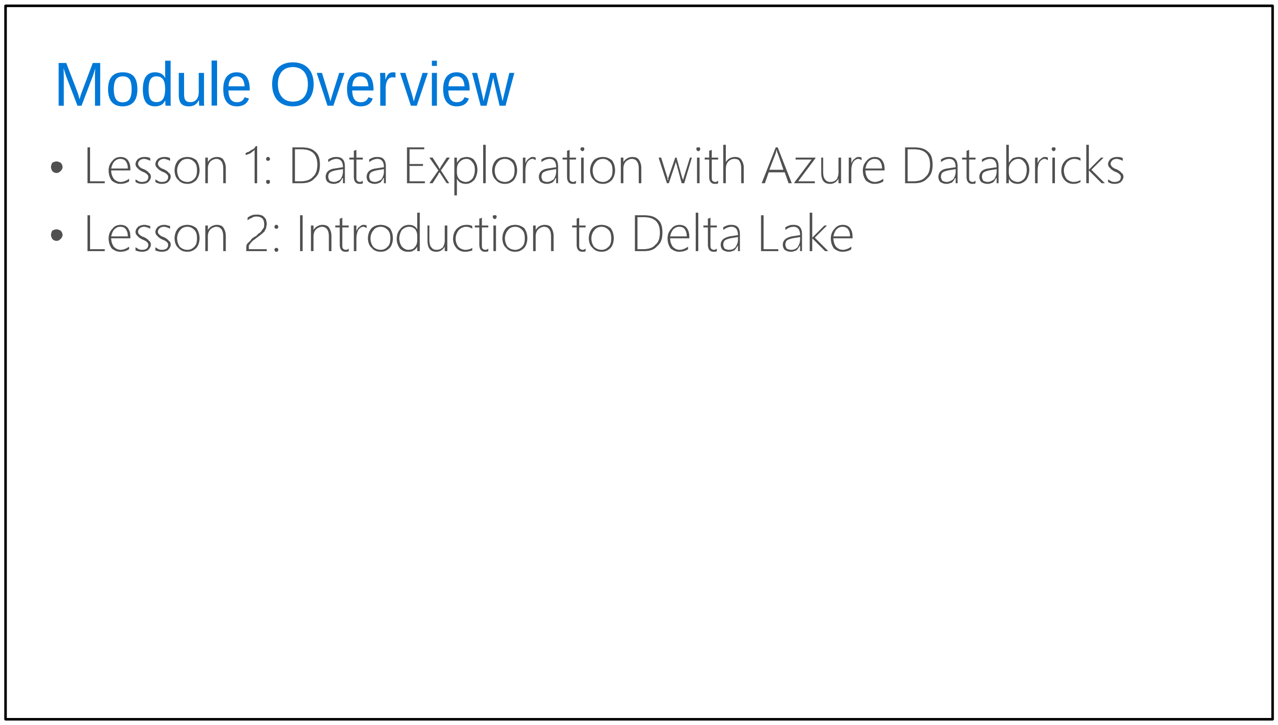
Module: Data Exploration in Azure Databricks and Introduction to Delta Lake

Microsoft Services



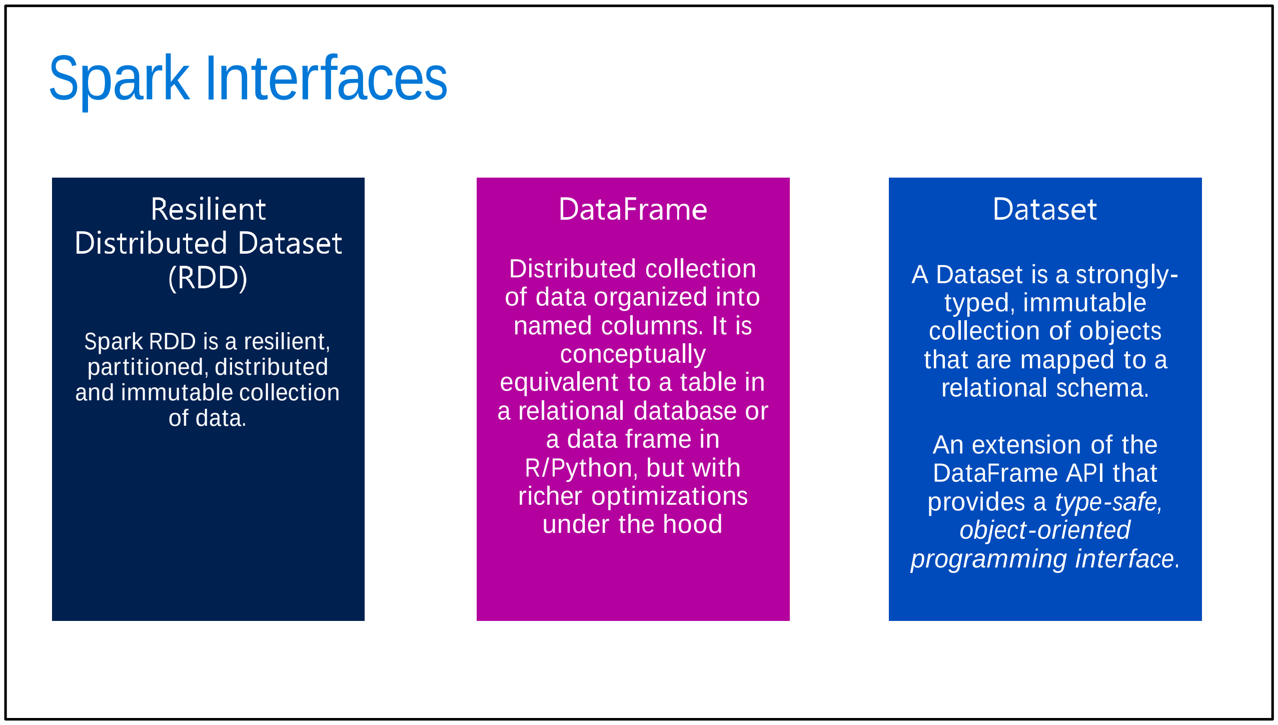
Module Overview

- Lesson 1: Data Exploration with Azure Databricks
- Lesson 2: Introduction to Delta Lake

Lesson 1: Data Exploration with Azure Databricks

After completing this lesson, you will be able to:

- Understand Apache Spark DataFrame and Dataset
- Understand data exploration techniques using Python or Scala



Spark Interfaces

Resilient Distributed Dataset (RDD)

Spark RDD is a resilient, partitioned, distributed and immutable collection of data.

DataFrame

Distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood

Dataset

A Dataset is a strongly-typed, immutable collection of objects that are mapped to a relational schema.

An extension of the DataFrame API that provides a *type-safe, object-oriented programming interface*.

All that you are going to do in Apache Spark is to read some data from a source and load it into Spark. You will then process the data and hold the intermediate results, and finally write the results back to a destination. But in this process, you need a data structure to hold the data in Spark. We have three alternatives to hold data in Spark.
RDD
Data Frame
Dataset
Apache Spark 2.x recommends to use the last two and avoid using RDDs. However, there is a critical fact to note about RDDs. Data Frames and Datasets, both of them are ultimately compiled down to an RDD. So, under the hood, everything in Spark is an RDD. And for that reason, I will start with RDDs and try to explain the mechanics of parallel processing.

Spark RDD
Let's define the RDD. The name stands for Resilient Drstributed Dataset. However, I can describe a RDD as below.
Spark RDD is a resilient, partitioned, distributed and immutable collection of data. Let's

quickly review this description.
Collection of data - RDDs hold data and appears to be a Scala Collection.
Resilient - RDDs can recover from a failure, so they are fault tolerant.
Partitioned - Spark breaks the RDD into smaller chunks of data. These pieces are called partitions.
Distributed - Instead of keeping those partitions on a single machine, Spark spreads them across the cluster. So they are a distributed collection of data.
Immutable - Once defined, you can't change a RDD. So Spark RDD is a read-only data structure.

When to use RDDs?
Consider these scenarios or common use cases for using RDDs when:
you want low-level transformation and actions and control on your dataset;
your data is unstructured, such as media streams or streams of text;
you want to manipulate your data with functional programming constructs than domain specific expressions;
you don’t care about imposing a schema, such as columnar format, while processing or accessing data attributes by name or column; and
you can forgo some optimization and performance benefits available with DataFrames and Datasets for structured and semi-structured data.

More Information: https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

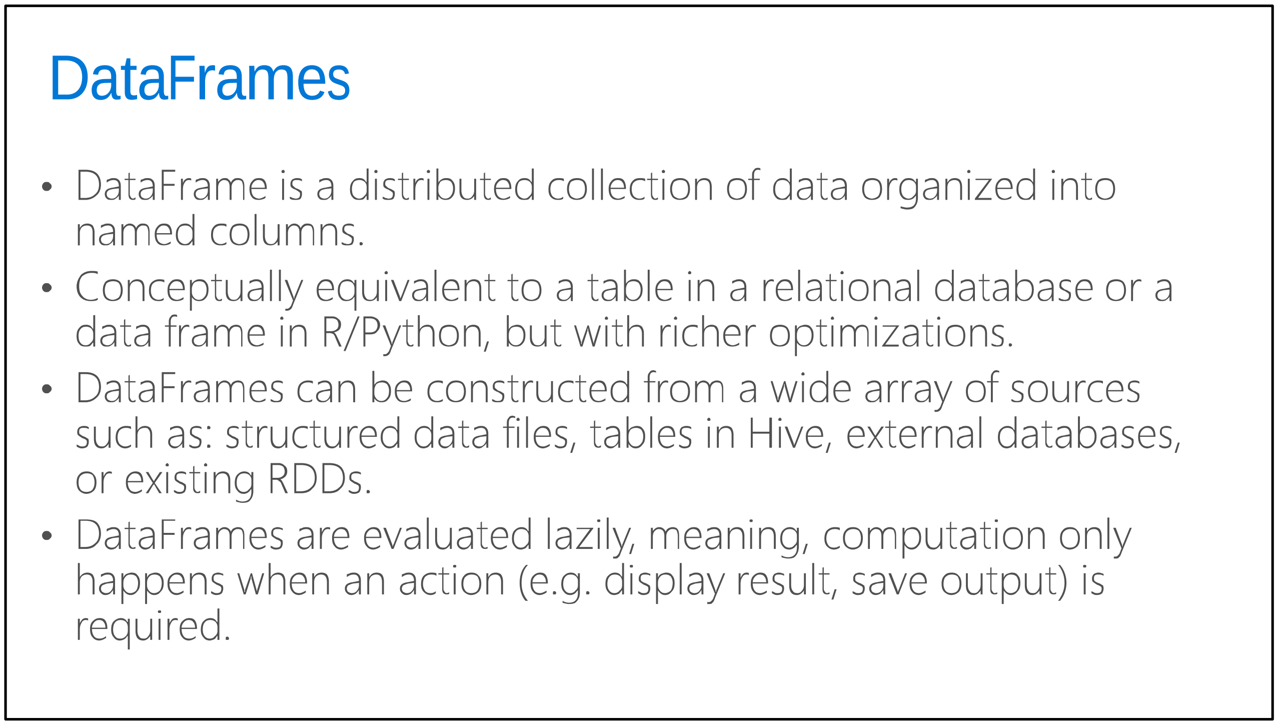
What Are DataFrames?
In Spark, a DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs.
Similar to RDDs, DataFrames are evaluated lazily. That is to say, computation only happens when an action (e.g. display result, save output) is required. This allows their executions to be optimized, by applying techniques such as predicate push-downs and bytecode generation, as explained later in the section “Under the Hood: Intelligent Optimization and Code Generation”. All DataFrame operations are also automatically parallelized and distributed on clusters.

More Information: https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html

DataSets
A Dataset is a strongly-typed, immutable collection of objects that are mapped to a relational schema. At the core of the Dataset API is a new concept called an encoder, which is responsible for converting between JVM objects and tabular representation. The tabular representation is stored using Spark's internal Tungsten binary format, allowing for operations on serialized data and improved memory utilization. Spark 1.6 comes with support for automatically generating encoders for a wide variety of types, including primitive types (e.g. String, Integer, Long), Scala case classes, and Java Beans.

