/work-dir/ch6/spark-warehouse/covid_nyt/*.parquet | wc -l
8
```

What if we flipped the problem around and had 9000, or even 1 million files representing the covid_nyt dataset? While this use case is extreme, we'll learn later on in the book (chapter 9) that streaming applications are a typical culprit with respect to creating tons of tiny files!

Let's create another empty table named *default.nonoptimal_covid_nyt and run some simple commands to unoptimize the table. For starters, execute the following command.*

```
$ from delta.tables import DeltaTable
  (DeltaTable.createIfNotExists(spark)
    .tableName("default.nonoptimal_covid_nyt")
    .property("description", "table to be optimized")
    .property("catalog.team_name", "dldg_authors")
    .property("catalog.engineering.comms.slack",
      "https://delta-users.slack.com/archives/CG9LR6LN4")
    .property("catalog.engineering.comms.email","dldg_authors@gmail.com")
    .property("catalog.table.classification","all-access")
    .addColumn("date", "DATE")
    .addColumn("county", "STRING")
    .addColumn("state", "STRING")
    .addColumn("fips", "INT")
    .addColumn("cases", "INT")
    .addColumn("deaths", "INT")
    .execute())
```

Now that we have our table, we can easily create way too many small files using the normal *default.covid_nyt* table as our source. The total number of rows in the table is 1,111,930. If we repartition the table, from the existing 8, to say 9000 partitions, this will split the table into an even 9000 files at around 5kb per file.

```
$ (spark
   .table("default.covid_nyt")
   .repartition(9000)
   .write
   .format("delta")
   .mode("overwrite")
   .saveAsTable("default.nonoptimal_covid_nyt")
  )
```

If you want to view the physical table files, you can run the following command.

```
$ docker exec -it delta_quickstart bash \
-c "ls -l /opt/spark/work-dir/ch6/spark-warehouse/nonop-
timal_covid_nyt/*parquet | wc -l"
```

You'll see there are exactly 9000 files.

We now have a table we can optimize. Next we'll introduce Optimize. As a utility, consider it to be your friend. It will help you painlessly consolidate the many small files representing our table into a few larger files. All in the blink of an eye.

Using Optimize to Fix the Small File Problem

Optimize is a Delta utility function that comes in two variants: z-order and bin-packing. The default is bin-packing.

Optimize

What exactly is bin-packing? At a high-level, this is a technique that is used to coalesce many small files into fewer large files, across an arbitrary number of bins. A bin is defined as a file of a maximum file size (the default for Spark Delta Lake is 1GB, Delta Rust is 250mb).

The OPTIMIZE command can be tuned using a mixture of configurations.

For tuning the optimize thresholds, there are a few considerations to keep in mind:

- **(spark only)** `spark.databricks.delta.optimize.minFileSize` (long) is used to group files smaller than the threshold (in bytes) together before being rewritten into a larger file by the `OPTIMIZE` command.
- ***(spark only)*** `spark.databricks.delta.optimize.maxFileSize` *(long)* is used to specify the target file size produced by the `OPTIMIZE` command
- **(spark-only)** `spark.databricks.delta.optimize.repartition.enabled` *(bool) is used to change the behavior of OPTIMIZE and will use repartition(1) instead of coalesce(1) when reducing*
- **(delta-rs and non-OSS delta)** The table property `delta.targetFileSize` (string) can be used with the `delta-rs` client, but is currently not supported in the OSS delta release. Example being **250mb**.

The OPTIMIZE command is deterministic and aims to achieve an evenly distributed Delta Lake table (or specific subset of a given table).

To see optimize in action, we can execute the optimize function on the `nonoptimal_covid_nyt` table. Feel free to run the command as many times as you want, Optimize will only take effect a second time if new records are added to the table.

```
$ results_df = (DeltaTable
   .forName(spark, "default.nonoptimal_covid_nyt")
   .optimize()
   .executeCompaction())
```

The results of running the optimize operation are returned both locally in a DataFrame (`results_df`) and available via the table history as well. To view the `OPTIMIZE` stats, we can use the `history` method on our DeltaTable instance.

```
$ from pyspark.sql.functions import col
 (
  DeltaTable.forName(spark, "default.nonoptimal_covid_nyt")
  .history(10)
  .where(col("operation") == "OPTIMIZE")
  .select("version", "timestamp", "operation", "operationMetrics.numRemoved-
Files", "operationMetrics.numAddedFiles")
  .show(truncate=False))
```

The resulting output will produce the following table.

```
+-------+----------------------+---------+--------------+-------------+
|version|timestamp             |operation|numRemovedFiles|numAddedFiles|
+-------+----------------------+---------+--------------+-------------+
|2      |2023-06-07 06:47:28.488|OPTIMIZE |9000          |1            |
+-------+----------------------+---------+--------------+-------------+
```

The important column for our operation shows that we removed 9000 files (`numRemovedFiles`) and generated one compacted file (`numAddedFiles`).

For Delta Streaming and Streaming Optimizations flip ahead to chapter 9.

Z-Order Optimize

Z-ordering is a technique to colocate related information in the same set of files. The related information is the data residing in your table's columns. Consider the `covid_nyt` dataset. If we knew we wanted to quickly *calculate the death rate by state over time* then utilizing `Z-ORDER BY` would allow us to *skip* opening files in our tables that don't contain relevant information for our query. This co-locality is automatically used by the Delta Lake data-skipping algorithms. This behavior dramatically reduces the amount of data that needs to be read.

For tuning the Z-ORDER BY:

- `delta.dataSkippingNumIndexedCols` (int) is the table property responsible for reducing the number of stats columns stored in the table metadata. This defaults to 32 columns.
- delta.checkpoint.writeStatsAsStruct (bool) is the table property responsible for enabling writing of columnar stats (per transaction) as parquet data. The default

value is false as not all vendor-based Delta Lake solutions support reading the struct based stats.

Chapter 12 will cover performance tuning in more detail, so we will dip our toes in now, and cover general maintenance considerations.

Table Tuning and Management

We just covered how to optimize our tables using the `OPTIMIZE` command. In many cases, where you have a table smaller than 1 GB, it is perfectly fine to just use OPTIMIZE, however, it is common for tables to grow over time, and eventually we'll have to consider partitioning our tables as a next step for maintenance.

Partitioning your Tables

Table partitions can work for you, or oddly enough also against you, similar to the behavior we observed with the small files problem, too many partitions can create a similar problem but through directory level isolation instead. Luckily, there are some general guidelines and rules to live by that will help you manage your partitions effectively, or at least provide you with a pattern to follow when the time comes.

Table Partitioning Rules

The following rules will help you understand when to introduce partitions.

1. **If your table is smaller than 1 TB**. Don't add partitions. Just use Optimize to reduce the number of files. If bin-packing optimize isn't providing the performance boost you need, you talk with your downstream data customers and learn how they commonly query your table, you may be able to use z-order optimize and speed up their queries with data co-location.
2. **If you need to optimize how you delete?** GDPR and other data governance rules mean that table data is subject to change. More often than not, abiding by data governance rules mean that you'll need to optimize how you delete records from your tables, or even retain tables like in the case of legal hold. One simple use case is N-day delete, for example 30 day retention. Using daily partitions, while not optimal depending on the size of your Delta Lake table, can be used to simplify common delete patterns like data older than a given point in time. In the case of 30 day delete, given a table partitioned by the column datetime, you could run a simple job calling `delete from {table} where datetime < current_timestamp() - interval 30 days`.

Choose the right partition column

The following advice will help you select the correct column (or columns) to use when partitioning. The most commonly used partition column is **date**. Follow these two rules of thumb for deciding on what column to partition by:

1. **Is the cardinality of a column very high?** Do not use that column for partitioning. For example, if you partition by a column userId and if there can be 1M+ distinct user IDs, then that is a bad partitioning strategy.
2. **How much data will exist in each partition?** You can partition by a column if you expect data in that partition to be at least 1 GB.

The correct partitioning strategy may not immediately present itself, and that is okay, there is no need to optimize until you have the correct use cases (and data) in front of you.

Given the rules we just set forth, let's go through the following use cases: defining partitions on table creation, adding partitions to an existing table, and removing (deleting) partitions. This process will provide a firm understanding for using partitioning, and after all, this is required for the long-term preventative maintenance of our Delta Lake tables.

Defining Partitions on Table Creation

Let's create a new table called *default.covid_nyt_by_day* which will use the `date` column to automatically add new partitions to the table with zero intervention..

```
$ from pyspark.sql.types import DateType
  from delta.tables import DeltaTable
  (DeltaTable.createIfNotExists(spark)
    .tableName("default.covid_nyt_by_date")
    ...
    .addColumn("date", DateType(), nullable=False)
    .partitionedBy("date")
    .addColumn("county", "STRING")
    .addColumn("state", "STRING")
    .addColumn("fips", "INT")
    .addColumn("cases", "INT")
    .addColumn("deaths", "INT")
    .execute())
```

What's going on in the creation logic is almost exactly the same as the last few examples, the difference is the introduction of the `partitionBy("date")` on the DeltaTable builder. To ensure the date column is always present the DDL includes a non-nullable flag since the column is required for partitioning.

Partitioning requires the physical files representing our table to be laid out using a unique directory per partition. This means all of the physical table data must

be moved in order to honor the partition rules. Doing a migration from a non-partitioned table to a partitioned table doesn't have to be difficult, but supporting live downstream customers can be a little tricky.

As a general rule of thumb, it is always better to come up with a plan to migrate your existing data customers to the new table, in this example that would be the new partitioned table, rather than introducing a potential breaking change into the current table for any active readers.

Given the best practice at hand, we'll learn how to accomplish this next.

Migrating from a Non-Partitioned to Partitioned Table

With the table definition for our partitioned table in hand, it becomes trivial to simply read all of the data from our non-partitioned table and write the rows into our newly created table. What's even easier is that we don't need to even specify how we intend to partition since the partition strategy already exists in the table metadata.

```
$ (
     spark
     .table("default.covid_nyt")
     .write
     .format("delta")
     .mode("append")
     .option("mergeSchema", "false")
     .saveAsTable("default.covid_nyt_by_date"))
```

This process creates a fork in the road. We currently have the prior version of the table (non-partitioned) as well as the new (partitioned) table, and this means we have a copy. During a normal cut-over, you typically need to continue to dual write until your customers inform you they are ready to be fully migrated. Chapter 9 will provide you with some useful tricks for doing more intelligent incremental merges, and in order to keep both versions of the prior table in sync, using merge and incremental processing is the way to go.

Partition Metadata Management

Because Delta Lake automatically creates and manages table partitions as new data is being inserted and older data is being deleted, there is no need to manually call `ALTER TABLE table_name [ADD | DROP PARTITION] (column=value)`. This means you can focus your time elsewhere rather than manually working to keep the table metadata in sync with the state of the table itself.

Viewing Partition Metadata

To view the partition information, as well as other table metadata, we can create a new DeltaTable instance for our table and call the `detail` method. This will return

a DataFrame that can be viewed in its entirety, or filtered down to the columns you need to view.

```
$ (DeltaTable.forName(spark,"default.covid_nyt_by_date")
    .detail()
    .toJSON()
    .collect()[0]
)
```

The above command converts the resulting DataFrame into a JSON object, and then converts it into a List (using `collect()`) so we can access the JSON data directly.

```
{
  "format": "delta",
  "id": "8c57bc67-369f-4c84-a63e-38b8ac19bdf2",
  "name": "default.covid_nyt_by_date",
  "location": "file:/opt/spark/work-dir/ch6/spark-warehouse/covid_nyt_by_date",
  "createdAt": "2023-06-08T05:35:00.072Z",
  "lastModified": "2023-06-08T05:50:45.241Z",
  "partitionColumns": ["date"],
  "numFiles": 423,
  "sizeInBytes": 17660304,
  "properties": {
    "description": "table with default partitions",
    "catalog.table.classification": "all-access",
    "catalog.engineering.comms.email": "dldg_authors@gmail.com",
    "catalog.team_name": "dldg_authors",
    "catalog.engineering.comms.slack": "https://delta-users.slack.com/archives/
CG9LR6LN4"
  },
  "minReaderVersion": 1,
  "minWriterVersion": 2,
  "tableFeatures": ["appendOnly", "invariants"]
}
```

With the introduction to partitioning complete, it is time to focus on two critical techniques under the umbrella of Delta Lake table lifecycle and maintenance: repairing and replacing tables.

Repairing, Restoring, and Replacing Table Data

Let's face it. Even with the best intentions in place, we are all human and make mistakes. In your career as a data engineer, one thing you'll be required to learn is the art of data recovery. When we recover data, the process is commonly called *replaying* since the action we are taking is to rollback the clock, or rewind, to an earlier point in time. This enables us to remove problematic changes to a table, and replace the erroneous data with the "fixed" data.

Recovering and Replacing Tables

When you can recover a table the catch is that there needs to be a data source available that is in a better state than your current table. In chapter 11, we'll be learning about the Medallion Architecture, which is used to define clear quality boundaries between your raw (bronze), cleansed (silver), and curated (gold) data sets. For the purpose of this chapter, we will assume we have raw data available in our bronze database table that can be used to replace data that became corrupted in our silver database table.

Conditional Table Overwrites using ReplaceWhere

Say for example that data was accidentally deleted from our table for 2021-02-17. There are other ways to restore accidentally deleted data (which we will learn next), but in the case where data is permanently deleted, there is no reason to panic, we can take the recovery data and use a conditional overwrite.

```
$ recovery_table = spark.table("bronze.covid_nyt_by_date")
  partition_col = "date"
  partition_to_fix "2021-02-17"
  table_to_fix = "silver.covid_nyt_by_date"


  (recovery_table
    .where(col(partition_col) == partition_to_fix)
    .write
    .format("delta")
    .mode("overwrite")
    .option("replaceWhere", f"{partition_col} == {partition_to_fix}")
    .saveAsTable("silver.covid_nyt_by_date")
  )
```

The previous code showcases the replace overwrite pattern, as it can either replace missing data or overwrite the existing data conditionally in a table. This option allows you to fix tables that may have become corrupt, or to resolve issues where data was missing and has become available. The replaceWhere with insert overwrite isn't bound only to partition columns, and can be used to conditionally replace data in your tables.

It is important to ensure the `replaceWhere` condition matches the where clause of the recovery table, otherwise you may create a bigger problem and further corrupt the table you are fixing. Whenever possible, it is good to remove the chance of human error, so if you find yourself repairing (replacing or recovering) data in your tables often, it would be beneficial to create some guardrails to protect the integrity of your table.

Next, let's look at conditionally removing entire partitions.

Deleting Data and Removing Partitions

It is common to remove specific partitions from our Delta Lake tables in order to fulfill specific requests, for example when deleting data older than a specific point in time, removing abnormal data, and generally cleaning up our tables.

Regardless of the case, if our intentions are to simply clear out a given partition, we can do so using a conditional delete on a partition column. The following statement conditionally deletes partitions (tpep_dropoff_date) that are older than the January 1st, 2023.

