```
  .load("/pathToMyDeltaTable")
)
```

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

```
-- SQL
set
delta.changeDataFeed.timestampOutOfRange.enabled
=true;
```

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

Schema

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

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

Commit version
: The `_commit_version` column is a long integer type column noting the Delta Lake file/table version from the transaction log that the change belongs to. When reading the change feed as a batch process, it will be at or in between the boundaries defined for the query. When read as a stream, it will be at or greater than the starting version and will continue to increase over time.

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

As an example, suppose there was a (fictional) discrepancy in the People 10 M dataset because the details actually belonged to a relative. We can update the errant record, and when we view the change feed, we will see the original record values denoted as the preimage and the updated values denoted as the postimage. We'll update the set on the mistakenly input name and correct the name and gender of the individual. Afterward, we'll view a subset of the table highlighting the before and after change feed records to see what it looks like. We can also note that it captures both the version and the timestamp from the commit at the same time:

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

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

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

Conclusion

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

CHAPTER 8

Advanced Features

In this chapter the focus is a bit less on how to interact with and use Delta Lake tables than you may have found in other chapters. Instead, the main focus here is a handful of advanced features that you'll find useful. At heart, these Delta Lake features have more to do with metadata than anything else. The first thing we'll look at is how you can use generated columns as part of table definitions to reduce the amount of insertion or transformation work required for data loading operations. After that, we'll look at how Delta Lake metadata helps drive higher data quality standards and provides richer information to users through constraints and comments. Last, we'll share some insight into how deletion vectors can speed up many operations against applicable tables. Each of these features shows how the power of Delta Lake is enhanced through well-thought-out uses of table metadata and the transaction log.

Generated Columns, Keys, and IDs

One of Delta Lake's lesser-utilized features is the ability to use generated columns in Spark to create column values dynamically. Put simply, generated columns allow you to add simple statements to a table definition that will create the values of a column when applied, rather than relying on the insertion of values for those columns as new data is inserted into the table. The use of these can vary, from identity columns to new columns that perform simple conversions of input columns.

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

You can include two types of generation expressions in a table definition that allow you to control whether values will always be generated or are generated by default. Columns that are always generated cannot be overwritten, whereas you can specify values during insertion operations for columns that are generated by default. Usually, the choice is to always generate columns because that option is simpler, but you may have cases in which you wish to explicitly be able to override a generated value with a specific value. For example, suppose you want to set a transaction at the beginning of each month to increment the beginning values of keys to the next thousand or million; you would then use generate by default so you could manually set that initial monthly transaction. Regardless, if you want to generate columns, you need to add the generation expression in your original table definition. In the following example, you can apply a Spark SQL function to an incoming date column to extract the year as a column. This can also be done to typecast columns or even to create more complex data structures such as structs out of incoming columns:

```
-- SQL
CREATE TABLE if not exists summary_cases(
    state STRING,
    fips INT,
    cases INT,
    deaths INT,
    county STRING,
    year INT GENERATED ALWAYS AS (YEAR(date))
    )
USING DELTA
```

One of the most common applications of generated columns is to create identity or surrogate key columns.[1] In the past you've been able to do this with other methods, such as leveraging external libraries to create UUIDs or using hashing methods to create unique keys. Delta Lake offers some advantages over these methods. By baking the ability to generate columns into the foundation of the format, you can avoid running into issues that stem from the nondeterminism of many of these previous methods and get simpler ID columns that are more human-readable than the results of the hashing methods.

Defining identity columns is just a slight extension of the generation expressions, except that there is no required SQL statement to perform some transformation. Instead, using the `IDENTITY` keyword triggers some actions behind the scenes that make this work. What you get is, in essence, a bit of automated tracking that maintains the incremental nature of the identity column(s):

1 If you'd like to further explore the use of various kinds of key-based relationships, we recommend *Learning SQL*, 3rd ed., by Alan Beaulieu (O'Reilly), *Deciphering Data Architectures* by James Serra (O'Reilly), and *Data Management at Scale*, 2nd ed., by Piethein Strengholt (O'Reilly).

```
-- SQL
id BIGINT GENERATED BY DEFAULT AS IDENTITY

-- SQL
id BIGINT GENERATED ALWAYS AS IDENTITY
```

These identity columns serve as surrogate keys that can be leveraged throughout your downstream applications to create primary and foreign key relationships, or potentially even for slowly changing dimension (SCD) types of tables. It's relevant to note that Databricks has a feature to make these primary and foreign key relationships enforceable via Unity Catalog.[2]

At an implementation level, there are a few secrets to the recipe to be learned from the Delta Lake protocol definitions for identity columns. The main takeaway is that whenever overwriting values is disallowed, a simple monotonic function generates the values for the column. This means you can also feel assured that the generation of values is an efficient operation, as it primarily relies on table metadata and simple integer mathematics.[3]

There are a couple of things under the covers that you will want to be aware of. First, when columns are generated using `ALWAYS`, there is a constraint applied to the table (you will find more information on constraints later in this chapter). This means that attempting to provide values for generated columns during insertion operations will yield an error for your transaction. Second, using generated columns poses some limitations on usage; for example, you cannot partition a table by a generated identity column, and concurrent transactions are disallowed. Last, for identity columns specifically, you must use the `BIGINT` type, whereas with other generated columns the type definition is more flexible, depending on your actual application.

Comments and Constraints

Delta Lake metadata helps to describe and/or provide you with more granular information about the table than would otherwise be available. Here the focus is on two specific components of the table metadata, *comments* and *constraints*, with each used distinctly. The creators and maintainers of tables commonly use the first kind, namely table comments. You can use these to provide richer context to column or table data. Savvy users, consumers of the data, may be able to gain a lot of additional information or save a lot of time by not having to reverse-engineer features. The second kind of metadata is more operational. Constraints are overall less common

2 See the Databricks documentation for usage examples and additional details.

3 For an extended discussion of the benefits and use of surrogate keys, see the blog post "Identity Columns to Generate Surrogate Keys Are Now Available in a Lakehouse Near You!" by Franco Patano.

in many applications, but they can play a huge role in improving the quality of data tables and detecting aberrations earlier than other methods.

Tags are map objects that contain additional metadata about transactional operations. They are an optional field in add or remove files, deletion vector files, and CDC files. When using the checkpoint V2 space, both the checkpoint and the associated sidecar files can also have tags. Note that remove actions in the checkpoint are tombstones used only by `VACUUM` and do not contain the stats field or the tags field. These are mostly intended for use at the implementation level to support or add new features to a particular Delta Lake implementation. A common use of tags is to annotate table properties in different processing engines. These are not explored in depth here, as most users will not use them, but we want to mention them, as they are distinctly different from the tags used in catalogs.

Comments

Comments should be used often and well. There are many kinds of comments you might wish to include for different kinds of informational purposes. They can convey important information about ownership or column design information. Possibilities for types of constructive comments might include:

Instructive
: Sometimes when creating different datasets, we make decisions about the layout that may not be transparent to end users. If a table does not have a unique key column but requires a combination of multiple columns to have a unique key, we might wish to capture in the comments for those columns which columns they can be combined with to have a unique key.

Explanatory
: In some cases, it might be useful to annotate the origin of data residing in a column, its security classification level, its intended users, or information about the derivation of calculated fields. Denoting the data origin is even more valuable when Delta Lake is used outside of environments that automatically capture lineage information. All of these provide enriched information to consumers on demand and can increase the delivered value of data products. This can be particularly useful in cases in which a table includes nonstandard key performance indicators (KPIs) with a reference to design documentation.

What you include in the comments is ultimately up to you and your organization. We recommend that you come up with a standard definition for usage and stick with it, as the many benefits that may potentially be gained from the additional information can greatly improve the experience.

We highly recommend the use of a catalog, such as Unity Catalog, that also supports table-level comments (and ideally tags) as well as additional features like lineage. Downstream consumers of a table, especially when accessing the table from multiple systems, might benefit from information about its maintainers or a point of contact, in case any issues arise with a means of contacting them (such as an email address). In many cases, information captured in table comments may also be beneficial if replicated to this table level for convenience.

Here is a quick example showing how you can easily add columns to a table at the time of creation. This allows you to provide clarifying or explanatory quick notes for all columns in the table at the same time; all you need to do is include the comments as part of the table schema definition:

```
-- SQL
CREATE TABLE example_table (
  id INT COMMENT 'uid column',
  content STRING COMMENT 'payload column for text'
)
USING delta
```

Sometimes your initial comments may not be as clear to table consumers as intended. In those cases, you can update individual column comments to refine them. This also gives you the flexibility you might need to include additional information not available at the time of table creation:

```
-- SQL
ALTER TABLE example_table
ALTER COLUMN id
COMMENT 'unique id column'
```

One last area that can be rather useful in many cases is adding transactional comments to table changes. You can set this option during individual operations as part of the table options when using the Python API, set it for a session in SQL and reuse it until you are done with those updates, or change it as many times as needed throughout the session.

When using the Python API as a table option, you just want to set the `userMetadata` option with your custom metadata:

```
# Python
(spark
 .read
 .table(<source>)
 .write
 .format("delta")
 .option("path", <destination>)
 .option("userMetadata", "custom commit metadata for the creation operation")
```

```
    .save()
)
```

In SQL the same option is set, but as mentioned, it will happen at a session level. This means you will need to remember to update it if you do not want the message to continue:

```
-- SQL
SET spark.databricks.delta.commitInfo.userMetadata='comment here'
```

You might find this useful, for example, when you are running multiple deletes and updates or other operations and want to denote that they all belong to the same set of actions. You can reset the `userMetadata` to `NULL` if you want to return to the default behavior.

Delta Table Constraints

Delta Lake table metadata can be used for more than just providing additional information about the table. In some cases, metadata can also create additional actions that help to provide safeguards and guarantees for your data assets in Delta Lake. You've already seen this with table versions stored in the metadata (Chapter 3) and how they allow for time travel views of the table to see prior versions without necessarily trying to roll back operations. Another kind of action-inducing metadata is when you include constraints on a table.[4]

Tables using writer version 7 and above need to have the feature name `checkConstraints` in the `writerFeatures`. Versions 3–6, however, always support `CHECK` constraints.

`CHECK` constraints are stored in your table metadata just like `userMetadata` and column comments. They are stored as key-value pair objects. You can see the value of any constraint for a particular table by name under the attribute `delta.constraints.<name>`. The value is stored as a SQL expression that will return a Boolean value. Because of this expression's nature, the columns specified in the expression must exist in the table. All rows in the table must satisfy the constraint expression by returning `true` when expressions are evaluated.

When you add a constraint to a table, it will check the existing table data to make sure it is compliant. In cases where it is not, the `ALTER TABLE` execution will fail. Similarly, when writing data to the table after a constraint has been added, every row

4 For an extended overview, we suggest Matt Powers's blog post on constraints for Delta Lake.

must satisfy the constraint expression or the write operation will fail. This can help you avoid writing malformed or noncompliant data to your table.

One of the most common uses of this feature is just adding a `NOT NULL` argument to guarantee that certain columns always get populated.[5]

To do this during table creation, you can include it as part of the column arguments, similar to and alongside column comments:

```
-- SQL
CREATE TABLE IF NOT EXISTS example_table (
  id INT COMMENT 'uid column' NOT NULL,
  content STRING COMMENT 'payload column for text'
)
USING delta
```

It's important to note here that only `CHECK` constraints added via `ALTER TABLE` commands will be represented in the table metadata, but you can feel assured that the null constraints set at creation will also be effective. Setting constraints via `ALTER TABLE` is also relatively straightforward. Consider the following example you could use to ensure you have nonnegative ID column values:

```
-- SQL
ALTER TABLE example_table
ADD CONSTRAINT id CHECK (id > 0)
```

Whichever way you choose to set various constraints, they are an effective way to increase data quality and enhance confidence in your data platform.

Deletion Vectors

Sometimes we can look at a problem and think of different ways to solve it. A feature in Delta Lake called *deletion vectors* is a great example of this idea. Chapter 10 provides a look at several ways you can optimize for either the table readers or the table writers for performance and the trade-offs you might need to make in that process. While deletion vectors certainly have a place in that discussion, they also deserve treatment and investigation as one of the advanced features in Delta Lake, so they are referenced here instead. The reason for this is that the way they work introduces a new concept that deserves a bit of explanation. Another reason is that the term *deletion vector* defines more the form and function of what the process does rather than how it helps you as a feature. One of the key benefits is that it gives you the ability to do a Merge-on-Read (MoR) operation. It dramatically reduces the

5 For an explanation of the relationship between constraints and nullability in Spark, as well as additional examples, see Matt Powers's blog post.

performance impact of doing simple delete operations and instead postpones those operations to run as a batch at a more convenient time in the future.

Merge-on-Read

What does *Merge-on-Read* mean? It means that, rather than going through the operation of rewriting a file at the time of deleting a record or set of records from a particular file, you instead make some kind of a note that the record or records are deleted. Thus you get to postpone the performance impact of actually performing the delete operation until a later time. Usually you will do this when you can run an `OPTIMIZE` operation or a more complicated `UPDATE` statement. With columnar files (Parquet, Delta, Iceberg, etc.), row-level deletions invoke relatively expensive rewrite operations of entire files containing those rows. Of course, if someone were to read the table after a Merge-on-Read operation has been initiated, then it will merge during that read operation. That's kind of the point, because it allows us to minimize the performance impact of performing a simple delete operation and to just perform it at a later time, when you are already filtering on the same set of files while reading them. For other cases, you can then avoid the deletions in situations where you don't need them to happen straight away. This further allows you to push multiple (or many) deletes into a single large batch later.

Deletion vectors are a way to get this kind of Merge-on-Read behavior.[6] Put simply, deletion vectors are just a file (or multiple files) adjacent to a data file that allows you to know which records are to be deleted out of the data file and to save the delete (rewrite) operation for a later point in time that is more efficient and convenient. *Adjacent* in this case is relative: deletion vector files are part of the larger set of files that make up a Delta Lake table, but in partitioned tables you will notice that the deletion vector files sit at the top directory level rather than within the partition directories. You can observe this in the coming examples.

We might call a deletion vector file a *sidecar* file since it is a file that sits alongside the other files in a table. In Delta Lake, however, we would want to distinguish this from sidecar files that are a formal component of the V2 checkpoint specification and that specify add or remove file operations.

For most cases in which performance is being optimized for the Delta Lake writer operations, deletion vectors present a unique opportunity to reduce latency in the write operations, as their use avoids cases of rewriting files where otherwise there is no data change. This does come at a small cost of an additional filtering operation

6 There's an excellent blog post by Nick Karpov exploring deletion vectors in great detail.

during subsequent read operations, but overall the performance impact is not very large.

To read a table with deletion vectors, you must use a client with at least reader version 3. This presents an area for potential conflicts. If you have an environment using older clients with lower reader versions, this could make tables inaccessible from those environments. Writing merely requires writer version 7. For example, in Databricks you need to use a Databricks Runtime (DBR) version of 14 or higher to write deletion vectors, but you only need to be on DBR version 12.1 or higher to be able to read them. Deletion vectors will work only when they are enabled via the `enableDeletionVectors` table property.

Setting the property is a simple `ALTER TABLE` command:

```
-- SQL
ALTER TABLE tblName
SET TBLPROPERTIES ('delta.enableDeletionVectors' = true);
```

Stepping Through Deletion Vectors

In this section we present an extended example to highlight the file-level changes that occur while using deletion vectors. You will see the familiar `covid_nyt` dataset used throughout the book, but with the size reduced and partitioned in such a way as to highlight deletion vector behavior specifically. As a bit of a roadmap to guide you, we'll show you these steps:

1. Create a table and identify some specific values to delete.
2. Enable deletion vectors.
3. Apply deletion operations against the table and inspect the file structure after each operation.

This example should help you to understand the nature of how the deletion vectors are operating. It's worth mentioning here that the original table creation does not need to occur in an environment that supports deletion vectors, but once the feature is enabled, read and write operations will be subject to the aforementioned version constraints.

First, create the reduced-size table; this makes it easier to view all the files simultaneously:

```
# Python
from pyspark.sql.functions import col
(
spark
 .read
 .load("rs/data/COVID-19_NYT/")
 .filter(col("state")=="Florida")
```

```
 .filter(
col("county").isin(
['Hillsborough', 'Pasco', 'Pinellas', 'Sarasota']
))
 .repartition("county")
 .write
 .format("delta")
 .partitionBy("county")
 .option("path", "nyt_covid_19/")
 .save()
 )

(
spark
.read
.load("nyt_covid_19/")
.write
.mode("overwrite")
.format("delta")
.saveAsTable("nyt")
)
```

Next, identify a single record from the table as a deletion target (for partition-level delete operations, you will be able to use any partition value):

```
# Python
spark.sql("""
select
  date,
  county,
  state,
  count(1) as rec_count
from
  nyt
where
  county="Pinellas"
and
  date="2020-03-11"
group by
  date,
  county,
  state
order by
  date
""").show()
```

```
date          county      state      rec_count
2020-03-11    Pinellas    Florida    1
```

Now enable deletion vectors on the table:

```
# Python
spark.sql("
ALTER TABLE nyt SET TBLPROPERTIES ('delta.enableDeletionVectors' = true);
")
```

Using `tree` or a file browser, verify the table structure before making any further changes. Since the table data was partitioned by county, you will see four resulting partition directories. Also, when the deletion vector operation was enabled, it incremented the table version and added a transaction to the *_delta_log* subdirectory. This allows for traceability across table transactions, which is useful if something downstream is not working right later:

```
# BASH
!tree spark-warehouse/nyt/
spark-warehouse/nyt/
├── county=Hillsborough
│   └── part-00000-6cf1fac7-1237-48b5-a7ca-ce824054a997.c000.snappy.parquet
├── county=Pasco
│   └── part-00003-dc22f540-c7f7-449c-8dc1-816f0f357075.c000.snappy.parquet
├── county=Pinellas
│   └── part-00001-42060e31-83e8-48d2-9174-02325ca5e686.c000.snappy.parquet
├── county=Sarasota
│   └── part-00002-dfb35d92-25bc-4caf-8aa0-1228143444a7.c000.snappy.parquet
└── _delta_log
    ├── 00000000000000000000.json
    └── 00000000000000000001.json
```

Apply a single deletion against the previously identified record:[7]

```
# Python
spark.sql("""
delete from
  nyt
where
  county='Pinellas'
and
  date='2020-03-11'
""").show()
```

7 Adding the `show` command at the end of the delete operations yields the number of affected rows in your output; otherwise you don't see this until checking the transaction log.

Check the results, and notice that all the original files still exist in the table. The only addition is a new file outside of the partition directories at the top level of the table. This is the deletion vector file from the delete operation:

```
# BASH
!tree spark-warehouse/nyt/
spark-warehouse/nyt/
├── county=Hillsborough
│   └── part-00000-6cf1fac7-1237-48b5-a7ca-ce824054a997.c000.snappy.parquet
├── county=Pasco
│   └── part-00003-dc22f540-c7f7-449c-8dc1-816f0f357075.c000.snappy.parquet
├── county=Pinellas
│   └── part-00001-42060e31-83e8-48d2-9174-02325ca5e686.c000.snappy.parquet
├── county=Sarasota
│   └── part-00002-dfb35d92-25bc-4caf-8aa0-1228143444a7.c000.snappy.parquet
├── deletion_vector_7de8988e-d96d-447c-9f99-1428e354907a.bin
└── _delta_log
    ├── 00000000000000000000.json
    ├── 00000000000000000001.json
    └── 00000000000000000002.json
```

Now apply two more deletion operations—one that aligns to a partition, and another that traverses multiple partitions:

```
# Python
spark.sql("""
delete
from
  nyt
where
  county='Pasco' # This is an entire partition
""").show()

spark.sql("""
delete
from
  nyt
where
  date='2020-03-13'# This has records in multiple partitions
""").show()
```

Inspect the files again. Notice that only one new deletion vector appears in this case:

```
# BASH
!tree spark-warehouse/nyt/
spark-warehouse/nyt/
├── county=Hillsborough
│   └── part-00000-6cf1fac7-1237-48b5-a7ca-ce824054a997.c000.snappy.parquet
├── county=Pasco
│   └── part-00003-dc22f540-c7f7-449c-8dc1-816f0f357075.c000.snappy.parquet
├── county=Pinellas
│   └── part-00001-42060e31-83e8-48d2-9174-02325ca5e686.c000.snappy.parquet
├── county=Sarasota
```

```
│   └── part-00002-dfb35d92-25bc-4caf-8aa0-1228143444a7.c000.snappy.parquet
├── deletion_vector_7de8988e-d96d-447c-9f99-1428e354907a.bin
├── deletion_vector_eda97b62-a3df-4b8f-885d-68295f324c2d.bin
└── _delta_log
    ├── 00000000000000000000.json
    ├── 00000000000000000001.json
    ├── 00000000000000000002.json
    ├── 00000000000000000003.json
    └── 00000000000000000004.json
```

So what happened? The answer is rather straightforward and is partially revealed if you check the `operationMetrics` from the transaction log. It shows that in the first delete operation, you get `numDeletionVectorsAdded: "1"`, which corresponds to the number of records we deleted (`numDeletedRows: "1"`) because it lies within a single partition file. The third deletion operation instead shows `numDeletionVectorsAdded: "3"`, which also directly corresponds to `numDeletedRows: "3"` as before. However, two additional entries appear that you should note: `numDeletionVectorsRemoved: "1"` and `numDeletionVectorsUpdated: "1"`. Delta Lake is compacting the deletion vectors into a single file when they apply to the same partition. You might then ask: *Why didn't I get another file for the second delete operation?* Since the operation aligned with the boundaries of the entire partition, Delta Lake simply removed the file that appears in the transaction log as `numRemovedFiles: "1"`. The original deletion vector is now a stale file, and the byte-level information in the new deletion vector also contains the old information. The extra data file and the original deletion vector file still behave by normal retention rules, so until you run a vacuum operation, they remain alongside the active table files.