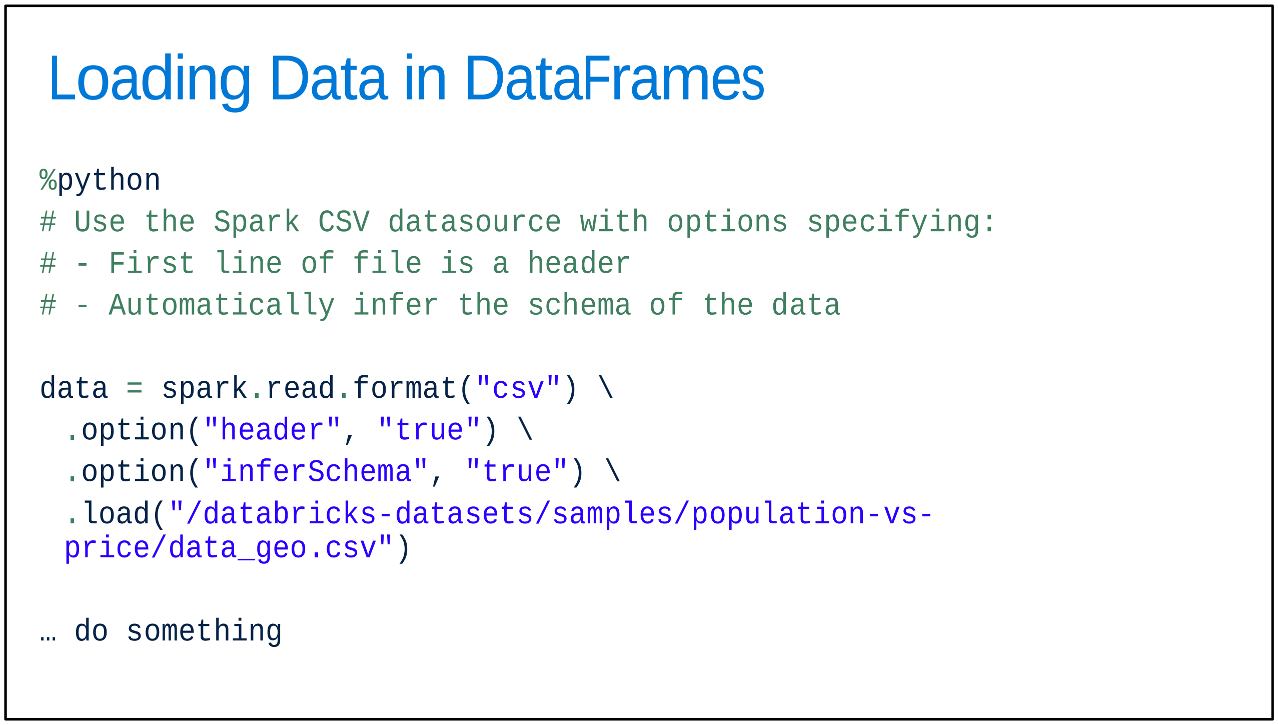
More Information: https://databricks.com/blog/2016/01/04/introducing-apache-spark-datasets.html

In many scenarios, especially with the performance optimizations embedded in DataFrames and Datasets, it will not be necessary to work with RDDs. But it is important to understand the RDD abstraction because:
The RDD is the underlying infrastructure that allows Spark to run so fast and provide data lineage.
If you are diving into more advanced components of Spark, it may be necessary to use RDDs.
The visualizations within the Spark UI reference RDDs.



DataFrames

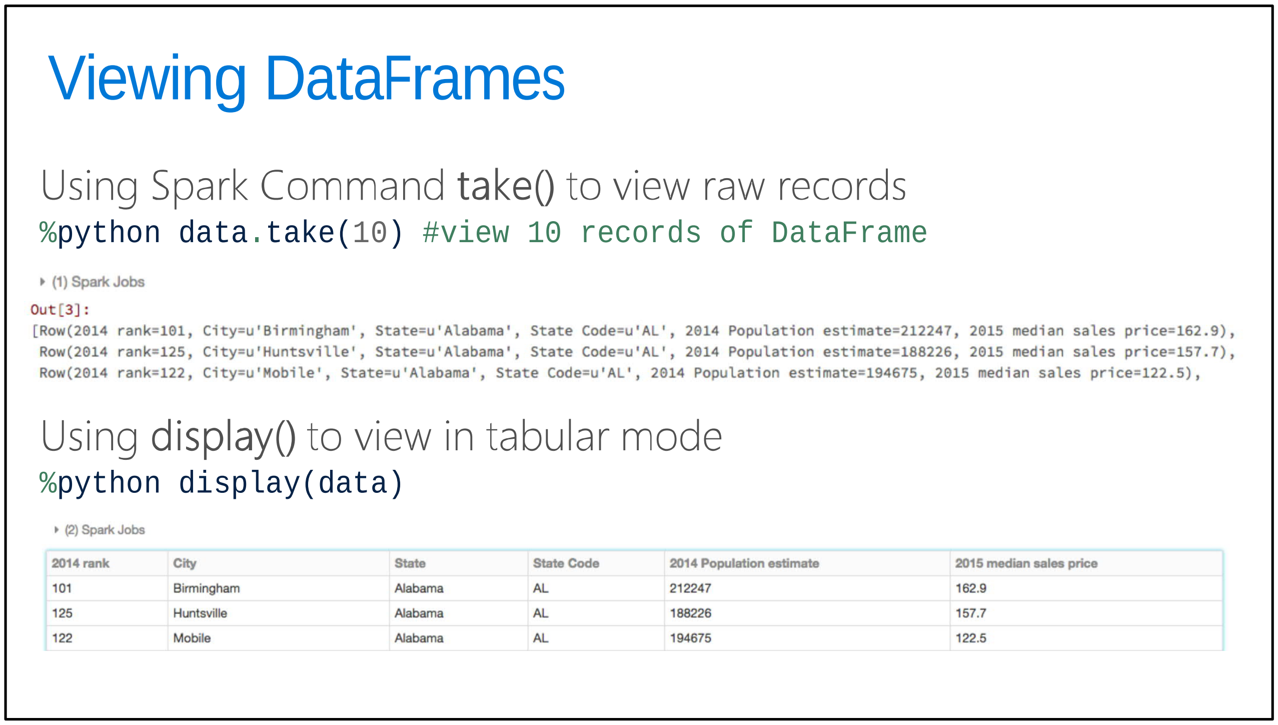
- DataFrame is a distributed collection of data organized into named columns.
- Conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations.
- DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs.
- DataFrames are evaluated lazily, meaning, computation only happens when an action (e.g. display result, save output) is required.



Loading Data in DataFrames

```
%python
# Use the Spark CSV datasource with options specifying:
# - First line of file is a header
# - Automatically infer the schema of the data

data = spark.read.format("csv") \
  .option("header", "true") \
  .option("inferSchema", "true") \
  .load("/databricks-datasets/samples/population-vs-
  price/data_geo.csv")

… do something
```

The easiest way to start working with DataFrames is to use an example Azure Databricks dataset available in the /databricks-datasets folder accessible within the Azure Databricks workspace. To access the file that compares city population versus median sale prices of homes, load the file /databricks-datasets/samples/population-vs-price/data_geo.csv.



Viewing DataFrames

Using Spark Command **take()** to view raw records

```
%python data.take(10) #view 10 records of DataFrame
```

▸ (1) Spark Jobs

```
Out[3]:
[Row(2014 rank=101, City=u'Birmingham', State=u'Alabama', State Code=u'AL', 2014 Population estimate=212247, 2015 median sales price=162.9),
 Row(2014 rank=125, City=u'Huntsville', State=u'Alabama', State Code=u'AL', 2014 Population estimate=188226, 2015 median sales price=157.7),
 Row(2014 rank=122, City=u'Mobile', State=u'Alabama', State Code=u'AL', 2014 Population estimate=194675, 2015 median sales price=122.5),
```

Using **display()** to view in tabular mode

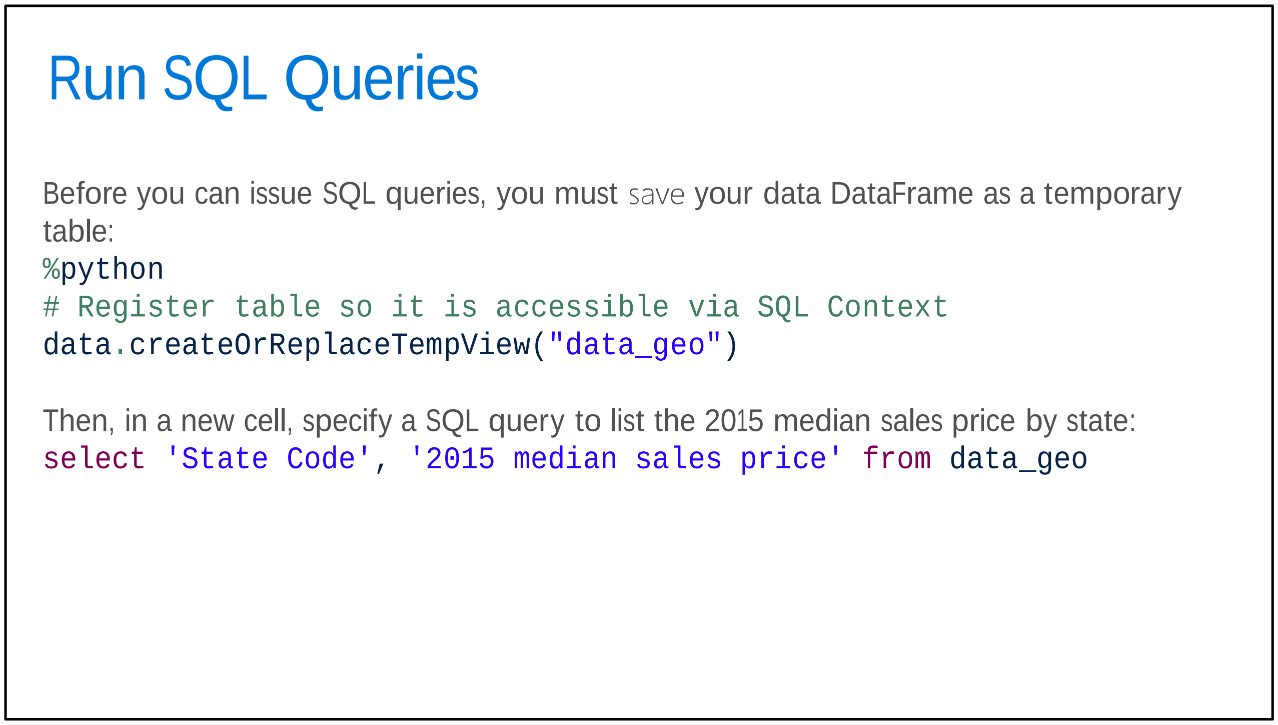
```
%python display(data)
```

▸ (2) Spark Jobs

2014 rank	City	State	State Code	2014 Population estimate	2015 median sales price
101	Birmingham	Alabama	AL	212247	162.9
125	Huntsville	Alabama	AL	188226	157.7
122	Mobile	Alabama	AL	194675	122.5

Now that you have created the data DataFrame, you can quickly access the data using standard Spark commands such as take(). For example, you can use the command data.take(10) to view the first ten rows of the data DataFrame. Because this is a SQL notebook, the next few commands use the %python magic command.

To view this data in a tabular format, you can use the Azure Databricks display() command instead of exporting the data to a third-party tool.



Run SQL Queries

Before you can issue SQL queries, you must save your data DataFrame as a temporary table:

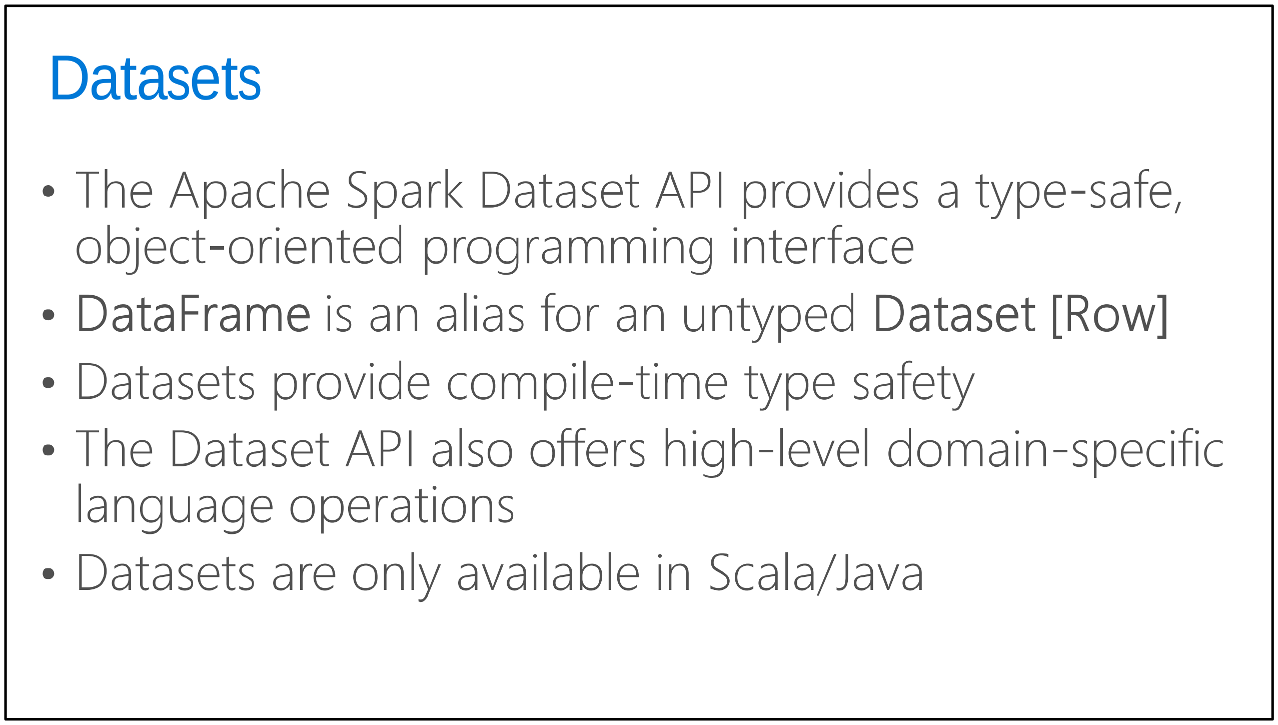
```
%python
# Register table so it is accessible via SQL Context
data.createOrReplaceTempView("data_geo")
```

Then, in a new cell, specify a SQL query to list the 2015 median sales price by state:

```
select 'State Code', '2015 median sales price' from data_geo
```

Using the dataframe “data” we will create a temporary view/table named “data_geo”. Once we have a tempview created, we can run sql queries against “data_geo”.