```
(
   DeltaTable
    .forName(spark, 'default.covid_nyt_by_date')
    .delete(col("date") < "2023-01-01"))
```

Removing data, or dropping entire partitions, can both be managed using conditional deletes. When you delete based on a partition column, this is an efficient way to delete data without the processing overhead of loading the physical table data into memory, and instead uses the information contained in the table metadata, to prune partitions based on the predicate. In the case of deleting based on non-partitioned columns, the cost is higher as a partial or full table scan can occur, however whether you are removing entire partitions or conditionally removing a subset of each table, as an added bonus, if for any reason you need to change your mind, you can "undo" the operation using time travel. We will learn how to restore our tables to an earlier point in time next.

Remember to never remove delta lake table data (files) outside of the context of the delta lake operations. This can corrupt your table, and cause headaches.

The Lifecycle of a Delta Lake Table

Over time, as each Delta table is modified, older versions of the table remain on disk in order to support table restoration, or to view earlier points in the table time (time-travel), and to provide a clean experience for streaming jobs that may be reading from various points in the table (which relate to different points in time, or history across the table). This is why it is critical to ensure you have a long enough lookback window for the *delta.logRetentionDuration*, so when you run vacuum on your table, you are not immediately flooded with pages or unhappy customers of a stream of data that just disappeared.

Restoring your Table

In the case where a transaction has occurred, for example a delete from on your table that was incorrect (cause life happens), rather than reloading the data (in the case where we have a copy of the data), we can rewind and restore the table to an earlier version. This is an important capability especially given that problems can arise where the only copy of your data was in fact the data that was just deleted. When there is nowhere left to go to recover the data, you have the ability to time-travel back to an earlier version of your table.

What you'll need to restore your table is some additional information. We can get this all from the table history.

```
$ dt = DeltaTable.forName(spark, "silver.covid_nyt_by_date")
  (dt.history(10)
    .select("version", "timestamp", "operation")
    .show())
```

The prior code will show the last 10 operations on the Delta Lake table. In the case where you want to rewind to a prior version, just look for the DELETE.

```
+-------+-------------------+--------------------+
|version|          timestamp|           operation|
+-------+-------------------+--------------------+
|      1|2023-06-09 19:11:...|              DELETE|
|      0|2023-06-09 19:04:...|CREATE TABLE AS S...|
+-------+-------------------+--------------------+
```

You'll see the DELETE transaction occurred at version 1, so let's restore the table back to version 0.

```
$ dt.restoreToVersion(0)
```

All it takes to restore your table is knowledge about the operation you want to remove. In our case, we removed the DELETE transaction. Because Delta Lake delete operations occur in the table metadata, unless you run a process called VACUUM, you can safely return to the prior version of your table.

Cleaning Up

When we delete data from our Delta lake tables this action is not immediate. In fact, the operation itself simply removes the reference from the Delta Lake table snapshot so it is like the data is now invisible. This operation means that we have the ability to "undo" in the case where data is accidentally deleted. We can clean up the artifacts, the deleted files, and truely purge them from the delta lake table using a process called "vacuuming".

Vacuum

The vacuum command will clean up deleted files or versions of the table that are no longer current, which can happen when you use the overwrite method on a table. If you overwrite the table, all you are really doing is creating new pointers to new files that are referenced by the table metadata. So if you overwrite a table often, the size of the table on disk will grow exponentially. Luckily, there are some table properties that help us control the behavior of the table as changes occur over time. These rules will govern the vacuum process.

- **delta.logRetentionDuration** defaults to `interval 30 days` and keeps track of the history of the table. The more operations that occur, the more history that is retained. If you won't be using time-travel operations then you can try reducing the number of days of history down to a week.
- **delta.deletedFileRetentionDuration** defaults to `interval 1 week` and can be changed in the case where delete operations are not expected to be undone. For peace of mind, it is good to maintain at least 1 day for deleted files to be retained.

With the table properties set on our table, the vacuum command does most of the work for us. The following code example shows how to execute the vacuum operation.

```
$ (DeltaTable.forName(spark, "default.nonoptimal_covid_nyt")
   .vacuum()
```

Running `vacuum` on our table will result in all files being removed that are no longer referenced by the table snapshot, including deleted files from prior versions of the table. While vacuuming is a necessary process to reduce the cost of maintaining older versions of a given table, there is a side effect that can accidentally leave downstream data consumers (consumers) high and dry, in the case where they need to read an early version of your table. There are other issues that can arise that will be covered in chapter 9 when we tackle streaming data in and out of our Delta Lake tables.

The `vacuum` command will not run itself. When you are planning to bring your table into production, and want to automate the process of keeping the table tidy, you can setup a cron job to call vacuum on a normal cadence (daily, weekly). It is also worth pointing out that vacuum relies on the timestamps of the files, when they were written to disk, so if the entire table was imported the vacuum command will not do anything until you hit your retention thresholds.

Dropping Tables

Dropping a table is an operation with no undo. If you run a `delete from {table}` you are essentially truncating the table, and can still utilize time travel (to undo the operation), however, if you want to really remove all traces of a table please read through the following warning box and remember to plan ahead by creating a table copy (or CLONE) if you want a recovery strategy.

Dropping a table is an operation with no undo. If you run a `delete from {table}` you are essentially truncating the table, and can utilize time travel (to undo the change). if you want to truly remove all traces of your table, the chapter will conclude and show you how to do that.

Removing all traces of a Delta Lake Table

If you want to do a permanent delete and remove all traces of a managed Delta Lake table, and you understand the risks associated with what you are doing, and really do intend to forgo any possibility of table recovery, then you can drop the table using the SQL DROP TABLE syntax.

```
$ spark.sql(f"drop silver.covid_nyt_by_date")
```

You can confirm the table is gone by attempting to list the files of the Delta Lake table.

```
$ docker exec \
  -it delta_quickstart bash \
  -c "ls -l /opt/spark/work-dir/ch6/spark-warehouse/sil-
ver.db/covid_nyt_by_date/
```

Which will result in the following output. This shows that the table really no longer exists on disk.

```
ls: cannot access './spark-warehouse/sil-
ver.db/covid_nyt_by_date/': No such file or directory
```

Summary

This chapter introduced you to the common utility functions available provided within the Delta Lake project. We learned how to work with table properties, explored the more common table properties we'd most likely encounter, and how to optimize our tables to fix the small files problem. This led us to learn about partitioning and restoring and replacing data within our tables. We explored using time travel to restore our tables, and concluded the chapter with a dive into cleaning up after ourselves and lastly permanently deleting tables that are no longer necessary. While not every use case can fit cleanly into a book, we now have a great reference to the common problems and solutions required to maintain your Delta Lake tables and keep them running smoothly over time.

CHAPTER 3
Streaming in and out of your Delta Lake

A Note for Early Release Readers

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

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

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

Now more than ever the world is infused with real time data sources. From e-commerce, social network feeds, and airline flight data to network security and IoT devices, the volume of data sources is increasing while the frequency with which data becomes available for usage is rapidly diminishing. One problem with this is while some event-level operations make sense, much of the information we depend upon lives in the aggregation of that information. So, we are caught between the dueling priorities of a.) reducing the time to insights as much as possible and b.) capturing enough meaningful and actionable information from aggregates. For years we've seen processing technologies shifting in this direction and it was this environment in which Delta Lake originated. What we got from Delta Lake was an open lakehouse format that supports seamless integrations of multiple batch and stream processes while delivering the necessary features like ACID transactions and scalable metadata processing which are commonly absent in most distributed data stores. With this in mind we can dig into some of the details for stream processing with Delta Lake, namely the functionality that's core to streaming processes, configuration options,

specific usage methods, and the relationship of Delta Lake to Databricks' Delta Live Tables.

Streaming and Delta Lake

As we go along we want to cover some foundational concepts and then get into more of the nuts and bolts of actually using Delta Lake for stream processing. We'll start with an overview of concepts and some terminology after which we will take a look at a few of the stream processing frameworks we can use with Delta Lake (for a more in depth introduction to stream processing see the *Learning Spark* book). Then we'll look at the core functionality, some of the options we have available, and some common more advanced cases with Apache Spark. Then to finish it out we will cover a couple of related features used in Databricks like Delta Live Tables and how it relates to Delta Lake and then lastly review how to use the change data feed functionality available in Delta Lake.

Streaming vs Batch Processing

Data processing as a concept makes sense to us: during its lifecycle we receive data, perform various operations on it, then store and or ship it onward. So what primarily differentiates a batch data process from a streaming data process? Latency. Above all other things latency is the primary driver because these processes tend not to differ in the business logic behind their design but instead focus on message/file sizes and processing speed. The choice of which method to use is generally driven by time requirements or service level/delivery agreements that should be part of requirements gathering at the start of a project. The requirements should also consider the required amount of time to get actionable insights from the data and will drive our decision in processing methodology. One additional design choice we prefer is to use a framework that has a unified batch and streaming API because there are so few differences in the processing logic, in turn providing us flexibility should requirements change over time.

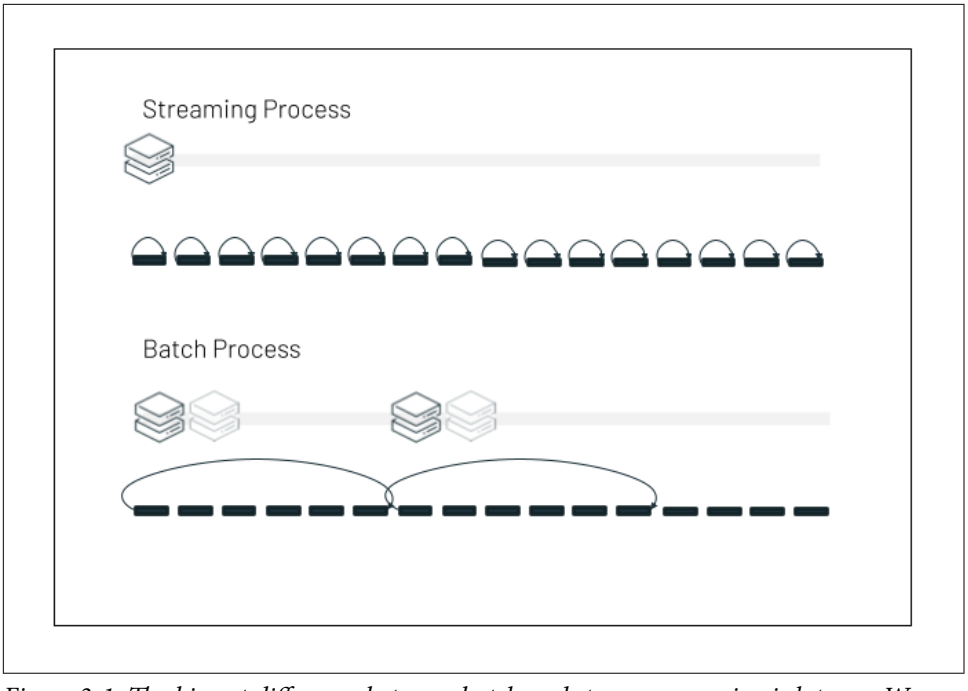


Figure 3-1. The biggest difference between batch and stream processing is latency. We can handle each individual file or message as they become available or as a group.

A batch process has defined beginning and ending points, i.e., there are boundaries placed in terms of time and format. We may process "a file" or "a set of files" in a batch process. In stream processing we look at it a little differently and treat our data as unbounded and continuous instead. Even in the case of files arriving in storage we can think of a stream of files (like log data) that continuously arrives. In the end this unboundedness is really all that is needed to make a source a data stream. In Figure 3-1 the batch process equates to processing groups of 6 files for each scheduled run where the stream process is always running and processes each file as it is available.

As we'll see shortly when we compare some of the frameworks with which we can use Delta Lake, stream processing engines like Apache Flink or Apache Spark can work together with Delta Lake as either a starting point or an ending destination for data streams. This multiple role means Delta Lake can be used at multiple stages of different kinds of streaming workloads. Often we will see the storage layer as well as a processing engine present for multiple steps of more complicated data pipelines where we see both kinds of operation occurring. One common trait among most stream processing engines is that they are just processing engines. Once we have decoupled storage and compute, each must be considered and chosen, but neither can operate independently.

From a practical standpoint the way we think about other related concepts like processing time and table maintenance is affected by our choice between batch or streaming. If a batch process is scheduled to run at certain times then we can easily measure the amount of time the process runs, how much data was processed, and chain it together with additional processes to handle table maintenance operations. We do need to think a little differently when it comes to measuring and maintaining stream processes but many of the features we've already looked at, like autocompaction and optimized writes for example, can actually work in both realms. In Figure 3-2 we can see how with modern systems batch and streaming can converge with one another and we can focus instead on latency tradeoffs once we depart from traditional frameworks. By choosing a framework that has a reasonably unified API minimizing the differences in programming for both batch and streaming use cases and running it on top of a storage format like Delta Lake that simplifies the maintenance operations and provides for either method of processing, we wind up with a more robust yet flexible system that can handle all our data processing tasks and minimize the need to balance multiple tools and avoid other complications necessitated by running multiple systems. This makes Delta Lake the ideal storage solution for streaming workloads. Next we'll consider some of the specific terminology for stream processing applications and follow it with a review of a few of the different framework integrations available to use with Delta Lake.

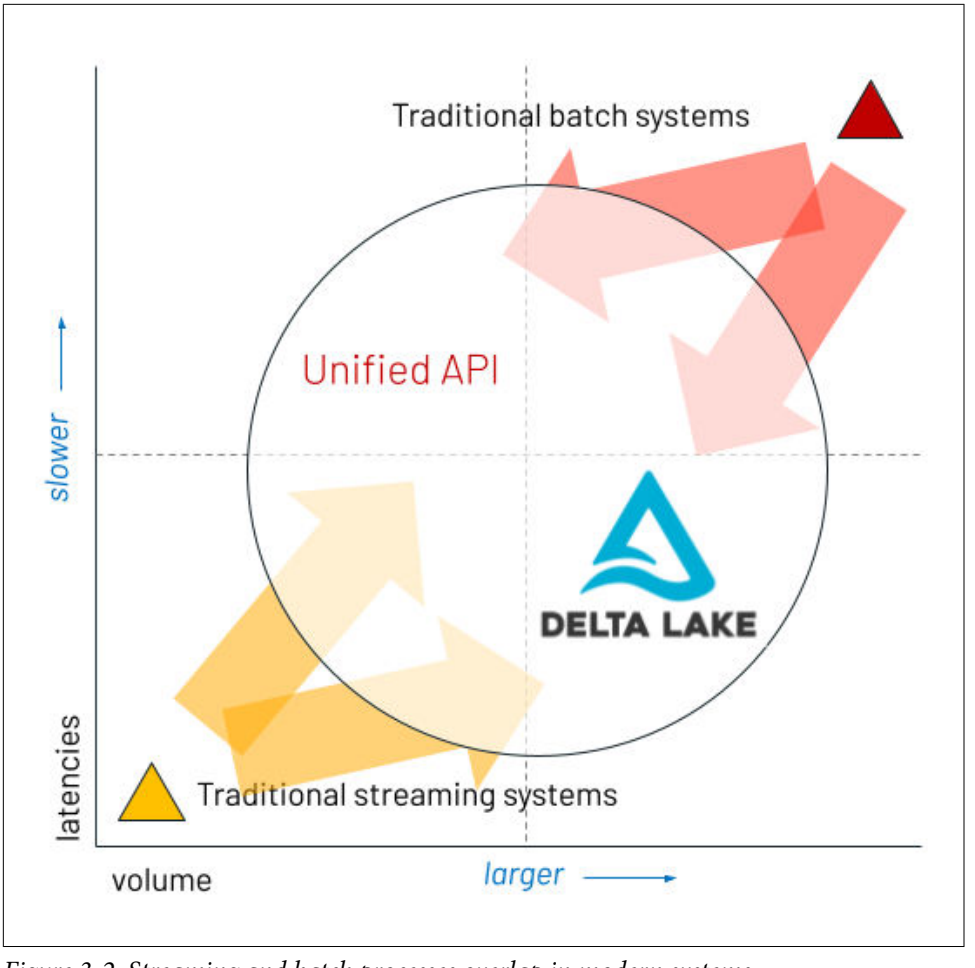


Figure 3-2. Streaming and batch processes overlap in modern systems.

Streaming Terminology

In many ways streaming processes are quite the same as batch processes with the difference being mostly one of latency and cadence. This does not mean, however, that streaming processes don't come with some of their own lingo. Some of the terms vary only a little from batch usage, like source and sink, while others don't really apply to batch, like checkpoint and watermark. It's useful to have some working familiarity with these terms but you can dig into them at a greater depth in *Stream Processing with Apache Flink* or *Learning Spark*.

Source. A stream processing source is any of a variety of sources of data that can be treated as an unbounded data set. Sources for data stream processing are varied and ultimately depend on the nature of the processing task in mind. There are a num-

ber of different message queue and pub-sub connectors used across the Spark and Flink ecosystems as data sources. These include many common favorites like Apache Kafka, Amazon Kinesis, ActiveMQ, RabbitMQ, Azure Event Hubs, and Google Pub/Sub. Both systems can also generate streams from files, for example, by monitoring cloud storage locations for new files. We will see shortly how Delta Lake fits in as a streaming data source.

Sink. Stream data processing sinks similarly come in different shapes and forms. We often see many of the same message queues and pub-sub systems in play but on the sink side in particular we quite often find some materialization layer like a key-value store, RDBMS, or cloud storage like AWS S3 or Azure ADLS. Generally speaking the final destination is usually one from the latter categories and we'll see some type of mixture of methods in the middle from origin to destination. Delta Lake functions extremely well as a sink, especially for managing large volume, high throughput streaming ingestion processes.

Checkpoint. Checkpointing is usually an important operation to make sure that you have implemented in a streaming process. Checkpointing keeps track of the progress made in processing tasks and is what makes failure recovery possible without restarting processing from the beginning every time. This is accomplished by keeping some tracking record of the offsets for the stream as well as any associated stateful information. In some processing engines, like Flink and Spark, there are built in mechanisms to make checkpointing operations simpler to use. We refer you to the respective documentation for usage details.

Let's consider an example from Spark. When we start a stream writing process and define a suitable checkpoint location it will in the background create a few directories at the target location. In this example we find a checkpoint written from a process we called 'gold' and named the directory similarly.