The schema of these deletion vectors themselves is relatively straightforward. A deletion vector will specify the application storage type, a path or inline specification, an offset when applicable (depending on the specified storage type), the size in bytes when applicable (also depending on the storage type), and the cardinality of the deletion operation. These specifications will also be present in the transaction log as part of later affected operations, like an `add` action taking place in the presence of a deletion vector. Since these operations are typically implemented at the engine level, there's no need to explore them further, but if you are curious and want additional details on the exact schema definition, check out the "Deletion Vector Descriptor Schema" section of the protocol document.

Conclusion

Delta Lake's advanced features such as generated columns, constraints, comments, and deletion vectors, while minimal to implement, can yield enormous impacts. These features enhance data quality, provide richer metadata, and optimize performance for deletion-related operations.

Generated columns allow for the dynamic creation of column values based on expressions, reducing data loading work. Constraints such as `CHECK` constraints enforce data quality rules and detect issues earlier. Comments enable the annotation of tables and columns with valuable context for users. Deletion vectors enable a Merge-on-Read approach, postponing the performance impact of deletes until a more convenient time, such as during reads or optimizations. Overall, these advanced Delta Lake metadata capabilities show how the power of Delta Lake is augmented through strategic uses of table metadata and the transaction log, delivering higher data quality standards and richer information to provide an enhanced experience for data consumers.

CHAPTER 9
Architecting Your Lakehouse

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

Our goal for this chapter is not just to offer a collection of ideas, patterns, and best practices but to offer you a field guide. We've provided the right information, reasoning, and mental models so that the lessons learned here can coalesce into clear blueprints for architecting your own data lakehouse. Whether you are new to the concept of the lakehouse, unfamiliar with the medallion architecture for incremental data quality, or attempting your first foray into working with streaming data, we'll take this journey together.

What we'll learn:

- What the lakehouse architecture is
- Using Delta Lake as the foundation for implementing the lakehouse architecture
- The medallion architecture
- Streaming medallion architecture

The Lakehouse Architecture

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

What Is a Lakehouse?

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

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

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

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

- If the lakehouse is a hybrid architecture combining the best of the data lake and the data warehouse, then mustn't it be better than the sum of its parts?
- Why does the flexibility, cost efficiency, and unbounded data scaling inspired by traditional data lakes matter for all of us today?
- Why do the benefits of the data lake only truly matter when coupled with the benefits of schema enforcement and evolution, ACID transactions, and proper data management, as inspired by traditional data warehouses?

Learning from Data Warehouses

The data warehouse emerged to fix the issue of data silos within large enterprises and to simplify business intelligence (BI) and analytical decision making. While the data warehouse exists as a centralized solution to solve structured data problems within a given data domain, physical limitations within the data warehouse architecture meant costs would increase proportionally to the size and scale of the data within the

warehouse. These physical limitations were attributable to data being stored locally (nondistributed) in what is known as a vertically scaling architecture.

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

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

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

Today, support for common machine learning and data science workflows—which require custom data types and formats supporting unstructured (images), semistructured (CSV, JSON), and fully structured data (Parquet/ORC), as well as the ability to easily read entire tables into memory using efficient file skipping, column pruning, and other data reduction techniques—is missing from traditional data warehouses. Rather than relying on the raw data required to train and test models, teams must actively query the warehouse to produce the correct input datasets, which can be tricky, especially when utilizing iterative algorithms due to multiple roundtrips if explicit cache points are skipped.

Learning from Data Lakes

The data lake emerged to store raw (unprocessed) data in a wide variety of formats (CSV, JSON, ORC, text, binary) within a distributed filesystem, the popular choice at the time being the Hadoop Distributed File System (HDFS). Utilizing commodity hardware, the data lake could be utilized to run distributed processing jobs (Map-Reduce) or be leveraged to act as a staging area for data to be loaded into the data warehouse. Today, many workloads still follow similar patterns, utilizing cloud-based object stores or other managed elastic storage and elastic compute to power data lakes. So how does this fit into the lakehouse story?

The data lake provides a solution for storing raw feeds of data (as files) that can be processed directly for data science and machine learning use cases, supporting data

formats that are unavailable within the data warehouse. These feeds of data found another use through being transformed to *keep the data warehouse in sync* using the dual-tier data architecture, which is covered in the next section.

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

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

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

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

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

The Dual-Tier Data Architecture

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

Rather than having a single hop from the operational data system (siloed data) into the data warehouse (shared) or into the data lake, the dual-tier architecture relies on extract, transform, load (ETL) jobs to manage consistency, utilizing data from the data lake to populate the data warehouse. This is what is shown in Figure 9-1.

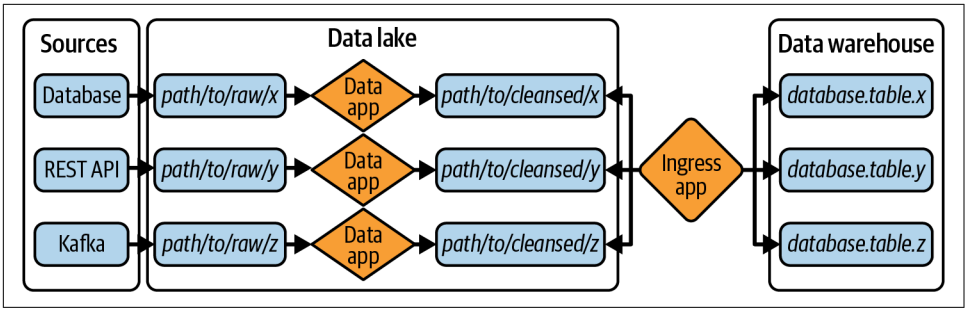

Figure 9-1. The dual-tier data architecture

The diagram shows the following flow:

1. Extract operational data from *siloed sources* for writing into landing zones (*/raw/**).
2. Read, clean, and transform the data from */raw* and write the changes to */cleansed*.
3. Read from */cleansed* (could do additional joining and normalizing with other data) before writing out to the data warehouse.

As long as the workflow completes, the data in the data lake will always be in sync with the warehouse. This pattern also enables support for unloading or reloading tables to save cost in the data warehouse. This makes sense in hindsight.

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

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

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

Lakehouse Architecture

The lakehouse is a hybrid data architecture that combines the best of the data warehouse with the best of the data lake. Figure 9-2 provides a simple flow of concepts through the lens of what use cases can be attributed to each of the three data architectures: the data warehouse, the data lake, and the data lakehouse.

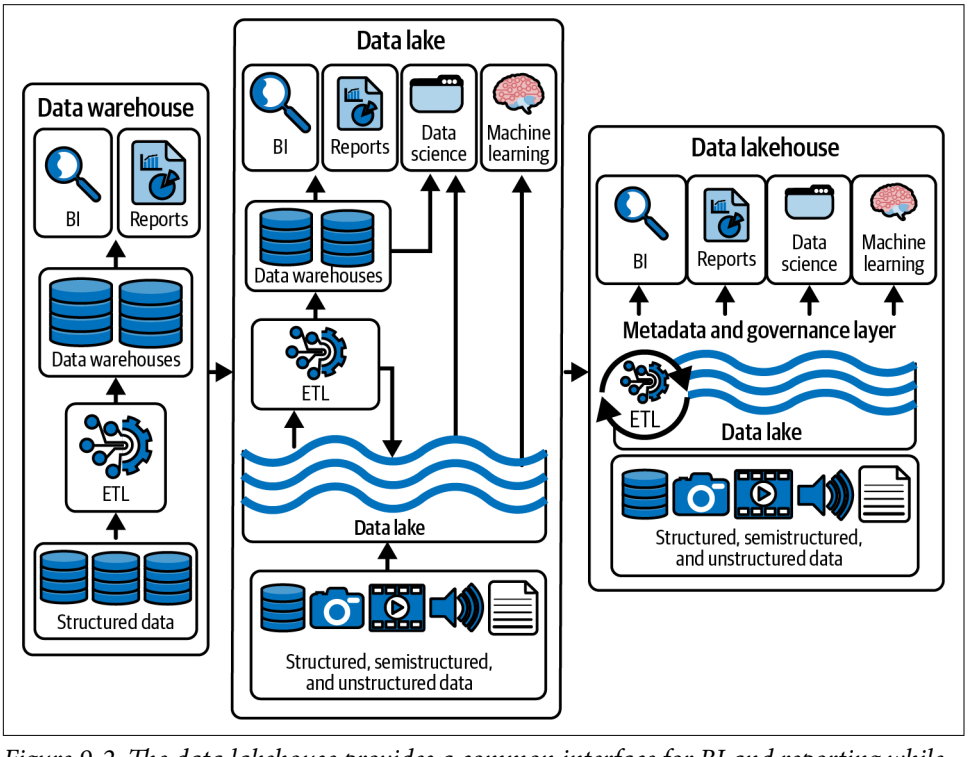

Figure 9-2. The data lakehouse provides a common interface for BI and reporting while ensuring that data science and machine learning workflows are supported in a single, unified way

This new architecture is enabled by marrying open standards in an opinionated overarching systems design—implementing data structures and data management features similar to those in a data warehouse directly on the kind of low-cost storage used for data lakes.

In fact, the lakehouse architecture intelligently provides the following:

- Transaction support
- Schema enforcement and governance/audit log and data integrity

- BI support through SQL and open interfaces such as JDBC
- Separation between storage and compute
- Open standards, open APIs, and open data formats
- End-to-end streaming
- Support for diverse workloads, from traditional SQL to deep learning

By merging the best of both worlds, we gain a single system that data teams can utilize to move faster, as they can utilize data for its explicit purpose without needing to access multiple systems (which always increases complexity). The dissolution of boundaries between the data warehouse and the data lake also makes it easier to utilize a single source of table truth. When compared against the dual-tier architecture, this is a huge win. This also prevents the problem of figuring out which side (warehouse or lake) has the correct data, who isn't in sync, and all the costly work involved to come up with a straight answer. The benefits also ensure teams have the most complete and up-to-date data available for data science, machine learning, and business analytics projects.

Foundations with Delta Lake

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

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

Open Source on Open Standards in an Open Ecosystem

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

Open file format

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

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

Parquet is the standard file format for column-oriented analytical data. So rather than our having to implement an internal, proprietary table format and access protocol, the Delta protocol is freely available to be used by the community to build new tooling and connectors (which we looked at in Chapter 4) and can be used natively within many offerings provided by key cloud service vendors such as Amazon and Microsoft, as well as Starburst and Databricks.

Self-describing table metadata

The metadata for each Delta table is stored alongside the physical table data. This design eliminates the need to maintain a separate metastore, like the Hive Metastore, simply to describe a given table. The design decision enables static tables to be more efficiently copied and moved using standard filesystem tools, while also enabling the existence of metadata-only copies of tables, which can be explored using the `SHALLOW CLONE` command.

Open table specification

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

Delta Universal Format (UniForm)

UniForm is a Delta feature introduced in Delta Lake 3.0. It enables reading Delta tables in the format needed by an application even if that application requires Iceberg or Hudi formats. By committing to interoperability, we can continue to utilize our Delta tables in an ever-expanding data ecosystem with ease of mind.

UniForm automatically generates the metadata needed for Apache Iceberg or Apache Hudi, so we don't need to decide on a given lakehouse format up-front or do manual conversions between formats, which can be error-prone. With UniForm, Delta is the universal format that works across ecosystems, providing interoperability for the lakehouse.

Enabling Delta UniForm Iceberg requires the Delta table feature `IcebergCompatV2`, a write protocol feature. Only clients that support this table feature can write to enabled tables. You must use Delta Lake 3.1 or above to write to Delta tables with this feature enabled.

Enabling Delta UniForm Iceberg requires `delta-iceberg` to be provided to Spark shell:

```
-packages io.delta:io.delta:delta-iceberg_2.12:<version>
```

Enabling Delta UniForm Hudi requires `delta-hudi` to be provided to Spark shell:

```
-packages io.delta:io.delta:delta-hudi_2.12:<version>
```

You can enable Iceberg or Hudi support using the Delta table properties:

```
% 'delta.universalFormat.enabledFormats' = 'iceberg, hudi'
```

You can create a table with support for Iceberg and Hudi as follows:

```
% CREATE TABLE T(c1 INT) USING DELTA TBLPROPERTIES(
  'delta.universalFormat.enabledFormats' = 'iceberg, hudi');
```

Or to add support for Iceberg after table creation:

```
ALTER TABLE T SET TBLPROPERTIES(
  'delta.columnMapping.mode' = 'name',
  'delta.enableIcebergCompatV2' = 'true',
  'delta.universalFormat.enabledFormats' = 'iceberg');
```

UniForm works by asynchronously generating the metadata for our Iceberg or Hudi tables after each successful Delta transaction.

Transaction Support

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

Serializable writes

Delta provides ACID guarantees for transactions while enabling multiple concurrent writers using a technique called *write serialization*. When new rows are simply being appended to the table, as with `INSERT` operations, the table metadata doesn't need

to be read before a commit can occur. However, if the table is being modified in a more complex way—for example, if rows are being deleted or updated—then the table metadata will be read before the write operation can be committed. This process ensures that before any changes are committed, the changes don't collide, as that could potentially corrupt the true sequential insert and operation order on a Delta table. Rather than risking corruption, collisions result in a specific set of exceptions raised by the type of concurrent modification.

Snapshot isolation for reads

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

Support for incremental processing

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

Incremental processing reduces the operational burden of maintaining a cursor (last offsets, IDs) or more complex state. Consider Example 9-1. We've probably seen a job like this in our careers, or can surmise that it is taking a starting timestamp and a set number of records to read, write, or maybe delete, and is also taking the last record identified of the last successful batch. The state management of traditional batch jobs can be tricky depending on the complexity of the job, due to the fact that we must maintain a manual checkpoint. Example 9-1 shows three variables that must be tracked: `startTime`, `recordsPerBatch`, and `lastRecordId`. The `startTime` variable in this example is intended to help create a time-based cursor in conjunction with the `lastRecordId`.

Example 9-1. Providing state to a stateless batch job

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

With Delta Lake, we can ignore the `startTime` and `lastRecordId` and simply use the `startingVersion` of the transaction log. This provides a specific point for us to read from. Example 9-2 shows the modified job.

Example 9-2. Providing the Delta `startingVersion` to a stateless batch job

```
% ./run-some-batch-job.py --startingVersion 10 --recordsPerBatch 10000
```

While there may not be a clear "Aha!" moment with this example, the power of incremental processing with Delta is that there is a transaction log that informs us of all the changes that happened on a table since our last run.

Support for time travel

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

Schema Enforcement and Governance

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

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

Schema-on-write

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

Consistent Data and Quality Expectations

In the real world, having invariants in place reduces the conversation about who broke what, when, and where. With Delta Lake, this means using the `mergeSchema` option infrequently and being very concerned if people want to use `overwriteSchema`. When you are using Delta Lake with some established ways of working, the Delta log will be your source of truth for arbitration, effectively removing useless meetings, since you can more or less automatically pinpoint root cause just by looking at the table history in the event that things do end up going off the rails—for example, we can take a look at the last 10 transactions using the `history` function from an instance of the `DeltaTable` class, like so: `DeltaTable.forName(spark, …).history(10)`. The result provides us with the exact sequence of changes made to the table and is an invaluable resource for root cause analysis.

Schema-on-read

Data lakes use the schema-on-read approach because there is no consistent form of governance or metadata native to the data lake, which is essentially a glorified distributed filesystem. While *schema-on-read* is flexible, its flexibility is also why data lakes are categorized as being like the Wild West—ungoverned, chaotic, and, more often than not, problematic.

What this means is that when there is data in some location (directory root) with some filetype (JSON, CSV, binary, Parquet, text, or other), with the ability of files being written to a specific location to grow unbounded, there is a high potential for problems to grow as the dataset ages.

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

Separation between storage and compute

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

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

Separation Between Logical Action and Physical Reaction

It is worth pointing out that there is even more separation between logical action within Delta Lake and the resulting physical action on the underlying physical storage layer. Take the example of cleaning up our tables from Chapter 5; there is a separation between calling `DELETE FROM` on a given table and when the physical files are affected (actually deleted). This is due to the time travel capabilities (rewind/undo) that enable us to remove accidental deletes—deletes that can otherwise harm the data integrity with no chance of restoration. The accidental deletion of data has happened to everyone at one point or another in their career; it's just that not everyone admits to it! This is why the `VACUUM` and `REORG` operations are so valuable. To really delete files, an action with a physical reaction must occur.

Support for transactional streaming

We introduced Delta's streaming capabilities in Chapter 7. The ability to switch easily between batch and streaming, across transactional tables, regardless of the specific operation (inbound reads or outbound writes) with Delta, may initially sound magical. Many the streaming pipeline has met its unexpected end due to distributed files suddenly disappearing on source tables due to changes made to tables by outside forces (such as overwrite jobs to replace missing data), but with Delta there is complete support for multiversion concurrency control, which means a streaming application reading from a table won't be interrupted due to a concurrent writer's operation.[1]

Delta Lake supports full end-to-end streaming without sacrificing quality for speed. Everything has trade-offs, and it is easy to go fast and operate blindly. In the real world, it is better to weigh the cost of delay against the need for speed and come to a general agreement on what trade-offs the business or data team is willing to make to

1 This is true for common append-style writes to the table. Other operations such as overwriting a table or deleting the table can affect streaming applications.

achieve the correct balance. While we can't guarantee that everything will always go smoothly, the safeguards provided to us when using Delta Lake can help calm even the choppiest of waters.

Unified access for analytical and ML workloads

Rounding things out, Delta provides a balanced approach to a wide range of data-related solutions. Data analysts and BI engineers can easily query using simple SQL, while there is also simultaneous support for efficient and direct physical file access for the data encompassing the Delta Lake tables; the latter provides the correct operating model for data science and ML workloads, where direct access to all columnar data, including the ability to run iterative algorithms (in place) within the scope of a job, is required.

The Delta Sharing Protocol

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

Figure 9-3 shows the Delta Sharing Protocol. The physical Delta table exists as a single source of truth, and the introduction of the Delta Sharing server adds the missing access controls and governance required to provide a safe and reliable exchange of data.

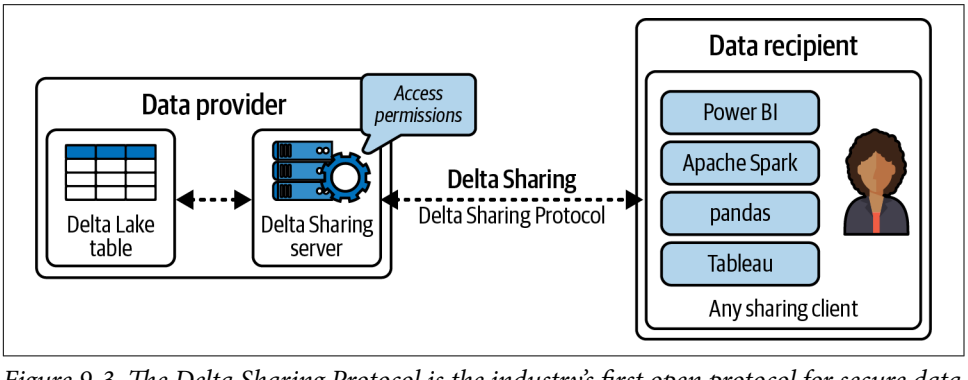

Figure 9-3. The Delta Sharing Protocol is the industry's first open protocol for secure data sharing, making it simple to share data with other organizations, regardless of which computing platforms they use

Using the Delta Sharing Protocol gives internal or external stakeholders secure direct access to Delta tables. This removes the operational costs incurred when exporting data, while saving time, money, and engineering sanity and providing a shared source of truth that is platform agnostic. We conclude this book with a deep dive into the Delta Sharing Protocol in Chapter 14.

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

The Medallion Architecture

Data in flight is messy, as it arrives in all shapes and sizes and with varying degrees of accuracy and completeness. Accepting that data will not always adhere to the myriad end-user expectations, existing data contracts, and established data quality checks, or even arrive on time—or ever—is key to addressing these data quality problems. Such challenges place a high degree of pressure on data engineering teams to continuously deliver across a dynamic landscape of subjective and objective requirements, and born from this collective toil is the *medallion architecture*.

The medallion architecture is a data design pattern used to logically organize data in the lakehouse. This is accomplished using a series of isolated data layers to provide a framework for progressively refining datasets. Figure 9-4 shows a high-level view of the architecture, with data flowing from *batch* or *streaming* sources across a variable lineage—from the point of initial ingestion (bronze) across multiple processing and enhancement phases, or stages.

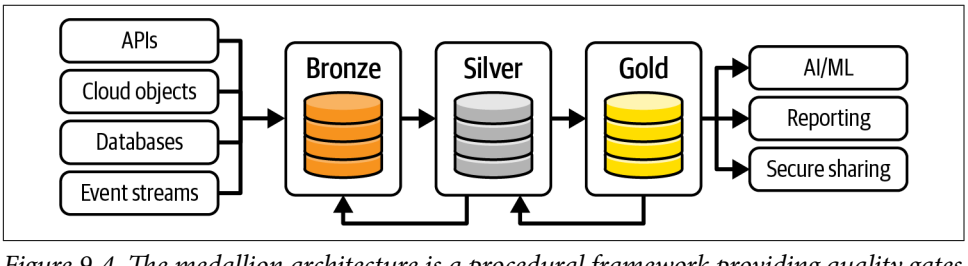

Figure 9-4. The medallion architecture is a procedural framework providing quality gates and tiers from the point of ingestion onward to the purpose-built curated data product

The medallion architecture provides a flexible framework for dealing with progressive enhancement of data in a structured way. It is worth pointing out that, while it is common to see three tiers (bronze, silver, gold), there is no rule stating that all use cases *require* three tiers. It may be that more mature data practitioners will have a two-tier system in which golden tables are joined with other golden tables to create even more golden tables. So the separation between silver and gold or between

bronze and silver may be fuzzy at times. The key reason for having a three-tiered framework is that it enables you to have a place to recover, or fall back on, when things go wrong or when requirements change.