Temporary tables persist for the duration of the spark session.

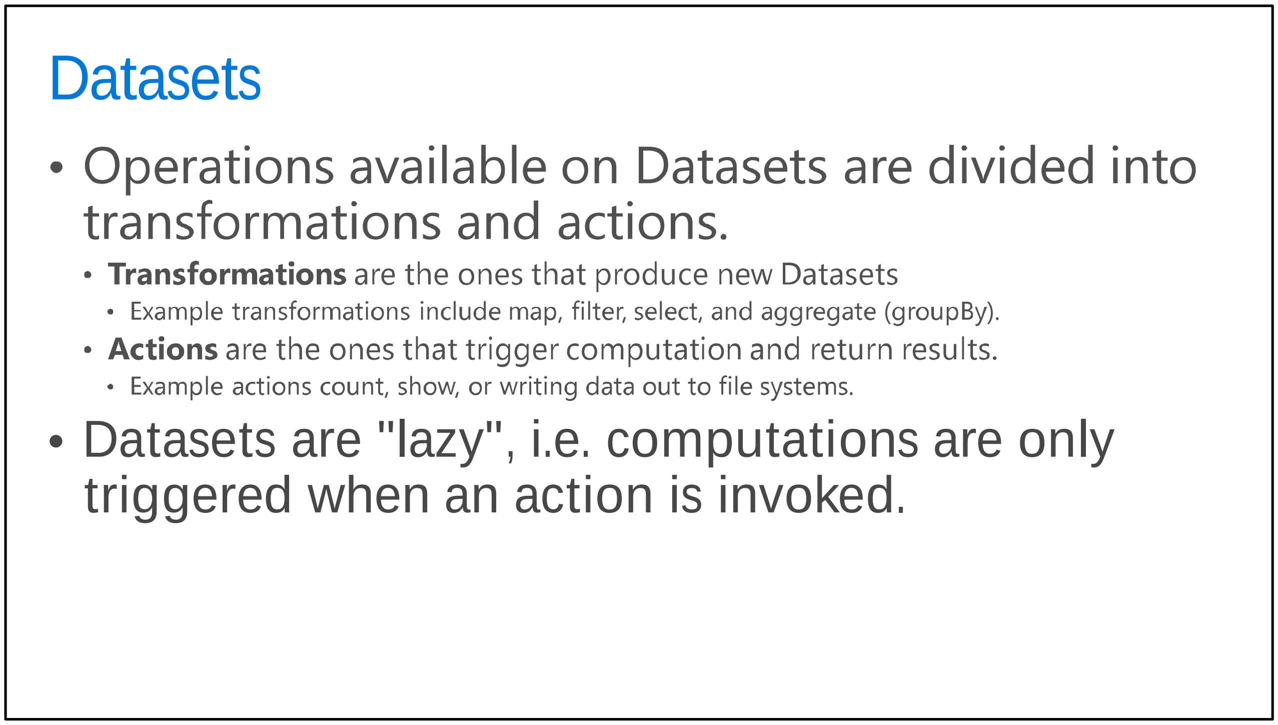


Datasets

- The Apache Spark Dataset API provides a type-safe, object-oriented programming interface
- DataFrame is an alias for an untyped Dataset [Row]
- Datasets provide compile-time type safety
- The Dataset API also offers high-level domain-specific language operations
- Datasets are only available in Scala/Java

The Apache Spark Dataset API provides a type-safe, object-oriented programming interface. DataFrame is an alias for an untyped Dataset [Row]. Datasets provide compile-time type safety—which means that production applications can be checked for errors before they are run—and they allow direct operations over user-defined classes. The Dataset API also offers high-level domain-specific language operations like sum(), avg(), join(), select(), groupBy(), making the code a lot easier to express, read, and write.

Must Read to understand DataSet:
https://spark.apache.org/docs/latest/api/java/org/apache/spark/sql/Dataset.html

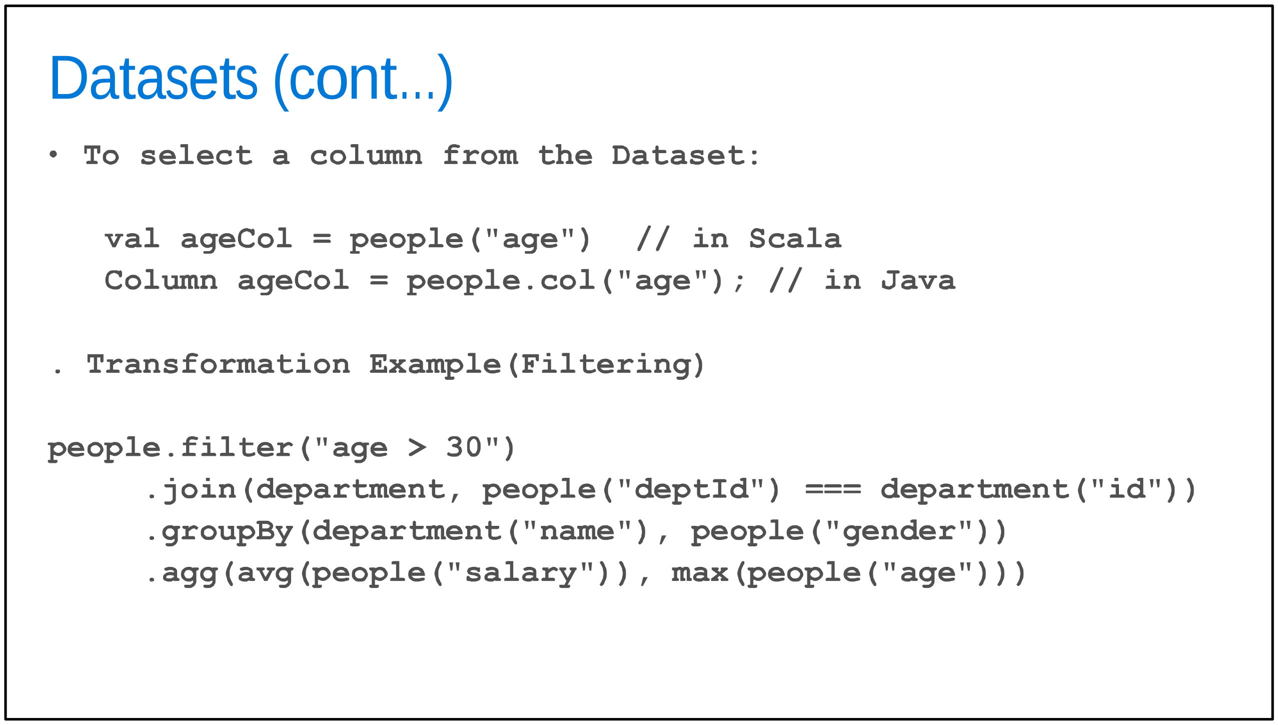


Datasets

- Operations available on Datasets are divided into transformations and actions.
  - **Transformations** are the ones that produce new Datasets
    - Example transformations include map, filter, select, and aggregate (groupBy).
  - **Actions** are the ones that trigger computation and return results.
    - Example actions count, show, or writing data out to file systems.
- Datasets are "lazy", i.e. computations are only triggered when an action is invoked.

The Apache Spark Dataset API provides a type-safe, object-oriented programming interface. DataFrame is an alias for an untyped Dataset [Row]. Datasets provide compile-time type safety—which means that production applications can be checked for errors before they are run—and they allow direct operations over user-defined classes. The Dataset API also offers high-level domain-specific language operations like sum(), avg(), join(), select(), groupBy(), making the code a lot easier to express, read, and write.

Must Read to understand DataSet:
https://spark.apache.org/docs/latest/api/java/org/apache/spark/sql/Dataset.html



Datasets (cont...)

• To select a column from the Dataset:

```
    val ageCol = people("age")   // in Scala
    Column ageCol = people.col("age"); // in Java
```

. Transformation Example(Filtering)

```
people.filter("age > 30")
      .join(department, people("deptId") === department("id"))
      .groupBy(department("name"), people("gender"))
      .agg(avg(people("salary")), max(people("age")))
```

https://spark.apache.org/docs/latest/api/java/org/apache/spark/sql/Dataset.html

Datasets (cont...)

```
case class Person(name: String, age: Long)

// Encoders are created for case classes
val caseClassDS = Seq(Person("Andy", 32)).toDS()
caseClassDS.show()

val path = "examples/src/main/resources/people.json"
val peopleDS = spark.read.json(path).as[Person]
peopleDS.show()
```

https://spark.apache.org/docs/latest/sql-getting-started.html#creating-datasets

Datasets (cont…)