```
tree -L 1 /…/ckpt/gold/

/…/ckpt/gold/
├── __tmp_path_dir
├── commits
├── metadata
├── offsets
└── state
```

The metadata directory will contain some information about the streaming query and the state directory will contain snapshots of the state information (if any) related to the query. The offsets and commits directories track at a micro batch level the progress of streaming from the source and writes to the sink for the process which for Delta Lake, as we'll see more of shortly, amounts to tracking the input or output files respectively.

Watermark. Watermarking is a concept of time relative to the records being processed. The topic and usage is somewhat more complicated for our discussion and we would recommend reviewing the appropriate documentation for usage. For our limited purposes we can just use a working definition. Basically, a watermark is a limit on how late data can be accepted during processing. It is most especially used in conjunction with windowed aggregation operations.[1]

Apache Flink

Apache Flink is one of the major distributed, in-memory processing engines that supports both bounded and unbounded data manipulation. Flink supports a number of predefined and built-in data stream source and sink connectors.[2] On the data source side we see many message queues and pub-sub connectors supported such as RabbitMQ, Apache Pulsar, and Apache Kafka (see the documentation for more detailed streaming connector information). While some, like Kafka, are supported as an output destination, it's probably most common to instead see something like writing to file storage or Elasticsearch or even a JDBC connection to a database as the goal. You can find more information about Flink connectors in their documentation.

With Delta Lake we gain yet another source and destination for Flink but one which can be critical in multi-tool hybrid ecosystems or simplify logical processing transitions. For example, with Flink we can focus on event stream processing and then write directly to a delta table in cloud storage where we can access it for subsequent processing in Spark. Alternatively, we could reverse this situation entirely and feed a message queue from records in Delta Lake. A more in-depth review of the connector including both implementation and architectural details is available as a blog post on the delta.io website.

Apache Spark

Apache Spark similarly supports a number of input sources and sinks.[3] Since Apache Spark tends to hold more of a place on the large scale ingestion and ETL side we do see a little bit of a skew in the direction of input sources available rather than the more event processing centered Flink system. In addition to file based sources there is a strong native integration with Kafka in Spark as well as several separately

1 To explore watermarks in more detail we suggest the "Event Time and Stateful Processing" section of *Spark: The Definitive Guide*.

2 We understand many readers are more familiar with Apache Spark. For an introduction to concepts more specific to Apache Flink we suggest the Learn Flink page of the documentation.

3 Apache Spark source and sink documentation can be found in the "Structured Streaming Programming Guide" which is generally seen as the go-to source for all things streaming with Spark: *https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html*

maintained connector libraries like Azure Event Hubs, Google Pub/Sub Lite, and Apache Pulsar.

There are still several output sinks available too, but Delta Lake is easily among one of the largest scale destinations for data with Spark. As we mentioned earlier, Delta Lake was essentially designed around solving the challenges of large scale stream ingestion with the limitations of the parquet file format. Largely due in part to the origins of Delta Lake and the longer history with Apache Spark, much of the details covered here will be Spark-centric but we should note that many of the concepts described have corollaries with other frameworks as well.

Delta-rs

The Rust ecosystem also has additional processing engines and libraries of its own and thanks to the implementation called delta-rs we get further processing options that can run on Delta Lake. This area is one of the newer sides and has seen some intensive build-out in recent years. Polars and Datafusion are just a couple of the additional ways you might use for stream data processing and both of these couple with delta-rs reasonably well. This is a rapidly developing area we expect to see a lot more growth in going forward.

One other benefit of the delta-rs implementation is that there is a direct Python integration which opens up additional possibilities for data stream processing tasks. This means that for smaller scale jobs, it is possible to use a Python API (like AWS boto3 for example) for services that otherwise require larger scale frameworks for interaction causing unneeded overhead. While you may not be able to leverage some of the features from the frameworks that more naturally support streaming operations you could also benefit from significant reductions in infrastructure requirements and still get lightning fast performance.

The net result of the delta-rs implementation is that Delta Lake gives us a format through which we can simultaneously make use of multiple processing frameworks and engines without relying on an additional RDBMS and still operate outside of more Java centered stacks. This means that even working in disparate systems we can build data applications confidently without sacrificing the built-in benefits we gain through Delta Lake.

Delta as Source

Much of the original intent in Delta Lake's design was as a streaming sink that added the functionality and reliability that was previously found missing in practice. In particular, Delta Lake simplifies maintenance for processes that tend to have lots of smaller transactions and files and provides ACID transaction guarantees. Before we look at that side in more depth though, let's think about Delta Lake as a streaming source. By way of the already incremental nature that we've seen in the transaction

log, we have a straightforward source of json files with well-ordered id values. This means that any engine can use the file id values as offsets in streaming messages with a complete transaction record of the files added during append operations and see what new files exist. The inclusion of a flag in the transaction log, *dataChange*, helps separate out compaction or other table maintenance events that generate new files as well but do not need to be sent to downstream consumers. Since the ids are monotonic this also makes offset tracking simpler so exactly once semantics are still possible for downstream consumers.

The practical upside of all of this is that with Spark Structured Streaming you can define the readStream format as "delta" and it will begin by processing all previously available data from the table or file targeted and then add incremental updates as they are added. This allows for significant simplification of many processing architectures like the medallion architecture which we have seen before and will discuss in more detail later, but for now we should assume that creating additional data refinement layers becomes a natural operation with significantly reduced overhead costs.

With Spark, the readStream itself defines the operation mode and "delta" is just the format and the operation proceeds as usual with much of the action taking place behind the scenes. The approach is somewhat flipped with Flink. There instead you start by building off of the Delta Source object in a Data Stream class and then you would use the forContinuousRowData API to begin incremental processing.

```
## Python
streamingDeltaDf = (
    spark
    .readStream
    .format("delta")
    .option("ignoreDeletes", "true")
    .load("/files/delta/user_events")
    )
```

Delta as Sink

Many of the features you would want for a streaming sink (like asynchronous compaction operations) were not available or scalable in a way that can support modern, high-volume streaming ingestion. The availability and increased connectivity of user activity and devices as well as the rapid growth in the internet of things (IoT) quickly accelerated the growth of large-scale streaming data sources. One of the most critical problems then comes in answering the question of "how can I efficiently and reliably capture all of the data?"

Many of the features of Delta Lake are there specifically to answer this problem. The way actions are committed to the transaction log, for example, fits naturally in the context of a stream processing engine where you are tracking the progress of the stream against the source and ensuring that only completed transactions are

committed to the log while corrupted files are not, allows you to make sure that you are actually capturing all of the source data with some reliability guarantees. The metrics produced and emitted to the delta log helps you to analyze the consistency (or variability) of the stream process with counts of rows and files added during each transaction.

Most large-scale stream processing happens in "micro-batches", which in essence are smaller scale transactions of similar larger batch processes. The result of which is that we may see many write operations coming from a stream processing engine as it captures the data in flight. When this processing is happening in an "always-on" streaming process it can become difficult to manage other aspects of the data ecosystem like running maintenance operations, backfilling, or modifying historical data. Table utility commands like `optimize` and the ability to interact with the delta log from multiple processes in the environment mean that on the one hand much of this was considered beforehand and because of the incremental nature we're able to interrupt these processes more easily in a predictable way. On the other hand we might still have to think a little more often about what kinds of combinations of these operations might occasionally produce conflicts we wish to avoid. Refer to the section on concurrency control in Chapter 7 for more details.

The medallion architecture with Delta Lake and Apache Spark in particular, which we will cover in depth in Chapter 11, becomes something of a middle ground where we see Delta Lake as both a streaming sink and a streaming source working in tandem. This actually eliminates the need for additional infrastructure in many cases and simplifies the overall architecture while still providing mechanisms for low-latency, high-throughput stream processing while preserving clean data engineering practices.

Writing a streaming DataFrame object to Delta Lake is straightforward, requiring only the format specification and a directory location through the `writeStream` method.

```
## Python
(streamingDeltaDf
.writeStream
.format("delta")
.outputMode("append")
.start("/<delta_path>/")
)
```

Similarly you can chain together a readStream definition (similarly formatted) together with a writeStream definition to set up a whole input-transformation-output flow (transformation code omitted here for brevity).

```
## Python
(spark
.readStream
```

```
.format("delta")
.load("/files/delta/user_events")
…
<other transformation logic>
…
.writeStream
.format("delta")
.outputMode(“append”)
.start("/<delta_path>/")
)
```

Delta streaming options

Now that we've discussed how streaming in and out of Delta Lake works conceptually, let's delve into some of the more technical side of the options we'll ultimately use in practice and a bit of background on instances where you may wish to modify them. We'll start by taking a look at ways we might limit the input rate and, in particular, how we can leverage that in conjunction with some of the functionality we get in Apache Spark. After that we'll delve into some cases where we might want to skip some transactions. Lastly, we'll follow up by considering a few aspects of the relation between time and our processing job.

Limit the Input Rate

When we're talking about stream processing we generally have to find the balance in the tradeoffs between three things: accuracy, latency, and cost. We generally don't want to forsake anything on the side of accuracy and so this usually comes down to a tradeoff between just latency and cost, i.e. we can either accept higher costs and scale up our resources to process data as fast as possible or we can limit the size and accept longer turnaround times on our data processing. Often this is largely under control of the stream processing engine, but we have two additional options with Delta Lake that allow us some additional control on the size of micro batches.

maxFilesPerTrigger
: This sets the limit of how many new files will be considered in every micro-batch. The default value is `1000`.

maxBytesPerTrigger
: Sets an approximate limit of how much data gets processed in each micro-batch. This option sets a “soft max”, meaning that a micro-batch processes approximately this amount of data but can process more when the smallest input unit is larger than this limit. In other words, this size setting operates more like a threshold value that needs to be exceeded, whether with one file or many files, however many files it takes to get past this threshold it will use that many files,

kind of like a dynamic setting for the number of files for the microbatch that uses an approximate size.

These two settings can be balanced with the use of triggers in Structured Streaming to either increase or reduce the amount of data being processed in each microbatch. You can use these settings, for example, to lower the size of compute required for processing or to tailor the job for the expected file sizes you will be working with. If you use `Trigger.Once` for your streaming, this option is ignored. This is not set by default. You can actually use both maxBytesPerTrigger and maxFilesPerTrigger for the same streaming query. What happens then is the micro-batch will just run until either limit is reached.

We want to note here that it's possible to set a shorter logRetentionDuration with a longer trigger or job scheduling interval in such a way that older transactions can be skipped if cleanup occurs. Since it has no knowledge of what came prior processing will begin at the earliest available transaction in the log which means data can be skipped in the processing. A simple example where this could occur is where the logRetentionDuration is set to, say, a day or two, but a processing job intending to pick up the incremental changes is only run weekly. Since any vacuum operation in the intervening period would remove some of the older versions of the files this will result in those changes not being propagated through the next run.

Ignore Updates or Deletes

So far in talking about streaming with Delta Lake, there's something that we've not really discussed that we really ought to. In earlier chapters we've seen how some of the features of Delta Lake improve the ease of performing CRUD operations, most notably those of updates and deletes. What we should call out here is that by default when streaming from Delta Lake it assumes we are streaming from an append-only type of source, that is, that the incremental changes that are happening are only the addition of new files. The question then becomes ***"what happens if I have update or delete operations in the stream source?"***

To put it simply, the Spark readStream operation will fail, at least with the default settings. This is because as a stream source we only expect to receive new files and we must specify how to handle files that come from changes or deletions. This is usually fine for large scale ingestion tables or receiving change data capture (CDC) records because these won't typically be subject to other types of operations. There are two ways you can deal with these situations. The hard way is to delete the output and checkpoint and restart the stream from the beginning. The easier way is to leverage the ignoreDeletes or ignoreChanges options, the two of which have rather different

behavior despite the similarity in naming. The biggest caveat is that when using either setting you will have to manually track and make changes downstream as we'll explain shortly.

The ignoreDeletes Setting

The `ignoreDeletes` setting does exactly what it sounds like it does, that is, it ignores delete operations as it comes across them *if a new file is not created*. The reason this matters is that if you delete an upstream file, those changes will not be propagated to downstream destinations, but we can use this setting to avoid failing the stream processing job and still support important delete operations like, for example, General Data Protection Regulation (GDPR) right to be forgotten compliance where we need to purge individual user data. The catch is that the data would need to be partitioned by the same values we filter on for the delete operation so there are no remnants that would create a new file. This means that the same delete operations would need to be run across potentially several tables but we can ignore these small delete operations in the stream process and continue as normal leaving the downstream delete operations for a separate process.

The ignoreChanges Setting

The `ignoreChanges` setting actually behaves a bit differently than `ignoreDeletes` does. Rather than skipping operations which are only removing files, `ignoreChanges` allows new files that result from changes to come through as though they are new files. This means that if we update some records within some particular file or only delete a few records from a file so that a new version of the file is created, then the new version of the file is now interpreted as being a new file when propagated downstream. This helps to make sure we have the freshest version of our data available, however, it is important to understand the impact of this to avoid data duplication. What we then need in these cases is to ensure that we can handle duplicate records either through `merge` logic or otherwise differentiating the data by inclusion of additional timekeeping information (i.e. add a `version_as_of` timestamp or similar). We've found that under many types of change operations the majority of the records will be reprocessed without changes so merging or deduplication is generally the preferred path to take.

Example

Let's consider an example. Suppose you have a Delta Lake table called user_events withdate, user_email, and action columns and it is partitioned by the date column. Let's also suppose that we are using the user_events table as a streaming source for a step in our larger pipeline process and that we need to delete data from it due to a GDPR related request.

When you delete at a partition boundary (that is, the WHERE clause of the query filters data on a partition column), the files are already in directories based on those values so the `delete` just drops any of those files from the table metadata.

So if you just want to delete data from some entire partitions aligning to specific dates, you can add the `ignoreDeletes` option to the read stream like this:

```
## Python
streamingDeltaDf = (
    spark
    .readStream
    .format("delta")
    .option("ignoreDeletes", "true")
    .load("/files/delta/user_events")
    )
```

If you want to delete data based on a non-partition column like user_email instead then you will need to use the `ignoreChanges` option instead like this:

```
## Python
streamingDeltaDf = (
    spark
    .readStream
    .format("delta")
    .option("ignoreChanges", "true")
    .load("/files/delta/user_events")
    )
```

In a similar way, if you update records against a non-partition column like user_email a new file gets created containing the changed records and any other records from the original file that were unchanged. With ignoreChanges set this file will be seen by the readStream query and so you will need to include additional logic against this stream to avoid duplicate data making its way into the output for this process.

Initial Processing Position

When you start a streaming process with a Delta Lake source the default behavior will be to start with the earliest version of the table and then incrementally process through until the most recent version. There are going to be times, of course, where we don't actually want to start with the earliest version, like when we need to delete a checkpoint for the streaming process and restart from some point in the middle or even the most recent point available. Thanks again to the transaction log we can actually specify this starting point to keep from having to reprocess everything from the beginning of the log similar to how checkpointing allows the stream to recover from a specific point.

What we can do here is define an initial position to begin processing and we can do it in one of two ways.[4] The first is to specify the specific version from which we want to start processing and the second is to specify the time from which we want to start processing. These options are available via startingVersion and startingTimestamp.

Specifying the startingVersion does pretty much what you might expect of it. Given a particular version from the transaction log the files that were committed for that version will be the first data we begin processing and it will continue from there. In this way all table changes starting from this version (inclusive) will be read by the streaming source. You can review the version parameter from the transaction logs to identify which specific version you might need or you can alternatively specify "latest" to get only the latest changes.

When using Apache Spark this is easiest by checking commit versions from the version column of the DESCRIBE HISTORY command output in the SQL context.

Similarly we can specify a startingTimestamp option for a more temporal approach. With the timestamp option we actually get a couple of slightly varying behaviors. If the given timestamp exactly matches a commit it will include those files for processing, otherwise the behavior is to process only files from versions occurring after that point in time. One particularly helpful feature here is that it does not strictly require a fully formatted timestamp string, we can also use a similar date string which can be interpreted for us.This means our startingTimestamp parameter should look like either :

- a timestamp string, e.g., "2023-03-23T00:00:00.000Z"
- a date string, e.g., "2023-03-23".

Unlike some of our other settings, we cannot use both options simultaneously here. You have to choose one or the other. If this setting is added to an existing streaming query with a checkpoint already defined then they will both be ignored as they only apply when starting a new query.

Another thing you will want to note is that even though you can start from any specified place in the source using these options, the schema will reflect the latest available version. This means that incorrect values or failures can occur if there is an incompatible schema change between the specified starting point and the current version.

4 *https://docs.delta.io/latest/delta-streaming.html#specify-initial-position*

Considering our user_events dataset again, suppose you want to read changes occurring since version 5. Then you would write something like this:

```
## Python
(spark
.readStream
.format("delta")
.option("startingVersion", "5")
.load("/files/delta/user_events")
)
```

Alternatively, if you wanted to read changes based on a date instead, say occurring since `2023-04-18`, use something like this instead:

```
## Python
(spark
.readStream
.format("delta")
.option("startingTimestamp", "2023-04-18")
.load("/files/delta/user_events")
)
```

Initial Snapshot with *EventTimeOrder*

The default ordering when using Delta Lake as a streaming source is based on the modification date of the files. We have also seen that when we are initially running a query it will naturally run until we are caught up to the current state of the table. We call this version of the table, the one covering the starting point through to the current state, the *initial snapshot* at the beginning of a streaming query. On Databricks we get an additional option for interpreting time for this initial snapshot. We may want to consider whether in the case of our data set this default ordering based on the modification time is correct or if there is an event time field we can leverage in the data set that might simplify the ordering of the data.

A timestamp associated with when a record was last modified (seen) doesn't necessarily align with the time an event happened. You could think of IoT device data that gets delivered in bursts at varying intervals. This means that if you are relying on a `last_modified` timestamp column, or something similar to that, records can get processed out of order and this could lead to records being dropped as late events by the watermark. You can avoid this data drop issue by enabling the option `withEventTimeOrder` which will prefer the event time over the modification time. This is an example for setting the option on a `readStream` with an associated watermark option on the `event_time` column.

```
## Python
(spark
.readStream
.format("delta")
.option("withEventTimeOrder", "true")
```

```
    .load("/files/delta/user_events")
    .withWatermark("event_time", "10 seconds")
    )
```

When the option is enabled the initial snapshot is analyzed to get a total time range and then divided into buckets with each bucket getting processed in turn as a microbatch which might result in some added shuffle operations. You can still use the maxFilesPerTrigger or maxBytesPerTrigger options to throttle the processing rate.

There are several callouts we want to make sure you're aware of related to this situation:

- The data drop issue only happens when the initial Delta snapshot of a stateful streaming query is processed in the default order.
- withEventTimeOrder is another of those settings that only takes effect at the beginning of a streaming query so it cannot be changed after the query is started and the initial snapshot is still being processed. If you want to modify the withEventTimeOrder setting, you must delete the checkpoint and make use of the initial processing position options to proceed.
- This option became available in Delta Lake 1.2.1. If you are running a stream query with withEventTimeOrder enabled, you cannot downgrade it to a version which doesn't support this feature until the initial snapshot processing is completed. If you need to downgrade versions, you can either wait for the initial snapshot to finish, or delete the checkpoint and restart the query.
- There are a few rarer scenarios where you cannot use `withEventTimeOrder`:
  - If the event time column is a generated column and there are non-projection transformations between the Delta source and watermark.
  - There is a watermark that with multiple Delta sources in the stream query.
- Due to the potential for increased shuffle operations the performance of the processing for the initial snapshot may be impacted.

Using the event time ordering triggers a scan of the initial snapshot to find the corresponding event time range for each micro batch. This suggests that for better performance we want to be sure that our event time column is among the columns we collect statistics for. This way our query can take advantage of data skipping and we get faster filter action.You can increase the performance of the processing in cases where it makes sense to partition the data in relation to the event time column. Performance metrics should indicate how many files are being referenced in each micro batch.

Setting `spark.databricks.delta.withEventTimeOrder.enabled` `true` can be set as a cluster level Spark configuration but also be aware that doing this will make it apply it to all streaming queries that run on the cluster.

Advanced Usage with Apache Spark

Much of the functionality we've covered to this point can be applied from more than one of the frameworks listed earlier. Here we turn our attention to a couple of common cases we've encountered while using Apache Spark specifically. These are cases where leveraging features of the framework can prevent us from using some of the built in features in Delta Lake directly.

Idempotent Stream Writes

Much of the previous discussion is centered around the idea of running a processing task from a single source to a single destination. In the real world, however, we may not always have neat and simple pipelines like this and instead find ourselves building out pipelines using multiple sources writing to multiple destinations which may also wind up overlapping. With the transaction log and atomic commit behavior we can support multiple writers to a single Delta Lake destination from a functional perspective as we've already considered. How can we apply this in our stream processing pipelines though?

In Apache Spark we have the method forEachBatch available on a structured streaming DataFrame that allows us to define more customized logic for each stream micro batch. This is the typical method we would use to support writing a single stream source to multiple destinations. The problem we encounter then is that if there are, say, two different destinations and the transaction fails in writing to the second destination then we have a somewhat problematic scenario where the processing state of each of the destinations is out of sync. More specifically, since the first write was completed and the second failed, when the stream processing job is restarted it will consider the same offsets from the last run since it did not complete successfully.

Consider this example where we have a sourceDf DataFrame and we want to process it in batches to two different destinations. We define a function that takes an input DataFrame and just uses normal Spark operations to write out each microbatch. Then we can apply that function using the foreachBatch method available from the writeStream method.

```
## Python
sourceDf = ... # Streaming source DataFrame

# Define a function writing to two destinations
def writeToDeltaLakeTables(batch_df):
```

```
    # location 1
    (batch_df
    .write
    .format(“delta”)
    .save("/<delta_path_1>/")
    )
    # location 2
    (batch_df
    .write
    .format(“delta”)
    .save("/<delta_path_2>/")
    )

# Apply the function against the micro-batches using ‘foreachBatch’
(sourceDf
.writeStream
.format("delta")
.queryName("Unclear status stream")
.foreachBatch(writeToDeltaLakeTables)
.start()
)
```

Now suppose an error occurs after writing to the first location but before the second completes. Since the transaction failed we know the second table won't have anything committed to the log, but in the first table the transaction was successful. When we restart the job it will start at the same point and rerun the entire function for that microbatch which can result in duplicated data being written to the first table. Thankfully Delta Lake has something available to help us out in this case by allowing us to specify more granular transaction tracking.

Idempotent writes

Let's suppose that we are leveraging foreachBatch from a streaming source and are writing to just two destinations. What we would like to do is take the structure of the foreachBatch transaction and combine it with some nifty Delta Lake functionality to make sure we commit the micro batch transaction across all the tables without winding up with duplicate transactions in some of the tables (i.e., we want idempotent writes to the tables). We have two options we can use to help get to this state.

`txnAppId`
: This should be a unique string identifier and acts as an application id that you can pass for each DataFrame write operation. This identifies the source for each write. You can use a streaming query id or some other meaningful name of your choice as `txnAppId`.

`txnVersion`
: This is a monotonically increasing number that acts as a transaction version and functionally becomes the offset identifier for a writeStream query.

By including both of these options we create a unique source and offset tracking at the write level, even inside a foreachBatch operation writing to multiple destinations. This allows for the detection at a table level of duplicate write attempts that can be ignored. This means that if a write is interrupted during processing just one of multiple table destinations we can continue the processing without duplicating write operations to tables for which the transaction was already successful. When the stream restarts from the checkpoint it will start again with the same micro batch but then in the foreachBatch, with the write operations now being checked at a table level of granularity, we only write to the table or tables which were not able to complete successfully before because we will have the same txnAppId and txnVersion identifiers.

The application ID (`txnAppId`) can be any user-generated unique string and does not have to be related to the stream ID so you can use this to more functionally describe the application performing the operation or identifying the source of the data. The same `DataFrameWriter` options can actually be used to achieve similar idempotent writes in batch processing as well.

In the case you want to restart processing from a source and delete/recreate the streaming checkpoint you must provide a new appId as well before restarting the query. If you don't then all of the writes from the restarted query will be ignored because it will contain the same txnAppId and the batch id values would restart so the destination table will see them as duplicate transactions.

If we wanted to update the function from our earlier example to write to multiple locations with idempotency using these options we can specify the options for each destination like this:

```
## Python
app_id = ... # A unique string used as an application ID.

def writeToDeltaLakeTableIdempotent(batch_df, batch_id):
    # location 1
    (batch_df
    .write
    .format(“delta”)
    .option("txnVersion", batch_id)
    .option("txnAppId", app_id)
    .save("/<delta_path>/")
    )
    # location 2
    (batch_df
    .write
    .format(“delta”)
```

```
    .option("txnVersion", batch_id)
    .option("txnAppId", app_id)
    .save("/<delta_path>/")
    )
```

Merge

There is another common case where we tend to see forEachBatch used for stream processing. Think about some of the limitations we have seen above where we might allow large amounts of unchanged records to be reprocessed through the pipeline, or where we might otherwise want more advanced matching and transformation logic like processing CDC records. In order to update values we need to merge changes into an existing table rather than simply append the information. The bad news is that the default behavior in streaming kind of requires us to use append type behaviors, unless we leverage forEachBatch that is.

We looked at the merge operation earlier in Chapter 3 and saw that it allows us to use matching criteria to update or delete existing records and append others which don't match the criteria, that is, we can perform upsert operations. Since forEachBatch lets us treat each micro batch like a regular DataFrame then at the micro batch level we can actually perform these upsert operations with Delta Lake. You can upsert data from a source table, view, or DataFrame into a target Delta table by using the MERGE SQL operation or its corollary for the Scala, Java, and Python Delta Lake API. It even supports extended syntax beyond the SQL standards to facilitate advanced use cases.

A merge operation on Delta Lake typically requires two passes over the source data. If you use nondeterministic functions like `current_timestamp` or `random` in a source DataFrame then multiple passes on the source data can produce different values in rows causing incorrect results. You can avoid this by using more concrete functions or values for columns or writing out results to an intermediate table. Caching the source data may help either because a cache invalidation can cause the source data to be partially or completely reprocessed resulting in the same kind of value changes (for example when a cluster loses some of its executors when scaling down). We've seen cases where this can fail in surprising ways when trying to do something like using a salt column to restructure DataFrame partitioning based on random number generation (e.g. Spark cannot locate a shuffle partition on disk because the random prefix is different than expected on a retried run). The multiple passes for merge operations increase the possibility of this happening.

Let's consider an example of using merge operations in a stream using `foreachBatch` to update the most recent daily retail transaction summaries for a set of customers. In this case we will match on a customer id value and include the transaction date, number of items and dollar amount. In practice what we do to use the mergeBuilder API here is build a function to handle the logic for our streaming DataFrame. Inside the function we'll provide the customer id as a matching criteria for the target table

and our changes source, and then allow for a delete mechanism and otherwise update existing customers or add new ones as they appear.[5] The flow of the operations in the function is to specify what to merge, with arguments for the matching conditions, and which actions we want to take when a record is matched or not (for which we can add some additional conditions).

```
## Python
from delta.tables import *

def upsertToDelta(microBatchDf, batchId):
    Target_table = "retail_db.transactions_silver"
    deltaTable = DeltaTable.forName(spark, target_table)
    (deltaTable.alias("dt")
    .merge(source=microBatchDf.alias("sdf"),
          condition="sdf.t_id = dt.t_id")
    .whenMatchedDelete(condition="sdf.operation='DELETE'")
    .whenMatchedUpdate(set={
          "t_id": "sdf.t_id",
          "transaction_date": "sdf.transaction_date",
          "item_count": "sdf.item_count",
          "amount": "sdf.amount"
          })
    .whenNotMatchedInsert(values={
          "t_id": "sdf.t_id",
          "transaction_date": "sdf.transaction_date",
          "item_count": "sdf.item_count",
          "amount": "sdf.amount"
          })
    .execute())
```

The function body itself is similar to how we specify merge logic with regular batch processes already. The only real difference in this case is we will run the merge operation for every received batch rather than an entire source all at once. Now with our function already defined we can read in a stream of changes and apply our customized merge logic with the `foreachBatch` in Spark and write it back out to another table.