Exploring the Bronze Layer

The bronze layer represents the initial point for our data lineage within the lakehouse. A common practice here is to apply minimal transformations (if any) on the data. These are the transformations that can't be ignored, like converting the source format into a compatible type for writing to Delta Lake. Going with the minimal transformations approach means we leave open the option to reprocess this raw data to support additional use cases, or modified requirements in the future.[2]

The Bronze Layer Is for Minimal Augmentation

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

Minimal transformations and augmentation

Because we are ingesting data that is as close to "raw" as possible, we need to remember to maintain a limited schema and do as little to transform the data as possible. Let's use a concrete example: say we are reading data from a streaming source such as Kafka, and we want to capture the topic name, binary key, and value as well as the timestamp for each record and write them into a Delta Lake table. These properties all exist in the Kafka DataFrame structure (if we are using the KafkaSource APIs with Spark) and can be extracted with the *kafka-delta-ingest* library (first explored in Chapter 4) as well.

Example 9-3 (*ch09/notebooks/medallion_bronze.ipynb*) is a concise example of minimal transformation and augmentation.

2 Remember that anything containing user data must be captured and processed according to the end-user agreed-upon consent and according to data governance bylaws and standards.

Example 9-3. This shows a simple bronze-style pipeline reading from Kafka, applying minimal transformations, and writing the data out to Delta

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

The extreme minimal approach applied in Example 9-3 takes only the information needed to preserve the data as close to its raw form as possible. This technique puts the onus on the silver layer to extract and transform the data from the `value` column.

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

In cases in which we are reading from an external database, such as Postgres, the minimum schema is simply the table DDL. We already have explicit guarantees and row-wide expected behavior given the schema-on-write nature of the database, and thus we can simplify the work required in the silver layer when compared to the example shown in Example 9-3.

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

Guarding the Bronze Layer with Permissive Mode in Spark

Example 9-4 shows a technique called *permissive passthrough* with Spark. This option allows us to add a gating mechanism using a predefined (consistent) schema to block corrupt data, while preserving the nonconformant rows for debugging.

Example 9-4. Preventing bad data with permissive passthrough

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

❶ We begin by loading a known schema using the `StructType.fromJson` method. We could just as easily have manually built the schema using the `StructType().add(...)` pattern.

❷ We then append the `_corrupt` field to our schema. This will provide a container for our bad data to sit in. Think of this as either the `_corrupt` column is null or it contains a value. The data can then be read using a filter `where(col("_corrupt").isNotNull())` or the inverse to separate the good from the bad.

❸ We then apply the reader options: `"inferSchema": "false"`, `"mode": "PERMISSIVE"`, `"columnNameOfCorruptRecord": "_corrupt"`. By turning off schema inference, we can opt into schema changes only by explicitly providing an updated schema. This means no runtime surprises. Schema inference is a powerful technique that scans (samples) a large number of rows of semistructured data (like CSV or JSON) to generate what it believes to be a stable `StructType` (schema). The problem with schema inference is that it doesn't understand the historical structure of the data and is limited to generating assumptions based on what it is provided in an initial batch.

The technique from Example 9-4 can be applied to streaming transforms just as easily using the `from_json` native function, which is located in the `sql.functions` package (`pyspark.sql.functions.*`, `spark.sql.functions.*`). This means we can

test things in batch and then turn on the streaming fire hose, understanding the exact behavior of our ingestion pipelines even in the inconsistent world of semistructured data.

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

Exploring the Silver Layer

While the bronze layer represents the initial point of lineage in the medallion architecture, the silver layer represents the point at which raw data is filtered, cleaned, and dressed up, and even augmented by joining across one or many other tables. If the bronze layer is data in its infancy, the silver layer is data in its teenage years, and as was true for all of us in our teens, our data coming-of-age story has its ups and downs.

Used for cleaning and filtering data

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

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

It is also normal to be filtering or even dropping some columns at this point. In Example 9-4, we saw the inclusion of the `_corrupt` column pattern. This information isn't valid for consumption in the silver or golden layer of the medallion architecture and is provided only to support data preservation techniques in the bronze layer and as a way of communicating problems to engineers.

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

Example 9-5 provides a continuation of Example 9-4, showing how we would pick up reading from the bronze Delta table and then filter, drop, and transform rows for receipt into the cleansed silver tables.

Example 9-5. Filtering, dropping, and transformations—all the things needed for writing to silver

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

The pipeline shown in Example 9-5 reads from the bronze Delta table (from Example 9-3) and decodes the binary data received (from the `value` column), while also enabling permissive mode, which we explored in Example 9-4:

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

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

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

After the transformations are taken care of, we write the data out to our silver Delta table. We also explicitly set the `mergeSchema:false`. While this is the default behavior, it is an important callout, since it flags for other engineers what the expected behavior is and ensures accidental columns don't mistakenly make their way to silver from bronze. We covered alternatives to automatic schema evolution using `ALTER TABLE` in Chapter 5.

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

Establishes a layer for augmenting data

There is no rule stating that a silver table must read from a bronze table. In fact, it is common for the silver layer to be used to join from one or many silver and even golden tables. For example, if the results of cleaning and filtering one of our bronze tables can be used to power multiple additional use cases, then we can save ourselves both time and additional complexity by reusing the fruits of our internal teams' and external partners' labor. Conceptually, Figure 9-5 shows the table lineage from left to right, starting with two bronze tables (on the left), followed by a series of joins and transformations (the silver layer), before yielding golden tables (on the right).

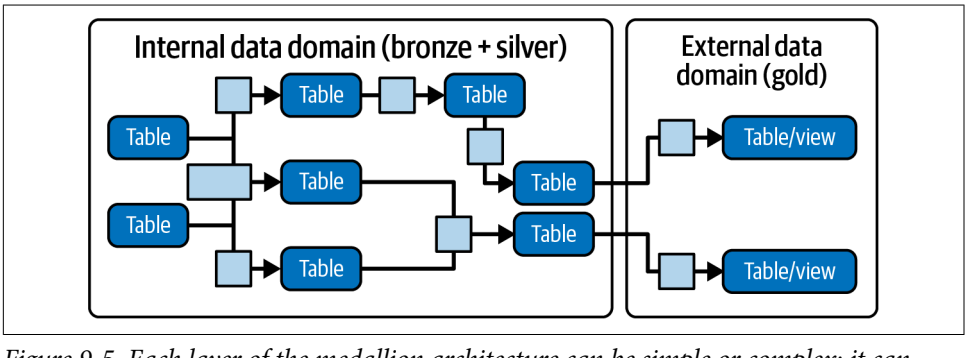

Figure 9-5. Each layer of the medallion architecture can be simple or complex; it can be easier to visualize the transformation of data across a lineage in terms of what is internal (left-hand side of the figure) and what is external (right-hand side)

Being able to view the lineage between bronze, silver, and gold can help provide additional context as the number of tables and views increases, and as the total data products and their owners naturally grow over time. We cover lineage in more detail in Chapter 13.

Enable data quality checks and balances

Delta provides capabilities for column-based constraints to enhance the functionality that can't be provided with simple schema enforcement alone. (Recall that schema enforcement and evolution was covered in Chapter 5.)

With column-level constraints, we can enforce more complex rules directly at the table level by applying predicates in the form of `CHECK`s:

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

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

Data quality frameworks can help simplify table constraints by separating the rules from the underlying physical table definition. Some popular open source frameworks are Great Expectations, Spark Expectations, and Delta Live Tables (DLT) expectations (the last of which is a paid offering from Databricks). Data quality is an important part of DataOps; it can help to block bad data before it leaves a specific layer within the medallion architecture.

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

Exploring the Gold Layer

The gold layer is the most mature data layer in the medallion architecture. Data in the gold layer has undergone multiple transformations and has been specifically curated, and it has a specific place in the data world. This is because data in the gold layer is purpose-built to solve explicit intended goals. If the bronze layer represents data as an infant, and silver is a teenager, then golden tables represent data in its late thirties or early forties—or at a point where it has established a concrete identity.

Establishes high trust and high consistency

While the analogy of data as people at different points in their lives might not be accurate, as a mental model it works. Data in the golden layer is much less likely to change drastically from day to day, in the same way that our personalities, wants, and wishes change at a slower pace as we age. Example 9-6 explores generating topN reports from the transformations out of our silver layer (Example 9-5).

Example 9-6. Creating intentional tables for business-level consumption

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

Example 9-6 shows how to do daily aggregations. It is typical for reporting data to be stored in the gold layer. This is the data we (and the business) care most about. It is our job to ensure that we provide purpose-built tables (or views) to ensure that business-critical data is available, reliable, and accurate.

For foundational tables—and really with any business-critical data—surprise changes are upsetting and may lead to broken reporting as well as to inaccurate runtime inference for machine learning models. This can cost the company more than just money; it can be the difference in whether or not the company retains its customers and reputation in a highly competitive industry.

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

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

Streaming Medallion Architecture

Earlier we learned that the medallion architecture is a data design pattern enabling us to solve common data problems encountered with any data in flight—such problems being:

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

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

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

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

By crafting our pipelines to block and correct data quality problems before they become more widespread, we can use the lessons learned across Examples 9-3 through 9-6 to stitch together end-to-end streaming workflows.

Figure 9-6 provides an example of the streaming workflow. Data arrives from our Kafka topic, as we saw in Example 9-3. The dataset is then appended to our bronze Delta table (`ecomm_raw`), which enables us to pick up the incremental changes in our silver application. The example providing the transformations was shown in Example 9-5. Last, either we create and replace temporary views (or materialized views in Databricks), or we create another golden application with the responsibility

of periodically ingesting data from `ecomm_silver` to produce purpose-built tables or views. Extending the pattern seen in Example 9-6, we can stitch together an end-to-end pipeline that incrementally ingests from its direct upstream, allowing us to trace the lineage of transformations all the way back to the initial point of inception (Kafka).

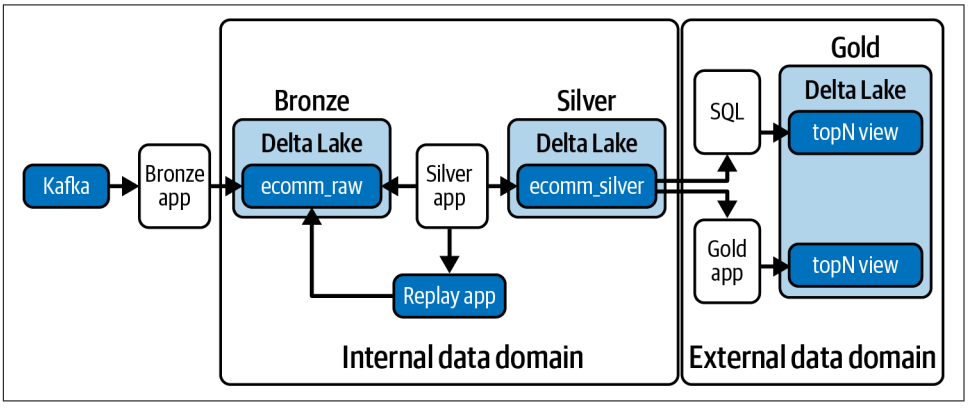

Figure 9-6. Streaming medallion architecture as viewed from the workflow level

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

For Delta Lake and Spark Structured Streaming

If you are migrating from a batch-first to a streaming-first architecture, it can be easiest to lean on triggers while you are ramping up—for example, `df.writeStream…trigger(availableNow=True).toTable(...`—so that you can continue to operate as if you are in batch while enabling your data applications to be easily converted into always-on streaming applications. A benefit of using structured streaming here is that all complex state management is handled via your application checkpoint; another added benefit is that `availableNow` triggering honors any rate-limiting options on the `DataStreamReader`, such as `maxFilesPerTrigger`.

Conclusion

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

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

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

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

CHAPTER 10

Performance Tuning: Optimizing Your Data Pipelines with Delta Lake

Up to this point, you've explored various ways of working with Delta Lake. You've seen many of the features that make Delta Lake a better and more reliable choice as a storage format for your data. Tuning your Delta Lake tables for performance, however, requires a solid understanding of the basic mechanics of table maintenance, which was covered in Chapter 5, as well as a bit of knowledge about and practice at manipulating or implementing some of the internal and advanced features introduced in Chapter 8. This performance side becomes the focus now, as we'll look at the impact of pulling the levers of some of those features in a bit more detail. We encourage you to review the topics laid out in Chapter 5 if you have not recently used or reviewed them.

In general, you will often want to maximize reliability and the efficiency with which you can accomplish data creation, consumption, and maintenance tasks without adding unnecessary costs to your data processing pipelines. By taking the time to optimize your workloads properly, you can balance the overhead costs of these tasks with various performance considerations to align with your objectives. What you should be able to gain here is an understanding of how tuning some of the features you've already seen can help to achieve your objectives.

First, there's some background work to provide a bit of clarity about the nature of your objectives. After that, there is an exploration into several of Delta Lake's features and how they impact these objectives. While Delta Lake can generally be used suitably with limited changes, when you think about the requirements put on modern data stacks, you should realize that you could always do better. In the end, taking on performance tuning involves striking balances and considering trade-offs to gain advantages where you need them. Because of this, it is best to make sure

you think about what other settings are affected when you consider modifying some parameters.

Performance Objectives

One of the biggest factors you need to consider is whether you want to try and optimize best for data producers or consumers. As discussed in Chapter 9, the medallion architecture is an example of a data architecture that allows you to optimize for both reading and writing where needed through data curation layers. This separation of processes helps you to streamline the process at the point of data creation and at the point of consumption by focusing on the goals of each at different points in the pipeline. Let's first consider some of the different objectives toward which you might want to orient your tuning efforts.

Maximizing Read Performance

Optimizing your processes for data consumers can be more simply thought of as improving the read performance on your datasets. You might have data scientists who rely on repeated reads on subsets of a dataset to build accurate machine learning models, or business analysts looking to derive specific information to convey to business stakeholders. The data consumer's needs should be considered in the design and layout of your processes. While this section won't contain a deep dive into requirements gathering or entity-relationship (ER) diagrams, proper data modeling is a high-value prerequisite to building a successful data platform, whether curation and governance happen centrally or are more distributed, such as with a data mesh architecture.[1] The data consumer needs you are primarily concerned with here are how those data consumers will access data the majority of the time. Broadly speaking, queries will fall into one of three types of patterns: narrow point queries, broader range queries, or aggregations.

Point queries

A *point query* is a query submitted by a data consumer, or user, with the intention of returning a single record from a dataset. For example, a user may access a database to look up individual records on a case-by-case basis. Such users are less likely to use advanced query patterns involving SQL-based join logic or advanced filtering conditions. Another example is a robust web-server process retrieving results programmatically and dynamically on a case-by-case basis. These queries are more likely to be evaluated with higher levels of scrutiny concerning perceived performance

1 If you wish to read more about data modeling and ER diagrams, check out Appendix A in *Learning SQL*, 3rd ed., by Alan Beaulieu (O'Reilly), or see the Wikipedia pages for data modeling and the entity–relationship model.

metrics. In both scenarios there is a human at the other end who is impacted by the query's performance, so you want to avoid any delays in record lookup without incurring high costs. This could mean that in some scenarios—such as the latter one, potentially—a high-performance, dedicated transactional system is required to meet latency requirements; this is often not the case, however, and through the tuning methods seen here you may be able to meet targets adequately without the need for secondary systems.

One of the things you'll consider is how things like file sizes, keys or indexing, and partitioning strategies can impact point query performance. As a rule of thumb, you should tend to steer toward smaller file sizes and try to use features such as indexes that reduce latency when searching for a needle in a haystack, even if the haystack is an entire field. You'll also see how statistics and file distribution impact lookup performance.

Range queries

A *range query* retrieves a set of records instead of retrieving a single record result like in a point query (which you can think of as just a special case with narrow boundaries). Rather than having an exact filter-matching condition, you'll find that range queries look for data within boundaries. Some common phrases that suggest such situations might be:

- Between
- At least
- Prior to
- Such that

Lots of other phrases are possible, but the general idea is that many records could satisfy such a condition (though it's still possible to wind up with just a single record). You will still encounter range queries when you use exact matching criteria describing broad categories, such as selecting *cats* as the type of animal from a list of pet species and breeds—you would have only one species but many different breeds. In other words, the result you look to obtain with a range query will generally be greater than one. Usually, you wouldn't know the specific number of records without adding some ordering element and further restricting the range.

Aggregations

On the surface, an *aggregation query* is similar to a range query, except that, instead of selecting down to a particular set of records, you'll use additional logical operations to perform some operation on each group of records. Borrowing from the pets example, you might want to get a count of the number of breeds per species or some other summary type of information. In such cases, you'll often see some type of

partitioning of the data by category or by breaking fine-grained timestamps down to larger periods (e.g., by year). Since aggregation queries will perform many of the same scanning and filtering operations as range queries, they will similarly benefit from the same kinds of optimizations.

One of the things you'll find here is that your preferences for how you create files in terms of size and organization depend on how you generally select the boundaries or define the groups for this type of usage. Similarly, indexing and partitioning should generally be aligned with the query patterns to produce more performant reads.

The similarities between point queries, range queries, and aggregation queries can be summarized as follows: *to deliver the best performance, you need to align the overall data strategy with the way the data is consumed.* This means you'll want to consider the data layout strategy in addition to the consumption patterns as you optimize tables. To do so, you will also have to consider how you maintain the data, and you'll have to consider how running maintenance processes such as `OPTIMIZE` or collecting statistics with `ANALYZE TABLE` impacts performance and then schedule any downtime as needed.

Maximizing Write Performance

Optimizing the performance for data producers is more than just reducing *latency*, the time lapse between receipt (ingestion) of a record and writing (committing) it to storage, where it is then available for consumption. While you usually will want to minimize this time as much as possible, striking a balance between SLAs, performance objectives, and cost, there is more you must consider. You've already seen a few of the ways you'll want to think about how the strategy you use for your data architecture should be driven by the data consumers, principally by aligning optimization goals to the kinds of query patterns that are used. What you must also remember is that you usually are not fortunate enough to have so much control as to be able to specify exactly how you'd like to receive data, and so you also have constraints driven by the upstream data producers—that is, the systems generating the data.

You might have to join numerous different data sources together to deliver the data asset your business requires. These sources can range from infrequently uploaded files in shared cloud storage locations and legacy RDBMS instances to memory stores and high-volume message bus pipelines. The types of systems that are involved will drive much of your decision making, because the volume of the data and the frequency with which you receive it will influence how your data application needs to perform. You'll also likely find that these sources will further impact the overall data strategy you choose to adopt.

Trade-offs

As has been noted, many of the constraints on your write processes will be determined by the producer systems. If you are thinking of large file-based ingestion or event- or microbatch-level stream processing, then the size and number of transactions will vary considerably. Similarly, if you are working with a single-node Python application or using larger distributed frameworks, you will have such variance. You will also need to consider the amount of time required for processing, as well as the cadence. Many of these things must be balanced, and so again, the medallion architecture lends a hand, because you can separate some of these concerns by optimizing for your core data-producing process at the bronze level and for your data consumers at the gold level, with the silver level forming a kind of bridge between them. Refer back to Chapter 9 if you want to review the medallion architecture.

Conflict avoidance

How frequently you perform write operations can limit when you can run table maintenance operations—for example, when you are using Z-Ordering. If you are using Structured Streaming with Apache Spark to write microbatch-level transactions to a Delta Lake table partitioned by the hour, then you have to consider the impacts of running other processes against that partition while it is still active.[2] How you choose options like autocompaction and optimized writes also impacts when or whether you need to run additional maintenance operations. Building indexes takes time to compute and could conflict with other processes too. It's up to you to make sure you avoid conflicts when needed, though it is much easier to do so than it was with things like read/write locks involved in every file access.

Performance Considerations

So far you've seen some of the criteria on which you'll want to base much of your decision making as far as how you interact with Delta Lake. You have many different tools built in, and how you use them usually will depend on how a particular table is interacted with. Our goal now is to look at the different levers you can pull and think about how the way you set different parameters can be better for any of the above cases. Some of this will review concepts discussed in Chapter 6 in the context of data producer/consumer trade-offs.

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

Partitioning

One of the great things about Delta Lake is that data can still be partitioned like Parquet files using Hive-style partitioning.[3] However, being able to partition tables in this way is also one of the drawbacks (be sure not to miss the section on liquid clustering, "Cluster By" on page 236). You can partition a Delta table by a column or even by multiple columns. The most commonly used partition column is date, but in high-volume processes it's not uncommon to find tables with multiple levels of partitioning using even hour and minute columns. This is a bit excessive for most processes, but technically you're not limited in how fine-grained you can make your partitioning structure. However, you may do so at your own peril, as overpartitioned tables can yield many headaches in terms of poor performance.

