topic will not be lost unless a critical failure occurs across the entire cluster—and if that happens, then we can only hope that a good disaster recovery (DR) plan has been set up to mitigate the risk of data loss.

Last, there are some invariants that make Kafka invaluable, especially for time-series data. Each Kafka topic has the ability to guarantee synchronous insertion within each topic partition without requiring the entire topic to coordinate insertion order across all partitions. This means that when the cluster is running in a normal state, you can trust the event order, which reduces stream processing complexity. This probably goes without saying, but not requiring expensive rereads and sorting when working with time-series data paves the path to analysis peace of mind when it comes to working with data supplied via a Kafka source into our Delta tables. Now, back to the `kafka-delta-ingest` connector.

The `kafka-delta-ingest` connector provides a daemon that simplifies the common step of streaming Kafka data into our Delta Lake tables. Getting started can also be done in four easy steps:

1. Install Rust.
2. Build the project.
3. Create your Delta table.
4. Run the ingestion flow.

Install Rust

This can be done using the `rustup` toolchain:

```
% curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Once `rustup` is installed, running `rustup update` will ensure we are on the latest stable version of Rust available.

Build the Project

This step ensures we have access to the source code.

Using `git` on the command line, simply clone the connector:

```
% git clone git@github.com:delta-io/kafka-delta-ingest.git \
  && cd kafka-delta-ingest
```

Set up your local environment

From the root of the project directory, run the Docker setup utility:

```
% docker compose up setup
```

After the setup flow completes, we have `localstack` (which runs a local Amazon Web Services [AWS] instance), `kafka` (redpandas), and the `confluent schema registry`, as well as `azurite` for local Azure Storage. Having access to run our cloud-based workflows locally greatly reduces the pain of moving from the design phase of our applications into production.

Build the connector

Rust uses `cargo` for dependency management and to build your project. The `cargo` utility is installed for us by the `rustup` toolchain. From the project root, execute the following command:

```
% cargo build
```

At this point we'll have the connector built and the Rust dependencies installed, and we can choose to either run the examples or connect to our own Kafka brokers and get started. The last section on using `kafka-delta-ingest` will cover running the end-to-end ingestion.[2]

Run the Ingestion Flow

For the ingestion application to function, we need to have two things—a source Kafka topic and a destination Delta table. There is a caveat with the generation of the Delta table, especially if you are familiar with Apache Spark–based Delta workflows: we must first create our destination Delta table in order to successfully run the ingestion flow.

There are a handful of variables that can modify the `kafka-delta-ingest` application. We will begin with a tour of the basic environment variables in Table 4-1, and then Table 4-2 will provide us with some of the runtime variables (args) that are available to us when using this connector.

Table 4-1. Using environment variables

Environment variable	Description	Default
`KAFKA_BROKERS`	The Kafka broker string; can be used to overwrite the location of the brokers for local testing, or for triage and recovery applications	`local host:9092`
`AWS_ENDPOINT_URL`	Used to run local tests via LocalStack	`none`
`AWS_ACCESS_KEY_ID`	Used to provide the application identity	`test`
`AWS_SECRET_ACCESS_KEY`	Used to authenticate the application identity	`test`
`AWS_DEFAULT_REGION`	Can be useful for running LocalStack or for bootstrapping separate S3 bucket locations	`none`

2 The full ingestion flow application is available in the book's GitHub repository under *ch04/rust/kafka-delta-ingest*.

Table 4-2. Using command-line arguments

Argument	Description	Example
`allowed_latency`	Used to specify how long to fill the buffer and await new data before processing	`--allowed_latency 60`
`app_id`	Used to run local tests via LocalStack	`--app_id ingest-app`
`auto_offset_reset`	Can be `earliest` or `latest`; this affects whether you read from the tail or the head of the Kafka topic	`--auto_offset_reset earliest`
`checkpoints`	Will record the Kafka metadata for each processed ingestion batch; this allows for you to easily stop the application and start it back up again without data loss (unless Kafka deletes the data between runs, which can be checked in the delete policy for the topic)	`--checkpoints`
`consumer_group_id`	Provides a unique consumer name for the Kafka brokers; using the group ID, the brokers can distribute the processing of a large topic among multiple consumer applications without duplication	`--consumer_group_id ecomm-ingest-app`
`max_mes sages_per_batch`	Use this option to throttle the number of messages per application tick (loop); this can help keep your applications from running out of memory if there is an unexpected increase in the volume of the records being written to the topic	`--max_messages_per _batch 1600`
`min_bytes_per_file`	Use this option to ensure that the underlying Delta table doesn't become riddled with small files	`--min_bytes_per_file 64000000`
`kafka`	Used to pass the Kafka broker string to the ingest application	`--kafka 127.0.0.1:29092`

Now all that is left to do is to run the ingestion application. If we are running the application using our environment variables, then the simplest command would provide the Kafka topic and the Delta table location. The command signature is as follows:

```
% cargo run ingest <topic> <delta_table_location>
```

Next, we'll see a complete example:

```
% cargo run \
  ingest ecomm.v1.clickstream file:///dldg/ecomm-ingest/ \
  --allowed_latency 120 \
  --app_id clickstream_ecomm \
  --auto_offset_reset earliest \
  --checkpoints \
  --kafka 'localhost:9092' \
  --max_messages_per_batch 2000 \
  --transform 'date: substr(meta.producer.timestamp, `0`, `10`)' \
  --transform 'meta.kafka.offset: kafka.offset' \
  --transform 'meta.kafka.partition: kafka.partition' \
  --transform 'meta.kafka.topic: kafka.topic'
```

With the simple steps we've explored together, we can now easily ingest data from our Kafka topics. We have set ourselves up for success by ensuring that the folks consuming our data do so with a high level of reliability. The more we can automate, the lower the chance of human error getting in the way and resulting in incidents or in the dreaded data loss.

In the next section, we are going to explore Trino. Both prior examples play nice alongside the Trino ecosystem, as they reduce the level of effort to ingest and transform data prior to writing solid tables that can be analyzed through more traditional SQL tooling.

Trino

Trino is a distributed SQL query engine designed to seamlessly connect to and interoperate with a myriad of data sources. It provides a connector ecosystem that supports Delta Lake natively.

Trino is the community-supported fork of the Presto project and was initially designed and developed in-house at Facebook. Trino was known as PrestoSQL before it was given its present name in 2020.

To learn more about Trino, check out *Trino: The Definitive Guide* (O'Reilly).

Getting Started

All we need to get started with Trino and Delta Lake is any version of Trino newer than version 373. At the time of writing, Trino is currently at version 459.

Connector requirements

While the Delta connector is natively included in the Trino distribution, there are still additional things we need to consider to ensure a frictionless experience.

Connecting to OSS or Databricks Delta Lake:

- Delta Tables written by Databricks Runtime 7.3 LTS, 9.1 LTS, 10.4 LTS, 11.3 LTS, and >= 12.2 LTS.
- Deployments using AWS, HDFS, Azure Storage, and Google Cloud Storage (GCS) are fully supported.
- Network access from the coordinator and workers to the Delta Lake storage.
- Access to the Hive Metastore (HMS).

- Network access to HMS from the coordinator and workers. Port 9083 is the default port for the Thrift protocol used by HMS.

Working locally with Docker:

- Trino Image
- Hive Metastore (HMS) service (standalone)
- Postgres or supported relational database management system (RDBMS) to store the HMS table properties, columns, databases, and other configurations (can point to managed RDBMS like RDS for simplicity)
- Amazon S3 or MinIO (for object storage for our managed data warehouse)

The Docker Compose configuration in Example 4-10 shows how to configure a simple Trino container for local testing.

Example 4-10. Basic Trino Docker Compose

```
services:
  trinodb:
    image: trinodb/trino:426-arm64
    platform: linux/arm64
    hostname: trinodb
    container_name: trinodb
    volumes:
      - $PWD/etc/catalog/delta.properties:/etc/trino/catalog/delta.properties
      - $PWD/conf:/etc/hadoop/conf/
    ports:
      - target: 8080
        published: 9090
        protocol: tcp
        mode: host
    environment:
      - AWS_ACCESS_KEY_ID=$AWS_ACCESS_KEY_ID
      - AWS_SECRET_ACCESS_KEY=$AWS_SECRET_ACCESS_KEY
      - AWS_DEFAULT_REGION=${AWS_DEFAULT_REGION:-us-west-1}
    networks:
      - dldg
```

The example in the next section assumes we have the following resources available to us:

- *Amazon S3 or MinIO (bucket provisioned, with a user, and roles set to allow read, write, and delete access).* Using local MinIO to mock S3 is a simple way to try things out without any upfront costs. See the `docker compose` examples in the book's GitHub repository under *ch04/trinodb/*.

- *MySQL or PostgreSQL.* This can run locally, or we can set it up on our favorite cloud provider; for example, AWS RDS is a simple way to get started.
- *Hive Metastore (HMS) or Amazon Glue Data Catalog.*

Next, we'll learn how to configure the Delta Lake connector so that we can create a Delta catalog in Trino. If you want to learn more about using the Hive Metastore (HMS), including how to configure the *hive-site.xml*, how to include the required JARs for S3, and how to run HMS, you can read through "Running the Hive Metastore". Otherwise, skip ahead to "Configuring and Using the Trino Connector" on page 79.

Running the Hive Metastore

If you already have a reliable metastore instance setup, you can modify the connection properties to use that instead. If you are looking to have a local setup, then we can begin with the creation of the *hive-site.xml*, which is shown in Example 4-11 and which is required to connect to both MySQL and Amazon S3.

Example 4-11. hive-site.xml for HMS

```
<configuration>
  <property>
    <name>hive.metastore.version</name>
    <value>3.1.0</value>
  </property>
  <property>
    <name>javax.jdo.option.ConnectionURL</name>
    <value>jdbc:mysql://RDBMS_REMOTE_HOSTNAME:3306/metastore</value>
  </property>
  <property>
    <name>javax.jdo.option.ConnectionDriverName</name>
    <value>com.mysql.cj.jdbc.Driver</value>
  </property>
  <property>
    <name>javax.jdo.option.ConnectionUserName</name>
    <value>RDBMS_USERNAME</value>
  </property>
  <property>
    <name>javax.jdo.option.ConnectionPassword</name>
    <value>RDBMS_PASSWORD</value>
  </property>
  <property>
    <name>hive.metastore.warehouse.dir</name>
    <value>s3a://dldgv2/delta/</value>
  </property>
   <property>
      <name>fs.s3a.access.key</name>
      <value>S3_ACCESS_KEY</value>
```

```
   </property>
   <property>
      <name>fs.s3a.secret.key</name>
      <value>S3_SECRET_KEY</value>
   </property>
  <property>
    <name>fs.s3.path-style-access</name>
    <value>true</value>
  </property>
  <property>
    <name>fs.s3a.impl</name>
    <value>org.apache.hadoop.fs.s3a.S3AFileSystem</value>
  </property>
</configuration>
```

The configuration provides the nuts and bolts we need to access the metadata database, using the JDBC connection URL, username, and password properties, as well as the data warehouse, using the `hive.metastore.warehouse.dir` and the properties prefixed with `fs.s3a`.

Next, we need to create a Docker Compose file to run the metastore, which we do in Example 4-12.

Example 4-12. Docker Compose for the Hive Metastore

```
version: "3.7"

services:
  metastore:
    image: apache/hive:3.1.3
    platform: linux/amd64
    hostname: metastore
    container_name: metastore
    volumes:
      - ${PWD}/jars/hadoop-aws-3.2.0.jar:/opt/hive/lib/
      - ${PWD}/jars/mysql-connector-java-8.0.23.jar:/opt/hive/lib/
      - ${PWD}/jars/aws-java-sdk-bundle-1.11.375.jar:/opt/hive/lib/
      - ${PWD}/conf:/opt/hive/conf
    environment:
      - SERVICE_NAME=metastore
      - DB_DRIVER=mysql
      - IS_RESUME="true"
    expose:
      - 9083
    ports:
      - target: 9083
        published: 9083
        protocol: tcp
        mode: host
    networks:
      - dldg
```

With the metastore running, we are now in the driver's seat to understand how to take advantage of the Trino connector for Delta Lake.

Configuring and Using the Trino Connector

Trino uses configuration files called *catalogs*. They are used to describe the catalog type (`delta_lake`, `hive`, and many more), and they enable us to tune a given catalog to optimize for reads and writes and to manage additional connector configurations. The minimum configuration for the Delta connector requires an addressable Hive Metastore location `thrift:hostname:port` (if using HMS). The other supported catalog at the time of writing is AWS Glue.

The code in Example 4-13 configures the connector pointing to the Hive Metastore.

Example 4-13. The Delta Lake connector properties

```
connector.name=delta_lake
hive.metastore=thrift
hive.metastore.uri=thrift://metastore:9083
delta.hive-catalog-name=metastore
delta.compression-codec=SNAPPY
delta.enable-non-concurrent-writes=true
delta.target-max-file-size=512MB
delta.unique-table-location=true
delta.vacuum.min-retention=7d
```

The property `delta.enable-non-concurrent-writes` must be set to `true` if there is a chance of multiple writers making nonatomic changes to a table. This is most often the case with Amazon S3; setting the property to `true` ensures that the table remains consistent.

The property file from Example 4-13 can be saved as *delta.properties*. As long as the file is copied into the Trino catalog directory (*/etc/trino/catalog/*), then we'll be able to read, write, and delete from the underlying `hive.metastore.warehouse.dir`, and do a whole lot more.

Let's look at what's possible.

Using Show Catalogs

Using `show catalogs` is a simple first step to ensure that the Delta connector has been configured correctly and shows up as a resource:

```
trino> show catalogs;

Catalog
---------
```

```
 delta
 ...
(6 rows)
```

As long as we see `delta` in the list, we can move on to creating a schema. This confirms that our catalog is correctly configured.

Creating a Schema

The notion of a schema is a bit overloaded. We have schemas that represent the structured data describing the columns of our tables, but we also have schemas representing traditional databases. Using `create schema` enables us to generate a managed location within our data warehouse that can act as a boundary for access and governance, as well as to separate the physical table data among bronze, silver, and golden tables. We'll learn more about the medallion architecture in Chapter 9, but for now let's create a `bronze_schema` to store some raw tables:

```
trino> create schema delta.bronze_schema;

CREATE SCHEMA
```

If we are greeted by an exception rather than seeing `CREATE SCHEMA` returned, then it's likely due to permissions issues writing to the physical warehouse. The following is an example of such an exception:

```
Query 20231001_182856_00004_zjwqg failed: Got excep-
tion: java.nio.file.AccessDeniedException s3a://com.new-
front.dldgv2/delta/bronze_schema.db: getFileStatus
on s3a://com.newfront.dldgv2/delta/bronze_schema.db:
com.amazonaws.services.s3.model.AmazonS3Exception: For-
bidden (Service: Amazon S3; Status Code: 403;
```

We can fix the problem by modifying our identity and access management (IAM) permissions or by ensuring we are using the correct IAM roles.

Show Schemas

This command allows us to query a catalog to view available schemas:

```
trino> show schemas from delta;

Schema
--------------------
 default
 information_schema
 bronze_schema
(3 rows)
```

If the schema we are looking for exists, then we are ready to move on to creating some tables.

Working with Tables

Table compatibility between the Trino and Delta ecosystems requires that we follow some guidelines. We'll look at data type interoperability and then create a table, add some rows, and view the Delta metadata, including the transaction history and tracking changes for Change Data Feed (CDF)–enabled tables. We'll conclude by looking at table optimization and vacuuming.

Data types

There are a few caveats to creating tables using Trino, especially when it comes to type mapping differences between Trino and Delta Lake. The table shown in Table 4-3 can be used to ensure that the appropriate types are used and to steer clear of incompatibility if our aim is interoperability.

Table 4-3. Delta to Trino type mapping

Delta data type	Trino data type
`BOOLEAN`	`BOOLEAN`
`INTEGER`	`INTEGER`
`BYTE`	`TINYINT`
`SHORT`	`SMALLINT`
`LONG`	`BIGINT`
`FLOAT`	`REAL`
`DOUBLE`	`DOUBLE`
`DECIMAL(p,s)`	`DECIMAL(p,s)`
`STRING`	`VARCHAR`
`BINARY`	`VARBINARY`
`DATE`	`DATE`
`TIMESTAMPNTZ (TIMESTAMP_NTZ)`	`TIMESTAMP(6)`
`TIMESTAMP`	`TIMESTAMP(3) WITH TIME ZONE`
`ARRAY`	`ARRAY`
`MAP`	`MAP`
`STRUCT(...)`	`ROW(...)`

CREATE TABLE options

The supported table options (shown in Table 4-4) can be applied to our table using the `WITH` clause of the `CREATE TABLE` operation. This enables us to specify options on our tables that Trino wouldn't otherwise understand. In the case of partitioning,

Trino won't automatically discover partitions, which could be a problem when it comes to the performance of SQL queries.

Table 4-4. `CREATE TABLE` options

Property name	Description	Default
`location`	Filesystem location uniform resource identifier (URI) for table. *This option is deprecated.*	Will use a managed table mapped to the location of the `hive.metastore.warehouse.dir` or Glue Catalog equivalent
`partitioned_by`	Columns to partition the table by	No partitions
`checkpoint_interval`	How often to commit changes to Delta Lake	Every 10 for open source software (OSS), and every 100 for Databricks (DBR)
`change_data_feed_enabled`	Track changes made to the table for use in change data capture (CDC)/Change Data Feed (CDF) applications	false
`column_mapping_mode`	How to map the underlying Parquet columns: options (ID, name, none)	none

Creating tables

We can create tables using the longform `<catalog>.<schema>.<table>` syntax, or the shortform syntax `<table>` after calling `use delta.<schema>`. Example 4-14 provides an example using the shortform create.

Example 4-14. Creating a Delta table with Trino

```
trino> use delta.bronze_schema;
CREATE TABLE ecomm_v1_clickstream (
  event_date DATE,
  event_time VARCHAR(255),
  event_type VARCHAR(255),
  product_id INTEGER,
  category_id BIGINT,
  category_code VARCHAR(255),
  brand VARCHAR(255),
  price DECIMAL(5,2),
  user_id INTEGER,
  user_session VARCHAR(255)
)
WITH (
    partitioned_by = ARRAY['event_date'],
    checkpoint_interval = 30,
    change_data_feed_enabled = false,
    column_mapping_mode = 'name'
);
```

The table generated using the DDL statement in Example 4-14 creates a managed table in our data warehouse that will be partitioned daily. The table structure represents the ecommerce data from the "Apache Flink" on page 61 section earlier in this chapter.

Listing tables

Using `show tables` will allow us to view the collection of tables within a given schema in the Delta catalog:

```
trino:bronze_schema> show tables;

Table
----------------------
 ecomm_v1_clickstream
(1 row)
```

Inspecting tables

If we are not the owners of a given table, we can use `describe` to learn about the table through its metadata:

```
trino> describe delta.bronze_schema."ecomm_v1_clickstream";

    Column     |     Type     | Extra | Comment
---------------+--------------+-------+---------
 event_date    | date         |       |
 event_time    | varchar      |       |
 event_type    | varchar      |       |
 product_id    | integer      |       |
 category_id   | bigint       |       |
 category_code | varchar      |       |
 brand         | varchar      |       |
 price         | decimal(5,2) |       |
 user_id       | integer      |       |
 user_session  | varchar      |       |
(10 rows)
```

Using INSERT

Rows can be inserted directly using the command line, or through the use of the Trino client:

```
trino> INSERT INTO delta.bronze_schema."ecomm_v1_clickstream"
    VALUES
        (DATE '2023-10-01', '2023-10-01T19:10:05.704396Z', 'view', ...),
        (DATE('2023-10-01'), '2023-10-01T19:20:05.704396Z', 'view', ...);
INSERT: 2 rows
```

Querying Delta tables

Using the `select` operator allows you to query your Delta tables:

```
trino> select event_date, product_id, brand, price
  -> from delta.bronze_schema."ecomm_v1_clickstream";

 event_date | product_id |  brand  | price
------------+------------+---------+--------
 2023-10-01 |   44600062 | nars    |  35.79
 2023-10-01 |   54600062 | lancome | 122.79
(2 rows)
```

Updating rows

The standard `update` operator is available:

```
trino> UPDATE delta.bronze_schema."ecomm_v1_clickstream"
    -> SET category_code = 'health.beauty.products'
    -> where category_id = 2103807459595387724;
```

Creating tables with selection

We can create a table using another table. This is referred to as `CREATE TABLE AS`, and it allows us to create a new physical Delta table by referencing another table:

```
trino> CREATE TABLE delta.bronze_schema."ecomm_lite"
    AS select event_date, product_id, brand, price
    FROM delta.bronze_schema."ecomm_v1_clickstream";
```

Table Operations

There are many table operations to consider for optimal performance, and for decluttering the physical filesystem in which our Delta tables live. Chapter 5 covers the common maintenance and table utility functions, and the following section covers what functions are available within the Trino connector.

Vacuum

The vacuum operation will clean up files that are no longer required in the current version of a given Delta table. In Chapter 5 we go into more detail about why vacuuming is required, as well as the caveats to keep in mind to support table recovery and rolling back to prior versions with time travel.

With respect to Trino, the Delta catalog property `delta.vacuum.min-retention` provides a gating mechanism to protect a table in case of an arbitrary call to vacuum with a low number of days or hours:

```
trino> CALL delta.system.vacuum('bronze_schema', 'ecomm_v1_clickstream', '1d');
```

```
Retention specified (1.00d) is shorter than the minimum retention configured
in the system (7.00d). Minimum retention can be changed with delta.vacuum.min-
retention configuration property or delta.vacuum_min_retention session property
```

Otherwise, the vacuum operation will delete the physical files that are no longer needed by the table.

Table optimization

Depending on the size of the table parts created as we make modifications to our tables with Trino, we run the risk of creating too many small files representing our tables. A simple technique to combine the small files into larger files is bin-packing `optimize` (which we cover in Chapter 5 and in the performance-tuning deep dive in Chapter 10). To trigger compaction, we can call `ALTER TABLE` with `EXECUTE`:

```
trino> ALTER TABLE delta.bronze_schema."ecomm_v1_clickstream" EXECUTE optimize;
```

We can also provide more hints to change the behavior of the `optimize` operation. The following will ignore files greater than 10 MB:

```
trino> ALTER TABLE delta.bronze_schema."ecomm_v1_clickstream"
    -> EXECUTE optimize(file_size_threshold => '10MB')
```

The following will only attempt to compact table files within the partition (`event_date = "2023-10-01"`):

```
trino> ALTER TABLE delta.bronze_schema."ecomm_v1_clickstream" EXECUTE optimize
WHERE event_date = "2023-10-01"
```

Metadata tables

The connector exposes several metadata tables for each Delta Lake table that contain information about their internal structure. We can query these tables to learn more about our tables and to inspect changes and recent history.

Table history

Each transaction is recorded in the `<table>$history` metadata table:

```
trino> describe delta.bronze_schema."ecomm_v1_clickstream$history";
        Column        |            Type             | Extra | Comment
----------------------+-----------------------------+-------+---------
 version              | bigint                      |       |
 timestamp            | timestamp(3) with time zone |       |
 user_id              | varchar                     |       |
 user_name            | varchar                     |       |
 operation            | varchar                     |       |
 operation_parameters | map(varchar, varchar)       |       |
 cluster_id           | varchar                     |       |
 read_version         | bigint                      |       |
```

```
 isolation_level      | varchar                    |       |
 is_blind_append      | boolean                    |       |
```

We can query the metadata table. Let's look at the last three transactions for our `ecomm_v1_clickstream` table:

```
trino> select version, timestamp, operation
    -> from delta.bronze_schema."ecomm_v1_clickstream$history";

 version |          timestamp          |  operation
---------+-----------------------------+--------------
       0 | 2023-10-01 19:47:35.618 UTC | CREATE TABLE
       1 | 2023-10-01 19:48:41.212 UTC | WRITE
       2 | 2023-10-01 23:01:13.141 UTC | OPTIMIZE
(3 rows)
```

Change Data Feed

The Trino connector provides functionality for reading Change Data Feed (CDF) entries to expose row-level changes between two versions of a Delta Lake table. When the `change_data_feed_enabled` table property is set to `true` on a specific Delta Lake table, the connector records change events for all data changes on the table:

```
trino> use delta.bronze_schema;
CREATE TABLE ecomm_v1_clickstream (
  ...
)
WITH (
    change_data_feed_enabled = true
);
```

Now each row of each transaction is recorded (with the operation type), enabling us to rebuild the state of a table or to walk through the changes to a table after a specific point in time.

For example, if we'd like to view all changes since version 0 of a table, we could execute the following:

```
trino> select event_date, _change_type, _commit_version, _commit_timestamp
from TABLE(
  delta.system.table_changes(
    schema_name => 'bronze_schema',
    table_name => 'ecomm_v1_clickstream',
    since_version => 0
  )
);
```

and view the changes made. In the example use case, we've simply inserted two rows:

```
 event_date | _change_type | _commit_version |      _commit_timestamp
------------+--------------+-----------------+-------------------------
 2023-10-01 | insert       |               1 | 2023-10-01 19:48:41.212 UTC
 2023-10-01 | insert       |               1 | 2023-10-01 19:48:41.212 UTC
(2 rows)
```

Viewing table properties

It is useful to be able to view the table properties associated with our tables. We can use the metadata table `<table>$properties` to view the associated Delta `TBLPROPERTIES`:

```
trino> select * from  delta.bronze_schema."ecomm_v1_clickstream$properties";

               key               | value
---------------------------------+-------
 delta.enableChangeDataFeed      | true
 delta.columnMapping.maxColumnId | 10
 delta.columnMapping.mode        | name
 delta.checkpointInterval        | 30
 delta.minReaderVersion          | 2
 delta.minWriterVersion          | 5
```

Modifying table properties

If we want to modify the underlying table properties of our Delta table, we'll need to use the Delta connectors alias for the supported table properties. For example, `change_data_feed_enabled` will map to the `delta.enableChangeDataFeed` property:

```
trino> ALTER TABLE delta.bronze_schema."ecomm_v1_clickstream"
SET PROPERTIES "change_data_feed_enabled" = false;
```

Deleting tables

Using the `DROP TABLE` operation, we can permanently remove a table that is no longer needed:

```
trino> DROP TABLE delta.bronze_schema."ecomm_lite";
```

There is a lot more that we can do with the Trino connector that is out of scope for this book; for now we will say goodbye to Trino and conclude this chapter.

Conclusion

During the time we spent together in this chapter, we learned how simple it can be to connect our Delta tables as either the source or the sink for our Flink applications. We then learned to use the Rust-based *kafka-delta-ingest* application to simplify the data ingestion process that is the bread and butter of most data engineers working with high-throughput streaming data. By reducing the level of effort required to simply read a stream of data and write it into our Delta tables, we end up in a much better place in terms of cognitive burden. When we start to think about all data in terms of tables—*bounded* or *unbounded*—the mental model can be applied to tame even the most wildly data-intensive problems. On that note, we concluded the chapter by exploring the native Trino connector for Delta. We discovered how simple configuration opens up the doors to analytics and insights, all while ensuring we continue to have a single source of data truth residing in our Delta tables.

CHAPTER 5

Maintaining Your Delta Lake

The process of keeping our Delta Lake tables running efficiently over time is akin to any kind of preventative maintenance for a car or motorcycle or any alternative mode of transportation (a bike, a scooter, rollerblades). We wouldn't wait for our tires to go flat before assessing the situation and finding a solution—we'd take action. We would 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 more air, or is this situation more dire, and the whole tire will need to be replaced?" The process of monitoring the situation, finding a remedy when we detect a problem, and applying the solution can be applied to our Delta Lake tables as well and is all part of the general process of maintaining the tables. In essence, we just need to think in terms of *cleaning*, *monitoring*, *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 build a firm understanding of the cause-and-effect relationships among 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 of our critical tables, all of which are valuable capabilities 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.

Delta Lake Table Properties Reference

The metadata stored alongside our table definitions includes `TBLPROPERTIES`. The common properties are presented in Table 5-1 and are used to control the behavior of our Delta tables. These properties enable automated preventative maintenance. When combined with the Delta Lake table utility functions, they also provide incredibly simple control over otherwise complex tasks. We simply add or remove properties to control the behavior of our tables.

Bookmark Table 5-1 for whenever you need a handy reference to these properties. Each row provides the property name, the internal data type, and the associated use case pertaining to cleaning, tuning, repairing, or replacing your Delta Lake tables.

Table 5-1. 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`	Boolean	Tuning	(none)
`delta.dataSkippingNumIndexedCols`	Int	Tuning	32
`delta.checkpoint.writeStatsAsStruct`	Boolean	Tuning	(none)
`delta.checkpoint.writeStatsAsJson`	Boolean	Tuning	true
`delta.randomizeFilePrefixes`	Boolean	Tuning	false

[a] Properties exclusive to Databricks.

The beauty behind using *table properties* 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 batch applications will simply read the revised properties on their next run).

The general behavior when adding or removing table properties is no different than using common data manipulation language (DML) operators, which consist of insert, delete, update, and, in more advanced cases, upsert, which will insert or update a row based on a match. Chapter 10 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, such as 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 table's Delta log.

There are other use cases that fall under the maintenance umbrella and 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 Lake 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/ch05
$ docker run --rm -it \
  --name delta_quickstart \
  -v $DLDG_DATA_DIR/:/opt/spark/data/datasets \
  -v $DLDG_CHAPTER_DIR/:/opt/spark/work-dir/ch05 \
  -p 8888-8889:8888-8889 \
  delta_quickstart
```

This 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 */ch05/ch05_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 5-1, 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 5-1. Creating a Delta Lake 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 six columns. In Example 5-1, 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; it contains only 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:

```
$ 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")
    )
```

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 we use `withColumn("date", to_date("date", "yyyy-MM-dd"))` to respect the existing data type of the table.

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 is thrown due to the default behavior of the `DataFrameWriter` in Spark, which defaults to `errorIfExists`. This is done for our benefit, to protect our precious data. So if the table exists, we raise an exception rather than trying to do anything that could damage the existing table.

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 by 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")
  ...
)
```

OK, 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 one column and our local schema (from reading the `covid_nyt` Parquet data) that is currently uncommitted and has six columns. This exception is another guardrail 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 two-step process provides a single source of truth for a table schema based on prior transactions:

Schema enforcement
: This 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
: This is the process of intentionally modifying an existing schema in a way that enables backward 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 to explicitly state that we approve of the schema changes we are bringing to the table, and that we 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") ❶
      .option("mergeSchema", "true") ❷
      .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:

❶ We updated the write mode to an append operation. This was necessary given that 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.

❷ We updated the write operation to include the `mergeSchema` option, enabling us to modify the `covid_nyt` table schema by adding the five additional columns required by the dataset within the same transaction in which we also physically added the `covid_nyt` 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` command, including the columns (and comments) and 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 preceding example, 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 schema, it comes at the price of our not being fully aware of the changes to our table schema. In cases in which 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.

If we knew that we had five 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 that 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 table 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()
```

The preceding 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 out this section, we'll learn how to remove unwanted table properties; then we can continue our journey 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 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.loRgeten tionDuratio` 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) property, we can execute `UNSET TBLPROPER TIES on our ALTER TABLE` command:

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

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 (see Example 5-1) and 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 works only for Spark workloads, you can probably imagine many scenarios in which 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, an engineering team, communication channels (Slack and email), and essentially anything else that helps to extend the utility of the descriptive table metadata. Utilizing table metadata can lead to simplified data discovery and capture information about 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 5-2 lists some example table properties that can be used to augment any Delta Lake table. The properties are broken down into prefixes and provide additional data catalog–style information alongside your existing table properties.

Table 5-2. Using table properties for data cataloging

Property	Description
`catalog.team_name`	Provides the team name and answers the question, "Who is accountable for the table?"
`catalog.engineering.comms.slack`	Provides 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`	Provides the email address for the engineering team—for example, *dldg_authors@gmail.com* (note that this isn't a real email address, 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 the scope of this book)

Delta Lake Table Optimization

Are you familiar with the idea that, for every action, there is an equal and opposite reaction?[1] Echoing the laws of physics, changes can be felt 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 not only that the table continues to serve its current use cases but also that it retains enough history to allow us to rewind (time travel) back to an earlier state (point in the table's time) and fix (overwrite) or recover (replace) the table in case larger problems are introduced to the table.

However, before we get into the more complicated maintenance operations, let's first look at common problems that can sneak into a table over time. Among the best known of these is the small file problem. Let's walk through the problem and its solution now.

The Problem with Big Tables and Small Files

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

How can too many small files hurt us? The answer is "in many different ways," but the common thread among 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 our 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.

One strategy for ensuring our tables remain in tip-top shape is to employ table-level monitoring. We cover some strategies for metadata-only monitoring in Chapter 13. These strategies can be extended to handle monitoring of the number of files for the current table snapshot, or to track how many versions of the table remain on disk. In the end, monitoring is a tool to help raise awareness of issues that pop up regularly and can be a lifeline with respect to your maintenance strategy.

1 This is Newton's third law of motion.

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 tends to be a result of cloud object storage, where each operation comes with its own variable latency, operational concurrency limits, and ultimately higher cost of doing business.

This is felt much more in traditional Hadoop-style ecosystems, such as MapReduce and Spark, where the unit of distribution is bound to a task, a file consists of "blocks," and each block takes one task. If we have one million files in a table and the files are 1 GB each, and we have a block size of 64 MB, 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 filesystem IO and network IO. When we encounter unoptimized files (the small files problem), then the performance of our tables suffers 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 5 kb 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 re-create 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 eight partitions, which is a small dataset:

```
$ ls -lh \
  /opt/spark/work-dir/ch05/spark-warehouse/covid_nyt/*.parquet | wc -l
8
```

What if we flipped the problem around and had nine thousand or even one million files representing the `covid_nyt` dataset? While this use case is extreme, we'll learn later in the book (Chapter 7) 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 eight partitions to, say, nine thousand partitions, this will split the table into an even nine thousand files at around 5 kb 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:

```
WAREHOUSE_DIR=/opt/spark/work-dir/ch05/spark-warehouse
FILE_PATH=$WAREHOUSE_DIR/nonoptimal_covid_nyt/*parquet

docker exec -it delta_quickstart bash \
-c "ls -l ${FILE_PATH} | wc -l"
```

You'll see that there are exactly nine thousand 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, and 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. As we look into fixing the small file problem, it is worth pointing out that the reverse can also be true—you may find yourself with many large files that need to be broken down in order to provide efficient processing.

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 1 GB; for Delta Rust, it's 250mb).

The `OPTIMIZE` command can be tuned using a mixture of configurations.[2]

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 together files smaller than the threshold (in bytes) 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)* The table property `delta.targetFileSize` (string)—an example being 250mb—can be used with the `delta-rs` client but is currently not supported in the OSS delta release.

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 `non optimal_covid_nyt` table. Feel free to run the command as many times as you want; `OPTIMIZE` will take effect a second time only 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 locally in a DataFrame (`results_df`) and are 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)
```

2 For the Spark ecosystem, Delta Lake >= 3.1.0 includes the option for auto compaction, using `delta.auto Optimize.autoCompact`.

```
    .where(col("operation") == "OPTIMIZE")
    .select(
                "version", "timestamp", "operation",
                "operationMetrics.numRemovedFiles",
                "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 nine thousand files (`numRemovedFiles`) and generated one compacted file (`numAddedFiles`).

Z-Order Optimize

Z-Ordering is a technique for colocating 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 `ZORDER BY` would allow us to *skip* opening files in our tables that don't contain relevant information for our query. This colocality 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 `ZORDER 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 10 will cover performance tuning in more detail, so for now we will just dip our toes in 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 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, against you, not unlike 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:[3]

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, 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 colocation.

If you need to optimize, how do 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, as in the case of legal hold. One simple use case is N-day delete—for example, 30-day retention. The use of daily partitions, while not optimal depending on the size of your Delta Lake table, can simplify common delete patterns, such as for 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`.

3 For the complete list of rules, you can always reference the Databricks documentation.

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 what column to partition by:

Is the cardinality of a column very high?
: If so, do not use that column for partitioning. For example, if you partition by a column `userId` and there can be more than a million distinct user IDs, then that is a bad partitioning strategy.

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 OK; 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 of 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` that 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 in 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 data definition language (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 nonpartitioned 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 Nonpartitioned to a Partitioned Table

With the table definition for our partitioned table in hand, it becomes trivial to simply read all the data from our nonpartitioned table and write the rows into our newly created table. What makes it even easier is that we don't need to 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 (nonpartitioned) 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 7 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 that 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/ch05/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/..."
  },
  "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 life cycle 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. The process of recovering data is commonly called *replaying*, since the action we are taking is to roll back 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

While it is possible to 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 9, 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) datasets. 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.

One technique for replacing corrupt or otherwise poor table partitions is to use the `replaceWhere` option alongside `overwrite` mode. 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")
  )
```

This 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. For example, say we write a simple command-line tool that takes a table and sets the conditions (`replaceWhere`, `overwrite`, or `restore`) and allows anyone to trigger a dry run—also known as a practice run—to see what would happen, to ensure the operation behaves correctly without causing additional problems. Rather than allowing teammates to run the command locally, and given that we are looking to remove human error, the command could be triggered using an API (with credentials) or via GitHub actions (after a PR and review to execute). In this way the operation intent can be recorded, and if things go wrong for any reason, the operation can be rolled back with limited impact and with no surprises.

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 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 simply to clear out a given partition, we can do so using a conditional delete on a partition column. The following statement conditionally deletes partitions (`date`) that are older than January 1, 2023:

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

Removing data, or dropping entire partitions, can 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; instead, it uses the information contained in the table metadata to prune partitions based on the predicate. In the case of deleting based on nonpartitioned columns, the cost is higher, as a partial or full table scan can occur. However, there is an added bonus: whether you are removing entire partitions or conditionally removing a subset of each table, if for any reason you 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 the context of the Delta Lake operations, as doing so can corrupt your table and cause headaches. This also means that any process that is not Delta aware should follow the same rules. Take cloud storage life cycle policies, for example: if your files are being automatically deleted every N days, this can also corrupt your Delta Lake tables.

The Life Cycle of a Delta Lake Table

Over time, as each Delta Lake table is modified, older versions of the table remain on disk to support table restoration or the viewing of earlier points in 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 that you 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 with unhappy customers because a stream of data just disappeared.

Restoring Your Table

In the case where a transaction has occurred—for example, an incorrect delete from your table (because 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 when 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 can 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 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 (or REORG), 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 as if the data is now invisible. This operation means that we have the ability to "undo" in cases where data is accidentally deleted. We can clean up the artifacts, the deleted files, and truly 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. With this in mind, it is best to utilize vacuum to enable short-lived time travel (up to 30 days is typical), and to employ a different strategy for storing strategic table backups. We'll look at the common scenario now.

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 vacuuming 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 cases where delete operations are not expected to be undone. For peace of mind, it is good to maintain at least one 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 the removal of all files 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, in that downstream data consumers can accidentally be left high and dry should they need to read an early version of your table.

If there is a need to store longer-retention table backups—for audit purposes, for disaster recovery, or for teams looking to read from earlier versions of the table—it is easiest to store the backup in another table. All we would need is the table version for the backup, and then a new Delta Lake table that can store the table permanently. Such backups could be postfixed with *_version_x* and can sit alongside the original table schema to reduce the number of places in which people need to look to find the earlier versions of the table.

Other issues that may arise will be covered in Chapter 7, where 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 set up 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. This is due to the way that the filesystem marks file creation time versus the actual time the files were originally created.

Dropping tables

Dropping a table is an operation with no undo. If you execute `delete from {table}`, you are essentially truncating the table and can still utilize time travel to undo the operation. However, if you really want to 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 truly want to remove all traces of a table, then read ahead.

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/.../silver.db/covid_nyt_by_date/"
```

The preceding code will result in the following output, which shows that the table really no longer exists on disk:

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

Conclusion

This chapter introduced you to the common utility functions provided within the Delta Lake project. We learned how to work with table properties, explored the table properties we'd most likely encounter, and learned how to optimize our tables to fix the small files problem. This led to our learning about partitioning and about restoring and replacing data within our tables. We explored using time travel to restore our tables, and we 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 for common problems and their required solutions in maintaining your Delta Lake tables and keeping them running smoothly over time.