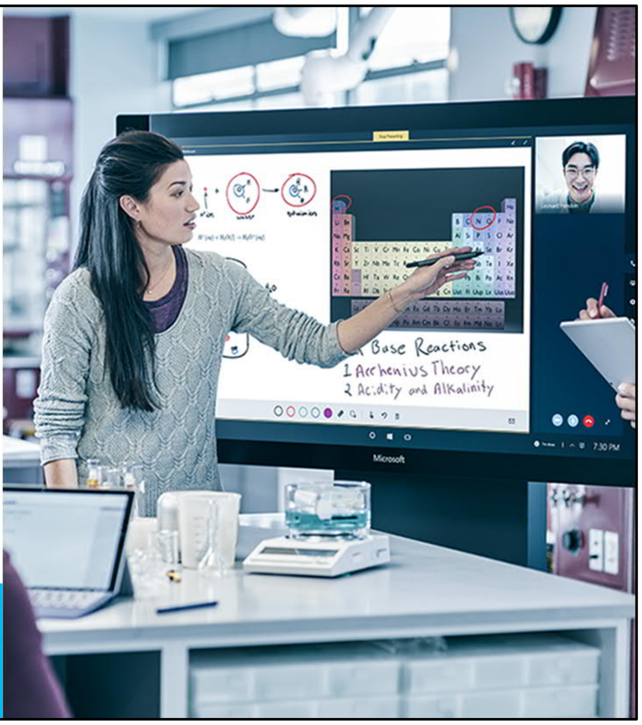
```
// Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

val sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

https://spark.apache.org/docs/latest/sql-getting-started.html#running-sql-queries-programmatically

Datasets (cont…)

- To efficiently support domain-specific objects, an Encoder is required.
  - The encoder maps the domain specific type T to Spark's internal type system.
  - For example, given a class Person with two fields, name (string) and age (int), an encoder is used to tell Spark to generate code at runtime to serialize the Person object into a binary structure.
- This binary structure often has much lower memory footprint as well as are optimized for efficiency in data processing (e.g. in a columnar format).

Lab: Data Exploration with Azure Databricks



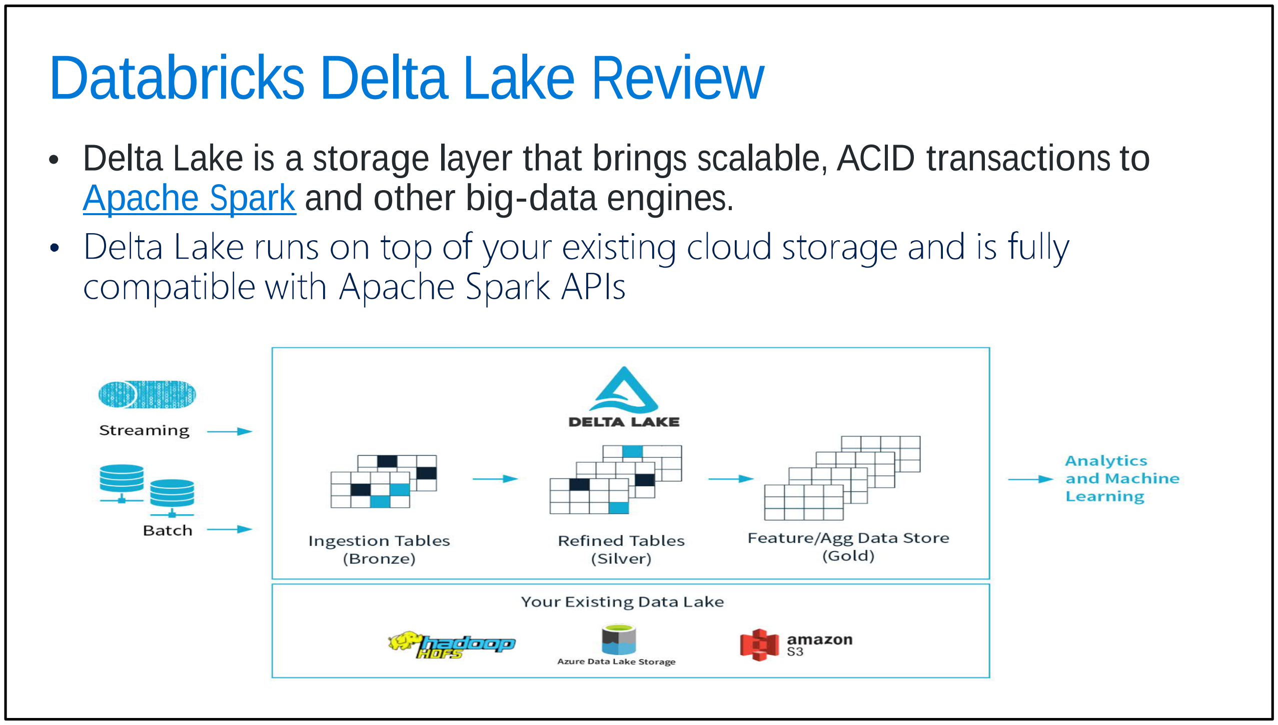
M03 L01 Lab 01
30 mins

Refer to M03_L01_Lab01.ipynb file for Databricks notebook code. Remember to import the notebook to your workspace and read the directions before executing the code

Lesson 2: Introduction to Delta Lake

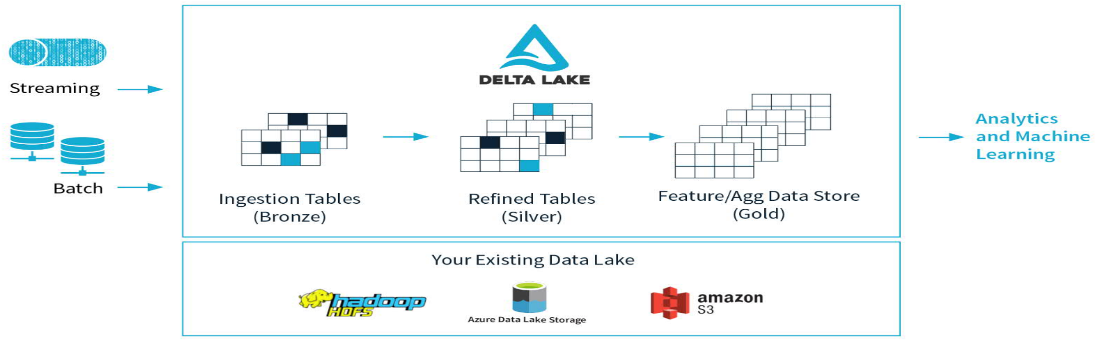
After completing this learning unit, you will be able to:

- Explore the main features of Delta Lake
- Learn how to read from and write to Delta tables from interactive queries.
- Learn how to optimize Delta Lake performances with file management



Databricks Delta Lake Review

- Delta Lake is a storage layer that brings scalable, ACID transactions to Apache Spark and other big-data engines.
- Delta Lake runs on top of your existing cloud storage and is fully compatible with Apache Spark APIs



Delta Lake is an open source storage layer that brings reliability to data lakes. Delta Lake provides ACID transactions, scalable metadata handling, and unifies streaming and batch data processing. Delta Lake runs on top of your existing data lake and is fully compatible with Apache Spark APIs.

Specifically, Delta Lake offers:

•ACID (Atomicity, Concurrency, Isolation, Durability) transactions on Data Lake: Serializable isolation levels ensure that readers never see inconsistent data.

•Scalable metadata handling: Leverages Spark's distributed processing power to handle all the metadata for petabyte-scale tables with billions of files at ease.

•Streaming and batch unification: A table in Delta Lake can be a batch as well as a streaming source and sink. Streaming data ingest, batch historic backfill, interactive queries all just work out of the box.

•Schema enforcement: Automatically handles schema variations to prevent insertion of bad records during ingestion.

•Time travel: Data versioning enables rollbacks, full historical audit trails, and reproducible machine learning experiments.

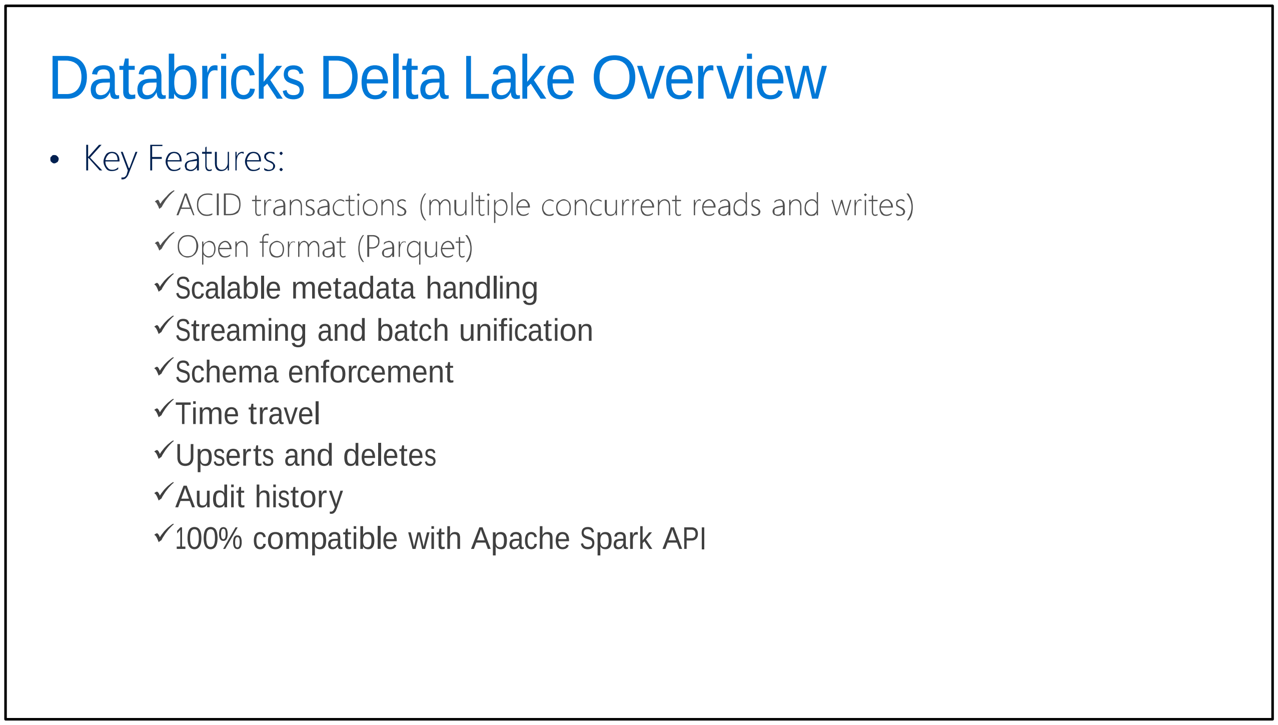
•Upserts and deletes: Supports merge, update and delete operations to enable complex use cases like change-data-capture, slowly-changing-dimension (SCD) operations, streaming upserts, and so on.

https://docs.delta.io/latest/delta-intro.html

Deltalake (cont...)

- Delta Lake is open source software that extends Parquet data files with a file-based transaction log for ACID transactions and scalable metadata handling.
- Delta Lake is the default format for all operations on Azure Databricks. Unless otherwise specified, all tables on Azure Databricks are Delta tables.
- Databricks originally developed the Delta Lake protocol and continues to actively contribute to the open source project.

https://learn.microsoft.com/en-us/azure/databricks/delta/tutorial



Databricks Delta Lake Overview

- Key Features:
  - ✓ACID transactions (multiple concurrent reads and writes)
  - ✓Open format (Parquet)
  - ✓Scalable metadata handling
  - ✓Streaming and batch unification
  - ✓Schema enforcement
  - ✓Time travel
  - ✓Upserts and deletes
  - ✓Audit history
  - ✓100% compatible with Apache Spark API

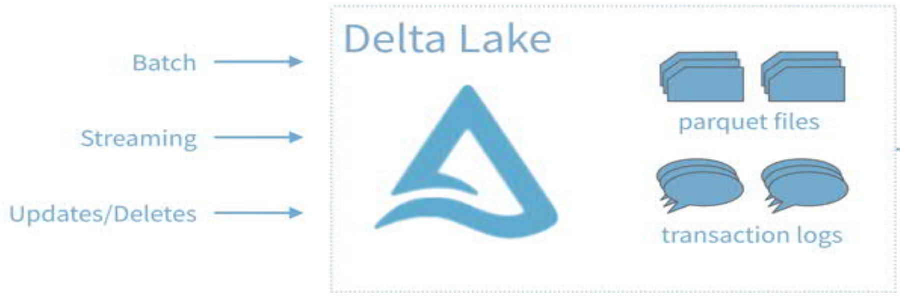
Key Features
ACID Transactions:
Data lakes typically have multiple data pipelines reading and writing data concurrently, and data engineers have to go through a tedious process to ensure data integrity, due to the lack of transactions. Delta Lake brings ACID transactions to your data lakes. It provides serializability, the strongest level of isolation level. Learn more at Diving into Delta Lake: Unpacking the Transaction Log.
Scalable Metadata Handling:
In big data, even the metadata itself can be "big data". Delta Lake treats metadata just like data, leveraging Spark's distributed processing power to handle all its metadata. As a result, Delta Lake can handle petabyte-scale tables with billions of partitions and files at ease.
Time Travel (data versioning):
Delta Lake provides snapshots of data enabling developers to access and revert to earlier versions of data for audits, rollbacks or to reproduce experiments. Learn more in Introducing Delta Lake Time Travel for Large Scale Data Lakes.
Open Format:
All data in Delta Lake is stored in Apache Parquet format enabling Delta Lake to leverage the efficient compression and encoding schemes that are native to Parquet.
Unified Batch and Streaming Source and Sink:
A table in Delta Lake is both a batch table, as well as a streaming source and sink.

Streaming data ingest, batch historic backfill, and interactive queries all just work out of the box.
Schema Enforcement:
Delta Lake provides the ability to specify your schema and enforce it. This helps ensure that the data types are correct and required columns are present, preventing bad data from causing data corruption. For more information, refer to Diving Into Delta Lake: Schema Enforcement & Evolution.
Schema Evolution:
Big data is continuously changing. Delta Lake enables you to make changes to a table schema that can be applied automatically, without the need for cumbersome DDL. For more information, refer to Diving Into Delta Lake: Schema Enforcement & Evolution.
Audit History:
Delta Lake transaction log records details about every change made to data providing a full audit trail of the changes.
Updates and Deletes:
Delta Lake supports Scala / Java APIs to merge, update and delete datasets. This allows you to easily comply with GDPR and CCPA and also simplifies use cases like change data capture. For more information, refer to Announcing the Delta Lake 0.3.0 Release and Simple, Reliable Upserts and Deletes on Delta Lake Tables using Python APIs which includes code snippets for merge, update, and delete DML commands.
100% Compatible with Apache Spark API:
Developers can use Delta Lake with their existing data pipelines with minimal change as it is fully compatible with Spark, the commonly used big data processing engine.

https://delta.io/

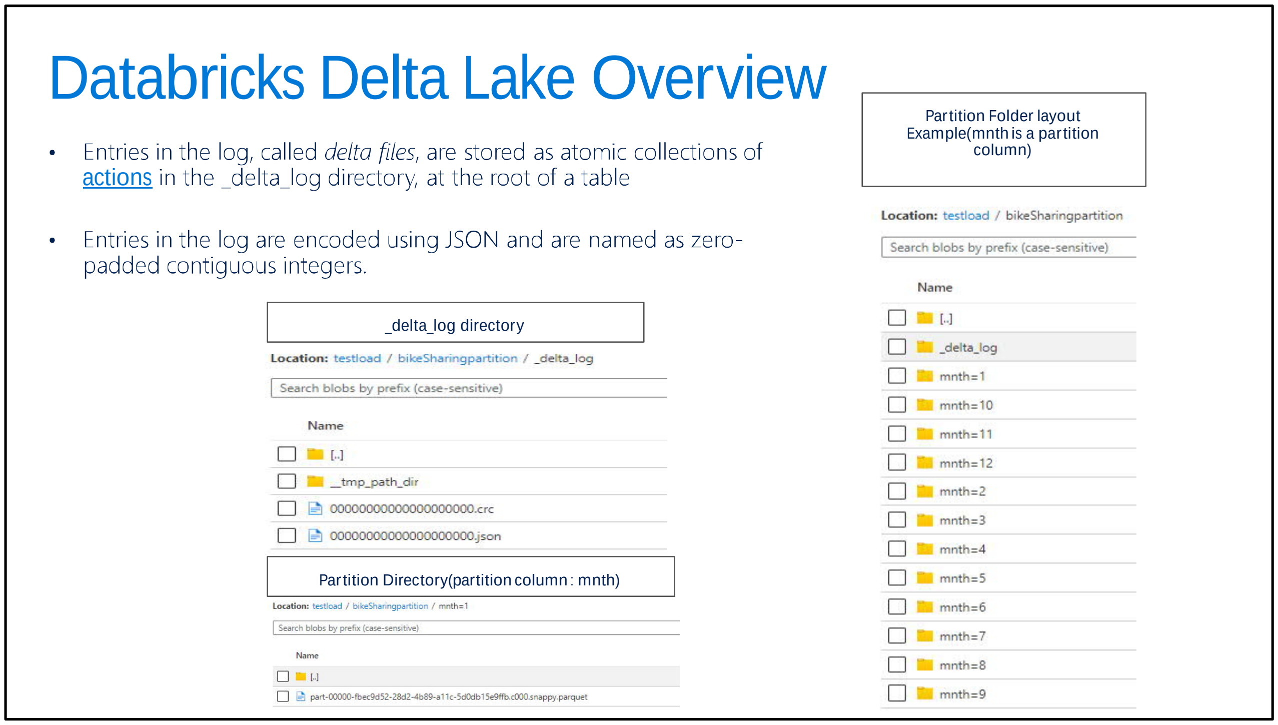
Databricks Delta Lake Overview

- Delta Lake uses versioned Parquet files to store your data in your cloud storage.
- Delta Lake also stores a transaction log to keep track of all the commits made to the table or blob store directory to provide ACID transactions



Refer to these links for review material related to Databricks Delta. Basically, Databricks Delta optimizes storage of data in the Databricks File System through the use of Delta tables which use the Parquet format as their underlying file format.

https://docs.azuredatabricks.net/delta/delta-intro.html#delta-intro
https://docs.azuredatabricks.net/delta/optimizations.html



Databricks Delta Lake Overview

- Entries in the log, called *delta files*, are stored as atomic collections of actions in the _delta_log directory, at the root of a table
- Entries in the log are encoded using JSON and are named as zero-padded contiguous integers.

_delta_log directory

Location: testload / bikeSharingpartition / _delta_log

Search blobs by prefix (case-sensitive)

Name
[..]
_tmp_path_dir
00000000000000000000.crc
00000000000000000000.json

Partition Directory(partition column : mnth)

Location: testload / bikeSharingpartition / mnth=1

Search blobs by prefix (case-sensitive)

Name
[..]
part-00000-fbec9d52-28d2-4b89-a11c-5d0db15e9ffb.c000.snappy.parquet

Partition Folder layout Example(mnth is a partition column)

Location: testload / bikeSharingpartition

Search blobs by prefix (case-sensitive)

Name
[..]
_delta_log
mnth=1
mnth=10
mnth=11
mnth=12
mnth=2
mnth=3
mnth=4
mnth=5
mnth=6
mnth=7
mnth=8
mnth=9

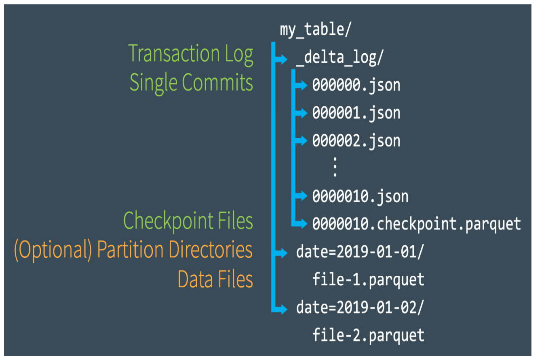
https://databricks.com/blog/2019/08/21/diving-into-delta-lake-unpacking-the-transaction-log.html

Walk through pictorial representation of _delta_log file, partition folder contents and partion root folder layout.

Databricks Delta Lake overview

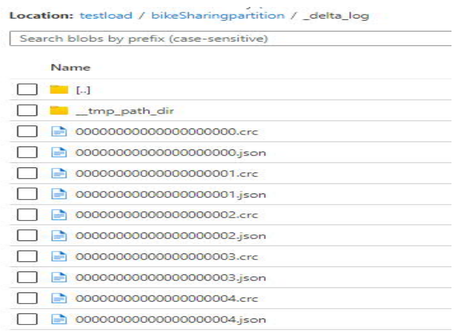
- Delta Lake automatically generates checkpoint files every 10 commits
- These checkpoint files save the entire state of the table at a point in time – in native Parquet format that is quick and easy for Spark to read

Sample Layout