```
## Python
changesStream = ... # Streaming DataFrame with CDC records

# Write the output of a streaming aggregation query into Delta table
(changesStream
.writeStream
.format("delta")
.queryName("Summaries Silver Pipeline")
.foreachBatch(upsertToDelta)
.outputMode("update")
```

5 For additional details and examples on using merge in forEachBatch, e.g. for SCD Type II merges, see *https://docs.delta.io/latest/delta-update.html#merge-examples*.

```
    .start()
)
```

So each micro-batch of the changes stream will have the merge logic applied to it and be written to the destination table or even multiple tables like we did in the example for idempotent writes.

Delta Lake Performance Metrics

One of the often overlooked but very helpful things to have for any data processing pipeline is insight into the operations that are taking place. Having metrics that help us to understand the speed and scale at which processing is taking place can be valuable information for estimating costs, capacity planning, or troubleshooting when issues arise. We've already seen a couple of cases where we are receiving metrics information when streaming with Delta Lake but here we'll look more carefully at what we are actually receiving.

Metrics

As we've seen there are cases where we want to manually set starting and ending boundary points for processing with Delta Lake and this is generally aligned to versions or timestamps. Within those boundaries we can have differing numbers of files and so forth and one of the concepts that we've seen important to streaming processes in particular is tracking the offsets, or the progress, through those files. In the metrics reported out for Spark Structured Streaming we see several details tracking these offsets.

When running the process on Databricks as well there are some additional metrics which help to track backpressure, i.e. how much outstanding work there is to be done at the current point in time. The performance metrics we see get output are numInputRows, inputRowsPerSecond, and processedRowsPerSecond. The backpressure metrics are numBytesOutstanding and numFilesOutstanding. These metrics are fairly self explanatory by design so we'll not explore each individually.

Comparing the inputRowsPerSecond metric with the processedRowsPerSecond metric provides a ratio that can be used to measure relative performance and might indicate if the job should have more resources allocated or if triggers should be throttled down a bit.

Custom Metrics

For both Apache Flink and Apache Spark, there are also custom metrics options you can use to extend the metrics information tracked in your application. One method we've seen using this concept was to send additional custom metrics information

from inside a forEachBatch operation in Spark. See the documentation for each processing framework as needed to pursue this option. This provides the highest degree of customization but also the most manual effort.

Auto Loader and Delta Live Tables

The majority of our focus is on everything freely available in the Delta Lake open source project, however, there are a couple of major topics only available in Databricks that rely on or frequently work in conjunction with Delta Lake that deserve mention. As the creators of Delta Lake and Apache Spark

Autoloader

Databricks has a somewhat unique Spark structured streaming source known as Auto Loader but is really better thought of as just the `cloudFiles` source. On the whole the `cloudFiles` source is more of a streaming source definition in Structured Streaming on Databricks, but it has rapidly become an easier entrypoint for streaming for many organizations where Delta Lake is commonly the destination sink. This is partly because it provides a natural way to incrementalize batch processes to integrate some of the benefits, like offset tracking, that are components of stream processing.

The cloudFiles source actually has two different methods of operation, one is directly running file listing operations on a storage location and the other is listening on a notifications queue tied to a storage location. Whichever method is used the utility should be quickly apparent that this is a scalable and efficient mechanism for regular ingestion of files from cloud storage as the offsets it uses for tracking progress are the actual file names in the specified source directories. Refer to the section on Delta Live Tables for an example of the most common usage.

One fairly standard application of Auto Loader is using it as a part of the medallion architecture design with a process ingesting files and feeding the data into Delta Lake tables with additional levels of transformation, enrichment, and aggregation up to gold layer aggregate data tables. Quite commonly this is done with additional data layer processing taking place with Delta Lake as both the source and the sink of streaming processes which provides low latency, high throughput, end to end data transformation pipelines. This process has become somewhat of a standard for file based ingestion and has eliminated some need for more complicated lambda architecture based processes, so much so that Databricks also built a framework largely centered around this approach.

Delta Live Tables

A declarative framework

Combining incremental ingestion, streamlined ETL, and automated data quality processes like *expectations*, Databricks offers a data engineering pipeline framework running on top of Delta Lake called Delta Live Tables (DLT). It serves to simplify building pipelines like those we just described in investigating the cloudFiles source, which actually explains the main reason for including it here in our discussion about streaming with Delta Lake, that is, it is a product built around Delta Lake that captures some of the key principles noted throughout this guide in an easy to manage framework.

Using Delta Live Tables

Rather than building out a processing pipeline piece by piece, the declarative framework allows you to simply define some tables and views with less syntax than, for example, many of the features we discussed by automating many of the best practices commonly used across the field. Some of the things that it will manage on your behalf include compute resources, data quality monitoring, processing pipeline health, and optimized task orchestration.

DLT offers static tables, streaming tables, views and materialized views to chain together many otherwise more complicated tasks. On the streaming side we see Auto Loader as a prominent and common initial source feeding downstream incremental processes across Delta Lake backed tables. Here is some example pipeline code based on examples in the documentation.

```
## Python
import dlt

@dlt.table
def autoloader_dlt_bronze():
    return (
        spark
        .readStream
        .format("cloudFiles")
        .option("cloudFiles.format", "json")
        .load("<data path>")
    )

@dlt.table
def delta_dlt_silver():
    return (
        dlt
        .read_stream("autoloader_dlt_bronze")
        …
        <transformation logic>
```

```
        …
    )

@dlt.table
def live_delta_gold():
    return (
        dlt
        .read("delta_dlt_silver")
        …
        <aggregation logic>
        …
    )
```

Since the initial source is a streaming process the silver and gold tables there are also incrementally processed. One of the advantages we gain for streaming sources specifically is simplification. By not having to define checkpoint locations or programmatically create table entries in a metastore we can build out pipelines with a reduced level of effort . In short, DLT gives us many of the same benefits of building data pipelines on top of Delta Lake but abstracts away many of the details making it simpler and easier to use.

Change Data Feed

Earlier we looked at what it might look like to integrate Change Data Capture (CDC) data into a streaming Delta Lake pipeline. Does Delta Lake have any options for supporting this type of feed? The short answer is: yes. To get around to the longer answer, let's first make sure we're on level terms of understanding.

By this point, we have worked through quite a few examples of using Delta Lake and we've seen that basically we have just 3 major operations for any particular row of data: inserting a record, updating a record, or deleting a record. This is similar to pretty much any other data system. So where does CDC come into play then exactly?

As defined by Joe Reis and Matt Housley in *Fundamentals of Data Engineering* "change data capture (CDC) is a method for extracting each change event (insert, update, delete) that occurs in a database. CDC is frequently leveraged to replicate between databases in near real time or create an event stream for downstream processing." Or, as they put it more simply, "CDC… is the process of ingesting changes from a source database system."[6]

Bringing this back around to our initial inquiry, tracking changes is supported in Delta Lake via a feature called Change Data Feed (CDF). What CDF does is it lets you track the changes to a Delta Lake table. Once it is enabled you get all of the

6 Fundamentals of Data Engineering: Plan and Build Robust Data Systems by Joe Reis and Matt Housley, p. 163, p. 256, O'Reilly, 2022.

changes to the table as they occur. Updates, merges, and deletes will be put into a new _change_data folder while append operations already have their own entries in the table history so they don't require additional files. Through this tracking we can read the combined operations as a feed of changes from the table to use downstream. The changes will have the required row data with some additional metadata showing the change type.

This feature is available in Delta Lake 2.0.0 and above. As of writing, this feature is in experimental support mode.

Levels of support for using Change Data Feed on tables with column mapping vary by the version you are using.

- Versions <= 2.0 do not support streaming or batch reads on change data feed on tables that have column mapping enabled.
- For version 2.1, only batch reads are supported for tables with column mapping enabled. This version also requires that there are no non-additive schema changes (no renaming or reordering).
- For version 2.2, both batch and streaming reads are supported on change data feeds from tables with column mapping enabled as long as there still are no non-additive schema changes.
- Versions >= 2.3 batch reads on change data feed for tables with column mapping enabled can now support non-additive schema changes. It uses the schema of the ending version used in the query rather than the latest version of the table available. You can still encounter failures in the case where the version range specified spans a non-additive schema change.

Using Change Data Feed

While it is ultimately up to you whether or not to leverage the CDF feature in building out a data pipeline, there are some common use cases where you can make good use of it to simplify or rethink the way you are handling some processing tasks. Here are a few examples of way you might think about leveraging it:

Curating Downstream Tables
: You can improve the performance of downstream Delta Lake tables by processing only row-level changes following initial operations to the source table to simplify ETL and ELT operations because it provides a reduction in logical complexity. This happens because you will already know how a record is being changed before checking against its current state.

Propagating Changes
: You can send a change data feed to downstream systems such as another streaming sink like Kafka or to some other RDBMS that can use it to incrementally process in later stages of data pipelines.

Creating an Audit Trail
: You could also capture the change data feed as a Delta table. This could provide perpetual storage and efficient query capability to see all changes over time, including when deletes occur and what updates were made. This could be useful for tracking changes across reference tables over time or security auditing of sensitive data.

We should also note that using CDF may not necessarily add any additional storage. Once enabled what we actually find is that there is no significant impact in processing overhead. The size of change records is pretty small and in most cases is much smaller than that actual data files written during change operations This means there's very little performance implication for enabling the feature.

Change data for operations is located in the `_change_data` folder under the Delta table directory similar to the transaction log. Simple operations, like appending files or deleting whole partitions, are much simpler than other types of changes. When the changes are of this simpler type Delta Lake detects it can efficiently compute the change data feed directly from the transaction log and these records may be skipped altogether in the folder. Since these operations are often among the most common this strongly aids in reducing overhead.

Since it is not part of the current version of table data the files in the `_change_data` folder follow the retention policy of the table. This means it is subject to removal during vacuum operations just like other transaction log files that fall outside of the retention policy.

Enabling the Change Feed

On the whole there's not much you need to do as far as configuring CDF for Delta Lake.[7] The gist of it really is to just turn it on, but doing this is slightly different depending on whether you are creating a new table or if you are implementing the feature for an existing one.

For a new table simply set the table property delta.enableChangeDataFeed = true within the CREATE TABLE command.

7 *https://docs.delta.io/latest/delta-change-data-feed.html#enable-change-data-feed*

```
## SQL
CREATE TABLE student (id INT, name STRING, age INT) TBLPROPERTIES (delta.enable
ChangeDataFeed = true)
```

For an existing table you can instead alter the table properties with the ALTER TABLE command to set delta.enableChangeDataFeed = true.

```
## SQL
ALTER TABLE myDeltaTable SET TBLPROPERTIES (delta.enableChangeDataFeed = true)
```

If you are using Apache Spark you can set this as the default behavior for the SparkSession object by setting spark.databricks.delta.properties.defaults.enableChangeDataFeed to true.

Reading the Changes Feed

Reading the change feed is similar to most read operations with Delta Lake. The key difference is that we need to specify in the read that we want the change the feed itself rather than just the data as it is by setting readChangeFeed to true. Otherwise the syntax looks pretty similar to setting options for time travel or typical streaming reads. The behavior between reading the change feed as a batch operation or as a stream processing operation differs, so we'll consider each in turn. We won't actually use it in our examples but rate limiting with `maxFilesPerTrigger` or `maxBytesPerTrigger` can be applied to versions other than the initial snapshot version. When used either the entire commit version being read will be rate limited as expected or the entire commit will be returned when below the threshold.

Specifying Boundaries for Batch Processes. Since batch operations are a bounded process we need to tell Delta Lake what bounds we want to use to read the change feed. You can either provide version numbers or timestamp strings to set both the starting and ending boundaries.[8] The boundaries you set will be inclusive in the queries, that is, if the final timestamp or version number exactly matches a commit then the changes from that commit will be included in the change feed. If you want to read the changes from any particular point all the way up to the latest available changes then only specify the starting version or timestamp.

When setting boundary points you need to either use an integer to specify a version or a string in the format `yyyy-MM-dd[ HH:mm:ss[.SSS]]`for timestamps in a similar way to how we set time travel options. An error will be thrown letting you know that the change data feed was not enabled if a timestamp or version you give is lower or older than any that precede when the change data feed was enabled.

```
## Python
# version as ints or longs
```

8 *https://docs.delta.io/latest/delta-change-data-feed.html#read-changes-in-batch-queries*

```
(spark.read.format("delta")
  .option("readChangeFeed", "true")
  .option("startingVersion", 0)
  .option("endingVersion", 10)
  .table("myDeltaTable")
)

# timestamps as formatted timestamp
(spark.read.format("delta")
  .option("readChangeFeed", "true")
  .option("startingTimestamp", '2023-04-01 05:45:46')
  .option("endingTimestamp", '2023-04-21 12:00:00')
  .table("myDeltaTable")
)

# providing only the startingVersion/timestamp
(spark.read.format("delta")
  .option("readChangeFeed", "true")
  .option("startingTimestamp", '2023-04-21 12:00:00.001')
  .table("myDeltaTable")
)


# similar for a file location
(spark.read.format("delta")
  .option("readChangeFeed", "true")
  .option("startingTimestamp", '2021-04-21 05:45:46')
  .load("/pathToMyDeltaTable")
)
```

Specifying Boundaries for Streaming Processes. If we want to use a readStream on the change feed for a table we can still set a startingVersion or startingTimestamp but they are more optional than they are in the batch case as if the options are not provided the stream returns the latest snapshot of the table at the time of streaming as an INSERT and then all future changes as change data.

Another difference for streaming is that we won't configure an ending position since a stream is unbounded and so does not have an ending boundary. Options like rate limits (`maxFilesPerTrigger`, `maxBytesPerTrigger`) and excludeRegex are also supported when reading change data and so other than that we proceed as we would normally.

```
## Python
# providing a starting version
(spark.readStream.format("delta")
  .option("readChangeFeed", "true")
  .option("startingVersion", 0)
  .load("/pathToMyDeltaTable")
)

# providing a starting timestamp
```

```
(spark.readStream.format("delta")
  .option("readChangeFeed", "true")
  .option("startingTimestamp", "2021-04-21 05:35:43")
  .load("/pathToMyDeltaTable")
)

# not providing either
(spark.readStream.format("delta")
  .option("readChangeFeed", "true")
  .load("/pathToMyDeltaTable")
)
```

If the specified starting version or timestamp is beyond the latest found in the table then you will get an error: `timestampGreaterThanLatestCommit`. You can avoid this error, which would mean choosing to receive an empty result set instead, by setting this option:

```
## SQL
set spark.databricks.delta.changeDataFeed.timestampOutOfRange.enabled = true;
```

If the starting version or timestamp value is in range of what is found in the table but an ending version or timestamp is out of bounds you will see with this feature enabled that all available versions falling within the specified range will be returned.

Schema

At this point you might wonder exactly how the data we are receiving in a change feed looks as it comes across. You get all of the same columns in your data as before. This makes sense because otherwise it wouldn't match up with the schema of the table. We do, however, get some additional columns so we can understand things like the change type taking place. We get these three new columns in the data when we read it as a change feed.

Change Type

The `_change_type` column is a string type column which, for each row, will identify if the change taking place is an `insert`, an `update_preimage` , an `update_postimage`, or a `delete` operation. In this case the preimage is the matched value before the update and the postimage is the matched value after the update.

Commit Version

The `_commit_version` column is a long integer type column noting the Delta Lake file/table version from the transaction log that the change belongs to. When reading the change feed as a batch process it will be at or in between the

boundaries defined for the query. When read as a stream it will be at or greater than the starting version and continue to increase over time.

Commit Timestamp
: The `_commit_timestamp` column is a timestamp type column (formatted as `yyyy-MM-dd[ HH:mm:ss[.SSS]]`) noting the time at which the version in `_commit_version` was created and committed to the log.

As an example, suppose we have the following example where there was a (fictional) discrepancy in the people10m dataset. We can update the errant record and when we view the change feed we will see the original record values denoted as the preimage and the updated values denoted as the postimage. We'll update the set on the mistakenly input name and correct the name and the gender of the individual. Afterwards we'll view a subset of the table highlighting the before and after change feed records to see what it looks like. We can also note that it captures both the version and timestamp from the commit at the same time.