CHAPTER 6
Building Native Applications with Delta Lake

By R. Tyler Croy

Delta Lake was created on the Java platform, but since the protocol became open source, it has been implemented with a number of different languages, allowing for new opportunities to use Delta Lake in native applications without requiring Apache Spark. The most mature implementation of the Delta Lake protocol after the original Spark-based library is delta-rs, which produces the `deltalake` library for both Python and Rust users.

In this chapter you will learn how to build a Python- or Rust-based application for loading, querying, and writing Delta Lake tables using these libraries. Along the way we will review some of the tools in the larger Python and Rust ecosystems that support Delta Lake, giving users substantial flexibility and performance when building data applications. Unlike its Spark-based counterpart, the `deltalake` library has no specific infrastructure requirements and can easily run in your command line, a Jupyter Notebook, an AWS Lambda, or anywhere else Python or compiled Rust programs can be executed. This extreme portability comes with a trade-off: there is no "cluster," and therefore native Delta Lake applications generally cannot scale beyond the computational or memory resources of a single machine.[1]

To demonstrate the utility of this "low overhead" approach to utilizing Delta Lake, in this chapter you will create an AWS Lambda, which will receive new data via its trigger, query an existing Delta Lake table to enrich its data, and store the new results in a new silver Delta Lake table. The pricing model of AWS Lambda incentivizes short

1 Some really interesting efforts such as Ballista are underway that will enable users to build Python- or Rust-based programs that run on a cluster, but they are still early in their maturity.

execution time and low memory utilization, which makes `deltalake` a powerful tool for building fast and cheap data applications. While the examples in this chapter run on AWS, the `deltalake` libraries for Python and Rust support a number of different storage backends from cloud providers such as Azure and Google Cloud Platform or on-premises tools like MinIO, HDFS, and more.

The general requirements for developing and deploying a Lambda function will be excluded from this chapter. To learn more, please consult with the AWS documentation for building a Python Lambda or a Rust Lambda.

Getting Started

To develop native Delta Lake applications, you will need to have Python 3 installed when building Python applications. Chances are your workstation either has Python 3 preinstalled or has it readily available as part of "developer tooling" packages. The Rust toolchain is necessary only when building Rust-based Delta Lake applications.[2] Rust, on the other hand, should be installed following the official documentation for installing the compiler and associated tooling, such as `cargo`.

Python

This example will largely be developed in the terminal on your workstation using `virtualenv` to manage the project-specific dependencies of the Lambda function:

```
% cd ~/dldg        # Choose the directory of your choice
% virtualenv venv  # Configure a Python virtualenv for managing deps
                   # in the ./venv/ directory
% source ./venv/bin/activate  # Activate the virtualenv in this shell
```

Once the `virtualenv` has been activated, the `deltalake` package can be installed with `pip`. It is also helpful to install the `pandas` package to do some data querying. The following example demonstrates some basic `deltalake` and `pandas` invocations to load and display a test dataset that is partitioned between two separate columns (c1, c2) containing a series of numbers:

```
% pip install 'deltalake>=0.18.2' pandas

% python
>>> from deltalake import DeltaTable
>>> dt = DeltaTable('./deltatbl-partitioned')
>>> dt.files()
```

2 The Rust compiler toolchain can easily be installed with the `rustup` installer.

```
['c2=foo0/part-00000-2bcc9ff6-0551-4401-bd22-d361a60627e3.c000.snappy.parquet',
'c2=foo1/part-00000-786c7455-9587-454f-9a4c-de0b22b62bbd.c000.snappy.parquet',
'c2=foo0/part-00001-ca647ee7-f1ad-4d70-bf02-5d1872324d6f.c000.snappy.parquet',
'c2=foo1/part-00001-1c702e73-89b5-465a-9c6a-25f7559cd150.c000.snappy.parquet']
```

```
>>> df = dt.to_pandas()
>>> df
```

```
   c1    c2
0   0  foo0
1   2  foo0
2   4  foo0
3   1  foo1
4   3  foo1
5   6  foo0
6   8  foo0
7   5  foo1
8   7  foo1
9   9  foo1
```

Reading data from Delta Lake tables is very easy thanks to the `to_pandas()` function, which loads data from the `DeltaTable` and produces a `DataFrame` that can be used to further query or inspect the data stored in the Delta Lake table. With a Pandas `DataFrame`, a wide world of data analysis is available in your terminal or notebook; to learn more about Pandas specifically, check out *Python for Data Analysis* (O'Reilly).

Getting started with Pandas is simple, but when reading large datasets, `to_pandas()` has some limitations; those will be covered in the next section.

Reading large datasets

Using Pandas and Delta Lake is a great way to start exploring data from within the terminal on your workstation. Behind the scenes of the `to_pandas()` function call mentioned in the previous section, the Python process must do the following:

1. Collect references to the necessary data files—in essence, the *.parquet* files returned from `dt.files()`.
2. Retrieve those data files from storage (the local filesystem in this example).
3. Deserialize and load those data files into memory.
4. Construct the `pandas.DataFrame` object using the data loaded in memory.

Steps 2 and 3 pose scaling limitations as the size of the data in the Delta Lake table grows. Modern workstations have *lots* of memory, which often means that loading a few gigabytes of data into memory is not that much of a concern, but the *retrieval* of that data can be a problem. For example, if the Delta Lake table is stored in AWS S3 but your Python terminal is running on your laptop, loading a few gigabytes over

coffee shop WiFi is time consuming, and it's unnecessary if you do not intend to query the *entire* table.

The design of Delta Lake provides a few mechanisms for reducing the size of the data that must be loaded to help make queries fast and efficient:

Partitions
: Structuring of data in storage to allow grouping of files by common prefixes, such as `mytable/year=2024/*.parquet`.

File statistics
: Additional metadata included by the writer in the transaction log about the *.parquet* file, whether Apache Spark or a native Python/Rust, that indicates the minimum or maximum values of columns contained in that data column.

By reducing the number of files that need to be loaded to perform queries, partitions and file statistics help lower execution times to produce results *faster*, which reduces developer iteration time and makes data processing workloads cheaper to run. The following examples will use these features to reduce the number of files loaded from storage and also the amount of memory needed to work with the Delta Lake table in Pandas.

The Delta protocol has a number of other design optimizations to allow for efficient operation, such as checkpoints, compaction, and Z-Ordering. These features are supported in the native Python and Rust libraries. They are discussed elsewhere in this book and will not be addressed explicitly in this chapter.

Partitions. Partitioning data into common prefixes is a pattern shared by several storage systems, including Delta Lake. Commonly referred to as *hive-style partitioning*, the `to_pandas()` function allows you to specify partitions with the optional parameter `partitions`. Consider the following example table layout:

```
deltatbl-partitioned
├── c2=foo0
│   ├── part-00000-2bcc9ff6-0551-4401-bd22-d361a60627e3.c000.snappy.parquet
│   └── part-00001-ca647ee7-f1ad-4d70-bf02-5d1872324d6f.c000.snappy.parquet
├── c2=foo1
│   ├── part-00000-786c7455-9587-454f-9a4c-de0b22b62bbd.c000.snappy.parquet
│   └── part-00001-1c702e73-89b5-465a-9c6a-25f7559cd150.c000.snappy.parquet
└── _delta_log
    └── 00000000000000000000.json
```

The table has a partition column of `c2` with two partitions defined. To work only with data contained within the first partition (`foo0`), the `to_pandas()` invocation can be modified as follows to use a partition filter that restricts data loaded only to the specified partition(s):

```
>>> dt.to_pandas(partitions=[('c2', '=', 'foo0')])
```

```
   c1    c2
0   0  foo0
1   2  foo0
2   4  foo0
3   6  foo0
4   8  foo0
```

If the datasets are particularly large, the same partition filter can be passed to the `files()` function on the `DeltaTable` for a low-overhead preview of the files that the `to_pandas` call would load:

```
>>> dt.files([('c2', '=', 'foo0')])
```

```
['c2=foo0/part-00000-2bcc9ff6-0551-4401-bd22-d361a60627e3.c000.snappy.parquet',
'c2=foo0/part-00001-ca647ee7-f1ad-4d70-bf02-5d1872324d6f.c000.snappy.parquet']
```

This partition filter shows that only two *.parquet* files must be loaded, rather than the four total in this table. That helps reduce the time spent retrieving data files from storage, but their contents must still be loaded into memory to build the `pandas.DataFrame`.

The `to_pandas()` function has two other optional parameters that are important to consider when trying to reduce the memory footprint while working with this dataset. The easiest one to use is `columns`, which simply restricts the columns that are projected from the *.parquet* file into the `DataFrame`. In this example, the `c2` partition is helpful to reduce the amount of data loaded from the table, but it's not needed in the `DataFrame` and is excluded via the `columns` parameter:

```
>>> dt.to_pandas(partitions=[('c2', '=', 'foo0')], columns=['c1'])
```

```
   c1
0   0
1   2
2   4
3   6
4   8
```

For particularly wide tables, this can be a helpful trick to reduce both the amount of data loaded into memory *and* the amount of data displayed in the terminal, making the results easier to visually inspect.

The final optional parameter that can further reduce the memory footprint is `filters`, which accepts DNF-style filter predicates that support a number of

operations such as <, >, <=, >=, =, and !=. The following snippet incorporates the optional parameters to `to_pandas()` that can be combined to produce a compact `DataFrame` containing only the desired data:

```
>>> dt.to_pandas(partitions=[('c2', '=', 'foo0')], columns=['c1'],
filters=[('c1', '<=', 4), ('c1', '>', 0)])
```

```
   c1
0   2
1   4
```

The semantics provided by the `to_pandas()` function are no substitute for the expressive power provided by the Pandas DataFrame API, but they offer a very useful mechanism for constraining the amount of data retrieved from a Delta Lake table and loaded into memory. Both are important to consider in the example discussed later in the "Building a Lambda" on page 131 section, in which the resource constraints of the AWS Lambda environment reward fast and lightweight runtimes.

File statistics. The Delta protocol allows for optional *file statistics* that can enable further optimization by query engines. When writing a *.parquet* file, most writers will put this additional metadata into the Delta transaction log, capturing each column's minimum and maximum values. The `deltalake` Python library can utilize this information to skip files that don't contain values in the specified column(s). This can be especially useful for append-only tables that have predictable and sequential data within a given partition.

Using an example dataset[3] that is partitioned by year but contains multiple Parquet files within each partition, the transaction log includes the following entry:


```
{
  "add": {
    "path": "year=2022/0-ec9935aa-a154-4ba4-ab7e-92a53369c433-2.parquet",
    "partitionValues": {
      "year": "2022"
    },
    "size": 3025,
    "modificationTime": 1705178628881,
    "dataChange": true,
    "stats": "{\"numRecords\": 4, \"minValues\": {\"month\": 9,
     \"decimal date\": 2022.7083, \"average\": 415.74,
     \"deseasonalized\": 419.02, \"ndays\": 24, \"sdev\": 0.27,
     \"unc\": 0.1}, \"maxValues\": {\"month\": 12, \"decimal date\": 2022.9583,
     \"average\": 418.99, \"deseasonalized\": 419.72, \"ndays\": 30,
     \"sdev\": 0.57, \"unc\": 0.22}, \"nullCount\": {\"month\": 0,
     \"decimal date\": 0, \"average\": 0, \"deseasonalized\": 0, \"ndays\": 0,
```


3 Scripts to download this sample data can be found in this book's GitHub repository. This particular example uses data from NOAA that tracks global CO_2 concentrations.

```
        \"sdev\": 0, \"unc\": 0}}",
      "tags": null,
      "deletionVector": null,
      "baseRowId": null,
      "defaultRowCommitVersion": null,
      "clusteringProvider": null
    }
}
```

The `stats` portion contains the relevant information for the file statistics–based optimization. Inspecting `minValues` and `maxValues` shows that *0-ec9935aa-a154-4ba4-ab7e-92a53369c433-2.parquet* contains data only for the months September to December in the year 2022. The following Pandas invocation will create a `DataFrame` that has loaded data *only* from this specific file utilizing the partition column and the month column. The file statistics help the underlying engine avoid loading *every* file in the `year=2022/` partition; instead, it selects only the one containing values where the month is greater than or equal to `9`, leading to a much faster and more efficient execution of data retrieval:

```
>>> from deltalake import DeltaTable
>>> dt = DeltaTable('./data/gen/filestats')
>>> len(dt.files())

198

>>> df = dt.to_pandas(filters=[('year', '=', 2022), ('month', '>=', 9)])
>>> df

   year  month  decimal date  average  deseasonalized  ndays  sdev   unc
0  2022      9     2022.7083    415.91          419.36     28  0.41  0.15
1  2022     10     2022.7917    415.74          419.02     30  0.27  0.10
2  2022     11     2022.8750    417.47          419.44     25  0.52  0.20
3  2022     12     2022.9583    418.99          419.72     24  0.57  0.22
```

Rather than loading every one of the files in the Delta Lake table to produce a `DataFrame` for experimentation, `filters` utilizes Delta's partitioning and file statistics to load a *single file* from storage for this example.

The Delta Lake transaction log provides a wealth of information that the `deltalake` native Python library utilizes to provide fast and efficient reads of tables; more examples can be found in the online documentation. Reading existing Delta Lake tables is exciting, but for many Python users, the *writing* of Delta Lake tables helps unlock new superpowers in the Python-based data analysis or machine learning environment.