```
my_table/
  _delta_log/
    000000.json
    000001.json
    000002.json
       ⋮
    0000010.json
    0000010.checkpoint.parquet
  date=2019-01-01/
    file-1.parquet
  date=2019-01-02/
    file-2.parquet
```

Partition Directory(partition column : mnth)



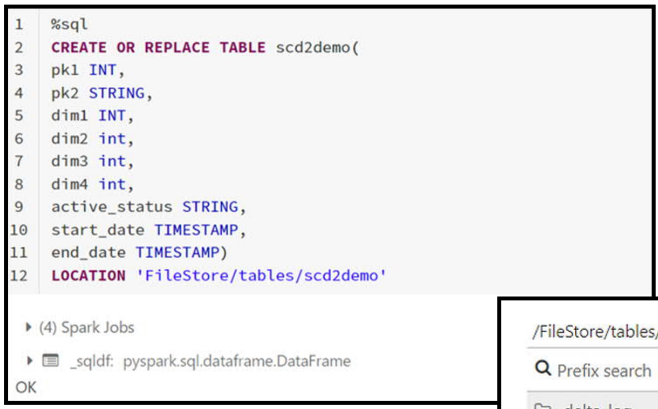
Location: testload / bikeSharingpartition / _delta_log

Search blobs by prefix (case-sensitive)

```
Name
[..]
__tmp_path_dir
00000000000000000000.crc
00000000000000000000.json
00000000000000000001.crc
00000000000000000001.json
00000000000000000002.crc
00000000000000000002.json
00000000000000000003.crc
00000000000000000003.json
00000000000000000004.crc
00000000000000000004.json
```

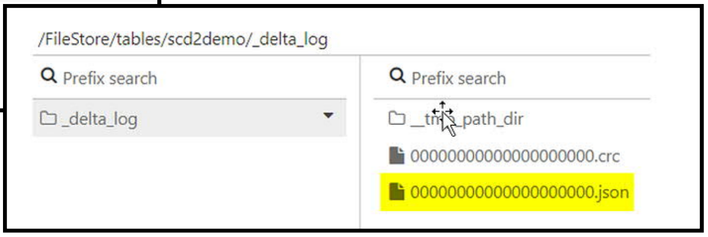
https://databricks.com/blog/2019/08/21/diving-into-delta-lake-unpacking-the-transaction-log.html

Deltalake (cont...)



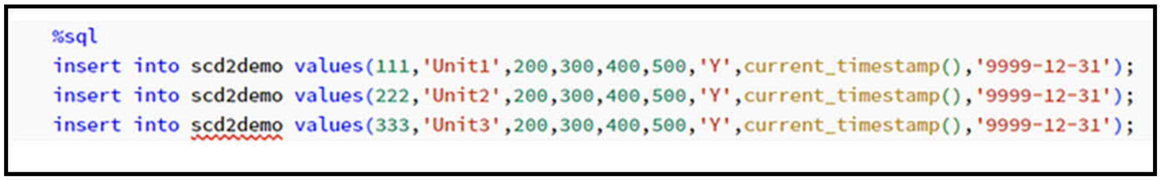
```
%sql
CREATE OR REPLACE TABLE scd2demo(
pk1 INT,
pk2 STRING,
dim1 INT,
dim2 int,
dim3 int,
dim4 int,
active_status STRING,
start_date TIMESTAMP,
end_date TIMESTAMP)
LOCATION 'FileStore/tables/scd2demo'
```

```
(4) Spark Jobs
_sqldf: pyspark.sql.dataframe.DataFrame
OK
```

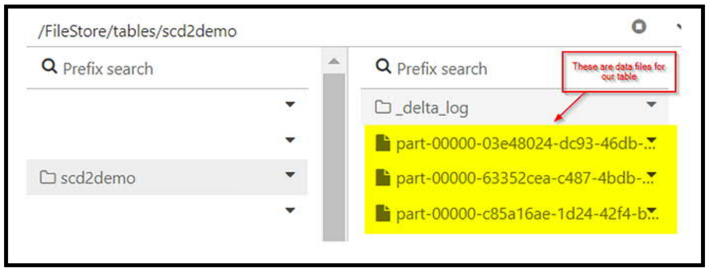


```
/FileStore/tables/scd2demo/_delta_log
Prefix search            Prefix search
_delta_log               __tmp_path_dir
                         00000000000000000000.crc
                         00000000000000000000.json
```

Deltalake (cont…)



```
%sql
insert into scd2demo values(111,'Unit1',200,300,400,500,'Y',current_timestamp(),'9999-12-31');
insert into scd2demo values(222,'Unit2',200,300,400,500,'Y',current_timestamp(),'9999-12-31');
insert into scd2demo values(333,'Unit3',200,300,400,500,'Y',current_timestamp(),'9999-12-31');
```


/FileStore/tables/scd2demo
Prefix search
scd2demo
Prefix search
These are data files for our table
_delta_log
part-00000-03e48024-dc93-46db-...
part-00000-63352cea-c487-4bdb-...
part-00000-c85a16ae-1d24-42f4-b...

https://medium.com/@amarkrgupta96/unlocking-performance-optimize-vacuum-and-z-ordering-in-databricks-delta-tables

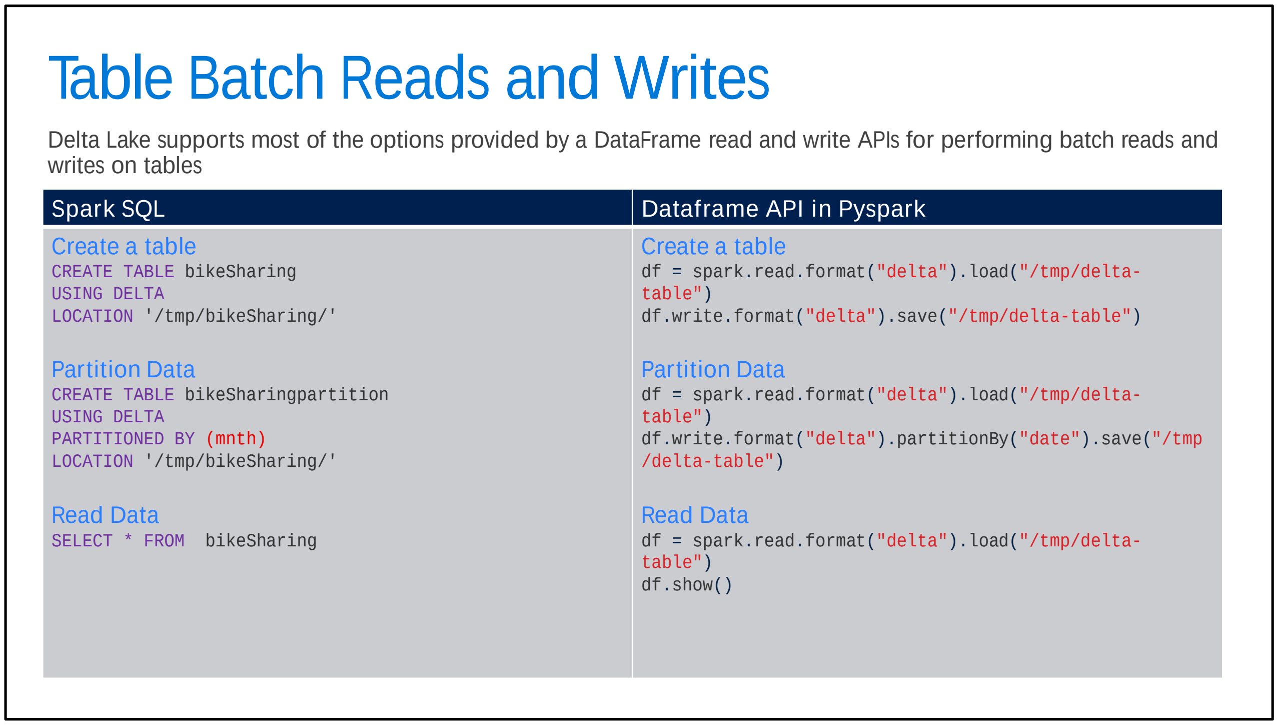


Table Batch Reads and Writes

Delta Lake supports most of the options provided by a DataFrame read and write APIs for performing batch reads and writes on tables

Spark SQL	Dataframe API in Pyspark

Create a table
`CREATE TABLE bikeSharing`
`USING DELTA`
`LOCATION '/tmp/bikeSharing/'`
	Create a table
	`df = spark.read.format("delta").load("/tmp/delta-table")`
	`df.write.format("delta").save("/tmp/delta-table")`

Partition Data
`CREATE TABLE bikeSharingpartition`
`USING DELTA`
`PARTITIONED BY (mnth)`
`LOCATION '/tmp/bikeSharing/'`
	Partition Data
	`df = spark.read.format("delta").load("/tmp/delta-table")`
	`df.write.format("delta").partitionBy("date").save("/tmp/delta-table")`

Read Data
`SELECT * FROM  bikeSharing`
	Read Data
	`df = spark.read.format("delta").load("/tmp/delta-table")`
	`df.show()`

https://docs.delta.io/latest/delta-batch.html

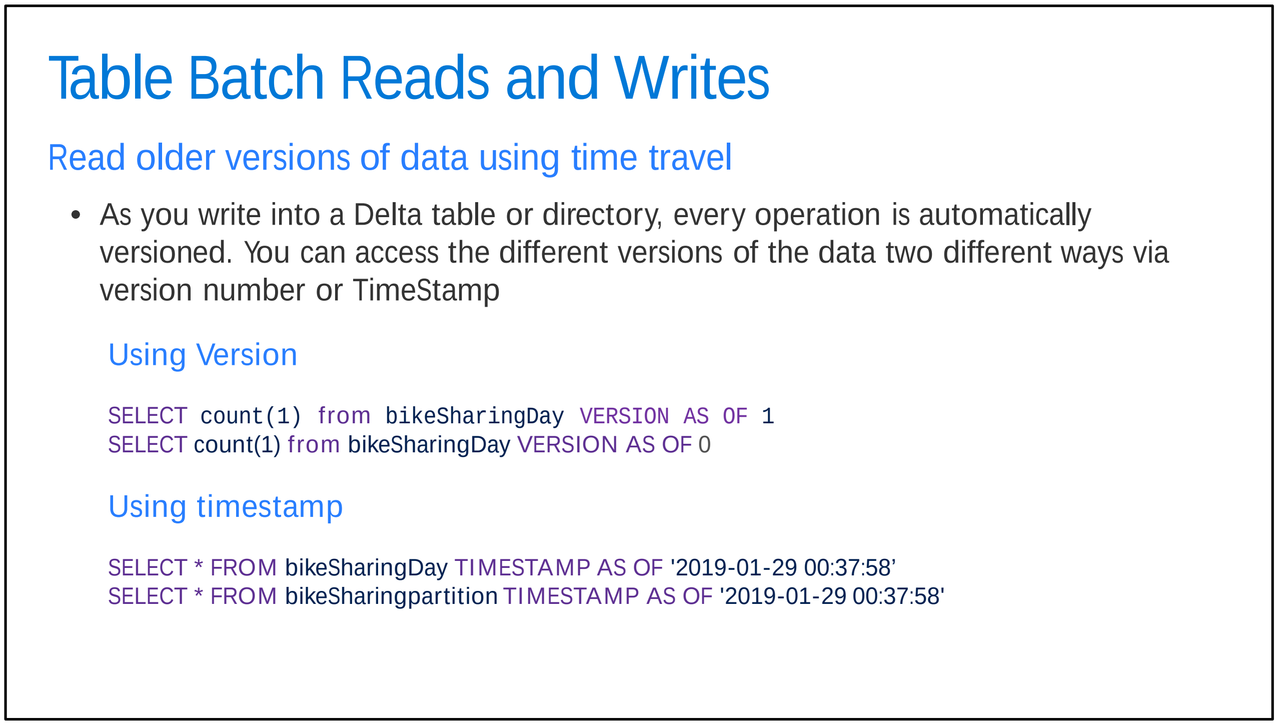


Table Batch Reads and Writes

Read older versions of data using time travel

- As you write into a Delta table or directory, every operation is automatically versioned. You can access the different versions of the data two different ways via version number or TimeStamp

Using Version