```
## SQL
UPDATE
people10m
SET
gender = 'F',
firstName='Leah'
WHERE
firstName='Leo'
and lastName='Conkay';

## Python
(
  spark
  .read.format("delta")
  .option("readChangeFeed", "true")
  .option("startingVersion", 5)
  .option("endingVersion", 5)
  .table("tristen.people10m")
  .select(
      col("firstName"),
      col("lastName"),
      col("gender"),
      col("_change_type"),
      col("_commit_version"))
  ).show()
```

```
+---------+--------+------+----------------+--------------+-------------------+
|firstName|lastName|gender|    _change_type|_commit_version|  _commit_timestamp|
+---------+--------+------+----------------+--------------+-------------------+
|      Leo|  Conkay|     M| update_preimage|             5|2023-04-05 13:14:40|
|     Leah|  Conkay|     F|update_postimage|             5|2023-04-05 13:14:40|
+---------+--------+------+----------------+--------------+-------------------+
```

Additional Thoughts

Here we have built upon many of the concepts covered in previous chapters and seen how they can be applied across several different kinds of uses. We explored several fundamental concepts used in stream data processing and how they come into play with Delta Lake. We indirectly saw how the core streaming functionality (particularly in Spark) is simplified with the use of a unified API due to the similarity in how it is used. Then we explored some different options for providing more direct control over the behavior of streaming reads and writes with Delta Lake. We followed this by looking a bit at some areas closely related to stream processing with Apache Spark or on Databricks but are built on top of Delta Lake. We finished by reviewing the Change Data Feed functionality available in Delta Lake and how we can use it in streaming or non-streaming applications. We hope this helps to answer many of the questions or curiosities you might have had about this area of using Delta Lake. After this we're going to explore some of the other more advanced features available in Delta Lake.

Key References

- Spark Definitive Guide
- Stream Processing with Apache Flink
- Learning Spark
- Streaming Systems

CHAPTER 4
Architecting your Lakehouse

A Note for Early Release Readers

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

This will be the 11th chapter of the final book. Please note that the GitHub repo will be made active later on.

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

Successful engineering initiatives begin with proper vision and clear purpose (what we are doing and why) as well as a solid design and architecture (how we plan to achieve the vision). Combining a thoughtful plan with the right building blocks (tools, resources, and engineering capabilities) ensures that the final result reflects the mission and performs well at scale. Delta Lake provides key building blocks enabling us to design, construct, test, deploy, and maintain enterprise grade data lakehouses.

The goal of this chapter is more than just offering a collection of ideas, patterns, and best practices, but rather to act as a field guide. By providing the right information, reasoning, and mental models, lessons learned here can coalesce into clear blueprints to use when architecting your own data lakehouse. Whether you are new to the concept of the lakehouse, unfamiliar with the medallion architecture for incremental data quality, or if this is your first foray into working with streaming data, we'll take this journey together.

What we'll learn:

- What is the Lakehouse Architecture?
- Using Delta Lake as the foundation for implementing the Lakehouse Architecture
- The Medallion Architecture
- Streaming Lakehouse Architecture

The Lakehouse Architecture

If successful engineering initiatives begin with clear vision and purpose, and our goal is ultimately to lay the foundation for our own data lakehouses, then we'll need to first define what a lakehouse is.

What is a Lakehouse?

> " The Lakehouse is an open data management architecture that combines the flexibility, cost-efficiency, and scale of the data lake, with the data management, schema enforcement, and ACID transactions of the traditional data warehouse. " - Databricks

There is a lot to unpack from this definition, namely, there are assumptions being made that all require some hands-on experience, or shared mental models, from both engineering and data management perspectives. Specifically, the definition assumes a familiarity with data warehouses and data lakes, as well as the trade-offs people must make when selecting one technology versus the other. The following section will cover the pros and cons of each choice, and describe how the lakehouse came to be.

The history and myriad use cases shared across the data warehouse and data lake should be second nature for anyone who has previously worked in roles spanning the delivery and consumption spaces. For anyone just setting out on their data journey, transitioning from data warehousing, or who has only worked with data in a data lake, this section is also for you.

In order to understand where the lakehouse architecture evolved from, we'll need to be able to answer the following:

- If the lakehouse is a hybrid architecture combining the best of the data lake and data warehouse, then in doing so, it must be better than the sum of its parts.
- Why does the flexibility, cost-efficiency, and unbounded data scaling, inspired by traditional data lakes, matter for all of us today?
- Why do the benefits of the data lake only truly matter when coupled with the benefits of schema-enforcement and evolution, ACID transactions, and proper data management, inspired by traditional data warehouses?

Learning from Data Warehouses

The data warehouse emerged to fix the issue of data silos within large enterprises and to simplify business intelligence and analytical decision making. While the data warehouse exists as a centralized solution to solve structured data problems within a given data domain, physical limitations within the data warehouse architecture meant costs would increase proportionally to the size and scale of the data within the warehouse. The root cause of the physical limitations were due to data being stored locally (non-distributed) in what is known as a vertically scaling architecture.

While cost is a limiting factor of large scale data warehouses (due to vertical scaling), the benefits of running the data warehouse can outweigh the higher bills when compared to operating many independent data silos. Architected with safe data management, access policies, and the enforcement of rules and standards in mind: data warehouses are built for consistency first. This means a lot when considering the correctness of data, which now falls under its own umbrella of *data quality*. With support of type-safe, structured data and schema enforcement, the data warehouse is commonly utilized for foundational business-intelligence and operational data systems that must provide consistent tables, and clear data definitions.

On the data management front, support for access control, through user and role based permissions, called grants, enable a secure and rule based system to gate which users can execute reads (select), writes (insert), updates, and deletes of the data within the warehouse's subsequent tables and views.

Outside of cost, issues preventing the data warehouse architecture from scaling to meet the demands of today, reside in a lack of flexibility supporting various kinds of workloads including data science and machine learning.

Today missing support for common machine learning and data science workflows, which require custom data types and formats - supporting unstructured (images), semi-structured (csv, json), and fully structured data (parquet / orc) - as well as the ability to easily read entire tables into memory–with efficient file skipping, column pruning–all without needing to make expensive queries multiple times for iterative algorithms.

Unfortunately, more data copying introduced silos due to missing support for data science, which requires data to be stored in the data lake, while supporting analysts and the business intelligence folks who needed their data to remain in the warehouse.

Learning from Data Lakes

The data lake emerged to store raw (unprocessed) data in a wide variety of formats (csv, json, orc, text, binary) within a distributed file system; the popular choice at the time being the Hadoop Distributed File System (HDFS). Utilizing commodity hardware, the data lake could be utilized to run distributed processing jobs (Map

Reduce), or be leveraged to act as a staging area for data to be loaded into the data warehouse. Today, many workloads still follow similar patterns, utilizing cloud based object stores, or other managed elastic storage and elastic compute to power data lakes. So how does this fit into the lakehouse story?

The data lake provides a solution for storing raw feeds of data (as files) that can be processed directly for data science and machine learning use cases, supporting data formats that are unavailable within the data warehouse. These feeds of data found another use though being transformed to *keep the data warehouse in sync* using the dual-tier data architecture, which is covered in the next section.

The benefits of the data lake are associated with the cost, which is comparatively low when weighed against data warehouse as well as well general support for file format flexibility.

The file format flexibility also acts as a double-edged sword. What exists in one format today, can just as easily shift tomorrow, as the data lake remains schema-less, allowing anything to be stored inside its filesystem.

On the upside, the separation of storage and compute means that costs remain low, requiring minimal overhead, until the point where data will be called into action. Sadly, due to the schema-less nature of the data lake, things don't always go well when older datasets are pulled out of storage. Corrupt data is one of the big reasons why the data lake also coined the name the "Data Swamp".

Further distancing itself from the data warehouse, the data lake doesn't support transactions, operation-level isolation, and as a consequence it lacks support for multiple simultaneous data producers or consumers sharing the same set of resources in the data lake. With respect to consistency, it is near impossible to achieve a consistent state between active readers and writers, or to support multiple access modes, like what is more common today with batch and streaming jobs operating on the same physical table.

Understanding that a data lake *without rules* eventually leads to data instability, unusable data, and in the worst examples completely "polluted" or "toxic" data lakes, there emerged this radical idea, "what if you could achieve the best of both worlds?"

The Dual-Tier Data Architecture

The dual-tier architecture is the natural evolution in the relationship between the data lake and warehouse. Set into your mind an orchestration platform like Airflow. The reason Airflow is popular rests on the fact that it is difficult to manage consistency between the data lake and the data warehouse. What if we had a way to manage both?

Rather than having a single hop from the operational data system (siloed data) into the data warehouse (shared), or into the data lake, the dual-tier architecture relied on extract-tranform-load (ETL) jobs to manage consistency. Consider the following set of jobs:

1. Write operational data from source database A into the data lake (location a).
2. Read, clean, transform the data from (location a) and write the changes to (location b)
3. Read from (location b), joining and normalizing with data from (location c) into a landing zone (location d)
4. Read the data from (location d) and write it into the data warehouse for consumption by the business.

As long as the workflow completes, the data in the data lake will be in sync with the warehouse, and enables support for unloading or reloading tables to save cost in the data warehouse.

This makes sense in hindsight.

In order to support direct read access on the data, the data lake is required for supporting machine learning use cases, while the data warehouse is required to support the business and analytical processing. However, the added complexity inadvertently puts a greater burden on data engineers to manage multiple sources of truth, the cost of maintaining multiple copies of all the same data (once or more in the data lake, and once in the data warehouse), and the headache of figuring out what data is stale, where, and why.

If you have ever played the game two truths and a lie, this is the architectural equivalent but rather than a fun game, the stakes are much higher; this is, after all, our precious operational data. Two sources of truth, by definition, mean both systems can be (and probably will be) out of sync, telling their own versions of the truth. This also means each source of truth is also lying. They just aren't aware.

So the question is still up in the air. What if you could achieve the best of both worlds and efficiently combine the data lake and the data warehouse? Well, that is where the data lakehouse was born.

Lakehouse Architecture

The lakehouse is a hybrid data architecture that combines the best of the data warehouse with the best of the data lake. Figure 4-1 provides a simple flow of concepts through the lens of what use cases can be attributed to each of the three data architectures: the data warehouse, data lake, and the data lakehouse.

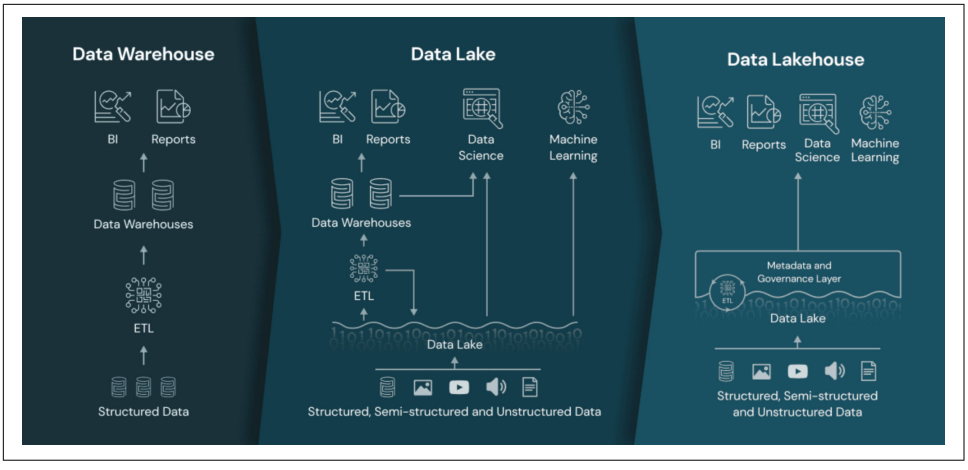


Figure 4-1. The Data Lakehouse provides a common interface for BI and reporting while ensuring that data science and machine learning workflows are supported in a single, unified, way.

This new architecture is enabled through an open system design: implementing similar data structures and data management features to those in a data warehouse, directly on the kind of low-cost storage used for data lakes.

Merging them into a single system means that data teams can move faster as they are able to use data without needing to access multiple systems. This dissolves the boundaries between the data warehouse and data lake, while also providing a single-source of truth, which is a huge win over the dual-tier architecture, and prevents the problem of figuring out which side (warehouse or lake) has the correct data, who isn't in sync, and all the costly work involved to come up with a straight answer.

The benefits also ensure teams have the most complete and up-to-date data available for data science, machine learning, and business analytics projects.

Architectural Pillars of the Data Lakehouse

- Transaction Support
- Schema Enforcement and Governance - audit log and data integrity
- BI Support through SQL and open interfaces like JDBC
- Separation between Storage and Compute
- Open-Standards: Open APIs, and Open Data Formats
- End to End Streaming
- Supports Diverse Workloads from traditional SQL to deep learning

Foundations with Delta Lake

We just learned about the successful marriage of ideas resulting in the Lakehouse. A design that isn't limited in the ways of the data warehouse and benefits from the high-availability, near boundless-scalability, and cost effective separation of storage and compute of the data lake.

This section will cover what we gain out of the box with Delta Lake and why it's the right tool to power the Lakehouse.

Open-Source on Open-Standards in an Open Ecosystem

Architecting your lakehouse with Delta Lake comes with *open-standards and a commitment to an open-ecosystem* focused on open-protocols, common sense, and standard conventions.

Open File Format

Apache Parquet is the physical file format for the data stored in our Delta tables. Parquet, being widely supported within the big data community, had already proved its value with respect to speed and scalability, but it remained difficult to maintain over time. Parquet on its own doesn't provide schema-validations or evolution. Nor does it support column remapping.

The big difference that Delta brings to the table is consistency and column-level guarantees enabling the underlying parquet to survive schema transformations and subtle changes over time that would leave standard parquet corrupted when processed as a contiguous collection of data over time.

Parquet is the standard file format for column oriented analytical data. So rather than implement an internal, proprietary table format and access protocol - the Delta protocol is freely available to be used by the community to build new tooling and connectors (which we looked at in Chapter 5) and can be used natively within many offerings provided by the key cloud service vendors like Amazon, Microsoft, as well as Starburst, and Databricks.

Self Describing Table Metadata

The metadata for each Delta table is stored alongside the physical table data. This design eliminates the need to maintain a separate metastore, like the Hive Metastore, to simply describe a given table. The design decision enables static tables to be copied more efficiently, and moved using standard file system tools, while also enabling metadata-only copies of tables to exist as we've seen with `SHALLOW CLONE` in Chapter 7.

Open Table Specification

Lastly, there is no fear of vendor lock-in; the entire Delta Lake project itself is provided freely to the entire open-source community through the Linux Foundation and has a good community around it.

Delta Universal Format (*[1]UniForm)

UniForm is a new feature introduced in Delta Lake 3.0. UniForm enables reading Delta in the format needed by an application, improving compatibility and expanding the ecosystem. Delta UniForm will automatically generate metadata needed for Apache Iceberg or Apache Hudi, so users don't have to choose upfront, or do manual conversions between formats which can be error prone. With UniForm, Delta is the universal format that works across ecosystems providing interoperability for the Lakehouse.

Transaction Support

Support for *transactions* is critical whenever data accuracy and sequential insertion order is important. Arguably this is required for nearly all production cases. *We should concern ourselves with achieving a minimally high bar at all times*. While transactions mean there are additional checks and balances, for example, if there are multiple writers making changes to a table there will always be an possibility for collisions. Understanding the behavior of the distributed Delta transaction protocol means we know exactly which write should win and how, and can guarantee the insertion order of data to be exact for reads.

Serializable Writes

Delta provides ACID guarantees for transactions while enabling multiple concurrent writers using a technique called write serialization. When new rows are simply being appended to the table, like with INSERT operations, the table metadata doesn't need to be read before a commit can occur. However, if the table is being modified in a more complex way, for example, if rows are being deleted, or updated, then the table metadata will be read before the write operation can be committed. This process ensures that before any changes are committed, the changes don't collide which could potentially corrupt the true sequential insert and operation order on a Delta table. Rather than risking corruption, collisions result in a specific set of exceptions raised by the type of concurrent modification.

1 UniForm is "coming soon" as of this Early Release

Snapshot Isolation for Reads

Processes reading a given Delta table are insulated from the complexities of multiple simultaneous writers and are guaranteed to read a consistent snapshot of the Delta table in exact serial order.

Support for Incremental Processing

Each table contains a single serial history of the atomic versions of the table, and for each version of the table the state is contained in a snapshot. This means that processes (jobs) reading from the Delta table at specific versions (points in time) can intuitively read only the specific changes between their local table snapshot, and the current (latest) version of the table.

Incremental processing reduces the operational burden of maintaining a cursor (last offsets, ids) or more complex state. Consider Example 4-1. We've probably seen a job like this in our careers, or can surmise that it is taking a starting timestamp, a set number of records to read, write, maybe delete, and is also taking the last record identified of the last successful batch. It is easier to say the batch job is using a checkpoint. But there is nothing easy about maintaining state.

Example 4-1. Providing state to a stateless batch job