Writing data

Numerous examples for performing data analysis or machine learning in Python start with loading data into a DataFrame of some form (typically Pandas) from a CSV- or TSV-formatted dataset. Comma-separated values (CSV) files are fairly

easy to produce and reason about and can be streamed into and out of different applications. The major downside for CSV datasets is that, to perform data analysis, they *typically* must fully be loaded into memory whenever they are needed; this can become problematic when they are quite large or slow to load.

This section will utilize the same small (tens of kilobytes) publicly available CSV dataset used earlier to demonstrate file statistics. The dataset contains annual atmospheric CO_2 concentrations provided by NOAA and demonstrates the ease with which Delta Lake tables can be created in Python using the `deltalake` package.

There are a couple of different options for writing Delta Lake tables, but this initial example will focus on the simple case of writing an unpartitioned Delta Lake table from a `pandas.DataFrame`:

```
>>> import pandas as pd
>>> from deltalake import write_deltalake, DeltaTable
>>> df = pd.read_csv('./data/co2_mm_mlo.csv', comment='#')
>>> len(df)

790

>>> write_deltalake('./data/co2_monthly', df)
>>> dt = DeltaTable('./data/co2_monthly')
>>> dt.files()

['0-6db689af-10fe-4350-82e8-bef6d962330a-0.parquet']

>>> df = dt.to_pandas()
>>> df

     year  month  decimal date  average  deseasonalized  ndays  sdev   unc
0    1958      3     1958.2027   315.70          314.43     -1 -9.99 -0.99
1    1958      4     1958.2877   317.45          315.16     -1 -9.99 -0.99
2    1958      5     1958.3699   317.51          314.71     -1 -9.99 -0.99
3    1958      6     1958.4548   317.24          315.14     -1 -9.99 -0.99
4    1958      7     1958.5370   315.86          315.18     -1 -9.99 -0.99
..    ...    ...           ...      ...             ...    ...   ...   ...
785  2023      8     2023.6250   419.68          421.57     21  0.45  0.19
786  2023      9     2023.7083   418.51          421.96     18  0.30  0.14
787  2023     10     2023.7917   418.82          422.11     27  0.47  0.17
788  2023     11     2023.8750   420.46          422.43     21  0.91  0.38
789  2023     12     2023.9583   421.86          422.58     20  0.69  0.29

[790 rows x 8 columns]
```

The resulting Delta Lake table is simple and includes only a single *.parquet* data file due to the compact size of the source dataset; for datasets in the tens of megabytes or larger, the `deltalake` writer may produce multiple data files when creating new transactions on the table. The `write_deltalake()` function has a number of optional parameters that allow for more advanced behaviors, such as partitioning:

```
>>> df = pd.read_csv('./data/co2_mm_mlo.csv', comment='#')
>>> write_deltalake('./data/gen/co2_monthly_partitioned', data=df,
partition_by=['year'])
```

This snippet will write the new Delta Lake table with hive-style partitions based on the `year` column of the provided `DataFrame`. The resulting table in storage is cleanly partitioned as follows:

```
co2_monthly_partitioned
├── _delta_log
│   └── 00000000000000000000.json
├── year=1958
│   └── 0-50ffe4cc-864d-4753-8f47-b0b55618a31a-0.parquet
├── year=1959
│   └── 0-50ffe4cc-864d-4753-8f47-b0b55618a31a-0.parquet
├── year=1960
│   └── 0-50ffe4cc-864d-4753-8f47-b0b55618a31a-0.parquet
```

In this example, the uncompressed dataset is small and easily fits entirely in memory, but for larger datasets, `write_deltalake()` can accept larger and more lazily loaded datasets and iterators, which allows for writing data incrementally.

To append or overwrite data, use the `mode` optional parameter, which currently supports the following modes:

- `error` (default): return an error if the table already exists
- `append`: add the provided `data` to the table
- `overwrite`: replace the table contents with the provided `data`
- `ignore`: do not write the table, or return an error if it already exists

The ability to write a Delta Lake table easily can accelerate local development or model training; in addition, it can enable building simple and fast ingestion applications in environments such as AWS Lambda, which will be covered later in the chapter.

Merging/updating

The `DeltaTable` object contains a number of simple functions for common merge or update tasks on the Delta Lake table, such as `delete`, `merge`, and `update`. These functions can be used in much the same way as delete, merge, and update operations in a relational database, but underneath the covers the Delta transaction log is doing a lot of important work to keep track of the data being modified.

For example, consider a Delta Lake table with 100 rows stored in a single *first.parquet* file, added via a single add transaction. A subsequent `delete` operation that deletes every other row will produce a new *second.parquet* file containing 50 records. Committing the deletion will create a new transaction on the table containing two actions—one removing the *first.parquet*, and a second action adding *second.parquet*:

```
>>> import pyarrow as pa
>>> from deltalake import DeltaTable, write_deltalake
>>> data = pa.table({'id' : list(range(100))}) # Create a sample dataset
>>> write_deltalake('delete-test', data)
>>> dt = DeltaTable('delete-test')
>>> dt.version()

0

>>> dt.to_pandas().count()

Id     100
dtype: int64

>>> dt.delete('id % 2 == 0')

{'num_added_files': 1, 'num_removed_files': 1, 'num_deleted_rows': 50,
 'num_copied_rows': 50, 'execution_time_ms': 35187, 'scan_time_ms': 33442,
 'rewrite_time_ms': 1}

>>> dt.version()  # There is a new version

1
```

Inspecting the *./delete-test/_delta_log/* directory after the `delete()` operation reveals two transaction entries; *00000000000000000001.json* contains the actions representing the `delete()` operation, revealing exactly how table modifications typically work:

```
{
  "add": {
    "path": "part-00001-4bc82516-2371-4004-9ff8...c000.snappy.parquet",
    "partitionValues": {},
    "size": 799,
    "modificationTime": 1708794006394,
    "dataChange": true,
    "stats": "{\"numRecords\":50,\"minValues\":{\"id\":1},\"maxValues\":
    {\"id\":99},\"nullCount\":{\"id\":0}}",
    "tags": null,
    "deletionVector": null,
    "baseRowId": null,
    "defaultRowCommitVersion": null,
    "clusteringProvider": null
  }
}
{
  "remove": {
    "path": "0-2684b307-3947-49ce-bc07-02688b10a204-0.parquet",
    "dataChange": true,
```

```
      "deletionTimestamp": 1708794006394,
      "extendedFileMetadata": true,
      "partitionValues": {},
      "size": 1074
    }
  }
  {
    "commitInfo": {
      "timestamp": 1708794006394,
      "operation": "DELETE",
      "operationParameters": {
        "predicate": "id % 2 = 0"
      },
      "clientVersion": "delta-rs.0.17.0"
    }
  }
```

The preceding snippet contains the two key actions: `remove` and `add`, with their respective files. There is an additional action called `commitInfo` that is optional in the Delta Lake table protocol but may contain additional information about what triggered this particular transaction. In this case, it describes the `DELETE` operation with its predicate, giving us insight into *why* the `remove` and `add` were necessary.

Whether the operation is a delete, update, or merge, when data is changed in the Delta Lake table, there is typically a removal of outdated Parquet files and a creation of *new* Parquet files with the modified data. This is the case except when using the newer *deletion vectors* feature, which the Python or Rust libraries do not support at the time of this writing.[4]

Going beyond Pandas

The `deltalake` Python package provides support for the Delta Lake table format to a number of different query engines and implementations. While this chapter utilizes the Pandas library as an example for reading, writing, and so on, integrations exist for using the `deltalake` package from DataFrame libraries such as Polars or Datafusion. Each of those libraries provides a compelling feature set for Python data applications.

The foundational library `pyarrow` binds all of these integrations together and implements shared abstractions such as `RecordBatch`, `DataSet`, and `Table`. In the documentation for `deltalake`, there are a number of functions that accept or return these objects. This chapter does not provide exhaustive documentation for each of these types, which are documented at a high level in the PyArrow project's online API documentation.

4 Development of deletion vectors can be followed in the delta-rs issue tracker.

RecordBatch. Most of the internal operations for reading and writing Delta Lake tables in Python will create or work with `RecordBatch` objects, which represent a collection of columns of equal length. Delta Lake data files are Apache Parquet formatted, which is a columnar data format, and `RecordBatch` is similarly columnar. Rather than expressing rows such as `[1, 'Will', True]`, `[2, 'Robert', True]`, `[3, 'Ion', True]`, and so on, the `RecordBatch` types are typically instantiated with *columns*, such as: `[1, 2, 3]`, `['Will', 'Robert', 'Ion']`, `[True, True, True]`.

Most applications can work with DataFrames built on top of the `RecordBatch` type, but there are a number of ways to squeeze higher performance or efficiency from a Python data application by understanding and working with `RecordBatch` objects directly.

Table. A `pyarrow.Table` is a collection of named, equal-length Arrow arrays and is effectively how "table" is commonly understood in most other data systems. As a container for schema plus data, the `DeltaTable` object can expose a PyArrow `Table` directly with the `to_pyarrow_table()` function that accepts filtering options similar to `to_pandas()`, such as `partitions` or `filters` keyword parameters. Calling `to_pyarrow_table()` will *also* load all the available data into memory.

When possible, it is more efficient to rely on `DataSet` rather than `Table`, as described below.

DataSet. A `pyarrow.DataSet` is similar to `Table` in that it has an associated schema of the full dataset, but unlike `Table`, it is lazily loaded and provides substantial flexibility for working with larger datasets. A `DataSet` object is very low overhead to create, since it usually results in very little data being read from storage and can be created from a `DeltaTable` object with `to_pyarrow_dataset()`.

Once a `DataSet` has been created, it is possible to lazily load data using functions such as `filter`, which provides a *new* filtered `DataSet` from which you can invoke `to_batches()` to provide a lazy iterator of `RecordBatch` data or invoke `to_table()` to produce a `pyarrow.Table` with all the data from the filtered dataset.

In many cases, `deltalake` uses `DataSet` internally to produce or consume data; it is a very flexible, efficient, and well-documented data type.

From simple data ingestion to transformation or complex query and machine learning tasks, the ability to interact with Delta Lake tables from practically any Python environment opens up innumerable possibilities and applications for data stored in Delta Lake.

Rust

Underneath the Python library described in the beginning of this chapter is a full implementation of the Delta Lake protocol in Rust, commonly referred to as delta-rs, which can also be used directly to build high-performance data applications. Adding the `deltalake` package to *Cargo.toml* in a Rust project is typically all that is needed to get started. The `deltalake` crate includes feature flags and dependencies that can optionally add dependencies to enable support for AWS, Azure, or Google Cloud. The `datafusion` feature flag can also be used to add integration with the Apache Arrow DataFusion project for doing sophisticated query, write, and merge operations within Rust.

There are many characteristics of Rust that make it increasingly sought after for handling data engineering tasks, such as its low memory overhead, ease of concurrency, and stability. In many cases, once a Delta Lake application is developed in Rust, it can and will run for years without issue.[5]

Examples in this section will assume the latest version of the `deltalake` package configured in a Rust project, with the `datafusion` feature enabled.

Rust is a compiled language, and with a large project such as delta-rs and DataFusion, link times can suffer with the default Clang or GNU `ld`. The mold linker is worth installing and configuring to improve development cycle times.

Following the same patterns as the Python examples, start by opening a contrived Delta Lake table:

```
#[tokio::main]
async fn main() {
    println!(">> Loading `deltatbl-partitioned`");
    let table = deltalake::open_table("../data/deltatbl-partitioned").await
      .expect("Failed to open table");
    println!("..loaded version {}", table.version());
    for file in table.get_files_iter() {
      println!(" - {}", file.as_ref());
    }
}
```

5 One of the earliest open source applications developed with delta-rs, kafka-delta-ingest, has been running in production environments for years without incident or substantial change in system resource requirements.

The full source code is located in the GitHub repository associated with this book. This simple example creates a `DeltaTable` from the provided path and inspects the files associated with the latest version, outputting the following:

```
>> Loading `deltatbl-partitioned`
..loaded version 0
 - c2=foo0/part-00000-2bcc9ff6-0551-4401-bd22-d361a60627e3.c000.snappy.parquet
 - c2=foo1/part-00000-786c7455-9587-454f-9a4c-de0b22b62bbd.c000.snappy.parquet
 - c2=foo0/part-00001-ca647ee7-f1ad-4d70-bf02-5d1872324d6f.c000.snappy.parquet
 - c2=foo1/part-00001-1c702e73-89b5-465a-9c6a-25f7559cd150.c000.snappy.parquet
```

Inspecting the file listing of the table is not particularly interesting, so the following example utilizes some DataFusion tooling to provide a SQL query–like interface to the same Delta Lake table:

```
use std::sync::Arc;
use deltalake::datafusion::execution::context::SessionContext;
use deltalake::arrow::util::pretty::print_batches;

#[tokio::main]
async fn main() {
  let ctx = SessionContext::new();
  let table = deltalake::open_table("../data/deltatbl-partitioned")
      .await
      .unwrap();
  ctx.register_table("demo", Arc::new(table)).unwrap();

  let batches = ctx
      .sql("SELECT * FROM demo LIMIT 5").await.expect("Failed to execute SQL")
      .collect()
      .await.unwrap();
  print_batches(&batches).expect("Failed to print batches");
}
```

Running this example will print the first five records found in the Delta Lake table, providing a simple interface and a distinctly simple but *unrusty* API for querying data in the table:

```
+----+------+
| c1 | c2   |
+----+------+
| 0  | foo0 |
| 2  | foo0 |
| 4  | foo0 |
| 1  | foo1 |
| 3  | foo1 |
+----+------+
```

DataFusion bundles its own SQL dialect but also provides a DataFrame API that should be familiar to users coming from Pandas or Apache Spark. In the book's GitHub repository, there are some additional examples that demonstrate the DataFrame equivalent of the DataFusion SQL examples. In fact, the `SessionContext::sql` function returns a `DataFrame` that allows the combining of simple SQL queries with more complex DataFrame chaining of logic for advanced use cases.