```
SELECT count(1) from bikeSharingDay VERSION AS OF 1
SELECT count(1) from bikeSharingDay VERSION AS OF 0
```

Using timestamp

```
SELECT * FROM bikeSharingDay TIMESTAMP AS OF '2019-01-29 00:37:58'
SELECT * FROM bikeSharingpartition TIMESTAMP AS OF '2019-01-29 00:37:58'
```

Introduction to time travel
https://databricks.com/blog/2019/02/04/introducing-delta-time-travel-for-large-scale-data-lakes.html

- timestamp_expression can be any one of:
  - '2018-10-18T22:15:12.013Z', that is, a string that can be cast to a timestamp
  - cast('2018-10-18 13:36:32 CEST' as timestamp)
  - '2018-10-18', that is, a date string
  - current_timestamp() - interval 12 hours
  - date_sub(current_date(), 1)
  - Any other expression that is or can be cast to a timestamp
- version is a long value that can be obtained from the output of DESCRIBE HISTORY table_spec.

Neither timestamp_expression nor version can be subqueries.

Data retention

- To time travel, you must retain both the log and the data files for that version
  - The data files are never deleted automatically. You need to run VACUUM for this to happen. Vacuum does not delete log files. Log files are automatically cleaned up after checkpoints are written
  - By default you can time travel up to 30 days unless you have
    - Run Vacuum
    - Changed the data or log retention periods by using the following table.properties
      - delta.logRetentionDuration = "interval <interval>" (default 30 days)
      - delta.deletedFileRetentionDuration = "interval <interval>" (default 7 days)

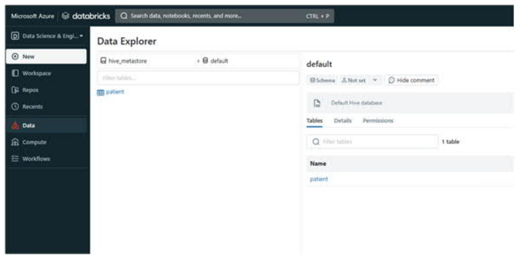
    In order to have 30 days full history, you'd have to have delta.deletedFileRetentionDuration set to "interval 30 days"

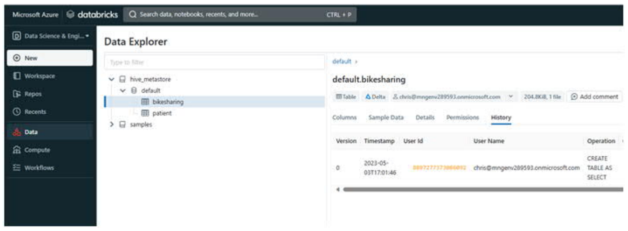
Table Batch Reads and Writes

Read older versions of data using time travel

You can look at the history of table changes using the DESCRIBE HISTORY command or through the UI – the Data Explorer (classic schema browser)

List change history on the table
DESCRIBE HISTORY `bikeSharingDay`





Version	Timestamp	User Id	User Name	Operation
0	2023-05-03T17:01:46	8897277373866692	chris@mngenv289593.onmicrosoft.com	CREATE TABLE AS SELECT

Introduction to time travel
https://databricks.com/blog/2019/02/04/introducing-delta-time-travel-for-large-scale-data-lakes.html

Be aware, if you wanted to demo the UI, you need to have a warehouse started. (you switch view to SQL, then SQL Warehouses and then either start the starter warehouse or create a new one)

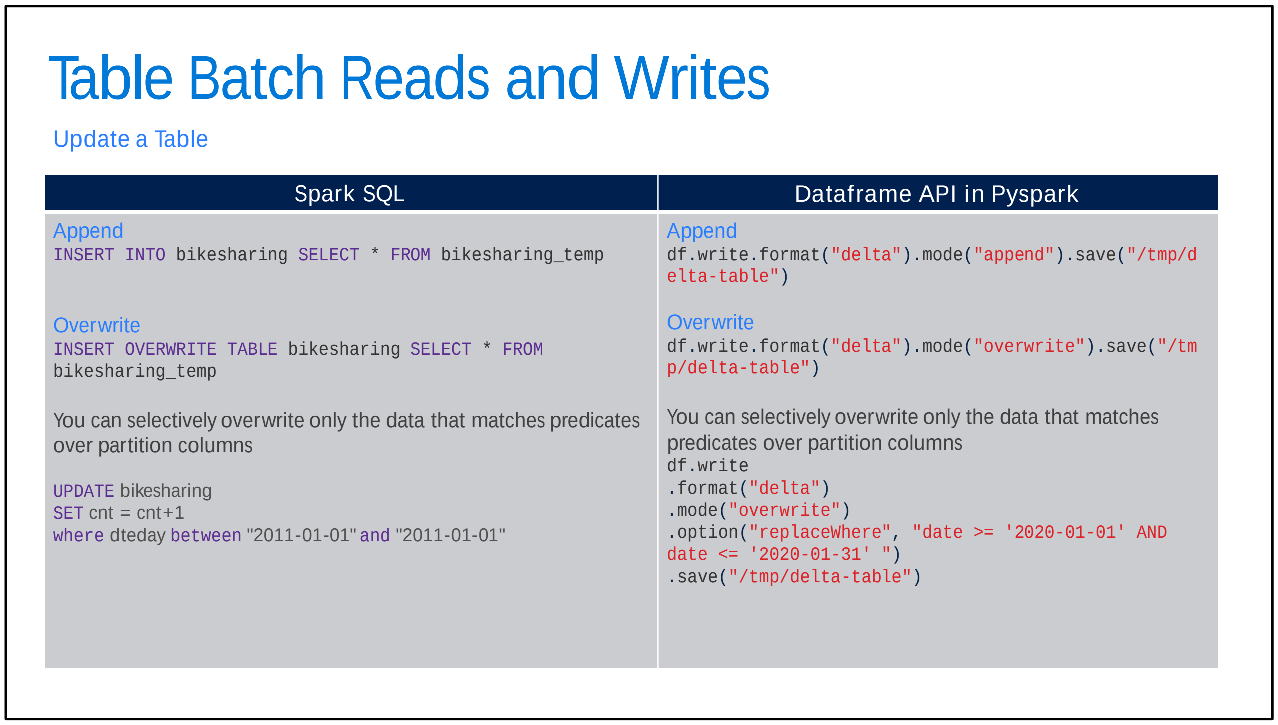


Table Batch Reads and Writes

Update a Table

Spark SQL	Dataframe API in Pyspark
Append `INSERT INTO bikesharing SELECT * FROM bikesharing_temp`	Append `df.write.format("delta").mode("append").save("/tmp/delta-table")`
Overwrite `INSERT OVERWRITE TABLE bikesharing SELECT * FROM bikesharing_temp`	Overwrite `df.write.format("delta").mode("overwrite").save("/tmp/delta-table")`

You can selectively overwrite only the data that matches predicates over partition columns
 	
`UPDATE bikesharing`
`SET cnt = cnt+1`
`where dteday between "2011-01-01" and "2011-01-01"`
	You can selectively overwrite only the data that matches predicates over partition columns
	`df.write`
	`.format("delta")`
	`.mode("overwrite")`
	`.option("replaceWhere", "date >= '2020-01-01' AND date <= '2020-01-31' ")`
	`.save("/tmp/delta-table")`

https://docs.delta.io/latest/delta-batch.html

Table Batch Reads and Writes

Update a Table – Fix incorrect updates

Spark SQL	Dataframe API in Pyspark

Fix the cnt column and increment it by one where date falls between "2011-01-01" and "2011-01-01“
 	
`UPDATE bikesharing`
`SET cnt = cnt+1`
`where dteday between "2011-01-01" and "2011-01-01"`
	You can update data that matches a predicate in a Delta table. For example, to fix a spelling mistake in the eventType, you can run the following:
	
	`from delta.tables import *`
	`from pyspark.sql.functions import *`
	`deltaTable = DeltaTable.forPath(spark, "/data/events/")`
	`deltaTable.update("eventType = 'clck'", { "eventType": "'click'" } ) # predicate using SQL formatted string`
	`/*`
	`deltaTable.update(col("eventType") == "clck", { "eventType": lit("click") } ) # predicate using Spark SQL functions`
	`*/`

https://docs.delta.io/latest/delta-batch.html

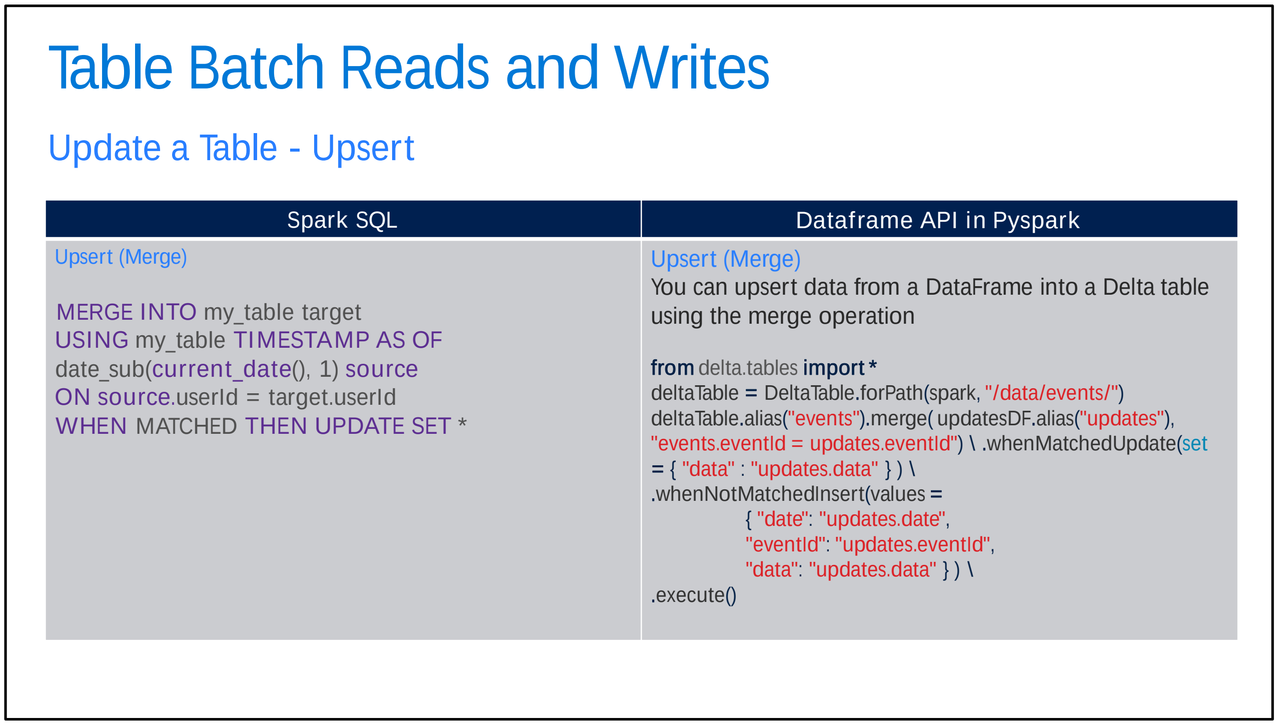


Table Batch Reads and Writes

Update a Table - Upsert

Spark SQL	Dataframe API in Pyspark
Upsert (Merge) MERGE INTO my_table target USING my_table TIMESTAMP AS OF date_sub(current_date(), 1) source ON source.userId = target.userId WHEN MATCHED THEN UPDATE SET *	Upsert (Merge) You can upsert data from a DataFrame into a Delta table using the merge operation from delta.tables import * deltaTable = DeltaTable.forPath(spark, "/data/events/") deltaTable.alias("events").merge(updatesDF.alias("updates"), "events.eventId = updates.eventId") \ .whenMatchedUpdate(set = { "data" : "updates.data" }) \ .whenNotMatchedInsert(values = { "date": "updates.date", "eventId": "updates.eventId", "data": "updates.data" }) \ .execute()

https://docs.delta.io/latest/delta-update.html#-delta-update

https://docs.databricks.com/delta/delta-batch.html#language-sql

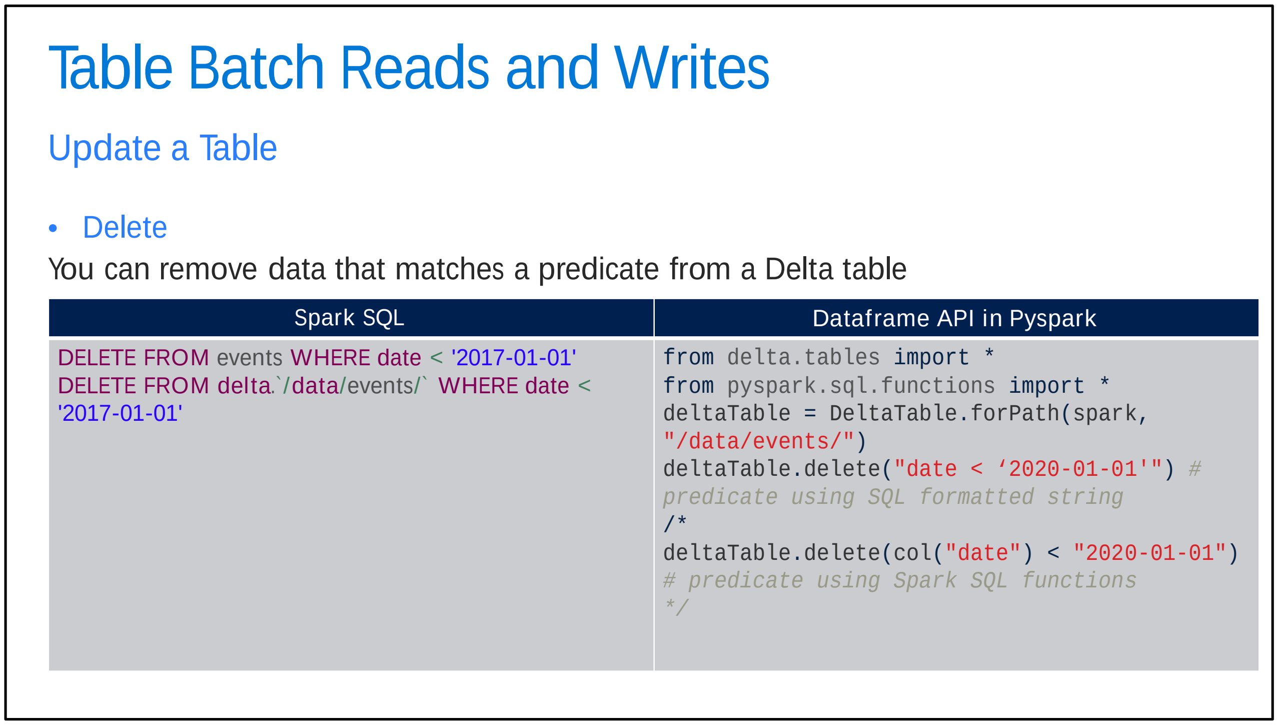


Table Batch Reads and Writes

Update a Table

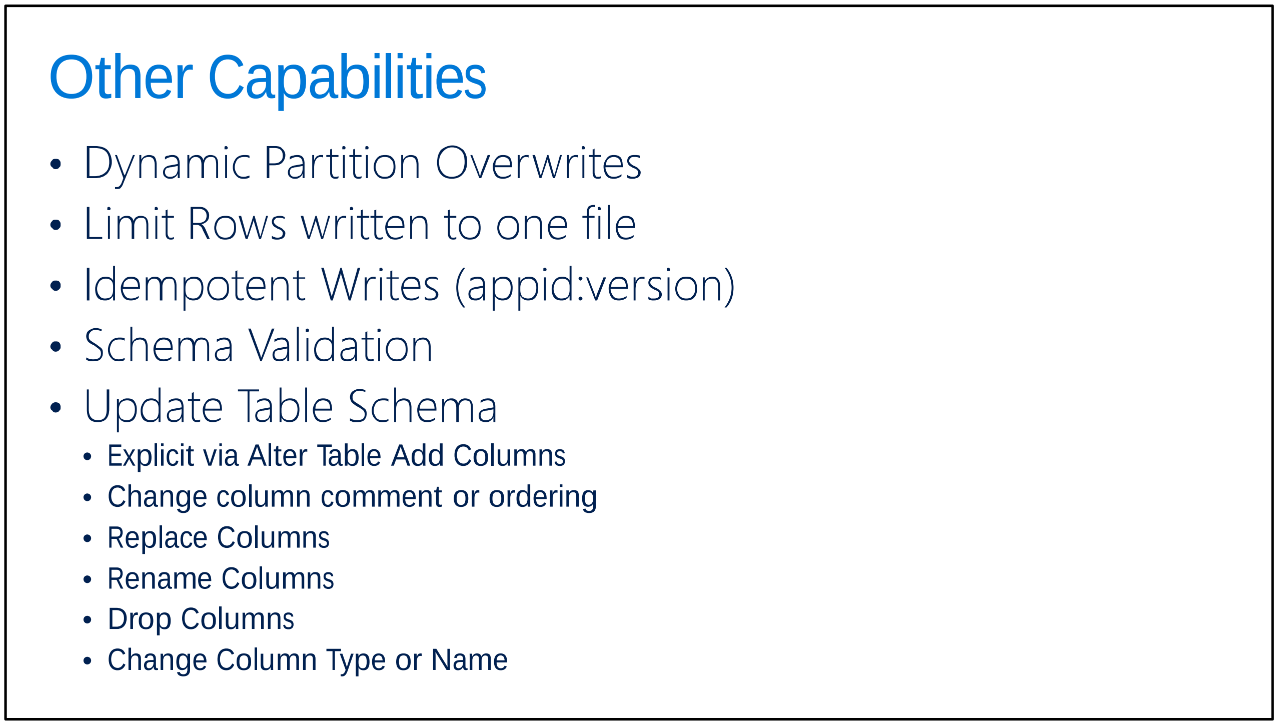
- Delete

You can remove data that matches a predicate from a Delta table

Spark SQL	Dataframe API in Pyspark

`DELETE FROM events WHERE date < '2017-01-01'`
`DELETE FROM delta.`/data/events/` WHERE date < '2017-01-01'`
	`from delta.tables import *`
	`from pyspark.sql.functions import *`
	`deltaTable = DeltaTable.forPath(spark, "/data/events/")`
	`deltaTable.delete("date < ‘2020-01-01'") # predicate using SQL formatted string`
	`/*`
	`deltaTable.delete(col("date") < "2020-01-01") # predicate using Spark SQL functions`
	`*/`

https://docs.delta.io/latest/delta-update.html#-delta-update



Other Capabilities

- Dynamic Partition Overwrites
- Limit Rows written to one file
- Idempotent Writes (appid:version)
- Schema Validation
- Update Table Schema
  - Explicit via Alter Table Add Columns
  - Change column comment or ordering
  - Replace Columns
  - Rename Columns
  - Drop Columns
  - Change Column Type or Name

- Dynamic partition overwrites

•When in dynamic partition overwrite mode, we overwrite all existing data in each logical partition for which the write will commit new data. Any existing logical partitions for which the write does not contain data will remain unchanged. This mode is only applicable when data is being written in overwrite mode: either INSERT OVERWRITE in SQL, or a DataFrame write with df.write.mode("overwrite").