```
% ./run-some-batch-job.py \
    --startTime x \
    --recordsPerBatch 10000 \
    --lastRecordId z
```

With Delta Lake, we can use the startingVersion to provide a specific point in the table to read from. Example 4-2 provides a glimpse at the same job with the startingVersion.

Example 4-2. Providing the Delta startingVersion to a stateless batch job

```
% ./run-some-batch-job.py --startingVersion 10
```

Support for Time Travel

The biggest gain from transactions, aside from the ability to rewind and reset tables based on incorrect inserts, is the ability to harness this power (time travel) to do new things like view the state of a given table at specific points in time to compare changes that have been made. This is a vantage point that few data engineers know they need, and a capability that can drastically reduce mean-time-to-resolution (MTTR) since each table has a history, and that history is very similar to git history or git blames for those familiar.

Schema Enforcement and Governance

Governance in the following context applies to the rules governing the structure of a given table definition (DDL) which manage the columns, column types, and descriptive metadata that make up a table. Schema enforcement pertains to the consequences of attempting to write invalid content into a table.

Delta Lake uses schema-on-write to achieve the high level of consistency required by the classic databases and supports the governance that people have come to rely on within database management systems (DBMS). For clarity, we'll cover the differences between schema-on-write and schema-on-read next.

Schema-On-Write

Because Delta Lake supports schema-on-write and declarative schema evolution, the onus of being correct falls to the producers of the data for a given Delta Lake table. However, this doesn't mean that anything goes just because you wear the 'producer of the data' hat. Remember that data lakes only become data swamps due to a lack of governance. With Delta Lake, the initial successful transaction committed automatically sets the stage identifying the table columns and types. With a governance hat on, we now must abide by the rules written into by the transaction log. This may sound a little scary, but rest assured, it is for the betterment of the data ecosystem. With clear rules around schema enforcement and proper procedures in place to handle schema evolution, the rules governing how the structure of a table is modified ultimately protect the consumers of a given table from problematic surprises.

Consistent Data & Quality Expectations

In the real world having invariants in place reduces the conversation about who broke what, when, and where. With Delta Lake this means to use the mergeSchema option infrequently and to be very concerned if people want to use overwriteSchema. When using Delta Lake with some established ways of working, the DeltaLog will be your source of truth for arbitration, effectively removing useless meetings since you can just about automatically pinpoint root cause in the case that things did end up going off the rails just by looking at DeltaTable.forName(spark, …).history(10).

Schema-On-Read

Data Lakes use the *schema-on-read* approach because is no consistent form of governance or metadata native to the data lake–which is essentially a glorified distributed file system. While *schema-on-read* is flexible, it's flexibility is also why data lakes are categorized like the wild west ; ungoverned, chaotic and more often than not problematic.

What this means is that while there is data in some location (directory root), with some file type (json, csv, binary, parquet, text, and more), with the ability for files being written to a specific location to grow unbounded, there is a high potential for problems to grow with the age of a dataset.

As a consumer of the data in the data lake at a specific location, If you're lucky, the data may be something you can extract and parse–it may even have some kind of documentation if you're really lucky–and with enough lead time and compute, you can probably accomplish your job. Without proper governance and type-saftey however, the data lake can grow quickly to multiple terabytes, petabytes if you love burning money, of essentially data garbage with a low-cost of storage overhead. While this is an extreme statement it is also a reality in many data organizations.

Separation between Storage and Compute

Delta Lake provides a clear separation between storage and compute. One of the biggest benefits of the data lake architecture is the flexibility of unbounded storage and file system scalability. The lakehouse architecture adopts the benefits of the data lake, since in today's day and age, producing and consuming tons of data comes with the territory of modern data, analytics and machine learning.

In theory, as long as you have strict governance in place around schema enforcement, conformance, and evolution - that comes along with the invariants of schema-on-write - coupled with opinionated support for the underlying file format (parquet), then you gain near limitless scalability (within reason) for the data living in your data lakehouse, using a file format that is interoperable and extremely portable. The portability aspect can be broken down even further. You can take your Delta Lake tables (pack the whole lakehouse up and go) from one cloud to another cloud, while retaining the integrity of all your tables - including the transaction logs.

Separation Between Logical Action and Physical Reaction

It is worth pointing out that there is even more of a separation between logical action within Delta Lake and the resulting physical action on the underlying physical storage layer. Take the example of cleaning up our tables from Chapter 6; there is a separation between calling *DELETE FROM* on a given table and when the physical files are affected (actually deleted). This is due to the time travel capabilities (rewind/undo) that enable us to remove accidental deletes. Delete's that can otherwise harm the data integrity with no chance of restoration. Deleting data accidentally has happened to everyone at one point or another in their career, just not everyone admits to it! This is why the VACUUM and REORG operations are so valuable. In order to really delete files an action with a physical reaction must occur.

Support for Transactional Streaming

We introduced Delta's streaming capabilities in Chapter 9. The ability to easily switch between batch and streaming, across transactional tables, regardless of the specific operation (inbound reads, or outbound writes) with Delta may initially sound magical. Many the streaming pipeline has met its unexpected end due to distributed files suddenly disappearing on source tables due to changes made to tables by outside forces (like overwrite jobs to replace missing data), but with delta there is complete support for multi-version concurrency control that means a streaming application reading from a table, won't be interrupted[2] due to another writers operation.

Delta Lake supports full end-to-end streaming, without sacrificing quality for speed. Everything has trade offs, and it is easy to go fast and operate blindly. In the real world it is better to weigh the cost of delay with the need for speed, and come to a general agreement on what tradeoffs the business or data team is willing to make to achieve the correct balance. We can't always have our cake and eat it too, but with time travel, almost anything is possible.

Unified Access for Analytical and ML Workloads

Rounding things out, Delta provides a balanced approach to a wide range of data related solutions. Data analysts and BI engineers can easily query using simple SQL while there is also simultaneous support for efficient direct physical file access for the data encompassing the Delta Lake tables, which provides the correct operating model for data science and ML workloads where direct access to all columnar data is

2 For common append style writes to the table. Other operations like overwriting a table, or deleting the table can affect streaming applications.

required including the ability to run iterative algorithms (in-place) within the scope of a job.

Delta Sharing Protocol

Sharing data safely and reliably between internal and external stakeholders is one of the hardest problems after data modeling. It is common practice to see ETL jobs that export data out of the data lake, for example from one S3 bucket to another. The reasons for essentially using file transfer protocol (FTP) to send and receive data rests on missing standards for identity and access management (IAM) and interoperable data formats. Delta Sharing protocol solves this problem.

Figure 4-2 shows the Delta Sharing Protocol. The physical Delta table exists as a single-source of truth and the introduction of the Delta sharing server adds the missing access controls and governance required to provide a safe and reliable exchange of data.

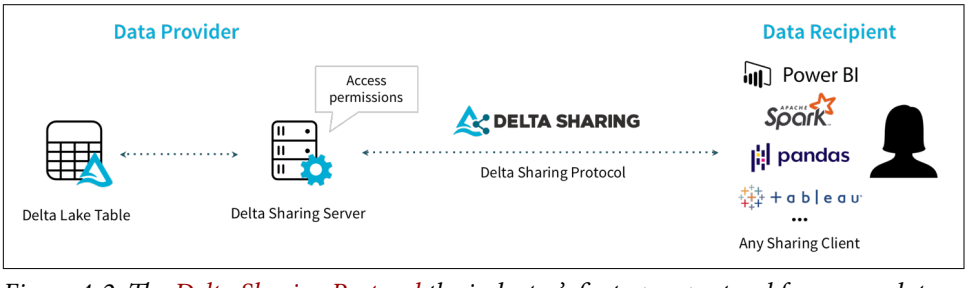


Figure 4-2. The Delta Sharing Protocol the industry's first open protocol for secure data sharing, making it simple to share data with other organizations regardless of which computing platforms they use

Using the Delta Sharing Protocol enables internal or external stakeholders secure direct-access to Delta tables. This removes the operational costs incurred when exporting data, while saving time, money, and engineering sanity while providing a shared source-of-truth that is platform agnostic.

The general capabilities provided by the Delta protocol support the foundational capabilities required by the data lakehouse. Now it is time for us to shift gears and look more specifically at architecting for data quality within the lakehouse using a purpose driven, layered data architecture called the *Medallion Architecture*.

The Medallion Architecture

Data in flight is messy, as it arrives–in all shapes, sizes and with varying degrees of accuracy and completeness. Accepting that not all data will adhere to the myriad end-user expectations, existing data contracts and established data quality checks,

arrive on time—or ever is key to addressing these data quality problem. These challenges place a high degree of pressure on data engineering teams to continuously deliver across a dynamic landscape of subjective and objective requirements, and borne from this collective toil came the Medallion Architecture.

The Medallion Architecture is a data design pattern used to logically organize data in the lakehouse. This is accomplished using series of isolated data layers to provide a framework for progressively refining datasets. Figure 4-4 shows a high-level view of the architecture, with data flowing from *batch* or *streaming* sources across a variable lineage from the point of initial ingestion (bronze), across multiple processing and enhancement phases, or stages.

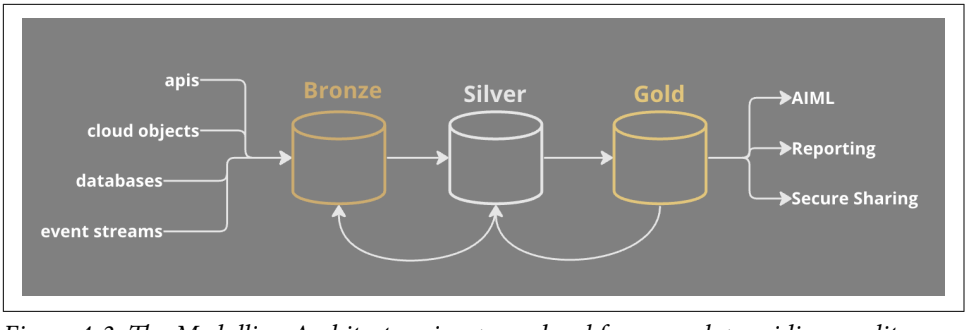


Figure 4-3. The Medallion Architecture is a procedural framework providing quality gates and tiers from the point of ingestion and onwards toward the purpose-built curated data product.

The Medallion Architecture provides a flexible framework for dealing with progressive enhancement of data in a structured way. It is worth pointing out that while it is common to see three tiers, there is no rule stating that all use cases require three tiers (bronze, silver, gold). It may be that more mature data practitioners will have a two-tier system where golden tables are joined with other golden tables to create even more golden tables. So the separation between silver and gold, or bronze and silver may be fuzzy at times. The key reason for having a three-tiered framework enables you to have a place to recover, or fall back on, when things go wrong, or requirements change.

Exploring the Bronze Layer

The bronze layer represents the initial point for our data lineage within the Lakehouse. A common practice here is to apply minimal transformations (if any) on the data. There are the transformations that can't be ignored, like converting the source format into a compatible type for writing to Delta Lake. The result of the minimal

transformations approach means we leave the option open for reprocessing this raw data to support additional use cases[3], or modified requirements in the future.

Bronze Layer is for Minimal Augmentation

The most important requirement of the bronze layer is to transform source data for writing into Delta Lake. When taking a minimal augmentation approach, it is also worth exploring ways to simplify and even automate this initial ingestion step. Using open data protocols that are interoperable with the DataFrame APIs–for example by using type-safe, binary serializable exchange formats like Apache Avro or Google Protocol Buffers–mean we can spend more time solving better problems than ingestion. For a small number of tables, it is arguable to ignore automation, but as the surface area increases, ignoring automation is simply bad for engineering mental health.

Minimal Transformations and Augmentation

Because we are ingesting data as close to "raw" as possible, we need to remember to maintain a limited schema and do as little to transform the data as possible. Let's use a concrete example. Say we are reading data from a streaming source like Kafka and want to capture the topic name, binary key and value, as well as the timestamp for each record and write them into a Delta Lake table. These properties all exist in the Kafka DataFrame structure (if we are using the KafkaSource api's with Spark) and can be extracted with the kafka-delta-ingest library (first explored in Chapter 5) as well.

Example 4-3 (*ch11/notebooks/medallion_bronze.ipynb)* is a concise example of minimal transformation and augmentation.

Example 4-3. Shows a simple bronze-style pipeline reading from Kafka, applying minimal transformations, and writing the data out to Delta.

```
% reader_opts: Dict[str, str] = …
  writer_opts: Dict[str, str] = …
  bronze_layer_stream = (
    spark.readStream
    .options(**reader_opts)
    .format("kafka").load()
    .select(col("key"),col("value"),col("topic"),col("timestamp"))
    .withColumn("event_date", to_date(col("timestamp")))
    .writeStream
    .format('delta')
```

3 Remembering that anything containing user data must be captured and processed according to the end-user agreed upon consent and according to data governance by-laws and standards.