Reading large data

For Delta Lake tables that represent data larger than what could reasonably fit in memory on a single machine, the Rust `deltalake` library offers users partitioning and file statistics semantics that are similar to those of the Python library.

When reading large datasets in Python, partitions and filters must be specified *before* creating a Pandas DataFrame. With DataFusion, the filters can be specified inline with the creation of the DataFrame because of the tight integration that the `delta lake` Rust crate provides with the DataFusion APIs. The `deltatbl-partitioned` table has a partition on the `c2` column that can be included in the DataFusion SQL query to avoid reading *.parquet* files in the partitions that don't match the predicate—for example:

```
let df = ctx
          .sql("SELECT * FROM demo WHERE c2 = 'foo0'")
          .await
          .expect("Failed to create data frame");
```

The DataFusion SQL will also use file statistics in the Delta transaction log to generate the appropriate and optimal query plan when creating the `DataFrame`. Generally speaking, when using the DataFusion SQL or DataFrame APIs with Delta Lake, the default behavior is almost always the correct and optimal one.

Writing data

At a fundamental level, a Delta Lake table consists of data files, typically in the Apache Parquet format, and transaction log files in a JSON format. The `deltalake` Rust crate supports writing both data and transaction log files, or writing *only* transactions. For example, kafka-delta-ingest translates streams of JSON data into Apache Parquet before creating a transaction to add the data to the configured Delta Lake table. Other Rust applications may use Parquet data files created by an external system, such as oxbow, which only needs to manage the Delta Lake table's transaction log.

Regardless of the specific application needs, the `deltalake` crate has several options. Covering each of the writer APIs in detail is out of the scope of this book,[6] but at present the crate supports:

- Transaction operations that allow direct interaction with the Delta log
- A DataFusion-based writer for inserting and/or merging records
- A simple high-level JSON writer that accepts `serde_json::Value` types
- A `RecordBatch` writer that allows developers to turn Arrow RecordBatches into Apache Parquet files written into Delta Lake tables

For most use cases, the decision about what *type* of writer is required will come down to whether the write should be an *append* or a *merge*.

For append-only writers, DataFusion is not necessary, and the `deltalake` package's `RecordBatchWriter` can be used to issue append-only writes to a `DeltaTable`.

DataFusion is an incredibly powerful data processing engine built in Rust, but it also adds a nontrivial increase in binary size, link-time overhead, and API surface area. The `deltalake` crate contains several integrations with DataFusion for performing queries, merges, and more, but they must be enabled by specifying the `datafusion` feature, such as with `cargo add -features datafusion deltalake`. For example, when building an AWS Lambda using Delta Lake, the binary built without DataFusion produces a binary size of 4.8 MB. When enabling the `datafusion` feature flag in *Cargo.toml*, the resulting *bootstrap.zip* grows to 8.2 MB.

Frequently the most challenging part of using `RecordBatchWriter` is constructing the necessary Arrow `RecordBatch` objects, which contain a schema and are columnar by nature. There are some utilities in the `arrow` crate that help with constructing `RecordBatch` objects, such as the JSON reader: `arrow::json::reader::ReaderBuilder`; but for the following example, an object will be manually created from in-memory data and assumes a Delta Lake table has already been created.

6 The writers are available as of the 0.19 release of the `deltalake` crate, but this may change as the project moves toward a 1.0 release.

Merging/updating

Modifying Delta Lake tables from Rust generally requires using the `datafusion` feature since the DataFusion engine provides the predicate handling required for deletes, merges, and updates of records. Unlike the Python library, which hangs these operations off the `DeltaTable` object, the Delta Lake table operations are available via the `DeltaOps` struct, which helps generate builders for various operations, such as the following `delete` example:

```
let table = deltalake::open_table("./test")?;
let (table, metrics) = DeltaOps::from(table).delete()
                        .with_predicate(col("id").rem(lit(2)).eq(lit(0)))
                        .await?
```

Similar to the Python example, this will produce a new transaction in the log with a remove and an add. The documentation for `DeltaOps` contains more information on exactly how to use `delete`, `update`, or `merge`. DataFusion is at the core of these operations, so it is highly useful to consult with the DataFusion documentation to learn more about constructing predicates, dataframes (needed for merges), and expressions. DataFusion SQL can also be used instead of Rust dataframe semantics via the `deltalake` crate's table provider.

Building a Lambda

Serverless functions represent an ideal use case for building native applications for Delta Lake, such as with AWS Lambda. The billing model for Lambda encourages low memory usage and fast execution time, which makes it a great platform for compact and efficient data processing applications. This section will adapt some of this chapter's previous examples to run within AWS Lambda to handle data ingestion or processing using `deltalake`. Other cloud providers have similar serverless offerings, such as Azure Functions and Google Cloud Run. The concepts in this section can be ported into those environments, but some of the interfaces may change.

For most applications, AWS Lambda is triggered by an external event such as an inbound HTTP request, an SQS message, or a CloudWatch event. Lambda will then translate this external event into a JSON payload, which the Lambda function will receive and can act upon. Imagine, for example, an application that receives an HTTP POST with a JSON array containing thousands of records that should be written to S3, as sketched out via the request flow diagram in Figure 6-1. Upon invocation, the Lambda receives the JSON array, which it can then append to a preconfigured Delta Lake table. Lambdas should conceptually be simple and complete their task as quickly and efficiently as possible.

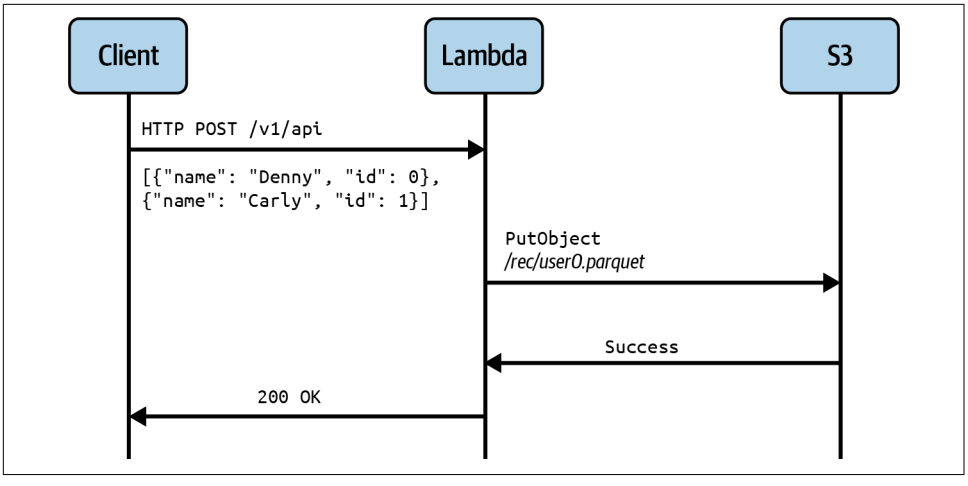

Figure 6-1. Request flow diagram of a hypothetical upload of user data for storage via AWS Lambda

Python

Lambdas can be written in Python directly within the AWS Lambda web UI. Unfortunately, the default Python runtime has only minimal packages built in, and developers wishing to include `deltalake` will need to package their Lambdas either with layers or as containers. AWS provides an "AWS SDK with Pandas" layer that can be used to get started, but some care must be taken to include the `deltalake` dependency due to the 250 MB size limitation of Lambda layers. How the Lambda is packaged doesn't have a significant impact on its execution, so this section will not focus heavily on packaging and uploading the Lambda. Please refer to the book's GitHub repository, as it contains examples that use layers and container-based approaches, along with the infrastructure code necessary to deploy the examples.

The `hello-delta-rust` example demonstrates the simplest possible Delta Lake application in Lambda. This example looks only at the table's metadata, rather than querying any of the data.

The *lambda_function.py* simply opens the Delta Lake table and returns metadata to the HTTP client:

```
import os
from deltalake import DeltaTable

def lambda_handler(event, context):
    url = os.environ['TABLE_URL']
    dt = DeltaTable(url)
    return { 'version' : dt.version(),
        'table' : url,
```

```
        'files' : dt.files(),
        'metadata' : {}}
```

This simple Python to create a `DeltaTable` object and then perform operations on the table (`dt`) demonstrates how easy interacting with Delta Lake from a Lambda can be. So long as the function returns a `list` or `dict` to the caller of `lambda_handler`, AWS Lambda will handle returning the information to the caller in JSON over HTTP.

The examples from the section "Reading large datasets" on page 117, which used Pandas or PyArrow for querying data in Python, can be reused inside the Lambda environment.

Similarly, the examples that cover *writing* data in Python can be reused in a Lambda. However, the Lambda execution environment is inherently *parallelized*, which presents concurrent write challenges when using AWS S3; these challenges and solutions are discussed later in this chapter. First we need the application, which will take the JSON array described above and append that to a Delta Lake table. Execution begins with the `lambda_handler` function, which is the entrypoint for AWS Lambda to execute your uploaded code:

```
def lambda_handler(event, context):
    table_url = os.environ['TABLE_URL']
        try:
            input = pa.RecordBatch.from_pylist(json.loads(event['body']))
            dt = DeltaTable(table_url)
            write_deltalake(dt, data=input, schema=schema(), mode='append')
            status = 201
            body = json.dumps({'message' : 'Thanks for the data!'})
        except Exception as err:
            status = 400
            body = json.dumps({'message' : str(err),
              'type' : type(err).__name__})

    return {
            'statusCode' : status,
            'headers' : {'Content-Type' : 'application/json'},
            'isBase64Encoded' : False,
            'body' : body,
    }
```

The preceding is a shortened example of the `ingest-with-python` example from the GitHub repository and could be taken and dropped into an arbitrary Python Lambda configuration. Upon uploading data, however, an error will be returned by default:

```
{"message": "Atomic rename requires a LockClient for S3 backends. Either con-
figure the LockClient, or set AWS_S3_ALLOW_UNSAFE_RENAME=true to opt out of
support for concurrent writers.",
"type": "DeltaProtocolError"}
```

By default, unsafe renames are *disabled* in the `deltalake` library. When you are faced with this error, it may be tempting to set `AWS_S3_ALLOW_UNSAFE_RENAME` to `true` in the configuration, but doing so can risk data loss or table corruption, because concurrent Delta writes cannot be done safely on AWS S3 without coordination. Skip ahead to the section "Concurrent writes on AWS S3" on page 135 to learn how to configure the Lambda to perform concurrent writes safely.

The append-only example can be further extended to load and merge data from other Delta Lake tables by creating more `DeltaTable` objects and then using the `pandas.DataFrame` functions for `merge`, `join`, or `concat`. Imagine a secondary table `s3://bucket/dietary_prefs` that needs to be joined with employee records being uploaded to the Lambda, which will produce the `s3://bucket/offsite_attendees` table:

```
def lambda_handler(event, context):
    table_url = os.environ['TABLE_URL']
        try:
            input = pd.DataFrame(json.loads(event['body']))
        # Expecting both `input` and `prefs` to have an `id` column to
        # perform the inner join
        prefs = DeltaTable('s3://bucket/dietary_prefs').to_pandas()
            dt = DeltaTable(table_url)
            write_deltalake(dt,
              data=pd.merge(input, prefs),
              schema=schema(), mode='append')
            # …
```

When performing joins of datasets in Lambda, it is important to remember that the function operates in a memory- and CPU-constrained environment! Using predicates or adopting PyArrow directly rather than working with Pandas can allow for improved performance, should the simplistic approach become too memory- or CPU-intensive. If the working datasets cannot fit into memory within Lambda, then the workload should be considered for running in another environment, such as the standalone services ECS/EKS/EC2, or for porting to Spark to take advantage of multiple machines.

Rust

Building AWS Lambdas in Rust is similarly straightforward to building their Python counterparts. Unlike Python, however, Rust can be compiled to native code and does not require a "runtime" in AWS Lambda; instead, a custom-formatted *bootstrap.zip* file containing the compiled executable must be uploaded to AWS. Additional tools such as cargo-lambda should be installed on your workstation to provide generators and the build/cross-compiling functionality needed to build the *bootstrap.zip* files required by Lambda. The following examples and those in the book's GitHub

repository rely on cargo-lambda, and to begin writing a Rust Lambda, the necessary scaffolding should be created:

```
% cargo lambda new deltadog --event-type s3::S3Event
% cd deltadog
% cargo add -features s3 deltalake
% cargo lambda build -release -output-format zip
```

The last command above will produce `./target/lambda/deltadog/bootstrap.zip`, which can be uploaded directly into AWS Lambda. Similar to the Python examples, there is a single entrypoint at which Rust code can be added. With the scaffolding above, any of the previous reading or writing Rust examples from this chapter can be copied and pasted into a Lambda function. Unlike their Python counterparts, the Rust Lambdas can typically process *much more* data because of the highly compact and efficient nature of Rust executables. Existing `deltalake` Rust code can be added into the function handler verbatim:

```
async fn function_handler(event: LambdaEvent<S3Event>) -> Result<(), Error> {
    // Extract some useful information from the request
    let _table = deltalake::open_table("s3://example/table").await?;
    Ok(())
}
```

The `ingest-with-rust` example in the GitHub repository can be used as a starting point, similar to `ingest-with-python`.