Structure

The easiest way to think about what partitioning does is that it breaks a set of files into sorted directories tied to your partitioning column(s). Suppose you have a customer membership category column in which every customer record will fall into either a "paid" membership or a "free" membership, as in the following example. If you partition by this `membership_type` column, then all the files with "paid" member records will be in one subdirectory, while all the files with "free" member records will be in a second directory:

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

df = pd.DataFrame(data=[
    (1, "Customer 1", "free"),
    (2, "Customer 2", "paid"),
    (3, "Customer 3", "free"),
    (4, "Customer 4", "paid")],
    columns=["id", "name", "membership_type"])

write_deltalake(
    "/tmp/delta/partitioning.example.delta",
  data=df,
  mode="overwrite",
  partition_by=["membership_type"])
```

3 For a more in-depth look at the Hive side of data layouts, see *Programming Hive* by Edward Capriolo, Dean Wampler, and Jason Rutherglen (O'Reilly).

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

By forcing the partitioning down and simultaneously partitioning by the `membership_type` column, you should see when you check the write path directory that you get a subdirectory for each of the distinct values in the `membership_type` column:

```
# Bash
tree /tmp/delta/partitioning.example.delta

/tmp/delta/partitioning.example.delta
├── _delta_log
│   └── 00000000000000000000.json
├── membership_type=free
│   └── 0-9bfd1aed-43ce-4201-9ef0-1d6b1a42db8a-0.parquet
└── membership_type=paid
    └── 0-9bfd1aed-43ce-4201-9ef0-1d6b1a42db8a-0.parquet
```

The following section can help you figure out when (or when not to) partition tables and the impact such decisions bear on other performance features, but understanding the larger partitioning concept is important, as even if you don't choose to partition tables yourself, you could inherit ownership of partitioned tables from someone who did.

Pitfalls

There are some cautions laid out for you here with regard to the partitioning structure in Delta Lake (remember the table partitioning rules from Chapter 5!). Your decision about the actual file sizes to use will be impacted by what kind of data consumers will use the table, but the way you partition your files has downstream consequences too. Generally, you will want to make sure that the total amount of data in a given partition is at least 1 GB, and you don't want partitioning at all for total table sizes under 1 TB. Anything less, and you can incur large amounts of unnecessary overhead with file and directory listing operations, especially if you are using Delta Lake in the cloud.[4] This means that if you have a high cardinality column, you should not use it as a partitioning column in most cases unless the sizing is still appropriate. In cases in which you need to revise the partitioning structure, you should use methods such as those outlined in Chapter 5 to replace the table with a more optimized layout. Overpartitioning tables is a problem that has been seen as

4 See more on this in the whitepaper "Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores".

causing performance problems for numerous people over time. It's far better to take the time to fix the problem than to pass poorer performance downstream.

File sizes

One direct implication of overpartitioning is that file sizes often turn out to be too small. File sizes of about 1 GB are recommended to handle large-scale data processes with relative ease. There are many cases, however, in which leveraging smaller file sizes, typically in the 32 MB to 128 MB range, can have performance benefits for read operations. A decision about the optimal file size comes down to the nature of the data consumer. High-volume append-only tables in the bronze layer generally function better with larger file sizes, as the larger sizes maximize throughput per operation with little regard to anything else. The smaller sizes will help a lot more with finer-grained read operations such as point queries, or in cases in which you have lots of merge operations, because of the higher number of file rewrites generated.

In the end, file size will often wind up being determined by the way you apply maintenance operations. When you run `OPTIMIZE`, and in particular when you run it with the included Z-Ordering option, you'll see that it affects your resulting file sizes. You do, however, have a couple of base options for trying to control the file sizes.

Table Utilities

You're probably pretty familiar with some version of the small files problem. While it was originally a condition largely affecting *elephantine* MapReduce processing, the underlying nature of the problem extends to more recent large-scale distributed processing systems as well.[5] In Chapter 5, you saw the need to maintain your Delta Lake tables and some of the tools available to do that. One of the scenarios covered was that for streaming use cases, where the transactions tend to be smaller, you need to make sure you rewrite those files into bigger ones to avoid a similar small files problem. Here you'll see how leveraging these tools can affect read and write performance while interacting with Delta Lake.

OPTIMIZE

The `OPTIMIZE` operation on its own is intended to reduce the number of files contained in a Delta Lake table (recall the exploration in Chapter 5). This is true in particular of streaming workloads, where you may have microbatches creating files and commits measured in just a couple of MB or less, and thus you can wind up with many comparatively small files. *Compaction* is a term used to describe the process of packing smaller files together, and it's one that is often used when talking about this

5 If you're not familiar with this problem, the blog post "The Small Files Problem" is probably worth a read.

operation. One of the most common performance implications of compaction is the failure to do it. While there could be some minute benefits to such small files (like rather fine-grained column statistics), these are generally heavily outweighed by the costs of listing and opening many files.

How it works is that when you run `OPTIMIZE`, you kick off a listing operation that lists all the files that are active in the table and their sizes. Then any files that can be combined will be combined into files around the target size of 1 GB. This helps to reduce issues that might occur from, for example, several concurrent processes committing smaller transactions to the same Delta Lake destination. In other words, `OPTIMIZE` is a mechanism to help avoid the small files problem.

Remember, there is some overhead to the operation; it has to read multiple files and combine them into the files that eventually get written, so it is a heavy I/O operation. Removing the file overhead is part of what helps to improve the read time for downstream data consumers. If you are using an optimized table downstream as a streaming source, as you explored in Chapter 9, the resulting files are not data change files and are ignored.

It's important to recall that there are some file size settings with `OPTIMIZE` that you can tweak to tune performance more to your preference. These settings and their behavior are covered in depth in Chapter 5. Next, we take a deeper look at Z-Ordering, which is instructive even if you're planning on using liquid clustering, as the underlying concepts are strongly related.

Z-Ordering

Sometimes the way you insert files or model the data you're working with will provide a kind of natural clustering of records. Say you insert one file to a table from something like customer transaction records, or you aggregate playback events from a video device every 10 minutes. Then say you want to go back an hour later to compute some KPIs from the data. How many files will you have to read? You already know it's six because of the natural time element you're working with (assuming you used event or transaction times). You might describe the data as having a natural, linear clustering behavior. You can apply the same description to any cases in which a natural sort order is inherent to the data. You could also artificially create a sorting or partitioning of the data by alphabetizing, using unique universal identifiers (UUIDs), or using a file insertion time and then reordering as needed.

At other times, however, your data may not have a native clustering that also lends itself to how it will be consumed. Sorting by an additional second range might improve things, but filtering for the first sorting range will almost always yield the strongest results. This trend continues to diminish in value as additional columns are added because it's still too linear.

There's a method used in multiple applications, one that extends well beyond just data applications, and it relies on remapping the data using a space-filling curve.[6] Without getting into too much of the rigorous detail (yet), this is a construction that lets us map multidimensional information, such as the values of multiple columns, into something more linear, such as a *cluster ID* in a sorted range. To be a bit more specific, what you need are locality-preserving, space-filling curves such as a Z-Order curve or a Hilbert curve, which are among the most commonly used options.[7] These allow us to create clusters of data in a far less linear style, which can provide great gains in performance for data consumers, especially for fine-grained point queries or more complex range queries.

In other words, this multidimensional approach means you can more easily filter on disjoint conditions. Consider a case in which you have a customer or device ID number column and an additional location information column. These columns wouldn't have any particular correlation, so there's no natural, linear clustering order. Space-filling curves would allow you to impose a clustering order on them anyway. You'll see more detail about how it works, but from a practical perspective, this means you can filter down to the combined clusters rather than get stuck having to read a full dataset.

For data producers, this represents an additional step in data production, which slows down processes, so the need for it downstream should be determined in advance. If no one benefits, then it wouldn't be worth the cost of applying it. That being said, the process is largely incremental and can be run on individual partitions when specified.

Compaction with `OPTIMIZE` using `ZORDER BY` is not idempotent (this is one of those cases in which the data change flag will be `False`) but is designed to be incremental when it runs. That is to say, when no new data is added to a partition (or to the table in the case of unpartitioned tables), then it will not try to cluster that partition or table again. This behavior expects that you are using the same column specifications for Z-Ordering, which makes sense, because a new column specification would require reclustering over the whole partition (or table).

6 For more information, see the Wikipedia article on space-filling curves.

7 See the original Databricks Engineering blog post on the initial implementation in Delta Lake. See Wikipedia for more information on Z-Order curves and Hilbert curves.

Z-Ordering attempts to create clusters of similar size in memory, which typically will be directly correlated with the size on disk, but there are situations in which this can become untrue. In those cases, task skewing can occur during the compaction process.

For example, if you have a string column containing JSON values, and this column has significantly increased in size over time, then when Z-Ordering by date, both the task durations and the resulting file sizes can become skewed during later processing.

Except for the most extreme cases, this should generally not significantly affect downstream consumers or processes.

One thing you might notice if you experiment with and without Z-Ordering of files in your table is that it changes the distribution of the sizes of the files. While `OPTIMIZE`, when left to its defaults, will generally create files that are fairly uniform in size, the clustering behavior you put in place means that file sizes can become smaller (or larger) than the built-in file size limiter (or one specified when available). This preference for the clustering behavior over strict file sizing is intended to provide the best performance by making sure the data gets colocated as desired.[8]

Optimization automation in Spark

Two settings available in Databricks—*autocompaction* and *optimized writes*—help make some of these table utilities easier to use and less interruptive (e.g., stream processing workloads). In the past, their combined usage was often called *auto-optimize*. Now they can be treated individually, because not only can they be used together, but they also can be flexibly used independently as needed in different situations, to great advantage.

Autocompaction. The first setting, `delta.autoCompact`, has been available in the Databricks Runtimes for a few years but is expected to become available across Delta Lake. The idea of `autoCompact` is that it can run `OPTIMIZE` on your table while a process is already running without additional commands. One of the biggest advantages is that you don't need to have a secondary process running that might conflict with a stream processing application, for example. The downside is that there could be a relatively minor effect on the processing latency. This is because after a file is committed, Spark will perform an `OPTIMIZE` operation as part of the same process. It analyzes the files available in the table and applies the compaction as necessary. This can be especially helpful with a streaming write based on a message bus, as the transactions tend to be smaller than you would find in many other workload

8 There is a more detailed example of Z-Ordering later in this chapter, but if you're in a hurry, the blog post "Optimize by Clustering not Partitioning Data with Delta Lake" by Denny Lee is a good and fast end-to-end walkthrough.

types, but it does come as a trade-off since it will insert additional tasks to do the compaction, which can hold up processing time. This means that for cases with tight SLA margins, you may wish to avoid using it.

Enabling the feature is just a Spark configuration setting:

```
delta.autoCompact.enabled true
```

There are a few additional settings that provide added flexibility and allow you to align the behavior of the compaction operations to your choosing.

While this feature can improve the way you use `OPTIMIZE` with Delta Lake, it will not allow the option of including a `ZORDER` on the files. You may still need additional processes, even when `auto Compact` is used to provide the best performance for downstream data consumers.

You can control the target output size of `autoCompact` with `spark.databricks .delta.autoCompact.maxFileSize`. While the default of 128 MB is often sufficient in practice, you might wish to tune this to a higher or lower number to balance between the impacts of rewriting multiple files during processing, whether or not you plan to run periodic table maintenance operations, and your desired target end state for file sizes.

The number of files required before compaction will be initiated is set through `spark.databricks.delta.autoCompact.minNumFiles`. The default number is `50`. This just makes sure you have a lower threshold to avoid any negative impact of additional operations on small tables with small numbers of files. Tables that are small but have many append and delete operations might benefit from setting this number lower, as this would create fewer files but would have less performance impacts due to the smaller size. A higher setting might be beneficial for rather large-scale processes where the number of writes to Delta Lake in a single transaction is generally higher. This would avoid running an `OPTIMIZE` step for every write stage, which could become burdensome in terms of added operational costs for each transaction.

Optimized writes. This setting is also a Databricks-specific implementation on Delta Lake that may become available in other versions.[9] In the past, you might often have ended up in scenarios in which the number of DataFrame partitions you were using grew much larger than the number of files you might want to write into, because the size of each file would be too small and create additional unneeded overhead. To solve this, you'd generally do something like `coalesce(`*`n`*`)` or `repartition(`*`n`*`)` before

9 You can find additional configuration options in the Databricks documentation for this feature.

the actual write operation to get your results compacted down to just *n* files being written. *Optimized writes* are a way to avoid needing to do this.

If you set `delta.optimizeWrites` to `true` on your table—or similarly, if you set `spark.databricks.delta.optimizeWrites.enabled` to `true` in your Databricks `SparkSession`—you get this different behavior. The latter setting will apply the former option setting to all newly created tables from the `SparkSession`. You might be wondering how this magical automation gets applied behind the scenes. How it works is that before the write part of the operation happens, you will get additional shuffle operations (as needed) to combine memory partitions so that fewer files can be added during the commit. This is beneficial on partitioned tables because the partitioning tends to make files even more granular. The added shuffle step can add some latency into write operations, so for data producer–optimized scenarios you might want to skip it, but it provides some additional compaction automatically, similar to `autoCompact` above, except that it occurs prior to the write operation rather than happening afterward. Figure 10-1 illustrates a case in which the distribution of the data across multiple executors would result in multiple files written to each partition (at left) and how the added shuffle improves the arrangement (at right).

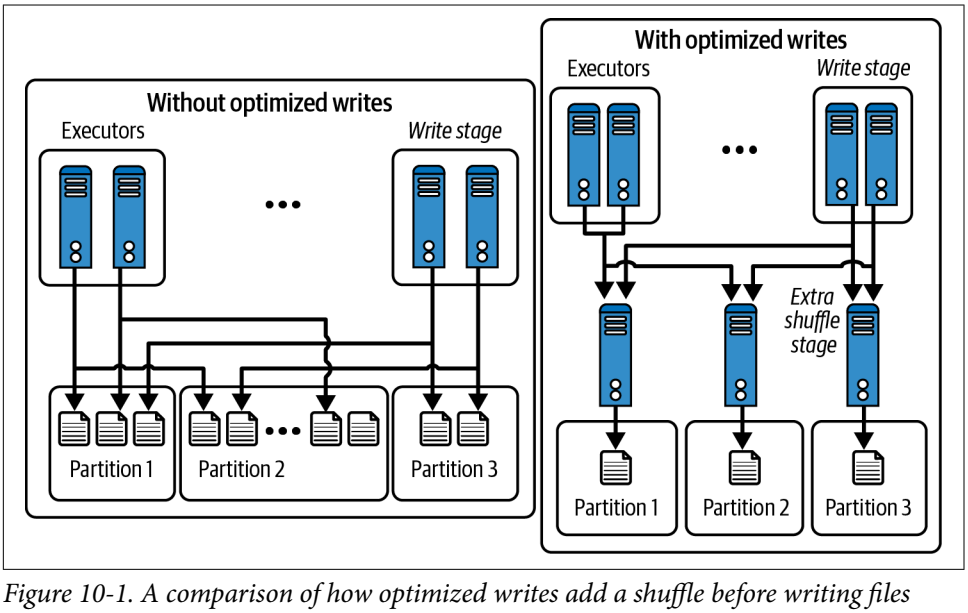

Figure 10-1. A comparison of how optimized writes add a shuffle before writing files

Vacuum

Because things like failed writes are not committed to the transaction log, you need to make sure you vacuum even append-only tables that don't have `OPTIMIZE` run on them. Write failures do occur from time to time, whether due to some cloud provider failure or perhaps because of something else, and the resulting stubs still live inside

your Delta Lake directory and could do with a little cleaning up. Not doing it early is another issue that can cause some pain. We've seen some fairly large Delta tables in production where cleaning up got overlooked during planning, and it wound up becoming a larger and costlier chore to handle later on, because by that point, millions of files needed removal (it took around three full days to fix in one case). In addition to the unnecessary associated storage costs, any external transactions hitting partitions containing extra files have many more files to sift through. It's much better to have a daily or weekly cleanup task or even to include maintenance operations in your processing pipeline. The details around the operation of vacuuming were shared in Chapter 5, but the implications of not doing it are worth mentioning here.[10]

Databricks autotuning

Databricks includes a couple of scenarios in which the respective options, when enabled, automatically adjust the `delta.targetFileSize` setting. One case is based on workload types, and the second is on the table size.

In Databricks Runtime (DBR) 8.2 and later, when `delta.tuneFileSizesForRewrites` is set to `true`, the runtime will check whether nine out of the last ten operations against the table were `merge` operations. In cases where that is the case, the target file size will be reduced to improve write efficiencies (at least some of the reasoning has to do with statistics and file skipping, which will be covered in the next section).

From DBR 8.4 onward, the table size is accounted for in determining this setting. For tables less than about 2.5 TB, the `delta.targetFileSize` setting will be put at a lower value of 256 MB. If the table is larger than 10 TB, the target will be set at a larger size of 1 GB. For sizes that fall in the intermediate range between 2.5 TB and 10 TB, there is a linearly increasing scale for the target, from 256 MB up to the 1 GB value. Please refer to the documentation for additional details, including a reference table for this scale.

Table Statistics

Up to this point, most of the focus has been centered around the layout and distribution of the files in your tables. The reason for this has a great deal to do with the underlying arrangement of the data within those files. The primary way to see what that data looks like is based on the file statistics in the metadata. Now you will see how you get statistics information and why it matters to you. You'll see what the process looks like, what the stats look like, and how they influence performance.

10 Matthew Powers and Nick Karpov's blog post on the `vacuum` command provides a more in-depth exploration of vacuuming, with examples and exploration of some of the nuances.

How statistics help

Statistics about our data can be pretty useful. You'll see more about what this means and what it looks like in a moment, but first, let's think about some reasons why you might want statistics on the files in your Delta Lake. Suppose that you have a table with a "color" field that takes 1 of 100 possible values, and each color value occurs in exactly 100 rows. This gives you 10,000 total rows. If these color values are randomly distributed throughout the rows, then finding all the "green" records would require scanning the whole set. Suppose you now add some more structure to the set by breaking it into ten files. In this case, you might guess that there are green records in each of the ten files. How could you know whether that is true without scanning all ten files? This is part of the motivation for having statistics on your files—namely, that if you do some counting operations at the time of writing the files or as part of your maintenance operations, then you can know from your table metadata whether or not specific values occur within files. If your records are sorted, this impact gets even bigger, because then you can drastically reduce the number of files that need to be read to find all your green records, or to find the row numbers between 50 and 150, as you can see in Figure 10-2. While this example is just conceptual, it should help to convince you why table statistics are important—but before you turn to a more detailed practical example, see first how statistics operate in Delta Lake.

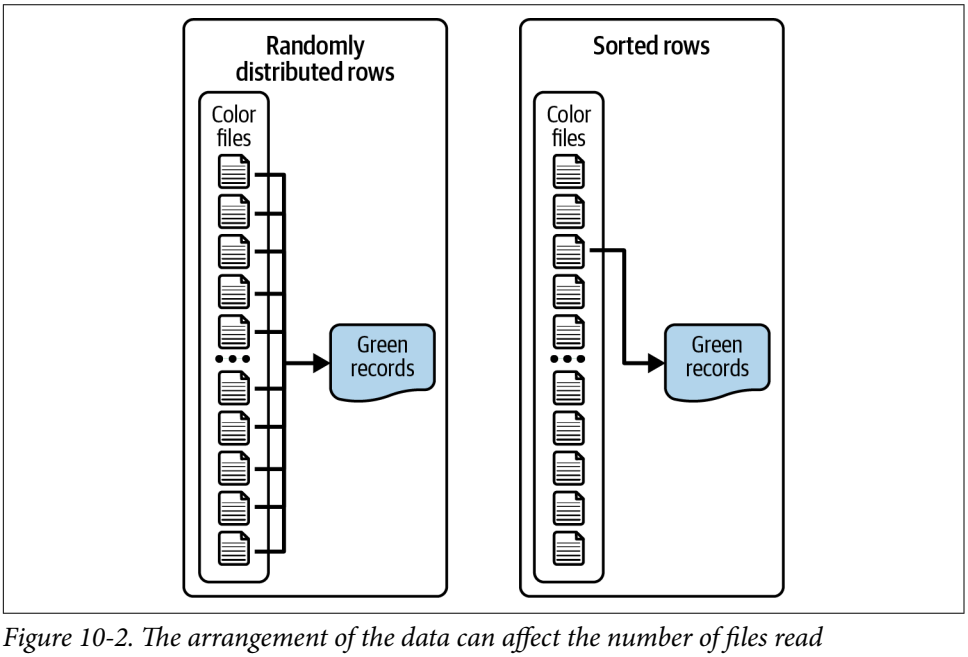

Figure 10-2. The arrangement of the data can affect the number of files read

File statistics

If you go back to the customer data table you created earlier, you can get a simple view of how statistics are generated during file creation by digging into the Delta Log. It's recommended to check the values or the relevant section of the Delta Lake protocol to see additional statistics that are added over time. Here you can use the path definition of your table and then add to that the initial JSON record from the table's creation in the *_delta_log* directory:

```
# Python
import json

basepath = "/tmp/delta/partitioning.example.delta/"
fname = basepath + "_delta_log/00000000000000000000.json"
with open(fname) as f:
    for i in f.readlines():
      parsed = json.loads(i)
      if 'add' in parsed.keys():
          stats = json.loads(parsed['add']['stats'])
          print(json.dumps(stats))
```

When you run this, you will get a collection of the statistics generated for each of the created files added to the Delta Lake table:

```
{
  "numRecords": 2,
  "minValues": {"id": 2, "name": "Customer 2"},
  "maxValues": {"id": 4, "name": "Customer 4"},
  "nullCount": {"id": 0, "name": 0}
}
{
  "numRecords": 2,
  "minValues": {"id": 1, "name": "Customer 1"},
  "maxValues": {"id": 3, "name": "Customer 3"},
  "nullCount": {"id": 0, "name": 0}
}
```

In this case, you see all the data values since the table has only four records, and there were no null values inserted, so those metrics are returned as zeros.

Notice in the example statistics pulled from the partitioning demonstration table that there is a count of records for each file. Apache Spark leverages this count to avoid reading any actual data files when running simple count operations that span partitions or entire tables by summing the statistics rather than scanning any data files, providing a significant performance advantage in many applications. Similarly, Spark can leverage these stats to performantly answer similar queries—for example:

```
-- SQL
SELECT max(id) FROM example_table
```

In Databricks (DBR 8.3 and above), you can additionally run an `ANALYZE TABLE` command to collect additional statistics, such as the number of distinct values, average length, and maximum length. These added statistics values can yield further performance improvements, so be sure to leverage them if you're using a compatible compute engine.

If you'll recall from Chapter 5, one of the settings you have available to you is `delta.dataSkippingNumIndexedCols`, which, with a default value of 32, determines how many columns statistics will be collected on. If you have a situation in which you are unlikely to run `SELECT` queries against the table, as in a bronze to silver layer stream process, for example, you can reduce this value to avoid additional overhead from the write operations. You could also increase the number of columns indexed in cases where query behavior against wider tables varies considerably more than would make sense to `ZORDER BY` (anything more than a few columns is usually not very beneficial). One other item to note here is that you can alter the table order to directly place larger valued columns after the number of indexed columns using `ALTER TABLE CHANGE` *`COLUMN`* `(FIRST | AFTER)`.[11]

If you want to make sure statistics are collected on columns you add after the initial table is created, you would use the `FIRST` parameter. You can reduce the number of columns and move a long text column, for example, after something like a timestamp column to avoid trying to collect statistics on the large text column and ensure that you still include your timestamp information to take advantage of filtering better. Setting each is fairly straightforward, but note that the `after` argument requires a *named* column:

```
-- SQL
ALTER TABLE
    delta.`example`
    set tblproperties("delta.dataSkippingNumIndexedCols"=5);
ALTER TABLE
    delta.`example`
    CHANGE articleDate first;
ALTER TABLE
    delta.`example` CHANGE textCol after revisionTimestamp;
```

Partition pruning and data skipping

So what's the actual goal of optimizing partitioning and collecting file-level statistics? The idea is to reduce the amount of data that needs to be read. Logically, the more you can skip reading, the faster you'll be able to retrieve the results of a query. At a surface level, you've already seen how statistics collection can be used to look for the maximum value of a column or to count the number of records without needing to

11 There is an example in the section covering the `CLUSTER BY` command that demonstrates this practice.

read the actual files. This is because the read part of that operation was done when the files were created, and by storing that result in the metadata, you get something like you'd expect from cached results because you don't have all the overhead required to reread all the data to compute the results. So that's great, but what about when you're doing something that isn't as trivial as getting a count of the records?

The next best thing would be to skip reading as many files as possible to retrieve results. Since these statistics are collected per file, what you get is a set of boundaries you can use to check for membership. Let's look again at the statistics you had for our small example table:

```
{
  "numRecords": 2,
  "minValues": {"id": 2, "name": "Customer 2"},
  "maxValues": {"id": 4, "name": "Customer 4"},
  "nullCount": {"id": 0, "name": 0}
}
{
  "numRecords": 2,
  "minValues": {"id": 1, "name": "Customer 1"},
  "maxValues": {"id": 3, "name": "Customer 3"},
  "nullCount": {"id": 0, "name": 0}
}
```

If you wanted to pull all the records contained for Customer 1, then you can easily see that you need to read only one of the two available files. That reduced the workload by half just in this simple case. This begins to highlight the impact of some of the points you've already seen, such as decisions you can make about file sizes or partitioning, and really kind of brings together the larger point.

Knowing that this behavior exists, you should try to target a partition layout and column organization that can leverage these statistics to maximize the performance according to your goals. If you are optimizing for write performance but frequently have to backfill values with a `merge` function to some previous point in time, then you will likely want to organize your data so that you can skip reading as many other days' data as possible to eliminate wasted processing time.

Similarly, if you want to maximize read performance, and you understand how your end users are accessing the data at the point of consumption, then you can seek a targeted layout that provides the most opportunity for skipping files at read time. There are some other cautions about overpartitioning tables because of the additional processing overhead, so next you'll see how you can use `ZORDER` to impact the downstream performance in conjunction with this knowledge of the statistics contained in each file.

Z-Ordering revisited

File skipping creates great performance improvements by reducing the number of files that need to be read for many kinds of queries. You might ask, though: how does adding the clustering behavior from `ZORDER BY` affect this process? This is fairly straightforward. Remember, Z-Ordering creates clusters of records using a space-filling curve. The implication of doing this is that the files in your tables are arranged according to the clustering of the data. This means that when statistics are collected on the files, you get boundary information that aligns with how your record clusters are segregated in the process. So now when seeking records that align with your Z-Ordered clusters, you can further reduce the number of files that need to be read.

You might further wonder how the clusters in the data get created in the first place. Consider the goal of optimizing the read task for a more straightforward case. Suppose you have a dataset with a timestamp column. If you wanted to create some same-sized files with definite boundaries, then a straightforward answer appears. You can sort the data linearly by the timestamp column and then just divide it into chunks that are the same size. What if you want to use more than one column, though, and create real clusters according to the keys, instead of just some linear sort you could have done on your own?

The more advanced task of using space-filling curves on multiple columns is not that hard to understand once you see the idea, but it's not as simple as the linearly sorted case either. At least not yet it isn't. That's actually part of the idea. You need to perform some additional work to construct a way to be able to similarly range partition data across multiple columns. To do this, you need a mapping function that can translate multiple dimensions onto a single dimension so that you can do the dividing step, just like in the linear ordering case. The actual implementation used in Delta Lake might be a little tricky to digest out of context, but consider this snippet from the Delta Lake repository:

```
// Scala
object ZOrderClustering extends SpaceFillingCurveClustering {
  override protected[skipping] def getClusteringExpression(
    cols: Seq[Column], numRanges: Int): Column = {
    assert(cols.size >= 1, "Cannot do Z-Order clustering by zero columns!")
    val rangeIdCols = cols.map(range_partition_id(_, numRanges))
    interleave_bits(rangeIdCols: _*).cast(StringType)
  }
}
```

This takes the multiple columns passed to the Z-Order modifier and then alternates the column bits to create a new temporary column that provides a linear dimension you can now sort on and then partition as a range. Now that you know how it works, consider a more discrete example that demonstrates this approach.

Lead by example

This example will show you how the differences in the layout can affect the number of files that need to be read with Z-Order clustering involved. In Figure 10-3, you have a two-dimensional array within which you want to match data files. Both the *x* range and the *y* range are numbered 1 to 9. The points are partitioned by the *x* values, and you want to find all the points at which both *x* and *y* are either 5 or 6.

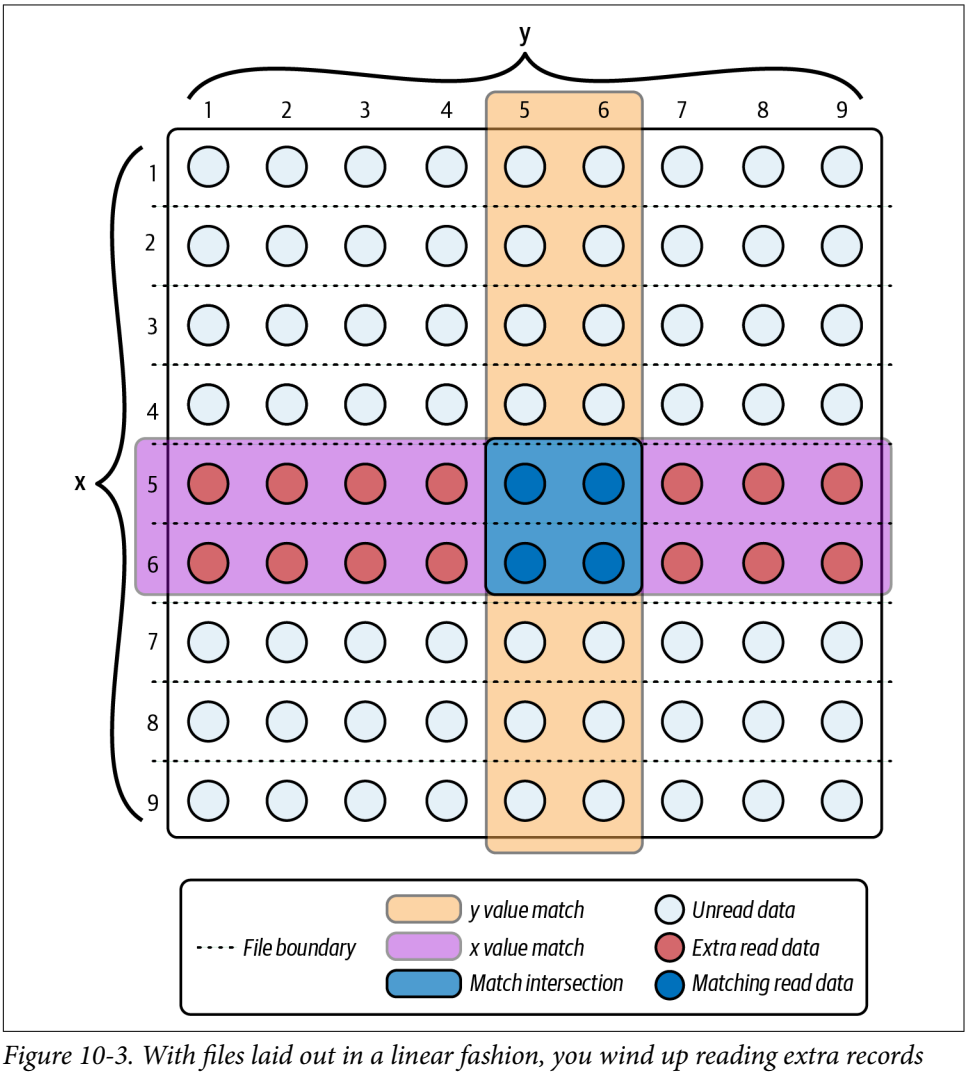

Figure 10-3. With files laid out in a linear fashion, you wind up reading extra records

First, find the rows that match the conditions $x = 5$ or $x = 6$. Then find the columns matching the conditions $y = 5$ or $y = 6$. The points where they intersect are the target values you want, but if the condition matches for a file, you have to read the whole file. So for the files you read (the ones that contain matching conditions), you can sort the data into two categories: data that matches your conditions specifically, and extra data in the files that you still have to read anyway.

As you can see, you have to read the entirety of the files (rows) where $x = 5$ or $x = 6$ to capture the values of y that match as well, which means nearly 80% of our read operation was unnecessary.

Now update your set to be arranged with a space-filling Z-Order curve instead. In both cases, you have a total of nine data files, but now the layout of the data (as shown in Figure 10-4) is such that by *analyzing the metadata* (checking the min/max values per file), you can skip additional files and avoid a large chunk of unnecessary records being read.

After applying the clustering technique to the example, you only have to read a single file. This is partly why Z-Ordering goes alongside an `OPTIMIZE` action. The data needs to be sorted and arranged according to the clusters. You might wonder if you still need to partition the data in these cases since the data is organized efficiently. The short answer is *yes*, as you may still want to partition the data, for example, in cases where you are not using liquid clustering and might run into concurrency issues. When the data is partitioned, `OPTIMIZE` and `ZORDER` will only cluster and compact data already colocated within the same partition. In other words, clusters will be created only within the scope of data inside a single partition, so the benefits of `ZORDER` still directly rely on a good choice of partitioning scheme.

The method for determining the closeness, or cluster membership, relies on interleaving the column bits and then range partitioning the dataset.[12]

12 There is a version of this written in Python to encourage additional exploration in the Chapter 10 section of the book's repository.

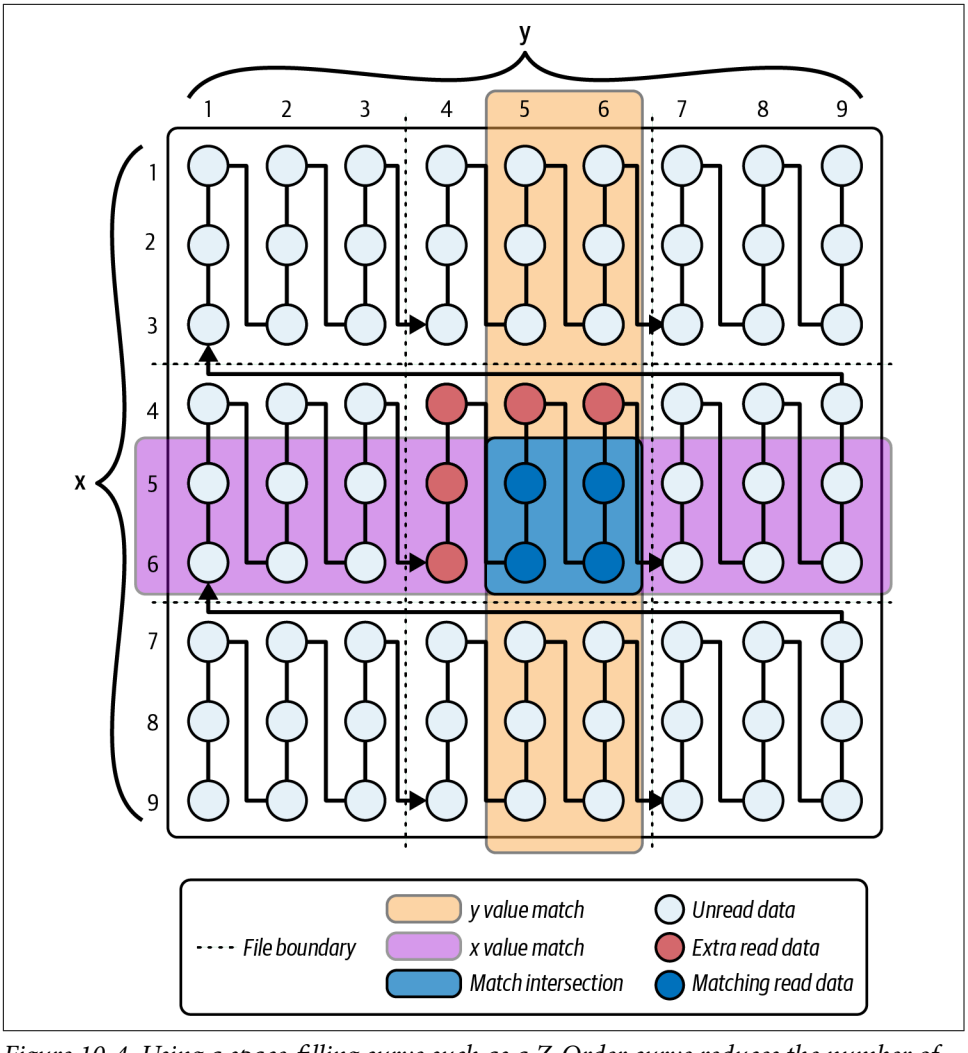

Figure 10-4. Using a space-filling curve such as a Z-Order curve reduces the number of files and unneeded data reads required for operations

You can use these steps to accomplish this:

1. Create columns containing the coordinate positions as integers.
2. Map them to binary values.
3. Bitwise interleave the binary values.
4. Map the resulting binary values back to integers.

5. Range partition the new one-dimensional column.
6. Plot the points by coordinates and bin identifier.

The results are shown in Figure 10-5. They don't show quite the same behavior as Figure 10-4, which is very neat and orderly, but they do clearly show that even with a self-generated and directly calculated approach, you could create your own Z-Ordering on a dataset.

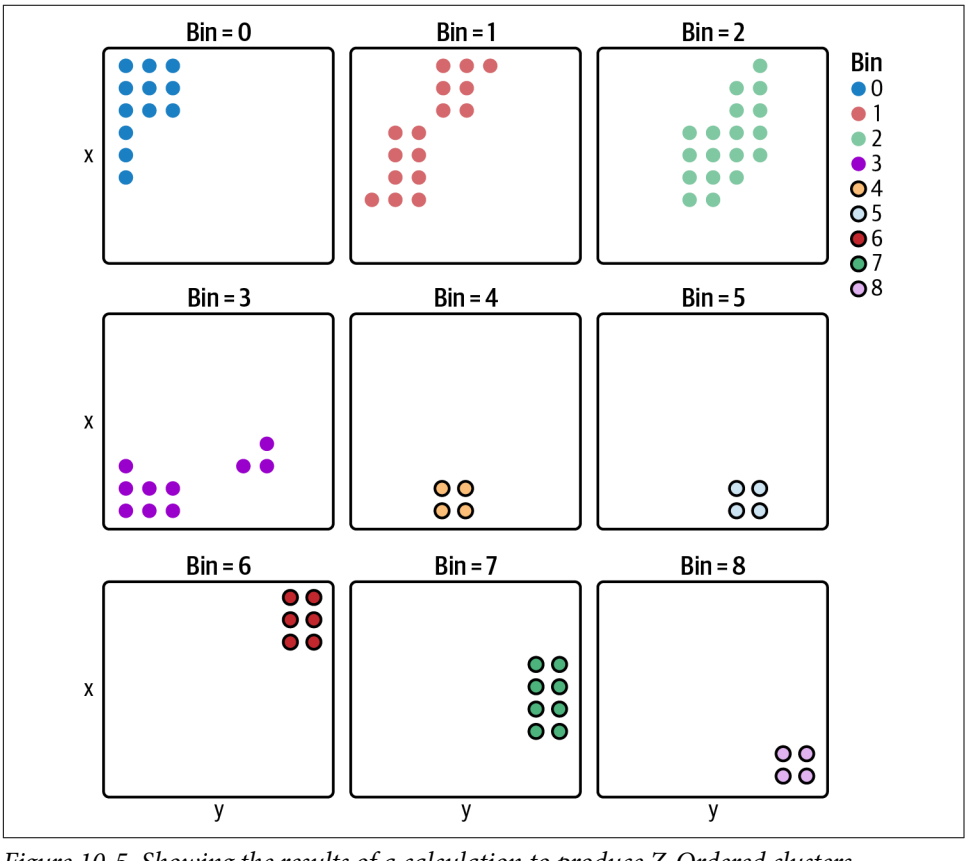

Figure 10-5. Showing the results of a calculation to produce Z-Ordered clusters

From a mathematical perspective, there are more details and even some enhancements that could be considered, but this algorithm is already built into Delta Lake, so for the sake of our sanity, this is the current limit of our rigor.[13]

13 For more technical details, refer to Mohamed F. Mokbel, Walid G. Aref, and Ibrahim Kamel, "Performance of Multi-dimensional Space-Filling Curves", in *Proceedings of the 10th ACM International Symposium on Advances in Geographic Information Systems (GIS '02)* (New York: Association for Computing Machinery, 2002), 149–54.

More recently, there have been questions about whether any table ought to be partitioned so that there are fewer constraints on the further development of ideas like Z-Ordering. This is partly because it can be very difficult to settle on the right partitioning columns from the outset, outside of highly static processes. Needs can also change over time, leading to added maintenance work in updating the table structure (see the example in Chapter 6 if you need to do this). One development in this area may reduce these maintenance burdens and decisions for good.

Cluster By

The end of partitioning? That's the idea. The newest and best-performing method for taking advantage of data skipping came in Delta Lake 3.0. Liquid clustering takes the place of traditional Hive-style partitioning with the introduction of the `CLUSTER BY` parameter during table creation. Like `ZORDER`, `CLUSTER BY` uses a space-filling curve to determine the best data layout but changes to other curve types that yield more efficiency. Figure 10-6 shows how different *partitions* may either get coalesced together or be broken down in different combinations within the same table structure.

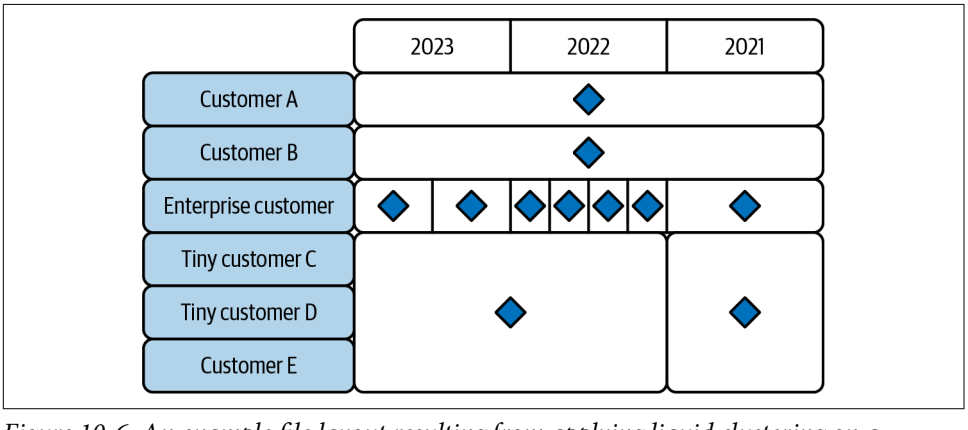

Figure 10-6. An example file layout resulting from applying liquid clustering on a dataset[14]

Where it starts to get different is in how you use it. Liquid clustering *must* be declared during table creation to enable it, and it is incompatible with partitioning, so you can't define both. When set, it creates a table property, `clusteringColumns`, which can be used to validate that liquid clustering is in effect for the table. Functionally, it

14 This example comes from a fuller walk-through highlighting how liquid clustering works both to split apart larger partitions as well as to coalesce smaller ones. For the full example, check out Denny Lee's blog post "How Delta Lake Liquid Clustering Conceptually Works".

operates similarly to `ZORDER BY` in that it still helps to know which columns might yield the greatest filtering behaviors on queries, so you should still make sure to keep our optimization goals in sight.

You also will not be able to `ZORDER` the table independently, as the action takes place primarily during compaction operations. A small side benefit worth mentioning is that liquid clustering reduces the specific information needed to run `OPTIMIZE` against a set of tables because there are no extra parameters to set, allowing you to even loop through a list of tables to run `OPTIMIZE` without worrying about matching up the correct clustering keys for each table. You also get row-level concurrency—a must-have feature for a partitionless table—which means that most of the time you can stop trying to schedule processes around one another and reduce downtime, since even `OPTIMIZE` can be run during write operations. The only conflicts that happen are when two operations try to modify the same row at the same time.

File clustering, like that shown in Figure 10-6, gets applied to compaction in two different ways. For normal `OPTIMIZE` operations, it will check for changes to the layout distribution and adjust as needed. This newer clustering enables a best-effort application of clustering the data during write processes, which makes it far more reliably incremental to apply. This means less work is required to rewrite files during compaction, which also makes that process more efficient as well. This feature is called *eager clustering*. This means that for data under the threshold (512 GB by default), new data appended to the table will be partially clustered at the time of the write (the best-effort part). In some cases, the size of these files will vary from the larger table until a larger amount of data accumulates and `OPTIMIZE` is run again. This is because the file sizes are still driven by the `OPTIMIZE` command.

To use the `CLUSTER BY` argument, you need at least a writer version of 7 in a Delta Lake release with the liquid clustering table feature present and enabled. To only consume the tables, you need a reader version of 3. This means that if you have other/older consumers in the environment, you are at risk of breaking workflows while migrating to newer versions and protocols.

Explanation

`CLUSTER BY` uses a different space-filling curve than `ZORDER`, but without the presence of partitions, it creates clusters across the whole table. Using it is fairly straightforward, as you simply include a `CLUSTER BY` argument as part of your table creation statement. You must do so at creation or else the table will not be compatible as a liquid partitioning table—it cannot be added afterward. You can, however, later update the columns chosen for the operation or even remove all columns from the clustering by using an `ALTER TABLE` statement and `CLUSTER BY` (use `NONE` instead of providing a column name or names for the latter case—there's an example of this

soon). This means you gain great flexibility with clustering keys, because they can be changed as needs arise or as consumption patterns evolve.

As you're creating tables that are optimized for either the downstream consumers or your write process, this presents an area in which you can make just such a decision between the two. Similar to other cases, if the goal is to get the speediest write performance, then you can elect not to include any clustering at all or to include as little as you wish. For the downstream consumers, though, you gain a considerable advantage. You saw in Chapter 5 that although it's possible to repartition a given table, it's not the most straightforward operation. Now you can adapt to downstream consumer needs more optimally by redefining the clustering columns, and this will be picked up during the next compaction process to apply the layout to the underlying files. This means that as usage patterns change, or even if you made questionable assumptions or errors in your original layout, they are more easily rectifiable. The following example shows how you can leverage liquid clustering in the Databricks environment.

If the initial write to a table is larger than 10 TB—for example, if you use a CTAS (`CREATE TABLE AS SELECT`) statement to do a one-time conversion—the first compaction operation can suffer from performance issues and take some time to complete. The clustering quality may also be affected somewhat. It is recommended to run the process in batches for large tables as a result, but otherwise even tables of 100 TB can have liquid clustering applied to them.

Hopefully, it has become apparent that liquid clustering offers several advantages over Hive-style partitioning and Z-Ordering tables whenever it's a good fit. You get faster write operations with similar read performance to other well-tuned tables. You can avoid problems with partitioning. You get more consistent file sizes, which makes downstream processes more resistant to task skewing. Any column can be a clustering column, and you gain much more flexibility to shift these keys as required. Last, thanks to row-level concurrency, conflicts with processes are minimized, allowing workflows to be more dynamic and adaptable.

Example

In this example, you'll see the Wikipedia articles dataset found in the */databricks-datasets/* directory available in any Databricks workspace. This Parquet directory has roughly 11 GB of data (disk size) across almost 1,100 gzipped files.

Start by creating a DataFrame to work with, add a regular date column to the set, and then create a temporary view to work with in SQL afterward:

```
# Python
articles _path = (
"/databricks-datasets/wikipedia-datasets/" +
"data-001/en_wikipedia/articles-only-parquet")

parquetDf = (
    spark
    .read
    .parquet(articles_path)
)
parquetDf.createOrReplaceTempView("source_view")
```

With a temporary view in place to read from, you can create a table simply by adding the `CLUSTER BY` argument to a regular CTAS statement to define the table:

```
-- SQL
CREATE TABLE
    example.wikipages
CLUSTER BY
    (id)
AS (SELECT *,
    date(revisionTimestamp) AS articleDate
    FROM source_view
    )
```

You still have a normal statistics collection action to think about, so you probably want to exclude the actual article text from that process, but you also created the `articleDate` column, which you probably want to use for clustering. To do this, you can add the following steps: reduce the number of columns you collect statistics on to only the first five, move both the `articleDate` and `text` columns, and then define the new `CLUSTER BY` column. You can do all of these using `ALTER TABLE` statements:

```
-- SQL
ALTER TABLE example.wikipages
  SET tblproperties ("delta.dataSkippingNumIndexedCols"=5);
ALTER TABLE example.wikipages CHANGE articleDate first;
ALTER TABLE example.wikipages CHANGE `text` after revisionTimestamp;
ALTER TABLE example.wikipages CLUSTER BY (articleDate);
```

After this step, you can run your `OPTIMIZE` command, and everything else will be handled for you. Then you can use a simple query like the following for testing:

```
-- SQL
SELECT
  year(articleDate) AS PublishingYear,
  count(distinct title) AS Articles
FROM
  example.wikipages
WHERE
  month(articleDate)=3
AND
  day(articleDate)=4
```

```
GROUP BY
  year(articleDate)
ORDER BY
  publishingYear
```

Overall, the process was easy, and the performance was comparable—only slightly faster than the Z-Ordered Delta Lake table. The initial write for liquid partitioning also took about the same amount of time. These results should be expected, because the arrangement is still basically linear. One of the biggest gains in value here, however, is the added flexibility. If at some point you decide to revert to clustering by the `id` column as in the original definition, you just need to run another `ALTER TABLE` statement and then plan for a bigger-than-usual `OPTIMIZE` process later on. Whether you end up using liquid clustering or rely on the familiar Z-Ordering, there's still an additional indexing tool you can put in place that further improves the query performance of chosen tables.

Bloom Filter Index

A *Bloom filter index* is a hashmap index that identifies whether a value *probably* exists in a file or *definitely* does not.[15] Hashmap indexes are considered space-efficient because an index file containing the hashed value (in a single row) is stored alongside the associated data file, and you can specify which columns you wish to be indexed. The catch is that you want to have a reasonable idea of the number of distinct values that need to be indexed, because this will determine the length of hashes needed to avoid collisions if that number is set too small or to avoid wasting space if it is set too large.

Bloom filter indexes can be used by either Parquet or Delta Lake tables in Apache Spark, even if they use liquid clustering. At runtime, Spark checks for the existence of the directory and uses the index if it exists. It does not need to be specified during query time.

A deeper look

A Bloom filter index is created at the time of writing files, so this has some implications to consider if you want to use the option. In particular, if you want all the data indexed, then you should define the index immediately after defining a table but before you write any data into it. The trick to this part is that defining the index correctly requires you to know the number of distinct values of any columns you want to index ahead of time. This may require some additional processing overhead, but for the example, you can add a `COUNT DISTINCT` statement and get the value

15 If you wish to dive more deeply into the mechanisms and calculations used to create Bloom filter indexes, consider starting with the "Bloom filter" Wikipedia article.

as part of the process to accomplish this using only metadata (another Delta Lake benefit). Use the same table from the `CLUSTER BY` example, but now insert a Bloom filter creation process right after the table definition statement (before you run the `OPTIMIZE` process):

```
# Python
from pyspark.sql.functions import countDistinct

cdf = spark.table("example.wikipages")
raw_items = cdf.agg(countDistinct(cdf.id)).collect()[0][0]
num_items = int(raw_items * 1.25)