•Configure dynamic partition overwrite mode by setting the Spark session configuration spark.sql.sources.partitionOverwriteMode to dynamic. You can also enable this by setting the DataFrameWriter option partitionOverwriteMode to dynamic. If present, the query-specific option overrides the mode defined in the session configuration. The default for partitionOverwriteMode is static.

Limit rows written to one file

You can use the SQL session configuration spark.sql.files.maxRecordsPerFile to specify the maximum number of records to write to a single file for a Delta Lake table. Specifying a value of zero or a negative value represents no limit.

You can also use the DataFrameWriter option maxRecordsPerFile when using the DataFrame APIs to write to a Delta Lake table. When maxRecordsPerFile is specified, the value of the SQL session configuration spark.sql.files.maxRecordsPerFile is ignored.

Idempotent writes

Sometimes a job that writes data to a Delta table is restarted due to various reasons (for example, job encounters a failure). The failed job may or may not have written the data to Delta table before terminating. In the case where the data is written to the Delta table, the restarted job writes the same data to the Delta table which results in duplicate data.
To address this, Delta tables support the following DataFrameWriter options to make the writes idempotent:

- txnAppId: A unique string that you can pass on each DataFrame write. For example, this can be the name of the job.
- txnVersion: A monotonically increasing number that acts as transaction version. This number needs to be unique for data that is being written to the Delta table(s). For example, this can be the epoch seconds of the instant when the query is attempted for the first time. Any subsequent restarts of the same job needs to have the same value for txnVersion.

The above combination of options needs to be unique for each new data that is being ingested into the Delta table and the txnVersion needs to be higher than the last data that was ingested into the Delta table. For example:

- Last successfully written data contains option values as dailyETL:23423 (txnAppId:txnVersion).
- Next write of data should have txnAppId = dailyETL and txnVersion as at least 23424 (one more than the last written data txnVersion).
- Any attempt to write data with txnAppId = dailyETL and txnVersion as 23422 or less is ignored because the txnVersion is less than the last recorded txnVersion in the table.
- Attempt to write data with txnAppId:txnVersion as anotherETL:23424 is successful writing data to the table as it contains a different txnAppId compared to the same option value in last ingested data.

You can also configure idempotent writes by setting the Spark session configuration spark.databricks.delta.write.txnAppId and spark.databricks.delta.write.txnVersion. In addition, you can set spark.databricks.delta.write.txnVersion.autoReset.enabled to true to automatically reset spark.databricks.delta.write.txnVersion after every write. When both the writer options and session configuration are set, we will use the writer option values.
Warning
This solution assumes that the data being written to Delta table(s) in multiple retries of the job is same. If a write attempt in a Delta table succeeds but due to some downstream failure there is a second write attempt with same txn options but different data, then that second write attempt will be ignored. This can cause unexpected results.

Schema validation

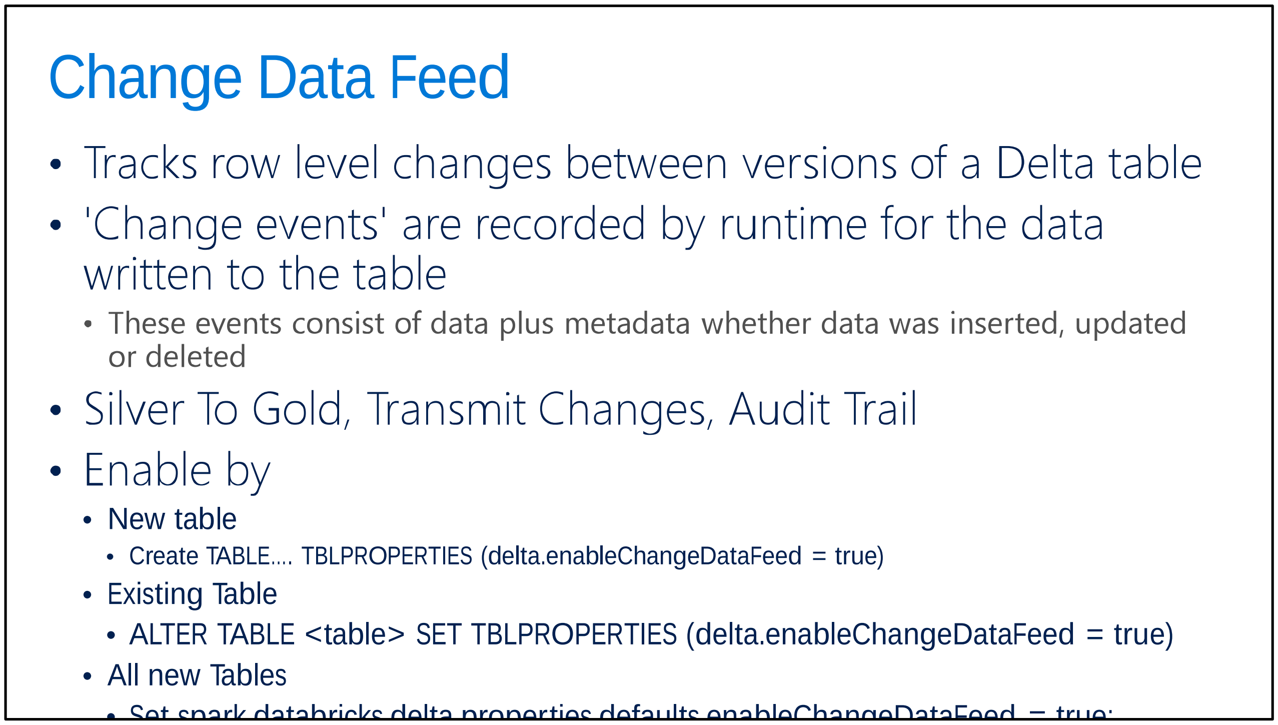
Delta Lake automatically validates that the schema of the DataFrame being written is compatible with the schema of the table. Delta Lake uses the following rules to determine whether a write from a DataFrame to a table is compatible:

- All DataFrame columns must exist in the target table. If there are columns in the

DataFrame not present in the table, an exception is raised. Columns present in the table but not in the DataFrame are set to null.
- DataFrame column data types must match the column data types in the target table. If they don’t match, an exception is raised.
- DataFrame column names cannot differ only by case. This means that you cannot have columns such as “Foo” and “foo” defined in the same table. While you can use Spark in case sensitive or insensitive (default) mode, Parquet is case sensitive when storing and returning column information. Delta Lake is case-preserving but insensitive when storing the schema and has this restriction to avoid potential mistakes, data corruption, or loss issues.

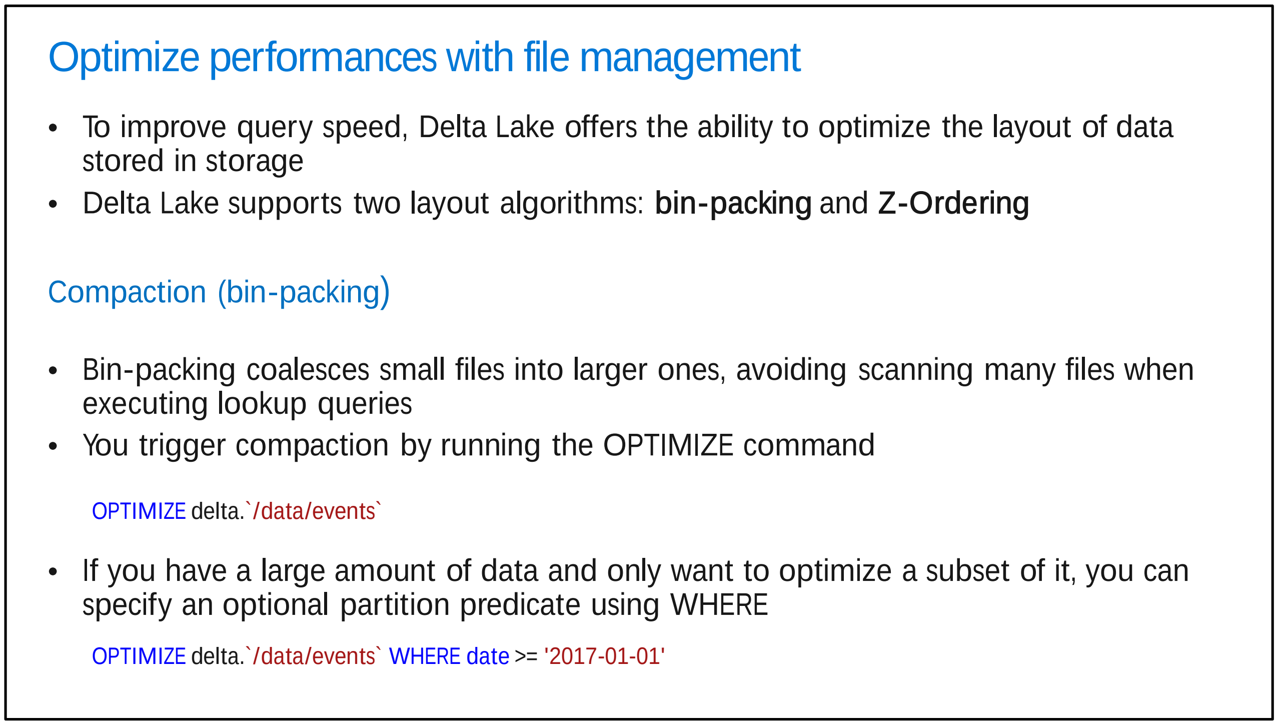
Delta Lake support DDL to add new columns explicitly and the ability to update schema automatically.
If you specify other options, such as partitionBy, in combination with append mode, Delta Lake validates that they match and throws an error for any mismatch. When partitionBy is not present, appends automatically follow the partitioning of the existing data.

- Update Table Schema
- https://docs.delta.io/2.3.0/delta-batch.html#id25


Change Data Feed

- Tracks row level changes between versions of a Delta table
- 'Change events' are recorded by runtime for the data written to the table
  - These events consist of data plus metadata whether data was inserted, updated or deleted
- Silver To Gold, Transmit Changes, Audit Trail
- Enable by
  - New table
    - Create TABLE.... TBLPROPERTIES (delta.enableChangeDataFeed = true)
  - Existing Table
    - ALTER TABLE <table> SET TBLPROPERTIES (delta.enableChangeDataFeed = true)
  - All new Tables
    - Set spark.databricks.delta.properties.defaults.enableChangeDataFeed = true;

See
https://docs.delta.io/2.3.0/delta-change-data-feed.html#change-data-feed



Optimize performances with file management

- To improve query speed, Delta Lake offers the ability to optimize the layout of data stored in storage
- Delta Lake supports two layout algorithms: **bin-packing** and **Z-Ordering**

Compaction (bin-packing)

- Bin-packing coalesces small files into larger ones, avoiding scanning many files when executing lookup queries
- You trigger compaction by running the OPTIMIZE command

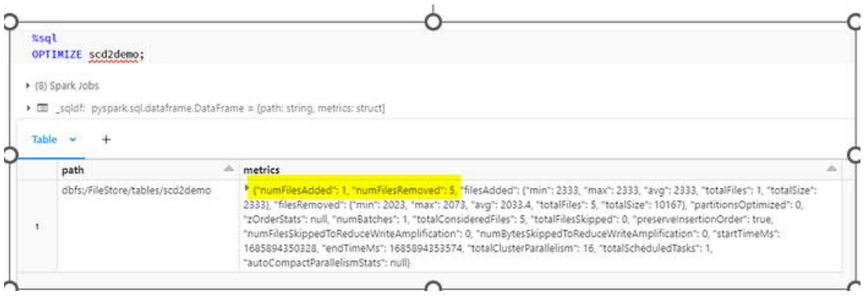
```
OPTIMIZE delta.`/data/events`
```

- If you have a large amount of data and only want to optimize a subset of it, you can specify an optional partition predicate using WHERE