```
    .options(**writer_opts)
    .partitionBy("event_date")
)
streaming_query = bronze_layer.toTable(...)
```

The extreme minimal approach applied in Example 4-3 takes only the information needed to preserve the data as close to its raw form as possible. This technique puts the onus on the silver layer to extract and transform the data from the `value` column.

While we are creating a minor amount of additional work, this bare bones approach enables the future ability to reprocess (reread) the raw data as it landed from Kafka without worrying about the data expiring (which can lead to data loss). Most data retention policies for delete in Kafka are between 24 hours and 7 days.

In the case where we are reading from an external database, like Postgres, the minimum schema is simply the table DDL. We already have explicit guarantees and row-wide expected behavior given the *schema-on-write* nature of the database, and therefor we can simplify the work required in the silver layer when compared to the example shown in Example 4-3.

As a rule of thumb, if the data source has a *type-safe* schema (avro, protobuf), or the data source implements *schema-on-write*, then we will typically see a significant reduction in the work requried in the bronze layer. This doesn't mean we can blindly write directly to silver either since the the bronze layer is the first guardian blocking unexpected or corrupt rows of data from its progression towards gold. In the case where we are importing non type-safe data — as seen with csv or json data — the bronze tier is incredibly important to weed out corrupt and otherwise problematic data.

Guarding the Bronze Layer with Permissive Mode in Spark

Example 4-4 shows a technique called permissive passthrough with Spark. This option allows us to add a gating mechanism using a predefined (consistent) schema to block corrupt data, while preserving the non-conformant rows for debugging.

Example 4-4. Preventing Bad Data with Permissive Passthrough

```
% from pyspark.sql.types import StructType, StructField, StringType
known_schema: StructType = (
  StructType.fromJson(...)
  .add(StructField('_corrupt', StringType(), True, {
    'comment': 'invalid rows go into _corrupt rather than simply being dropped'
}))
happy_df = (
  spark.read.options(**{
    "inferSchema": "false",
    "columnNameOfCorruptRecord": "_corrupt",
    "mode": "PERMISSIVE",
```

```
})
.schema(known_schema)
.json(...)
```

1. We begin by loading a known schema using the `StructType.fromJson` method, we could just as easily have manually built the schema using the `StructType().add(...)` pattern.
2. We then append the `_corrupt` field to our schema. This will provide a container for our bad data to sit. Think of this like either the _corrupt column is null or it contains a value. The data can then be read using a filter `where(col("_corrupt").isNotNull())` or the inverse to separate the good from the bad.
3. We then apply the reader options: `inferSchema:false`, `mode:Permissive`, `columnNameOfCorruptRecord:_corrupt`. By turning off schema inference we opt-into schema changes only by explicitly providing an updated schema. This means no runtime surprises. Schema inference is a powerful technique that scans (samples) a large number of rows of semi-structured data (like csv or json) to generate what it believes to be a stable StructType (schema). The problem with schema inference is it doesn't understand the historical structure of the data, and is limited to generating assumptions based on what it is provided in an initial batch.

The technique from Example 4-4 can be applied to streaming transforms just as easily using the `from_json` native function which is located in the `sql.functions` package (`pyspark.sql.functions.*, spark.sql.functions.*`). This means we can test things in batch, and then turn on the streaming firehose, understanding the exact behavior of our ingestion pipelines even in the inconsistent world of semi-structured data.

Summary

While the bronze layer may feel limited in scope and responsibility, it plays an incredibly important role in debugging, recovery, and as a source for new ideas in the future. Due to the raw nature of the bronze layer tables, it is also unadvisable to broadcast the availability of these tables widely. There is nothing worse than getting paged or called into an incident for issues arising from the misuse of raw tables.

Exploring the Silver Layer

With the bronze layer representing the initial point of lineage in the medallion architecture, the silver layer represents the point where raw data is filtered, cleaned and dressed up, and even augmented by joining across one or many other tables. If the bronze layer is data in its infancy, the silver layer is data in its teenage years, and just like we all were in our teens, our data coming of age story has its ups and downs.

Used for Cleaning and Filtering Data

Depending on the source of the data that first landed in the bronze layer, we may be in for a wild ride. Just like no two people are exactly alike, the general consistency and baseline quality of each data source can vary wildly. This is where initial cleaning and filtering come into play.

We clean up our data to normalize and present a consistent source of reliable data for downstream consumption. Our downstream consumers may be ourselves, teams within our organization, or even external stakeholders. On one extreme, we may be extracting and decoding binary data that originated from streaming sources–like Kafka–to convert from avro or protobuf, then applying additional transformations on the resulting data. The output of our pipeline may result in nested or flattened rows.

It is also normal to be filtering or even dropping some columns at this point. In Example 4-4, we saw the inclusion of the _corrupt column. This information isn't necessary for consumption in the silver or golden layer of the medallion architecture. These are only provided to support data preservation techniques in the bronze layer and as a form of communication between engineers.

It isn't uncommon for engineers to provide `_*` columns like `_corrupt` or `_debug` that contain simple information or more specific structs or maps. This technique can also be used to carry observability metadata or additional context for reporting purposes.

Example 4-5 provides a continuation to Example 4-4, showing how we would pickup reading from the bronze Delta table, then filter, drop, and transform rows for receipt into the cleansed silver tables.

Example 4-5. Filtering, Dropping, and Transformations. All the things needed for writing to Silver.

```
% medallion_stream = (
  delta_source.readStream.format("delta")
  .options(**reader_options)
  .load()
  .transform(transform_from_json)
  .transform(transform_for_silver)
  .writeStream.format("delta")
  .options(**writer_options))
  .option('mergeSchema': 'false'))
  streaming_query = (
    medallion_stream
    .toTable(f"{managed_silver_table}"))
```

The pipeline shown in Example 4-5 reads from the bronze delta table (from Example 4-3), decodes the binary data received (from the value column), while also enabling permissive mode which we explored in Example 4-4.

```
def transform_from_json(input_df: DataFrame) -> DataFrame:
    return input_df.withColumn("ecomm",
        from_json(
            col("value").cast(StringType()),
            known_schema,
            options={
                'mode': 'PERMISSIVE',
                'columnNameOfCorruptRecord': '_corrupt'
            }
        ))
```

Then a second transformation is required as we make preparations for writing into the silver layer. This is minor secondary transformation removing any corrupt rows, and applying aliasing to declare the ingestion data and timestamp which could be different from the event timestamp and date.

```
def transform_for_silver(input_df: DataFrame) -> DataFrame:
    return (
        input_df.select(
            col("event_date").alias("ingest_date"),
            col("timestamp").alias("ingest_timestamp"),
            col("ecomm.*")
        )
        .where(col("_corrupt").isNull())
        .drop("_corrupt"))
```

After the transformations are taken care of, we write the data out to our silver Delta table. We also explicitly set the mergeSchema:false. While this is the default behavior, it is an important call out since it flags to other engineers what the expected behavior is, and to ensure accidental columns don't mistakenly make their way to silver from bronze. We covered alternatives to automatic schema evolution using ALTER TABLE in chapter 6.

Regardless of why we clean and filter the bronze data, the results of our efforts provide our stakeholders with more consistent and reliable data to power their myriad use cases. We can consider the silver layer to be the first stable layer in the medallion architecture.

Establishes a Layer for Augmenting Data

There is no rule stating that a silver table must read from a bronze table. In fact, it is common for the silver layer to be used to join from one or many silver and even golden tables. For example, if the results of cleaning and filtering one of our bronze tables can be used to power multiple additional use cases, then we can save ourselves both time and additional complexity by reusing the fruits of our internal teams and external partners' labor. Being able to view the lineage visually between bronze, silver, and gold can help provide additional context as the number of tables and views, data products and owners, naturally grows over time.

Enables Data Checks and Balances

Delta provides capabilities for column based constraints to enhance the functionality that can't be provided with simple schema enforcement alone— Schema Enforcement and Evolution was covered in Chapter 6.

With column level constraints, we can enforce more complex rules directly at the table level by applying predicates in the form of `CHECKs`.

```
ALTER TABLE <tablename>
ADD CONSTRAINT <name>
CHECK <sql-predicate>
```

The upside here is that we can guarantee that the data in our table will never not meet the constraint criteria. The downside is that if any row doesn't meet the constraint's check, a `DeltaInvariantViolationException` will be thrown, short-circuiting the job.

Data Quality frameworks can help simplify table constraints by separating the rules from the underlying physical table definition. Popular frameworks in the open-source world are Great Expectations, Spark Expectations, and Delta Live Table (DLT) expectations–which is a paid offering by Databricks. Data quality is an important part of DataOps, and it can help to block bad data before it leaves a specific layer within the Medallion Archtiecture.

Summary

Remember, as data engineers we need to act like owners and provide excellent customer service to our data stakeholders. The earlier in the refinement process we can establish good quality gates, the happier our downstream data consumers will be.

Exploring the Gold Layer

The gold layer is the most mature data layer in the medallion architecture. Just like silver was on the path to being all grown up, but not quite, data in the gold layer has undergone multiple transformations, and has been specifically curated and has a specific place in the data world. This is because data in the gold layer is curated, and purpose built to solve explicit intended goals. If bronze represents data as an infant, and silver is a teenager, then golden tables represent data in its late thirties or early forties — or at a point where they have established a concrete identity.

Establishes High-Trust and High Consistency

While the analogy to data as people at different points in their lives might not be accurate, as a mental model data it works. Data in the golden layer is much less likely to change drastically from day to day in the same way that our personalities, wants,

and wishes change with a slower pace as we age. Example 4-6 explores generating topN reports from the transformations out of our silver layer (Example 4-5).

Example 4-6. Creating Intentional Tables for Business-Level Consumption

```
% pyspark
silver_table = spark.read.format("delta")...
top5 = (
  silver_table
  .groupBy("ingest_date", "category_id")
  .agg(
    count(col("product_id")).alias("impressions"),
    min(col("price")).alias("min_price"),
    avg(col("price")).alias("avg_price"),
    max(col("price")).alias("max_price")
  )
  .orderBy(desc("impressions"))
  .limit(5))
(top5
  .write.format("delta")
  .mode("overwrite")
  .options(**view_options)
  .saveAsTable(f"gold.{topN_products_daily}"))
```

The prior example shows how to do daily aggregations. It is typical for reporting data to be stored in the gold layer. This is the data we (and the business) show care about. It is our jobs to ensure that we provide purpose built tables (or views) to ensure business critical data is available, reliable, and accurate.

For foundational tables–and really with any business critical data–surprise changes are upsetting and may lead to broken reporting as well as inaccurate runtime inference for machine learning models. This can cost the company more than just money, it can be the difference between retaining customers and reputation in a highly competitive industry.

Summary

The gold layer can be implemented using physical tables or virtual tables (views). This provides us with ways of optimizing our curated tables that result in either a full physical table when not using a view, and simple metadata providing any filters, column aliases, or join criteria required when interacting with the virtual table. The performance requirements will ultimately dictate the usage of tables vs views, but in many cases a view is good enough to support the needs of many gold layer use cases.

Now that we've explored the medallion architecture, the last stop on our journey will be to dive into patterns for decreasing the level of effort and time requirements from the point of data ingestion to the time when the data becomes available for consumption for downstream stakeholders at the gold edge.

Streaming Medallion Architecture

Earlier we learned that the Medallion Architecture is a data design pattern enabling us to solve common data problems encountered with any data in flight. The problems being:

- Lack of replay or recovery (which is solved with the bronze layer)
- Broken column-level expectations (which is solved with the Delta protocol and turning off mergeSchema, and ignoring overwriteSchema unless needed as a last resortl).
- Problems with column specific data quality and correctness. Which can be solved with constraints, or by using utility libraries like spark-expecations, or Delta Live Tables with `@dlt.expect`).

While we've already looked at patterns to refine data using the medallion architecture to remove imperfections, adhere to explicitly defined schemas, and provide data checks and balances, what we didn't cover was how to provide a seamless flow for transformations from bronze to silver and silver to gold.

Time tends to get in the way more often than not — with too little time, there is not enough information to make informed decisions, and with too much time, there is a tendency to become complacent and sometimes even a little bit lazy. Time is much more of a goldilocks problem, especially when we concern ourselves with reducing the end-to-end latency for data traversing our lakehouse. In the next section, we will look at common patterns for reducing the latency of each tier within the medallion architecture, focusing on end-to-end streaming.

Reducing End to End Latency within your Lakehouse

As we've seen across the book, the Delta protocol supports both batch or streaming access to tables. We can deploy our pipelines to take specific steps ensuring that the datasets that are output meet both our quality standards and result in the ability to trust the upstream sour es of data, enabling us to drastically reduce the end-to-end latency from data ingestion (bronze) on through (silver), and ultimately into the hands of the business or data product owners in the (gold) layer.

By crafting our pipelines to block and correct data quality problems before they become more widespread, we can use the lessons learned across Example 4-3 through Example 4-5 to stitch together end-to-end streaming workflows.

Figure 4-4 provides an example of the streaming workflow. Data arrives from our Kafka topic, as we saw in Example 4-3. The dataset is then appended to our bronze delta table (ecomm_raw) which enables us to pick up the incremental changes in our silver application. The example providing the transformations was shown in

Example 4-5. Lastly, we either create and replace temporary views (or materialized views in Databricks), or create another golden application with the responsibility of periodically ingesting data from ecomm_silver to produce purpose built tables or views. Extending the pattern seen in Example 4-6, we can stitch together an end-to-end pipeline that incrementally ingests from its direct upstream allowing us to trace the lineage of transformations all the way back to the initial point of inception (kafka).

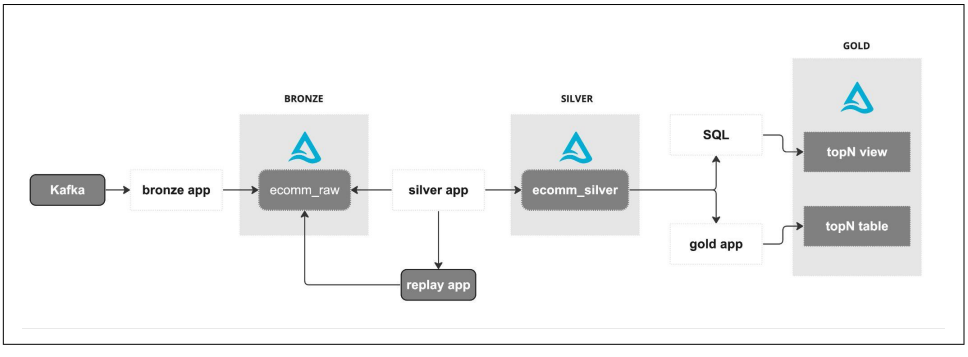


Figure 4-4. Streaming Medallion Architecture as viewed from the workflow level.

There are many ways to orchestrate end-to-end worlkflows, using scheduled jobs, or full-fledged frameworks like Apache Airflow, Databricks Workflows or Delta Live Tables. The end result provides us with reduced latency from the edge all the way to our most important, business-critical, golden tables.

Summary

This chapter introduced the architectural tenets of the modern Lakehouse architecture and showed how Delta Lake can be used for foundational support for this mission.

Built on open-standards, with open-protocols and formats, supporting ACID transactions, table-level time-travel, simplified interoperability with UniForm, as well as out-of-the-box data sharing protocols to simplify the exchange of data both for internal and external stakeholders. We skimmed the surface of the Delta protocol and learned more about the invariants that provide us with rules of engagement as well as table-level guarantees, by looking at how schema-on-write, and schema enforcement protect our downstream data consumers from accidental leakage of corrupt or low quality data.

We then looked at how the medallion architecture can be used to provide a standard framework for data quality, and how each layer is utilized across the common bronze-silver-gold model.

The quality gating pattern enables us to build a consistent data strategy and provide guarantees and expectations based on a model of incremental quality from bronze (raw) to silver (cleansed and normalized) up to gold (curated and purpose driven). How data flows within the lakehouse, between these gates enables a higher level of trust within the lakehouse, and even allows us to reduce the end-to-end latency by enabling end-to-end streaming in the lakehouse.