spark.sql(f"""
    create bloomfilter index
    on table
        example.wikipages
    for columns
        (id options (fpp=0.05, numItems={num_items}))
""")
```

Here the previously created table is loaded, and you can bring in the Spark SQL function `countDistinct` to get the number of items for the column you want to add an index for. Since this number determines the overall hash length, it's probably a good idea to pad it—for example, where `raw_items` is multiplied by 1.25, there was an additional 25% added to get `num_items` to allow for some growth to the table (adjust according to your projected needs). Then define the Bloom filter index itself using SQL. Note that the syntax of the creation statement details exactly what you wish to do for the table and is pretty straightforward. Then specify the column(s) to index and set a value for `fpp` (more details are in the following section on configuration) and the number of distinct items you want to be able to index (as already calculated).

Configuration

The `fpp` value in the parameters is short for *false positive probability*. This number sets a limit on what rate of false positives is acceptable during reads. A lower value increases the accuracy of the index but takes a little bit of a performance hit. This is because the `fpp` value determines how many bits are required for each element to be stored, so increasing the accuracy increases the size of the index itself.

The less commonly used configuration option, `maxExpectedFpp`, is a threshold value set to `1.0` by default, which disables it. Setting any other value in the interval `[0, 1)` sets the maximum expected false positive probability. If the calculated `fpp` value exceeds the threshold, the filter is deemed to be more costly to use than it is beneficial, and so it is not written to disk. Reads on the associated data file would then fall back to normal Spark operation, since no index remains for it.

You can define a Bloom filter index on numeric types, datetime types, strings, and bytes, but you cannot use Bloom filter indexes on nested columns. The filtering actions that work with these columns are `and`, `or`, `in`, `equals`, and `equalsnullsafe`. One additional limitation is that null values are not indexed in the process, so filtering actions related to null values will still require a metadata or file scan.

Conclusion

When you set out to refine the way you engineer data tables and pipelines with Delta Lake, you may have a clear optimization target, or you might have conflicting objectives. In this chapter, you saw how partitioning and file sizes influence the statistics generated for Delta Lake tables. Further, you saw how compaction and space-filling curves can influence those statistics. In any case, you should be well equipped with knowledge about the different kinds of optimization tools you have available to you in working with Delta Lake. Most specifically, note that file statistics and data skipping are probably the most valuable tools for improving downstream query performance, and you have many levers you can use to impact those statistics and optimize for any situation. Whatever your goal is, this should prove to be a valuable reference as you evaluate and design data processes with Delta Lake.

CHAPTER 11

Successful Design Patterns

Considering Delta Lake's flexibility and applicability to data applications, trying to capture all the cases for which you can use Delta Lake is like trying to describe all the potential uses of paper. The variety feels limitless, and its value is legion. That said, we will do our best in this chapter to capture exemplary cases of using Delta Lake and to highlight the value in doing so.

We will start by showing how the performance optimizations and simplified maintenance operations in Delta Lake helped Comcast slash the amount of resources needed to run its smart remote process by a factor of 10. We will then describe how Scribd helped evolve the Delta Lake landscape and created the Delta Rust implementation, which is one hundred times cheaper than the equivalent structured streaming applications. Finally, we'll see how Delta Lake feeds high-volume operational CDC ingestion and supports real-time workloads from Flink at DoorDash, creating a single-source-of-truth lakehouse from many different operational systems. Each section is accompanied by several resources you may wish to review to explore the stories found here in greater detail.

Slashing Compute Costs

The focus of this section reaches many audiences—literally! It's no secret that there has been somewhat of an eruption in the number of streaming entertainment services over the last several years. Organizations supporting these kinds of services tend to have large volumes of high-throughput streaming data that they need to manage to help support the service.

High-Speed Solutions

Streaming media services usually capture data from individual end-user devices, which include several different components. To run such services successfully, you may require varying kinds of information about device health, application status, playback event information, and interaction information. This usually translates to a need for building high-throughput stream processing applications and solutions.

One of the most critical components of these streaming applications is ensuring the capture of the data with reliability and efficiency. In Chapter 7, several implementation methods and their benefits demonstrate how Delta Lake can play a critical role in doing exactly these kinds of data capture tasks. Delta Lake is often the destination for many of these ingestion processes because it has ACID transaction guarantees and additional features like optimized writes that make high-volume stream processing better and easier.

Let's say you want to monitor the Quality of Service (QoS) across all your users in near real time. To accomplish this task, you usually need not just playback event information but also the relevant context from each user's session, a sequence of interactions bound together over some time span. Sessionization is often an important cornerstone of many downstream operations beyond ingestion and typically falls into the data engineering stages of a larger data process, as shown in Figure 11-1. With session information and other system information in Delta Lake, you can power downstream analytics use cases such as Quality of Service measurement or trending item recommendations while maintaining a low turnaround time in processing.

Building out these pipelines is often fairly complex and will involve the interaction of multiple pipelines and processes. At the core, you will find that each component boils down to the idea of needing to build a robust data processing pipeline to serve multiple business needs.

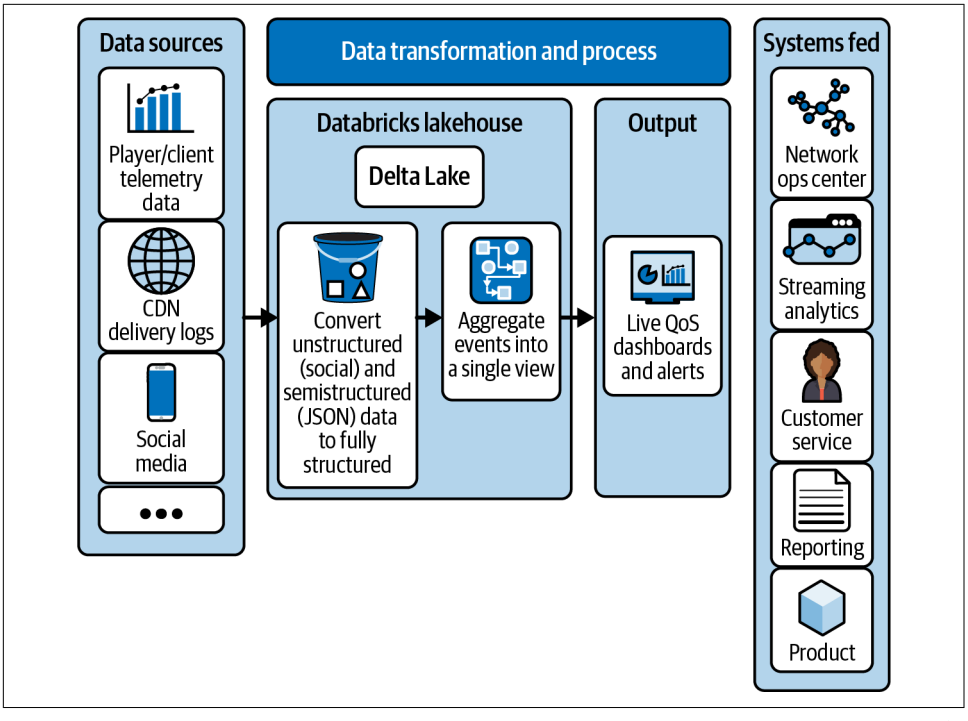

Figure 11-1. A reference architecture for Quality of Service monitoring with Delta Lake[1]

Smart Device Integration

Comcast developed a successful smart remote control device to change the way people watch television. The crux of the company's data problem was that this kind of system requires large amounts of data processing and poses several technical and organizational challenges. Through the use of Delta Lake as a data format, many of these challenges were overcome, and Comcast was able to slash the cloud infrastructure requirements for one of its most critical workloads by 90%. It was also able to solve many quality-of-life issues around these data processes. Here you can see how Comcast solved many of those challenges.

1 For an extended exploration of a QoS solution end-to-end, we recommend the blog post "How to Build a Quality of Service (QoS) Analytics Solution for Streaming Video Services" and its accompanying notebooks from Databricks.

Comcast's smart remote

Comcast is the largest American multinational telecommunications and media conglomerate, and in this section you will see how the company was able to drastically reduce the amount of cloud resources required to run its most important workloads. Comcast has strived to change how people interact with their televisions through its Xfinity Voice Remote, which acts as a central point of access. So, as you might expect, there are a lot of critical data workloads that center around the device at the edge. Figure 11-2 shows a high-level example of the interaction flow.

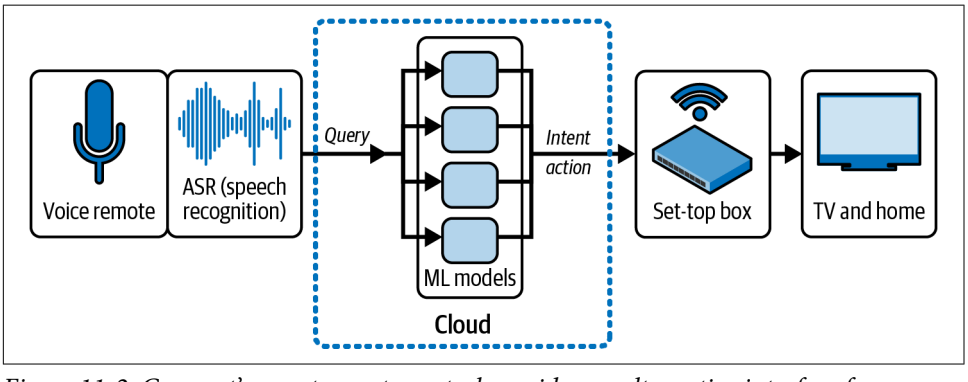

Figure 11-2. Comcast's smart remote control provides an alternative interface for entertainment

Before we explore how Comcast is building its solutions on Delta Lake, it might be useful to review more specific information about the scale of its operations. Comcast drives interactions through its voice remote, and its customers used this remote 14 billion times in 2018–2019 (Figure 11-3 illustrates the relative scale to data processing).[2] Users expect many things in their experience with the applications, such as getting accurate searches and feeling enabled to find the right content for consumption. Each user's individual experience should also have elements of personalization that make the experience their own. With the voice remote, users can interact with the whole system; anything they want is just a quick phrase away. On top of this, Comcast uses user data to create personalized experiences.

2 For additional detail, see the Databricks videos "Comcast Makes Home Entertainment Accessible to Everyone with Voice, Data and AI" and "Winning the Audience with AI: How Comcast Built an Agile Data and AI Platform at Scale".

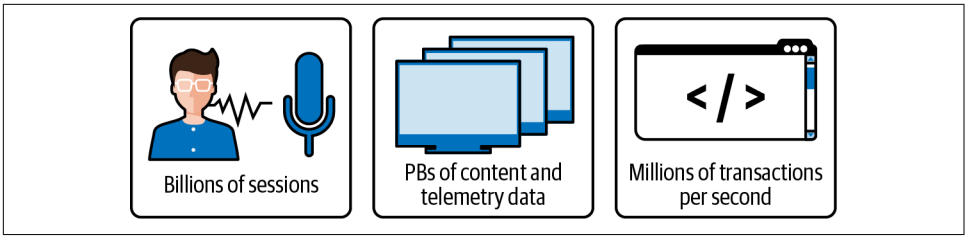

Figure 11-3. High data throughput volumes occur across large consumer groups

Consider the technical components essential to running such services behind the scenes. First, receiving voice commands as input (something that's exploded in popularity more recently) is a technically challenging problem. There's the transformation of voice to a digital signal, which then has to be mapped to each needed command. There's often an additional component to this mapping of correcting for intent. Is it more likely for someone to be searching for a show called *How It's Made* or to be asking about other shows that explain how some particular thing is made? If it is a search command, there is still a need to find similar content through a matching algorithm. All of this gets wrapped together into a single interface point in a setting in which the user experience needs to be measured against accuracy, so getting bits of data about these processes and enabling analytics to assess immediate problems or long-term trends is also critical.

So now we have voice inputs that have to be converted to embedding vectors (vectors of numeric data capturing semantic meaning as "tokens"), as well as contextual data (this could be what type of page the user is on, other recent searches, date-time parameters, etc.) for each interaction with the remote.[3] The goal is to collect all this and provide inference back through the user interface (UI) in nearly real time. From a functional standpoint, there's also a large amount of telemetry information that needs to be collected to maintain insights into things such as device health, connectivity status, viewing session data, and other similar concerns.

Once the problem of getting this data from individual devices to a centralized processing platform is solved, there are still additional challenges in deciding how to standardize the data sources, as multiple versions of devices may have differing available information, or usage regions may have differing collection laws that mean fuller or lesser contents of captured events. Downstream from standardization, there is still a need to organize the data and create actionable steps in a fit-for-function format.

3 For a more robust treatment of embeddings, see Marcos Garcia, "Embeddings in Natural Language Processing: Theory and Advances in Vector Representations of Meaning", *Computational Linguistics* 47, no. 3 (2021): 699–701.

For all of this to happen from a single team would require a huge amount of effort and a lengthy amount of time, so enabling multiple teams to collaborate to tackle the complexity would be beneficial, if not an absolute necessity.

Earlier attempts

To support the voice remote, Comcast needed to be able to analyze queries and look at user journeys to do things like measure the intention of a query. At a rate of up to 15 million transactions per second, Comcast needed to enable sessionization across billions of sessions on multiple petabytes of data. Running on native AWS services, it would overrun limits and increase the concurrency it was using until it was eventually running 32 concurrent job runs across 640 virtual machines to be able to get to the scale it needed for sessionization. The processing flow is shown in Figure 11-4. This led Comcast to seek a scalable, reliable, and performant solution.

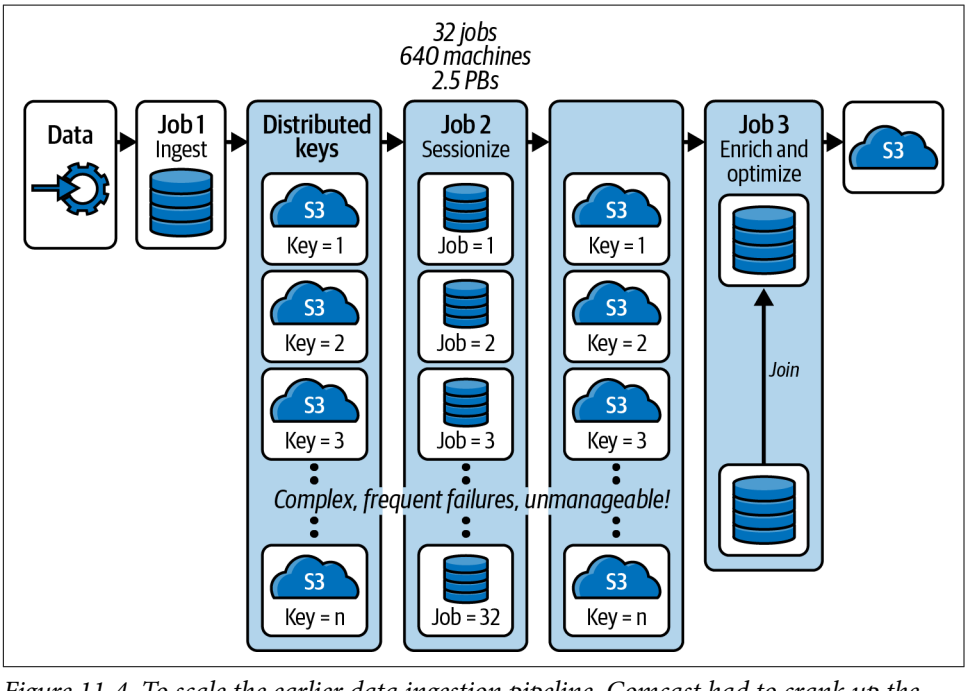

Figure 11-4. To scale the earlier data ingestion pipeline, Comcast had to crank up the concurrency

Delta Lake reduces the complexity

Delta Lake was built to help solve exactly these kinds of problems. ACID transactions and support for multiple writers with features like optimized writes and autocompaction each play a role in simplifying and overcoming the challenges involved with large-scale stream processing tasks. The problem here originates in the nature of the

data and the partitioning by key values. Many natural keys (e.g., user ID values) will result in skewing of the data. This means that high-volume key lookups become increasingly burdensome as data volumes increase, and the highest-frequency keys can become a bottleneck in your application.[4] Enabling additional features such as `delta.randomFilePrefixes` for high transaction rates with cloud providers allows you as an engineer to achieve massive scale with improved efficiency, as doing so removes potential barriers caused by prefix limitations.[5] By allowing a distributed framework to handle the key partitioning rather than forcing a manual parallelization of the tasks, you can gain significant performance improvements. By making this change, Comcast was able to run the same ingestion process with a single Spark job on just 64 virtual machines. The resulting process flow is shown in Figure 11-5.[6]

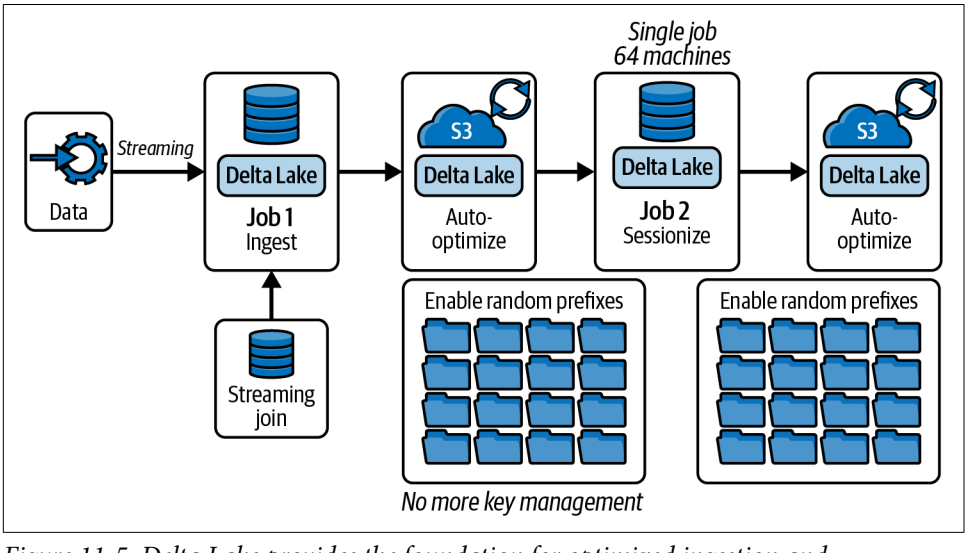

Figure 11-5. Delta Lake provides the foundation for optimized ingestion and sessionization

If this was the whole story, you would probably already be convinced of the value Delta Lake can bring to ease processing burdens. What's great is that it's *not* the whole story. In its Databricks environment, Comcast was able to readily access this sessionized data for multiple downstream purposes.

4 There is some good discussion of hot-spot keys in key-value stores in the section "Partitioning of Key-Value Data" in Martin Kleppmann's *Designing Data-Intensive Applications* (O'Reilly).

5 AWS states in its performance guidance for S3 that sequential prefixes can also be effective.

6 Databricks, "Customer Story: Comcast".

It was mentioned already that building a process like this may involve different kinds of machine learning tasks, such as the creation of embedding vectors or model inference. In particular, there would be a need to transform that voice input into meaningful action. By capturing the sessionized data and storing it efficiently, data scientists can build modeling pipelines quickly and easily.

MLflow, another open source product, offers many features for improving the end-to-end MLOps process. MLflow's key features include tracking and comparing multiple model versions in experiments, a registry for management, and mechanisms enabling the easier deployment of model objects. MLflow also includes support for large language models (LLMs), other generative models, and AI agents in addition to traditional machine learning models.

Since Comcast is using MLflow, it gets additional side benefits from Delta Lake in its machine learning processes. With the data source tracking available in the experiment for a project, MLflow can track information about the Delta Lake table being used for the experiment without having to make a copy of the data, in the same way as you would with a CSV file or other data sources.[7] Figure 11-6 shows where MLflow sits in the data life cycle. Since Delta Lake also has time travel capabilities, machine learning experiments can have enhanced reproducibility, which would benefit anyone maintaining data science products in production.

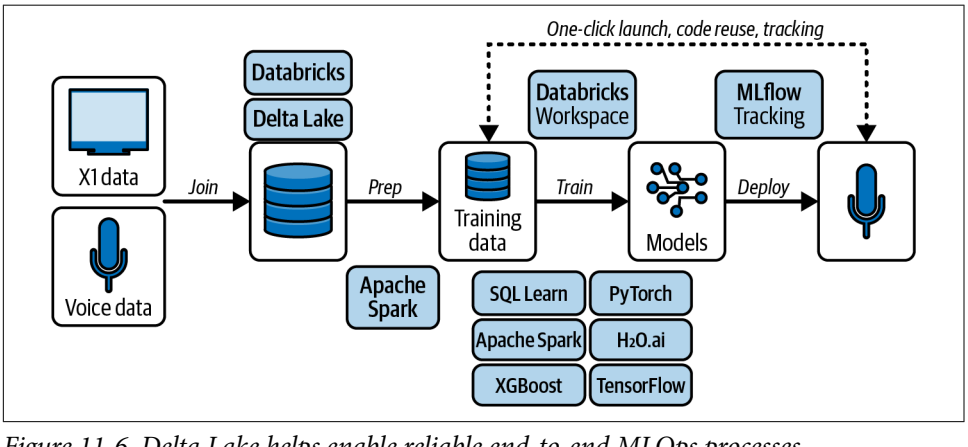

Figure 11-6. Delta Lake helps enable reliable end-to-end MLOps processes

7 To compare the entire capabilities for tracking different kinds of files in MLflow experiments, we suggest you look at the "`mlflow.data`" section of their documentation.

Another important target is to be able to monitor the telemetry data involved for QoS or similar types of analytical applications. In Comcast's case, it used Databricks SQL to run analytical workloads directly on its Delta Lake tables instead of in Redshift, as it had previously.[8] The company reported for a pilot of this approach that it chose its 10 worst-performing queries to evaluate the performance. It observed a reduction of more than 70% in the time spent running queries (see Figure 11-7).

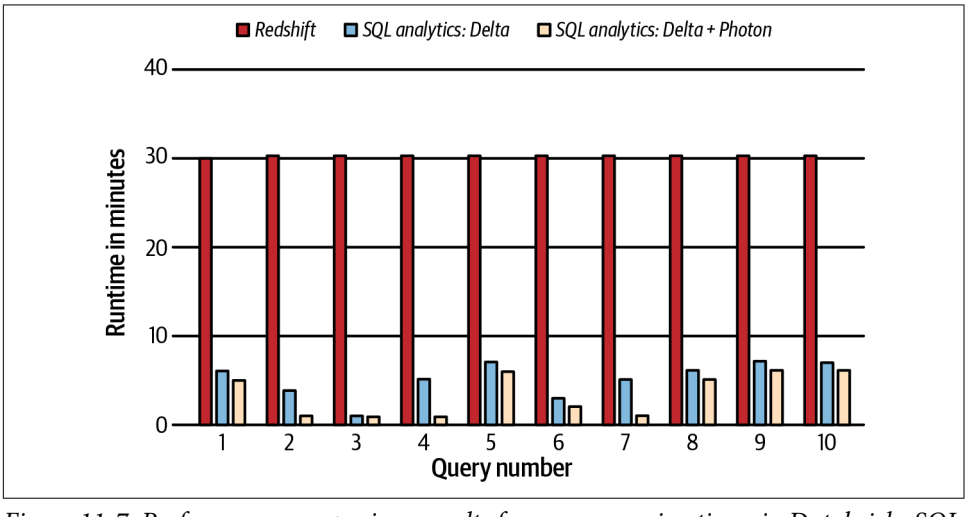

Figure 11-7. Performance comparison results for query running times in Databricks SQL on Delta Lake versus Redshift

In the end, it's looking to be highly advantageous for Comcast to continue innovating with Delta Lake. It has so far experienced huge savings gains in its data ingestion processes and has a promising outlook on improving reporting. This should allow Comcast to further improve end-user experiences for its smart remotes and increase overall satisfaction rates.

8 Molly Nagamuthu and Suraj Nesamani, "SQL Analytics Powering Telemetry Analysis at Comcast", posted September 16, 2021, by Databricks, YouTube.

Efficient Streaming Ingestion

Suppose you have some large ingestion pipelines running on Kafka and Databricks to feed your Delta Lake environment. Now suppose you have a crack engineering team that decides to invest significant effort into reducing costs by crafting a solution for small streams that doesn't require the heavy-lifting capabilities of Spark. You also want to bring all that data together downstream from those ingestion processes. What you might be looking for then is something like what the team at Scribd has done.

Streaming Ingestion

Stream processing applications for ingestion tasks are relatively common. We have a large array of streaming frameworks out there to choose from. Among the most common ones are the open source Apache Kafka, Kinesis from AWS, Event Hubs in Azure, and Google's Pub/Sub.

While there is certainly a wide variety of applicability covering interesting subjects like real-time telemetry monitoring of IoT devices and fraudulent transaction monitoring or alerting, one of the most common cases for stream processing is large-scale and dynamic data ingestion.[9] For many organizations, collecting data about activities by end users on mobile applications or point-of-sale (POS) data from retailers directly translates to success in supporting mission-critical business analytics applications. Acquiring large amounts of data from widely dispersed sources quickly and correctly allows businesses to become more rapidly adaptable to changing conditions as well (Figure 11-8 shows a unified architecture across many streaming sources).

Great flexibility, as achieved through the enablement of real-time processes and the use of artificial intelligence applications, is fueled by dynamic and resilient data pipelines often falling into this category.[10] In all of these, there's usually an element of capturing inbound data for later analytical or evaluation purposes, so while there might be additional components in some processing pipelines, at the end of the day this process applies to most stream processing applications.[11]

9 Many teams document their own journey of landing streaming data sources in Delta Lake; for example, the Michelin team captured a step-by-step implementation guide to building a Kafka + Avro + Spark + Delta Lake IoT data ingestion pipeline in a Microsoft Azure environment.

10 The term *artificial intelligence* is used here in the classical software development sense of "narrow AI," meaning the application of machine learning algorithms to make automated business decisions without human interaction—see the definitions of artificial intelligence posted by the Stanford Institute of Human-Centered Artificial Intelligence.

11 Refer to the discussion of the medallion architecture in Chapter 7 or 9 for more details on implementing stream processing applications and Delta Lake.

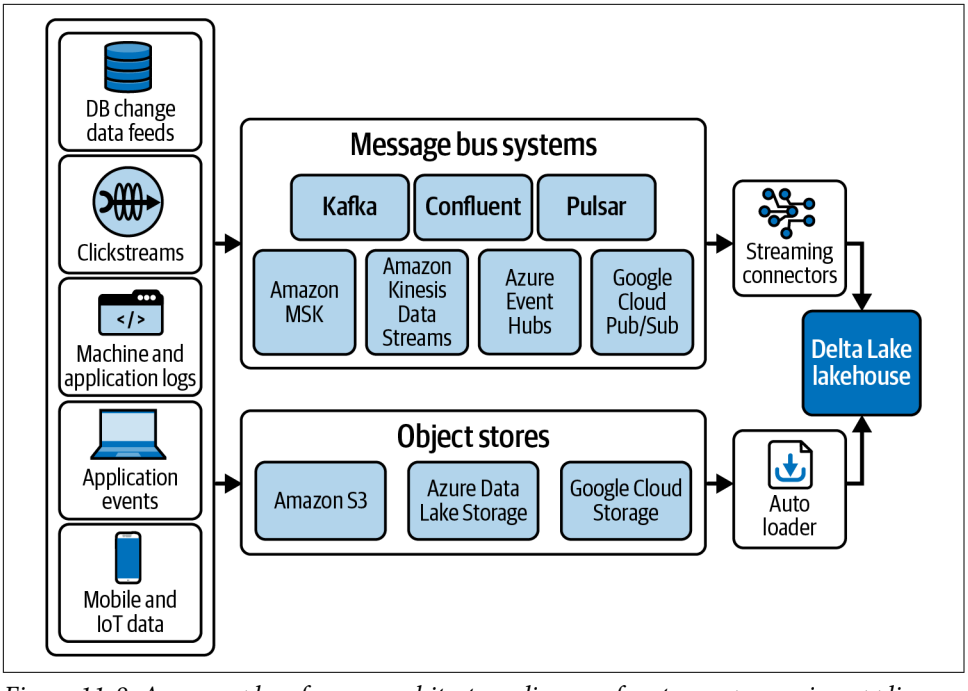

Figure 11-8. An example reference architecture diagram for stream processing applications with a Delta Lake sink from Databricks[12]

Consider the case of IoT data coming in from devices. If you send all the data into Kafka, you can build a Spark application to consume that stream and capture all the original data as it is received, following the model of the medallion architecture. Then you can create business-level reporting and send those results out to be consumed in a downstream application. Naturally, there are many variations on this approach, but the general pipeline model is similar, as shown in Figure 11-9. At Scribd, this application was so common that they built a new framework around implementing this process.