Concurrent writes on AWS S3

Delta Lake has supported concurrent reads from multiple clusters since its inception, but safe concurrent writes require special care with AWS S3, since it lacks `putIfAbsent` consistency guarantees. Without separate coordination, there is no way to guarantee that writes originating from different writer processes—Python, Rust, or Spark—won't conflict with each other. Most Delta Lake applications built with AWS S3 use some variation of an AWS DynamoDB table to coordinate writers. Prior to the `deltalake` Python release 0.15 and the Rust release 0.17, those libraries used `dynamodb_lock`, while more recent releases use the `S3DynamoDBLogStore`-compatible implementation, which allows Python and Rust applications to interoperate using the same protocol adopted by Delta Lake Spark writers requiring multicluster support.

S3DynamoDBLogStore. The de facto standard for performing concurrent writes with AWS S3 relies on a DynamoDB table but utilizes it in a different fashion. Starting with the `deltalake` Rust crate version 0.17 and the `deltalake` Python version 0.15, native applications can interoperate seamlessly with Spark applications using the `S3DynamoDBLogStore` protocol. The protocol relies on the coordination of commits to the Delta log via a DynamoDB table, which provides both serialization of commits

to the log *and* increased resiliency in case of unexpected crashes of writers (see Figure 6-2).

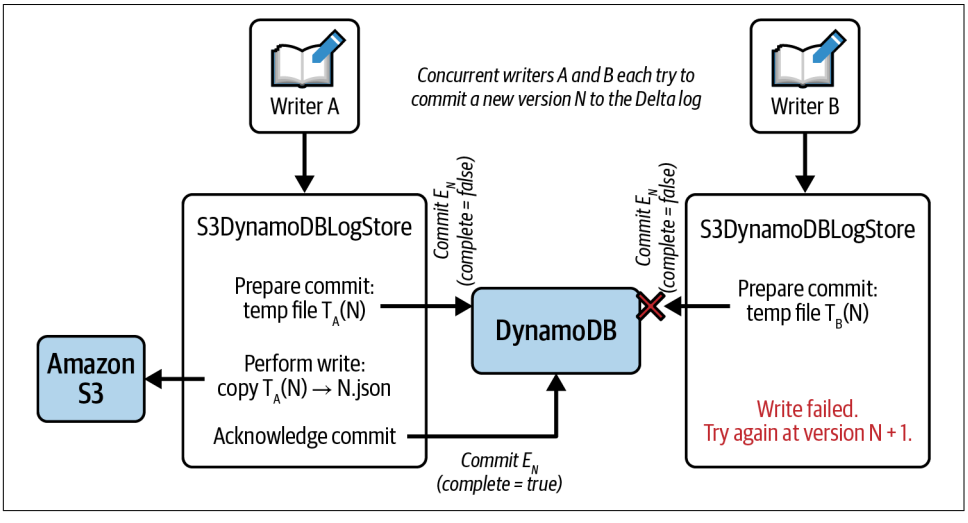

Figure 6-2. Coordination process for two concurrent writers using the `S3DynamoDBLogStore` process

The design considerations for `S3DynamoDBLogStore` are explained more in depth on the Delta Lake blog. Consult the Delta Lake documentation for the most up-to-date details on configuring the required DynamoDB table, or start with some of the examples in this book's GitHub repository.

DynamoDB lock. Applications with older dependencies may still rely on `dynamodb_lock`, but since this approach is deprecated, this section will not dive too deeply into its function and design. At a high level, a DynamoDB table is configured as a simple key-value store alongside the Python or Rust application. Prior to executing a write operation, the `deltalake` library will check DynamoDB for the presence of a lock item—essentially a key representing the table it wishes to write against. If that key does not exist, the library will:

- Write a time-to-live (TTL) lock item with the table's identifier
- Commit its Delta transactions
- Delete the item from DynamoDB

If a key already exists, however, applications must enter a retry/backoff loop and wait until the lock item is cleared from the DynamoDB table. Aside from only supporting Python/Rust writers, this approach has been deprecated because it provides poor resiliency in cases of writer failures. If a writer crashes or exits with errors in the critical section after a lock item has been created, all other writers must wait until either the original writer is able to reclaim its lock *or* the TTL expires. There are few guarantees of recoverability with this approach, and "single giant table lock" leads to concurrency limitations, which can acutely affect Lambda invocations.[7]

Concurrency with S3-compatible stores

A number of other storage systems implement the AWS S3 APIs, such as the open source MinIO and Cloudflare R2. However, not every S3-like implementation suffers from the eventually consistent behaviors of AWS S3, which may mean there is no need for coordination among concurrent writers.

Consult the service's documentation to determine whether it can support an atomic "copy if not exists" operation (sometimes referred to as `putIfAbsent`). Cloudflare R2, for example, supports atomic behavior *only* when custom headers are supplied to its REST APIs, which can be toggled in the `deltalake` packages via the `AWS_COPY_IF_NOT_EXISTS` environment variable.

For other services, the environment variable `AWS_S3_ALLOW_UNSAFE_RENAME` can be set to true to disable the coordinator/locking requirements of the `deltalake` packages.

What's Next

The native data processing ecosystem is blossoming, with dozens of great tools in Python and Rust being developed and coming to maturity. Most of this innovation is being done by passionate and inspired developers in the larger open source ecosystem.

Delta Lake plays a pivotal role via the `deltalake` Python package or Rust crate, allowing data applications to benefit from the optimized storage and transactional nature of Delta. The list of integrations and great tools continues to grow; following is a list of interesting projects that are worth learning more about:

7 Check out the blog post "Concurrency Limitations for Delta Lake on AWS" for more details on the DynamoDB lock's limitations.

Python:

- Pandas
- Polars
- Dask
- Daft
- LakeFS
- PyArrow

Rust:

- ROAPI
- kafka-delta-ingest
- Ballista
- DataFusion
- arrow-rs
- Arroyo
- ParadeDB

The delta-rs project and those listed here are only as productive as the people who show up, so you're invited to get involved! File bug reports, write user documentation, or create new open source projects that use Delta Lake to solve new problems!

CHAPTER 7

Streaming In and Out of Your Delta Lake

Now more than ever, the world is infused with real-time data sources. From ecommerce, social network feeds, and airline flight data to network security and IoT devices, the volume of data sources is increasing alongside the speed with which you're able to access it. One problem with this is that, while some event-level operations make sense, much of the information we depend on 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 that are commonly absent in most distributed data stores. With that in mind, in this chapter we dig into some of the details for stream processing with Delta Lake—namely, the functionality that is 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 *Learning Spark* by Jules S. Damji, Brooke Wenig, Tathagata Das, and Denny Lee [O'Reilly]). Then we'll look at the core functionality, some of the options we have available, and some common more advanced cases with Apache Spark. And to finish out the chapter, we will cover

a couple of related features used in Databricks, such as Delta Live Tables and how it relates to Delta Lake, and then review how to use the Change Data Feed functionality available in Delta Lake.

Streaming Versus Batch Processing

Data processing as a concept makes sense to us: during the data processing life cycle, we receive data, perform various operations on it, and then store it or ship it onward. So what primarily differentiates a batch data process from a streaming data process? Latency. There are different points at which to measure it, but *latency* is just the measure of time between records coming in and records going out. 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 latency in getting actionable insights from the raw data and will drive your decisions 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, which in turn provides us flexibility should requirements change over time. This is a simpler alternative to approaches like a lambda architecture running different systems for batch and stream processing.[1]

Each batch that we process has defined beginning and ending points—that is, there are boundaries placed in terms of time and format. For example, we might process data for each distinct calendar date as a batch. We may process "a file" or "a set of files" as a batch. In stream processing, we look at things 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 arrive. In the end, this unboundedness is really all that is needed to make a source a data stream. In Figure 7-1, the batch process equates to processing groups of six 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 such as Apache Flink or Apache Spark can work together with Delta Lake as either a starting point or an ending destination for data streams. These multiple roles mean Delta Lake can be used at multiple stages of different kinds of streaming workloads. Often we will see the storage layer as well

1 For a review of the lambda architecture pattern, we suggest starting with the Wikipedia page. It is essentially a parallel path architecture with a stream processing component and a batch processing component, both reading from the same source. The streaming process provides a faster view of the data, and the batch process ensures eventual accuracy.

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.

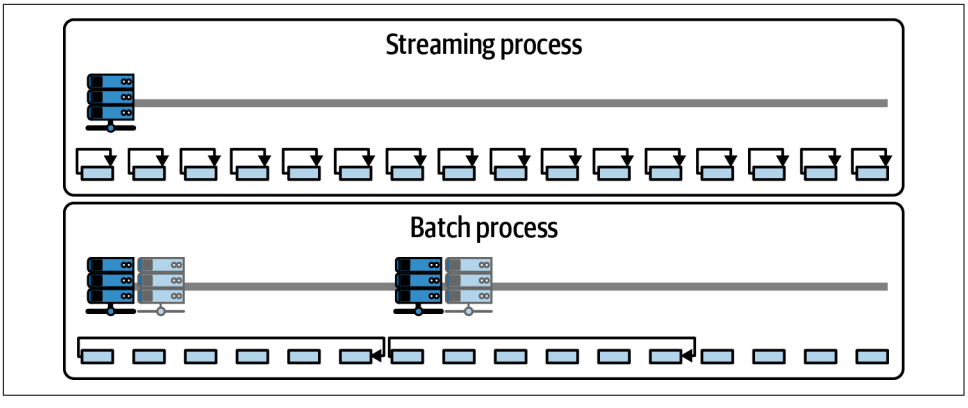

Figure 7-1. The biggest difference between batch and stream processing is latency; we can handle the files or messages individually as each becomes available or as a group

From a practical standpoint, the way we think about other related concepts such as processing time and table maintenance is affected by our choice between batch and streaming. If a batch process is scheduled to run at certain times, then we can easily measure the amount of time the process runs and how much data was processed and then 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—such as autocompaction and optimized writes, for example—can work in both realms. In Figure 7-2, we can see how, with modern systems, batch and streaming can converge, and we can focus instead on latency trade-offs 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 then 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 we 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 up with a review of a few of the different framework integrations available for use with Delta Lake.

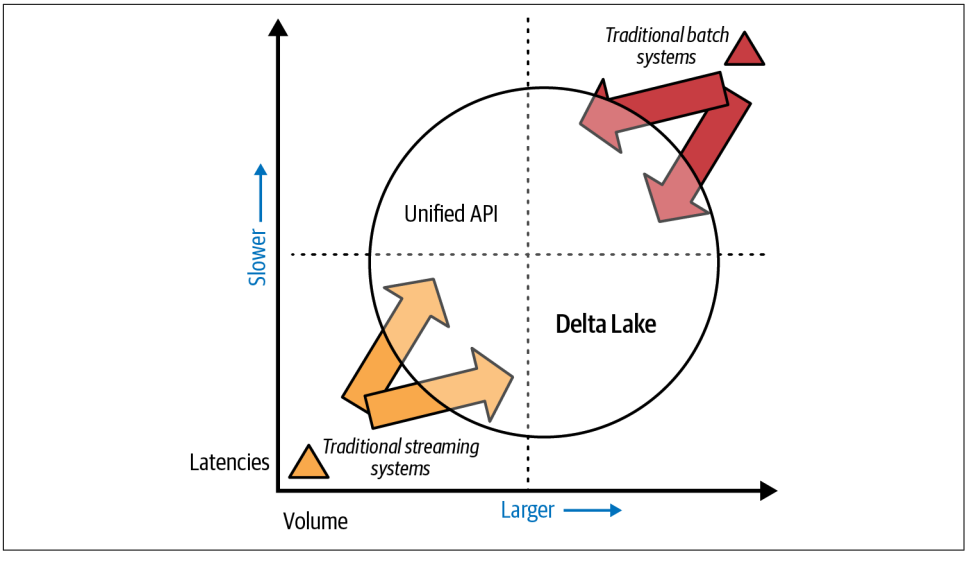

Figure 7-2. Streaming and batch processes overlap in modern systems

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 terms, such as *source* and *sink*, vary only a little from batch usage, while terms like *checkpoint* and *watermark* don't really apply to batch. 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* by Fabian Hueske and Vasiliki Kalavri (O'Reilly) or *Learning Spark*.

Source. A stream processing source is any of a variety of sources of data that can be treated as an unbounded dataset. Sources for data stream processing are varied and ultimately depend on the nature of the processing task in mind. There are a number of different message queue and pub/sub connectors used as data sources across the Spark and Flink ecosystems. These include many common favorites such as Apache Kafka, Amazon Kinesis, ActiveMQ, RabbitMQ, Azure Event Hubs, and Google's 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 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 such as 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, between origin and destination. Delta Lake functions extremely well as a sink, especially for managing large-volume, high-throughput streaming ingestion processes.

Checkpoint. It is usually important to make sure that you have implemented checkpointing 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, such as 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.

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

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 the following 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 microbatch level the progress of streaming from the source and writing to the sink, which for Delta Lake amounts to tracking the input or output files, respectively, as we'll see more of shortly.

Watermark. Watermarking is a concept of time relative to the records being processed. The topic and usage are somewhat more complicated for our discussion, and we would recommend reviewing the appropriate documentation. For our limited purposes, we can just use a working definition: a *watermark* is basically a limit on how late data can be accepted during processing. It is most especially used in conjunction with windowed aggregation operations.[2]

Apache Flink

Apache Flink is one of the major distributed, in-memory processing engines that supports both bounded and unbounded data manipulation. Flink supports many predefined and built-in data stream sources and sink connectors.[3] 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 Flink documentation for more detailed streaming connector information). While some, such as 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 that can be critical in multitool hybrid ecosystems or can 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 many input sources and sinks.[4] 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, in contrast to 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 maintained

2 To explore watermarks in more detail, we suggest the "Event-Time and Stateful Processing" chapter of *Spark: The Definitive Guide* by Bill Chambers and Matei Zaharia (O'Reilly).