12 This architecture diagram comes from the Databricks blog post "Simplifying Streaming Data Ingestion into Delta Lake" (accessed December 7, 2023).

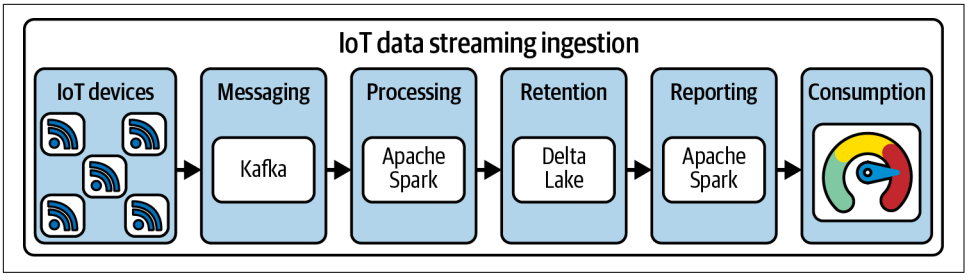

Figure 11-9. A simplified streaming data ingestion architecture for IoT devices specific to Kafka

The Inception of Delta Rust

While it started as an open publishing platform, Scribd is now a digital document library, with over 170 million documents in more than 150 categories and counting. Part of Scribd's mission is to change the way the world reads. Scribd aims to do so by providing a wide range of reading material at a fair price for both creators and consumers, providing intellectual property protection for creators, and keeping costs low, preferring to build its brand on community rather than on advertising.

Inherent to its existence as a digital library, Scribd runs its website as well as mobile applications.[13] Users can utilize Scribd's website and mobile applications to browse through a digital library with millions of presentations, research papers, templates, and other kinds of documents. All the documents in the library are uploaded by creators, writers, and editors using multiple common document formats like *.pdf*, *.txt*, *.doc*, *.ppt*, *.xls*, and *.docx*. There is also a subscription system. All these different system components translate to events that must be collected and handled accordingly. Scribd accomplishes this by using a fairly large number of event streams through Kafka.

Building a streaming ingestion pipeline typically requires multiple components. Putting this into the immediate context, a straightforward design approach would be to build a stream processing application for each topic stream coming from Kafka. In the case of Scribd, we can easily build a list of some of the probable event topic streams: creator uploads, reading events, system login or authentication events, subscription events, web traffic events, searches, item bookmarking or saving events, and item sharing events. This means many different stream processing applications will be involved, which usually leads to the development of some kind of framework to reduce development and maintenance overhead across all the applications.

13 Christian Williams, "Streaming Data into Delta Lake with Rust and Kafka", posted July 19, 2022, by Databricks, YouTube.

Maintaining a stream processing framework for many event streams can be quite a complex task, and without careful planning it can be quite expensive as well. Here is the story of the evolution of Scribd's stream processing framework, leading up to its creation of the *kafka-delta-ingest* library, and of how it cut ingestion costs by 95%.

The Evolution of Ingestion

The stream processing platform at Scribd has been revamped a couple of different times. Early on, all the processing was done in Kafka and Hadoop, which used to be a fairly standard stream processing approach. This version of the platform was later subsumed by a move to Kafka and Databricks using Spark Structured Streaming and Delta Lake. This was a favorable move for Scribd, in part because of Delta Lake's features, such as the `optimize` and `vacuum` utilities and the addition of ACID transactions.

However, in Scribd's case, there were many topic streams, and many of them were on the small side. This led to some attempts to reduce spiraling ingestion costs. At Scribd, larger dedicated clusters were still used "when it didn't seem wasteful" to do so—that is, when there were large tasks that efficiently utilized the cluster resources. Many small streams were instead *stacked* (run simultaneously on the same cluster), which produces a similar level of efficient resource utilization and thus reduces overall processing costs. There are still some challenges in doing this, however. Making decisions about how to logically group topics can be frustrating. There's always the possibility that one of the processing tasks could fail, causing all the stacked streams on that cluster to subsequently fail. This is in addition to the already slightly challenging task of trying to accommodate maintenance tasks in your ingestion processes.

The Scribd team had a few desires for improving the situation:

- Further reducing the costs, if possible
- Different observability of the ingestion processes
- Better handling of job failures
- More flexible adjustment to changes in the throughput size of event streams

This also led to thoughtful reflection on how the team might approach the problem. Would it be possible to do this without Spark or to find some more minimal overhead method? How would the team still maintain its standardization on Delta Lake, since that made stewardship so much easier?

To the Scribd team at the time, it seemed like with some invested effort, there might be another way to approach the problem. The team has relatively simple ingestion processes that are append-only operations with no filters, joins, or aggregations and uses only a subset of Delta Lake's features, which proved to simplify the development of an alternative.

The scenario at Scribd led to its investment in developing two projects that are now well-supported and accepted parts of the larger Delta Lake ecosystem. The first project is delta-rs, the Rust-based implementation of the Delta Lake protocol explored in depth in Chapter 6. The second project is *kafka-delta-ingest* (a short guide to using *kafka-delta-ingest* can be found in Chapter 4), a lightweight companion framework designed to quickly and easily ingest data from a Kafka topic stream into a Delta Lake table.[14] Together the projects form an efficient operating pair (Figure 11-10 shows the simplified data flow).

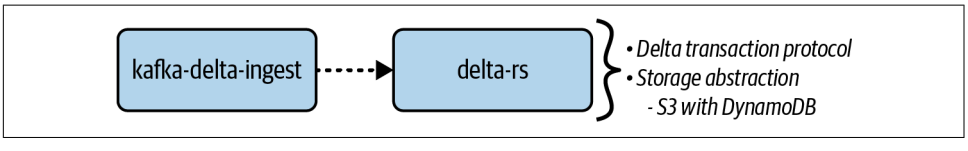

Figure 11-10. Scribd's kafka-delta-ingest in tandem with delta-rs for efficient ingestion

Undertaking such an endeavor was not without risks or potential blocking issues. The risk of corrupting the Delta log posed one challenge, as did the need to manually control offset tracking in Kafka to avoid duplicate or dropped records. Scribd also needs to support multiple writers to tables, and furthermore, some limitations in AWS S3 require specific handling (e.g., S3 lock coordination).[15]

Scribd runs anywhere from 70 to 90 of these *kafka-delta-ingest* and delta-rs pipelines in production. It runs serverless computation of these pipelines through AWS Fargate and monitors everything in Datadog. Some of the things it monitors include message deserialization logs and several metrics: the number of transformations and failures, the number of Arrow batches in memory, the sizes of Parquet data files written, and the current time lag in Kafka streams.

14 Christian Williams, "Kafka to Delta Lake, as Fast as Possible", *Scribd Technology* (blog), Scribd, May 19, 2021.

15 Some of these S3 issues are discussed in the D3L2 web series episode "The Inception of Delta Rust" on YouTube.

All of this led to rather significant cost savings in ingestion processing, as with the tools the Scribd team built, the cost for running some of the stream processing applications is reduced to as much as 100 times lower (shown in Figure 11-11). Another feature that rounds out this fantastic achievement is that this is accomplished in such a way (by remaining standardized on Delta Lake) that the ingested data is immediately available for analytics and machine learning processes or for further integration with other batch processes in Scribd's Databricks environment, and it maintains queryability.

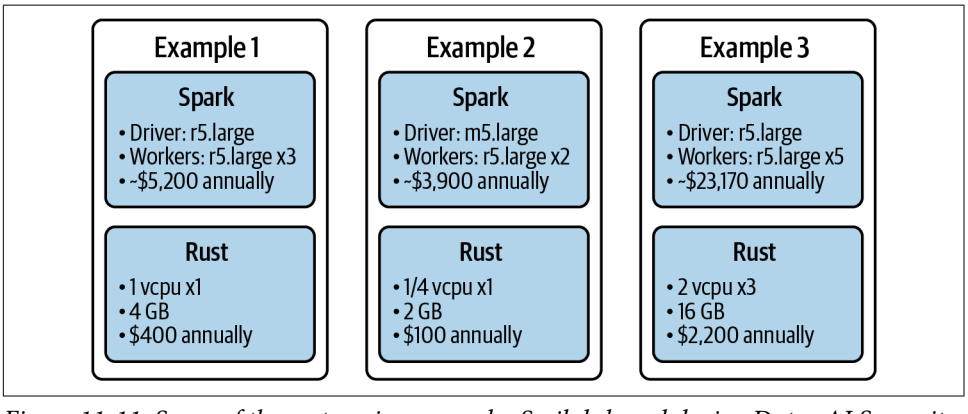

Figure 11-11. Some of the cost-saving examples Scribd shared during Data+AI Summit 2022 that show the cost of running a process originally in Spark and then using delta-rs[16]

Coordinating Complex Systems

From smart devices and entertainment to security and digital payment systems, there is no shortage of high-volume data sources. With Scribd, much of the focus was on simple event capture, with less stress on the operational systems where *kafka-delta-ingest* is a viable solution. Now let's consider cases in which the edge of interaction with the outside world is less straightforward and requires more services. It's more messy and more human. Complex applications that continuously evolve tend to have many more integrated operational components that need to stay in harmony over time, or else you might find yourself spending too much time curating existing data, rather than thinking about new requirements, sources, or processes as you would probably prefer.

The inclusion of multiple real-time operational databases and the demand for generating business value often mean that the information from those databases needs to

16 Note that the Rust resources show individual vCPU and memory allocation, whereas the Spark resources show clusters composed of multiple EC2 instances; r5.large EC2 instances each have two vCPUs and 16 GB of RAM. Amazon EC2 R5 instance metrics can be found on the AWS website.

be collected into a unified location for the development of analytics and machine learning applications. Other systems may not have operational databases but rely on event-driven systems. Oftentimes this data will be needed in conjunction with data from other systems, creating a relatively complex data ecosystem, such as customer transaction data with anonymized trend data available on the open market, for example. Figure 11-12 shows how to combine data sources such as these to support multiple downstream applications. Relying on a lakehouse format such as Delta Lake with its broad array of connectors for different systems reduces this complexity and enables analytical and artificial intelligence–based applications.

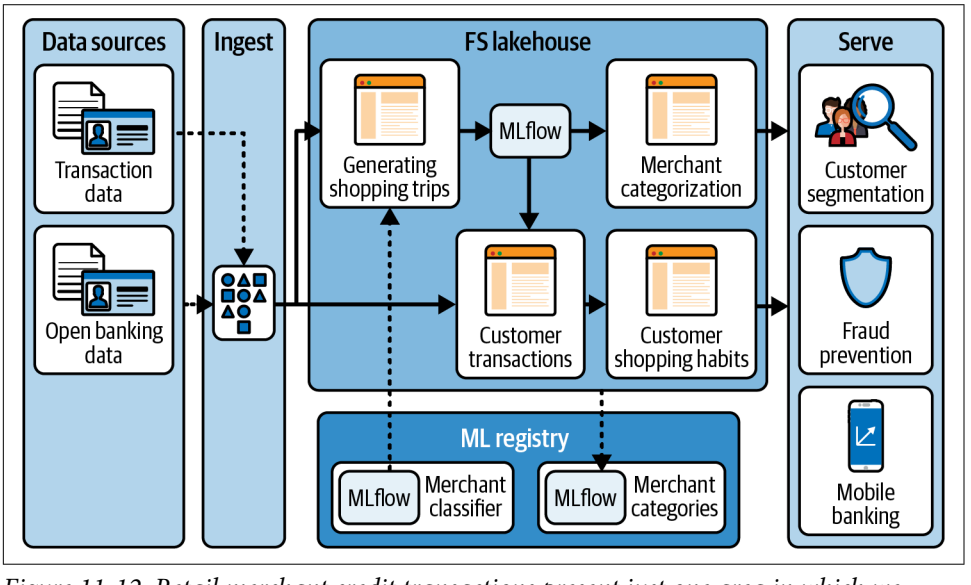

Figure 11-12. Retail merchant credit transactions present just one area in which we might see complex system interactions

Combining Operational Data Stores at DoorDash

Many people have found themselves in situations in which it would be convenient if someone could pick up meals, groceries, electronics, or pretty much anything else for them and maybe save them a trip out. DoorDash helps to fill all kinds of needs by providing flexibility and convenience through its delivery services. While most are familiar with DoorDash's gig-based operating methodology, it may be helpful to consider a particular couple of points.

Multiple parties are involved in the purchase process through DoorDash. Typically there are the requesters, the people who make deliveries, and restaurants or merchants who will prepare orders or make products available. Without even stepping into the larger IT ecosystem of the DoorDash organization, there is already an apparent need for large-scale low-latency data pipelines, i.e., streaming data applications,

because each "event" itself is a collation of many events as it steps through the process.

DoorDash is leveraging Delta Lake as part of its data ecosystem in two ways. The first is to simplify the management of large-scale change data capture and downstream exposure of data for analytics. The second is in support of real-time workloads in Flink. Both capture some of the benefits of utilizing Delta Lake in your architectural designs.

Change Data Capture

Change data capture, or CDC, is a common application pattern that often needs to be supported for a variety of reasons (see Chapter 7 for some additional discussion of CDC).[17] DoorDash uses CDC for replication of operational databases supporting multiple services into the analytical environment.[18] This is driven by a historical need to answer the question "How many orders did DoorDash do yesterday?" Earlier on this was an easier task, as the question could be answered by creating a copy of the database and using queries against the copy to answer analytical questions or perform data science tasks.

As DoorDash grew, its service architecture evolved, leading to an environment with multiple operational databases that also come in multiple flavors, such as CockroachDB, PostgreSQL, and Apache Cassandra. Seeking to get data from these databases in the simplest way, the DoorDash team initially got snapshots from the databases and pulled them in daily. While this approach worked, it did pose problems—specifically, the challenge of tracking data versioning, and a need to filter the snapshots to incrementalize the data process efficiently. After trying various changes in the environment, the team eventually set out to develop a more robust system.

For our purpose, the key system requirements were:

- Maintain less than a day of data latency
- Use a lakehouse design pattern
- Support schema evolution
- Allow for data backfilling
- Enable analytical workloads
- Write once, read many times

17 If you want to spend more time exploring CDC, also known as logical log replication, we recommend *Designing Data-Intensive Applications* by Martin Kleppmann (O'Reilly).

18 Ivan Peng and Phani Nalluri, "Unlocking Near Real Time Data Replication with CDC, Apache Spark Streaming, and Delta Lake", posted July 26, 2023, by Databricks, YouTube.

- Avoid late-arriving data
- Build with open source software

The design that arose from these requirements (see Figure 11-13) is a streaming CDC framework built on Spark Structured Streaming that replicates change feeds into a unified source of truth built on Delta Lake that supports downstream integrations across a wide range of query interfaces. Features such as merge support and ACID transactions helped make Delta Lake a critical component of the design.

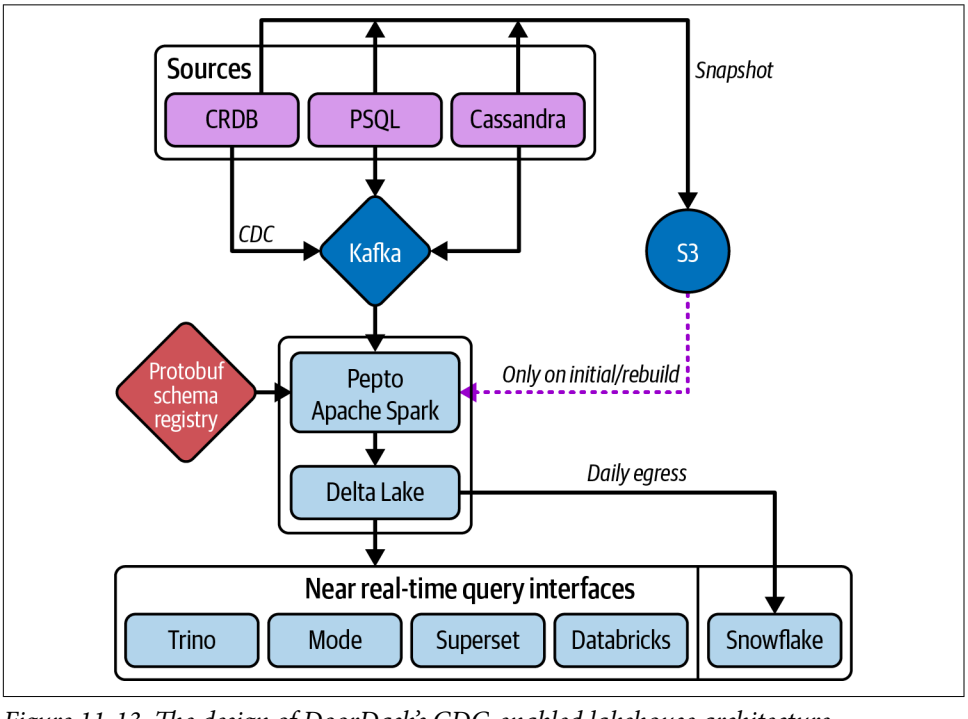

Figure 11-13. The design of DoorDash's CDC-enabled lakehouse architecture

The success of this design could be measured in many ways, but there are several aspects that the team highlights. The system supports 450 streams (one-to-one with tables) running 24/7 on more than one thousand EC2 nodes. This translates to about 800 GB ingested daily from Kafka, with a total daily processing volume of about 80 TB. The design far exceeded the initial requirements and attained a data freshness of less than 30 minutes. The team has enabled the self-service creation of tables for data users in the environment that become available in less than an hour.

Delta and Flink in Harmony

With real-time events being of central importance to DoorDash, its heavy use of Kafka is hardly surprising. Apache Spark is a natural choice for many stream processing applications; however, it's not the only choice. Some teams at DoorDash use Apache Flink for many real-time processes, and therefore it should also be easily supportable. In Chapter 4 you saw how the Flink/Delta Connector works operationally, but here it could be useful to see how this can be pulled into a larger data ecosystem to provide both flexibility and reliability.[19]

The real-time platform team at DoorDash is managing petabytes of vital customer events every day and needs to provide a platform to enable data users and applications to capture, create, or access this information (see Figure 11-14).[20] Adding the Flink/Delta Connector extends the number of ways that users and applications can interact with Delta Lake, which combines the fast operational nature of Flink with a storage format built to handle exactly those kinds of workloads and provides a common format usable across the whole data platform, even while different teams choose to leverage different application processing frameworks.

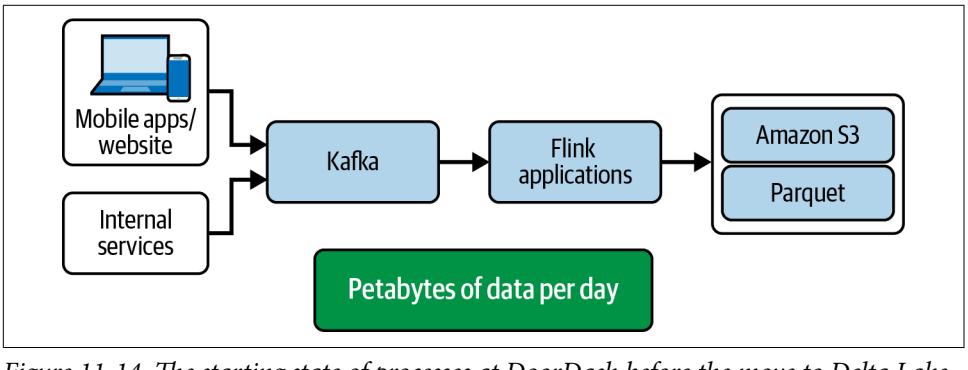

Figure 11-14. The starting state of processes at DoorDash before the move to Delta Lake

Figure 11-15 shows exactly what this change at DoorDash enabled: easy integration with its current tooling with the addition of ACID guarantees at a massive scale. Previously this process was taking place with regular Parquet files, adding additional complications in the form of write locks and other challenges. Additionally, the quality-of-life improvements gained through easy-to-use compaction operations and the ability to do these operations while stream processing applications are still

19 Fabian Paul, Pawel Kubit, Scott Sandre, Tathagata Das, and Denny Lee, "Writing to Delta Lake from Apache Flink", Delta Lake (blog), April 27, 2022.

20 Allen Wang, "Building Scalable Real Time Event Processing with Kafka and Flink", *DoorDash Engineering* (blog), DoorDash, August 2, 2022; Allison Cheng, "Flink + Delta: Driving Real-Time Pipelines at DoorDash", posted July 26, 2023, by Databricks, YouTube.

running are highly valuable, as is the efficiently queryable state achieved through the inclusion of Z-Ordering clusters on the data.

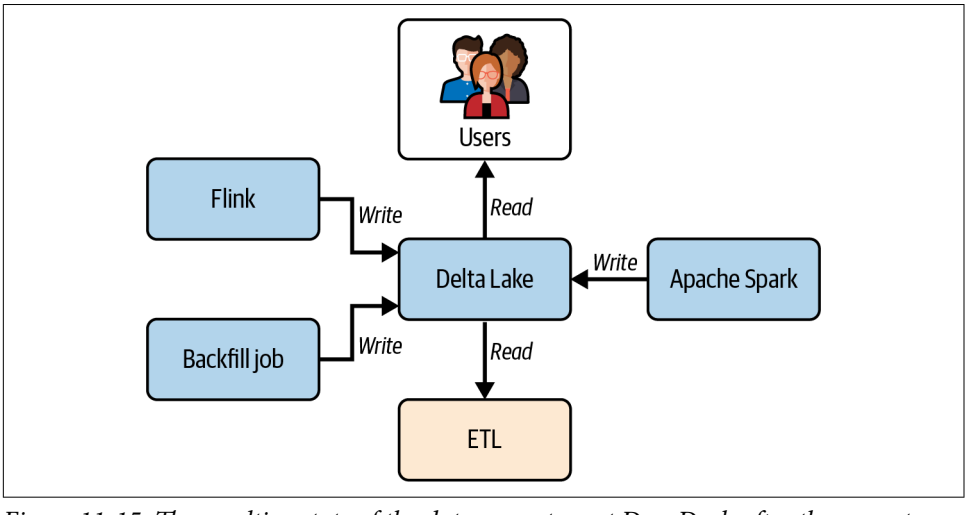

Figure 11-15. The resulting state of the data ecosystem at DoorDash after the move to Delta Lake

The moral of the story of the DoorDash decision to adopt Delta Lake is this: even for data systems with multiple types of tooling operating at massive scale and with a need to support things like efficiently capturing data from real-time event streams or the changes coming through operational databases, Delta Lake provides reliability and usability, making it a winning choice.

Conclusion

Data applications come in many different forms and formats. Authoring those data applications can be complex and painful. In this chapter you've seen a few ways to alleviate this pain through the many benefits of Delta Lake. In particular, the features of Delta Lake help create a robust data environment that supports broad tooling choices, reduces costs, and improves your quality of life as a developer.

CHAPTER 12

Foundations of Lakehouse Governance and Security

We do many things every day without consciously thinking about them. These rote actions, or automatic behaviors, are based on our daily routines and on information we've grown to trust over time. Our routines can be simple or complex, with actions grouped and categorized into different logical buckets. Consider, for example, the routine of locking up before leaving for the day; this is a common behavior for mitigating risk, because we simply can't trust everyone to have our best interests in mind. Think about this risk mitigation as a simple story: *to prevent unauthorized access to a physical location (entity: home, car, office), access controls (locking mechanism) have been introduced to secure a physical space (resource) and provide authorized admittance only when trust can be confirmed (key, credentials).*

In the simplest sense, the only thing preventing intrusion is *the key*. While a key grants access to a given physical space via a lock, the bearer of a given key must also know the physical location of a protected resource; otherwise, the key has no use. This is an example of site security, and as a mental model, it is useful when constructing a plan for the layered governance and security model for resources contained within our lakehouse. After all, the lakehouse is a safe space that protects what we hold near and dear *only* if we collectively govern the resources contained within.

But what exactly is the governance of a data resource, and how do we get started when there are many components of the governance landscape?

This chapter provides a foundation for architecting a scalable data governance strategy for the data assets (resources) contained within the lakehouse. While we aim to cover as much surface area here as possible, consider this a referential chapter just scratching the surface of the myriad components of lakehouse data governance. For example, we won't cover governance with respect to compliance and enforcement of region-specific rules and regulations (GDPR, CCPA, right-to-be-forgotten policies, and so on), nor will we cover general governance from a nonengineering or nontechnical perspective.

Lakehouse Governance

Before we dive deeper into lakehouse governance, it is important to introduce the many components of governance today. The reason for this is that *governance* is an overloaded term that means many different things, depending on who you ask. Therefore, in order to go beyond basic access controls and traditional database-level governance, we need to introduce the systems and services that can come together to provide a comprehensive governance solution for our lakehouse.

There are many components to lakehouse governance, as seen in Figure 12-1, but at a high level, we can simply break them down between *identity and access management* or IAM (1) and *catalog services* (2–8). This allows us to build a working model that is easier to adopt.

For example, unless we understand who (or what) is requesting access to our data (identity services), we cannot manage the permissions enabling access—seen as the union between identity services and policies. Furthermore, without integrating the policies and rules contained within our IAM services (1) with the physical filesystem (3), we will not be capable of governing the databases (schemas), tables, views, and other assets stored in our catalog metastores (2). Given the modern separation between metadata management (2) and physical filesystem resources (3), the foundation for any lakehouse governance begins with the basic delegation of access to resources in a unified and controlled way.

Modern lakehouse governance includes (4) robust auditing across data management operations on a per-action basis, commonly captured through event logs for state changes made via IAM (1) for the resources registered within the catalog or metastore (3).

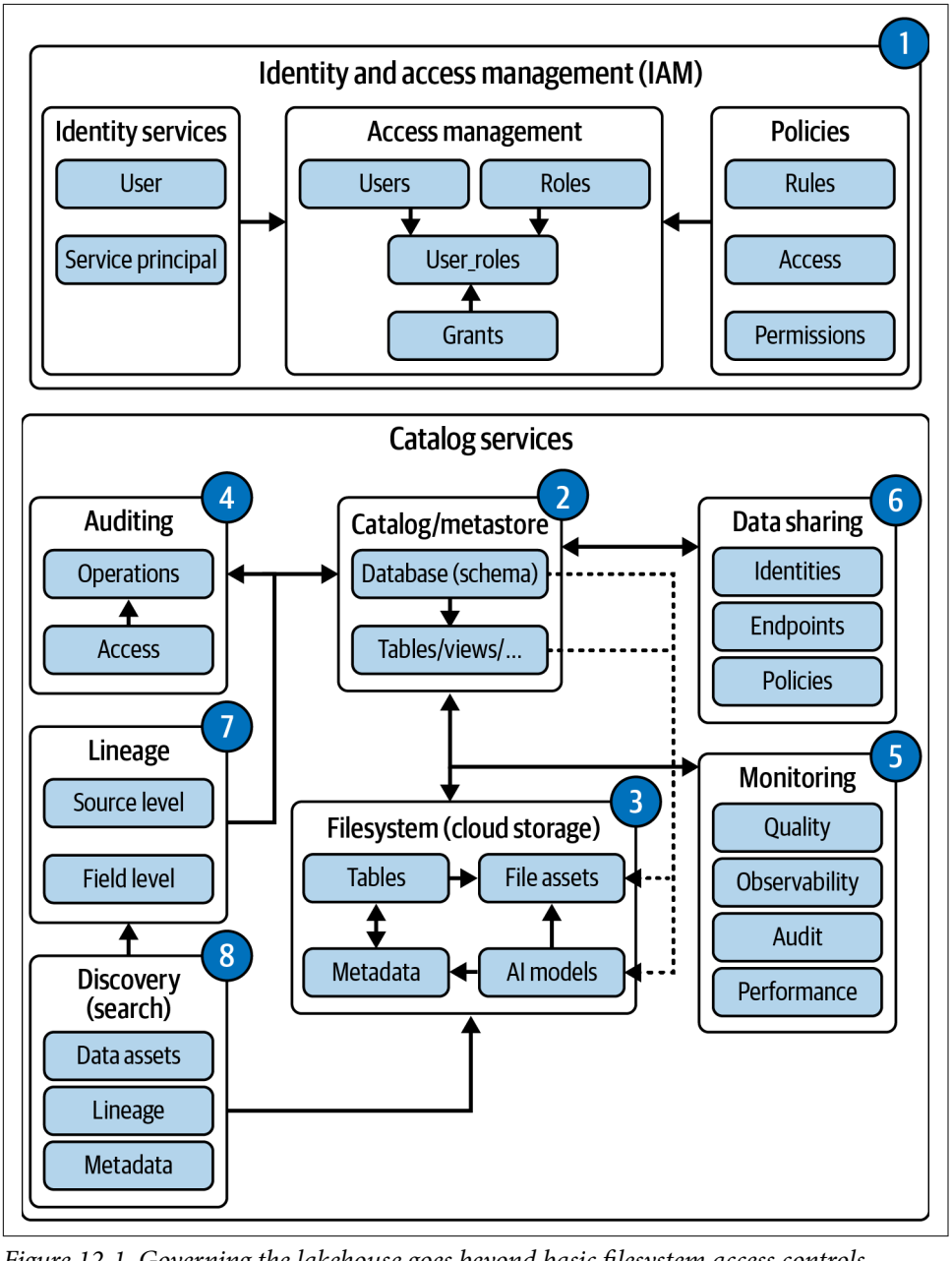

Figure 12-1. Governing the lakehouse goes beyond basic filesystem access controls