3 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 Flink documentation.

4 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.

connector libraries, such as 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. Due in large part to the origins of Delta Lake and the longer history with Apache Spark, much of what's covered here will be Spark-centric, but we should note that many of the concepts 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 other options for stream data processing, and both couple with delta-rs reasonably well. This is a rapidly developing area that 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 (such as AWS Boto3, for example) for services that otherwise require larger-scale frameworks for interaction and thus cause 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 benefit from a significant reduction 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 when 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 targeted table or file and then add incremental updates as they are added. This allows for significant simplification of many processing architectures (such as 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, with `"delta"` denoting 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, where you instead start by building off of the Delta source object in a `DataStream` class and then 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 (such as asynchronous compaction operations) were not available or scalable in a way that could 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 trying to answer the question *How can I efficiently and reliably capture all the data?*

Many of the features of Delta Lake are there specifically to remedy 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; this allows you to make sure that you are actually capturing all the source data with some reliability guarantees. The metrics produced and emitted to the Delta log help you to analyze the consistency (or variability) of the streaming process, with counts of rows and files added during each transaction.

Most large-scale stream processing happens in *microbatches*, which in essence are smaller-scale transactions of similar larger batch processes. The result of this 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, such as 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 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.[5]

The medallion architecture with Delta Lake and Apache Spark in particular, which we will cover in depth in Chapter 9, becomes something of a middle ground in which we see Delta Lake as both a streaming sink and a streaming source working in tandem (see Figure 7-3). 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 and preserving clean data engineering practices.

5 You can find detailed descriptions, including error messages, in the "Concurrency Control" section of the Delta Lake documentation.

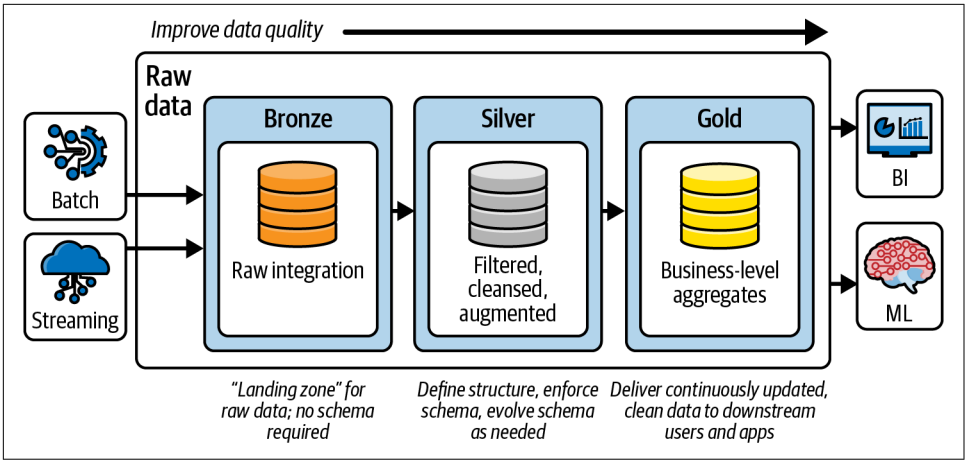

Figure 7-3. A visualization of the Databricks medallion architecture definition; you can see both a streaming source coming in with a Delta Lake table as a sink and then that table also becoming the source for the next process

Writing a streaming DataFrame object to Delta Lake is straightforward; it requires 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) and 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 the more technical side of the options we'll ultimately use in practice and go over a bit of background on instances in which you may wish to modify them. We'll start by looking 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. Last, we'll follow up by considering a few aspects of the relationship between time and our processing job.

Limit the Input Rate

When we're talking about stream processing, we typically have to find a balance among three concerns: accuracy, latency, and cost. We generally don't want to forsake anything on the side of accuracy (except in cases where we might want to drop stale records or limit scope), and so this usually comes down to a trade-off between 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 the control of the stream processing engine, but we have two additional options with Delta Lake that allow us more control over the size of microbatches:

`maxFilesPerTrigger`
: This sets the limit for how many new files will be considered in every microbatch. The default value is `1000`.

`maxBytesPerTrigger`
: This sets an approximate limit for how much data gets processed in each microbatch. This option sets a *soft max*, meaning that a microbatch 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 with 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 in each 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, these two options are ignored. This is not generally set by default. You can actually use both `maxBytesPerTrigger` and

`maxFilesPerTrigger` for the same streaming query, in which case the microbatch 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 does not know what came before, processing will begin at the earliest available transaction in the log, which means data can be skipped in the processing. A simple example of where this could occur is when the `logRetentionDuration` is set to, say, a day or two, but a processing job intending to pick up the incremental changes is run only 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, we've not really discussed something that we really ought to address. In earlier chapters we've seen how some 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 when we are streaming with Delta Lake, it assumes by default that we are streaming from an append-only type of source—that is, it assumes that the incremental changes that are happening are only the addition of new files. A question then arises: *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 expect to receive only 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 for receiving change data capture (CDC) records, because those typically won't be subject to other types of operations. There are two ways you can deal with these situations. The harder 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, which have rather different behaviors from each other despite the similarity in their names. 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: 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, such as when we need to purge individual user data to comply with the GDPR's right to be forgotten. 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 that 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 a particular file or 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 through `merge` logic or otherwise differentiate the data by inclusion of additional timekeeping information (i.e., add a `version_as_of` timestamp, or something 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.

Example

Let's consider an example. Suppose you have a Delta Lake table called `user_events` with `date`, `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 `readStream`:

```
# Python
streamingDeltaDf = (
    spark
    .readStream
    .format("delta")
```

```
        .option("ignoreDeletes", "true")
        .load("/files/delta/user_events")
        )
```

If you want to delete data based on a nonpartition column like `user_email` instead, then you will need to use the `ignoreChanges` option:

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

Similarly, if you update records against a nonpartition column like `user_email`, a new file is created that contains 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 to the most recent version. There are going to be times, of course, when we don't actually want to start with the earliest version, such as when we need to delete a checkpoint for the streaming process and restart from some point in the middle, or even from 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 to define an initial position to begin processing, and we can do that in one of two ways. 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. 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 the process 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 most easily done 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 that can be interpreted for us. This means our `startingTimestamp` parameter should look like one of the following:

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

Unlike with some of our other settings, we cannot use both options simultaneously here; we 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 apply only 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.

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

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

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

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

Initial Snapshot with withEventTimeOrder

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 dataset, this default ordering based on the modification time is correct, or if there is an event time field we can leverage in the dataset that might simplify the ordering of the data.

A timestamp associated with when a record was last modified (i.e., seen) doesn't necessarily align with the time an event happened. 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. Following is an example of 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` option to throttle the processing rate.

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

- The data drop issue happens only when the initial Delta snapshot of a stateful streaming query is processed in the default order.
- `withEventTimeOrder` is another of those settings that takes effect only at the beginning of a streaming query, so it cannot be changed after the query is started and while 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.
- If you are running a stream query with `withEventTimeOrder` enabled, you cannot downgrade it to a version that 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 in which you cannot use `withEventTimeOrder`:
  - If the event time column is a generated column and there are nonprojection transformations between the Delta source and the watermark
  - If there is a watermark 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 microbatch. 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 microbatch.

The setting `spark.databricks.delta.withEventTimeOrder.enabled true` can be set as a cluster-level Spark configuration, but be aware that doing this will make it apply 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 in which 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; instead, we may 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 microbatch. This is the method we would typically use to support writing a single stream source to multiple destinations. The problem we encounter is that if there are, say, two different destinations, and the transaction fails in writing to the second destination, then we have a scenario in which the processing state of each of the destinations is out of sync. More specifically, since the first write was completed and the second write 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 in which 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 microbatches 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 write 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 that can help us out 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 microbatch 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.

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.

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, at a table level, for the detection of duplicate write attempts that can be ignored. This means that if a write is interrupted during the processing of 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 microbatch, but then in the `foreachBatch`, with the write operations now being checked at a table level of granularity, we write only to the table or tables that were not able to complete successfully before, because we will have the same `txnAppId` and `txnVersion` identifiers.

In the case that 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 will 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 could 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 in which we tend to see `foreachBatch` used for stream processing. Think about some of the limitations we have seen 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, such as processing CDC records. 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 in Chapter 3 and saw that it allows us to use matching criteria to update or delete existing records and append others that don't match the criteria—that is, we can perform upsert operations. Since `foreachBatch` lets us treat each microbatch like a regular DataFrame, then at the microbatch 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 such as `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 by writing out results to an intermediate table. Caching the source data may help as well, 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 in which 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 to 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 we'll allow for a delete mechanism and otherwise update existing customers or add new ones as they appear.[6] 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 that we will run the merge operation for every received batch rather than for 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)
```

6 For additional details and examples on using merge in `foreachBatch`, e.g., for SCD Type II merges, see the Delta Lake documentation.

```
    .outputMode("update")
    .start()
    )
```

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

Delta Lake Performance Metrics

An often overlooked but very helpful thing 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 cost estimating, capacity planning, or troubleshooting when issues arise. We've already seen a couple of cases in which 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 in which we want to manually set starting and ending boundary points for processing with Delta Lake, and these are generally aligned to versions or timestamps. Within those boundaries, we can have differing numbers of files and so forth, and one of the concepts we've seen is 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 that help to track backpressure—that is, 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 won't explore them individually.

Comparing the `inputRowsPerSecond` metric with the `processedRowsPerSecond` metric provides a ratio that can be used to measure relative performance and that might indicate whether a job should have more resources allocated to it or whether 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 is 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 requires 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 available only in Databricks that rely on or frequently work in conjunction with Delta Lake and that deserve mention.

Auto Loader

Databricks has a somewhat unique Spark Structured Streaming source known as Auto Loader, though it is really better thought of as 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 in which Delta Lake is commonly the destination sink. This is partly because it provides a natural way to incrementalize batch processes so as to integrate some of the benefits of stream processing, such as offset tracking.

The `cloudFiles` source actually has two different methods of operation: one is to directly run file-listing operations on a storage location, and the other is to listen on a notifications queue tied to a storage location. Whichever method is used, it will quickly become 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 filenames in the specified source directories. Refer to the section "Delta Live Tables" on page 163 for an example of the most common usage.

One fairly standard application of Auto Loader is to use 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. This is quite commonly 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 of the need for more complicated processes based on lambda architecture—so much so that Databricks also built a framework largely centered around this approach.

Delta Live Tables

Databricks offers a data engineering pipeline framework running on top of Delta Lake called Delta Live Tables (DLT) that combines incremental ingestion, streamlined ETL, and automated data quality processes like *expectations*. DLT 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: 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.

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 a lot of the features we discussed by automating many of the best practices commonly used across the field. The things that it can 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 Delta Live Tables 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 the integration of 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 we basically have just three 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 then where exactly does CDC come into play?

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 succinctly, CDC "is the process of ingesting changes from a source database system."[7]

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 to let you track the changes to a Delta Lake table. Once it is enabled, you get all the 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

7 Joe Reis and Matt Housley, *Fundamentals of Data Engineering: Plan and Build Robust Data Systems* (O'Reilly), 163, 256.

changes will have the required row data with some additional metadata showing the change type.

CDF is available in Delta Lake 2.0.0 and above. Levels of support for using CDF on tables with column mapping vary by the version you are using:

- Versions ≤ 2.0 do not support streaming or batch reads for CDF on tables that have column mapping enabled.
- For version 2.1, only batch reads are supported for tables with column mapping enabled. It also requires that there are no nonadditive schema changes (no renaming or reordering).
- For version 2.2, both batch and streaming reads are supported for CDF from tables with column mapping enabled as long as there still are no nonadditive schema changes.
- For versions ≥ 2.3, batch reads for CDF for tables with column mapping enabled can now support nonadditive schema changes. CDF 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 cases in which the version range specified spans a nonadditive schema change.

Using Change Data Feed

While ultimately it is up to you whether or not to leverage the CDF feature in building out a data pipeline, there are some common use cases in which 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 the way you might think about leveraging CDF:

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 (extract, transform, load) and ELT (extract, load, transform) operations, because CDF 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 for security auditing of sensitive data.

We should also note that using CDF may not necessarily add any additional storage. Once it is enabled, what we actually find is that there is no significant impact on processing overhead. The size of change records is pretty small; in most cases their size is much less than that of 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. 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 that it can efficiently compute the change data feed directly from the transaction log, and thus these records can be skipped altogether in the folder. Since these are often among the most common operations, this capacity strongly aids in reducing overhead.

Since the *_change_data* folder is not part of the current version of table data, the files in the 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 the retention policy.

Enabling the change data feed

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

For a new table, simply set the table property `delta.enableChangeDataFeed` to `true` within the `CREATE TABLE` command:

```
-- SQL
CREATE TABLE student (id INT, name STRING, age INT)
TBLPROPERTIES (delta.enableChangeDataFeed = true)
```

For an existing table, you can instead alter the table properties with the `ALTER TABLE` command to set `delta.enableChangeDataFeed` to `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 to 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 and reading it 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 that is 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 provide either version numbers or timestamp strings to set both the starting and ending boundaries. 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 use either 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 precedes when the change data feed was enabled:

```
# Python
# version as ints or longs
(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—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, so we otherwise 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")
```