```
OPTIMIZE delta.`/data/events` WHERE date >= '2017-01-01'
```

https://databricks.com/blog/2018/07/31/processing-petabytes-of-data-in-seconds-with-databricks-delta.html

OPTIMIZE



```
%sql
OPTIMIZE scd2demo;
```

```
▸ (8) Spark Jobs
▸ _sqldf: pyspark.sql.dataframe.DataFrame = [path: string, metrics: struct]
```

Table

	path	metrics
1	dbfs:/FileStore/tables/scd2demo	{"numFilesAdded": 1, "numFilesRemoved": 5, "filesAdded": {"min": 2333, "max": 2333, "avg": 2333, "totalFiles": 1, "totalSize": 2333}, "filesRemoved": {"min": 2023, "max": 2073, "avg": 2033.4, "totalFiles": 5, "totalSize": 10167}, "partitionsOptimized": 0, "zOrderStats": null, "numBatches": 1, "totalConsideredFiles": 5, "totalFilesSkipped": 0, "preserveInsertionOrder": true, "numFilesSkippedToReduceWriteAmplification": 0, "numBytesSkippedToReduceWriteAmplification": 0, "startTimeMs": 1685894350328, "endTimeMs": 1685894353574, "totalClusterParallelism": 16, "totalScheduledTasks": 1, "autoCompactParallelismStats": null}

Optimize performances with file management- Continued

Data skipping

- As new data is inserted into a Delta table, file-level min/max statistics are collected for all columns of supported types
- When there's a lookup query against the table, Delta table first consults these statistics in order to determine which files can safely be skipped
- You do not need to configure data skipping - the feature is activated whenever applicable. However, its effectiveness depends on the layout of your data. For best results, apply Z-Ordering.

https://databricks.com/blog/2018/07/31/processing-petabytes-of-data-in-seconds-with-databricks-delta.html

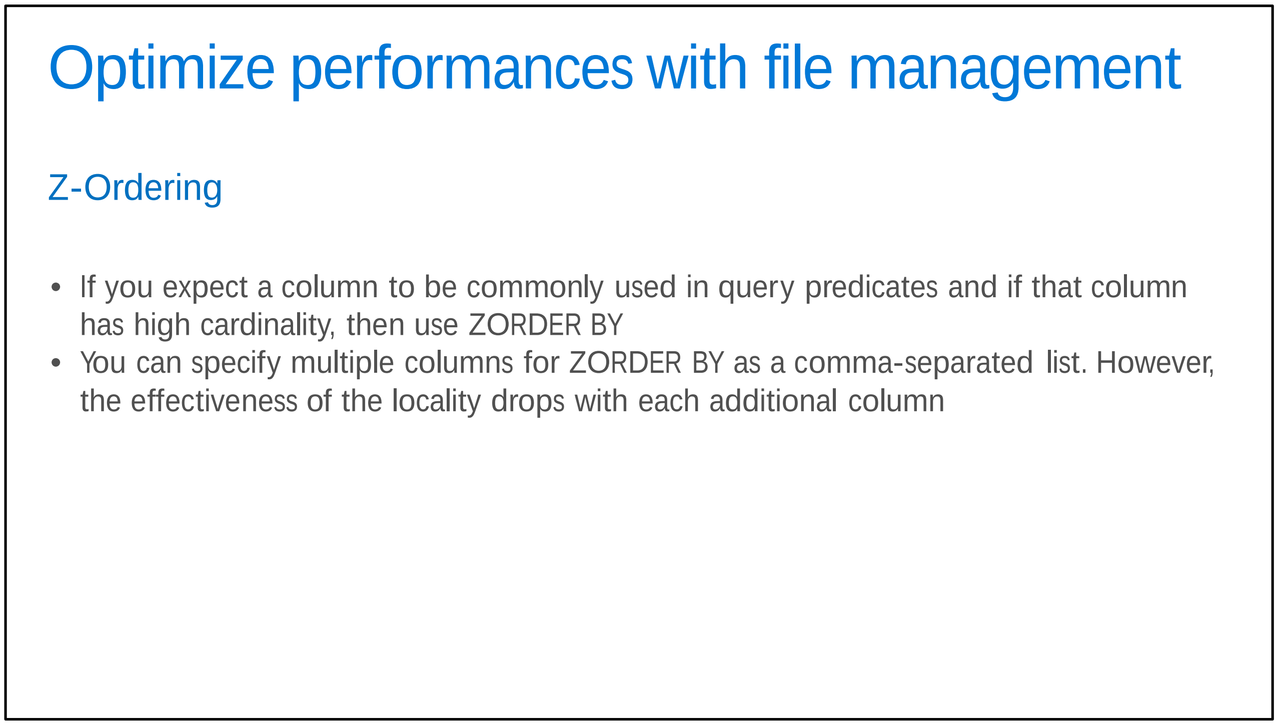
Optimize performances with file management

Z-Ordering

- Z-Ordering is a technique to colocate related information in the same set of files
- This co-locality is automatically used by data-skipping algorithms to dramatically reduce the amount of data that needs to be read
- To Z-Order data, you specify the columns to order on in the ZORDER BY clause:

```
%sql
OPTIMIZE events WHERE date >= current_timestamp() - INTERVAL 1 day
ZORDER BY (eventType)
```

https://databricks.com/blog/2018/07/31/processing-petabytes-of-data-in-seconds-with-databricks-delta.html

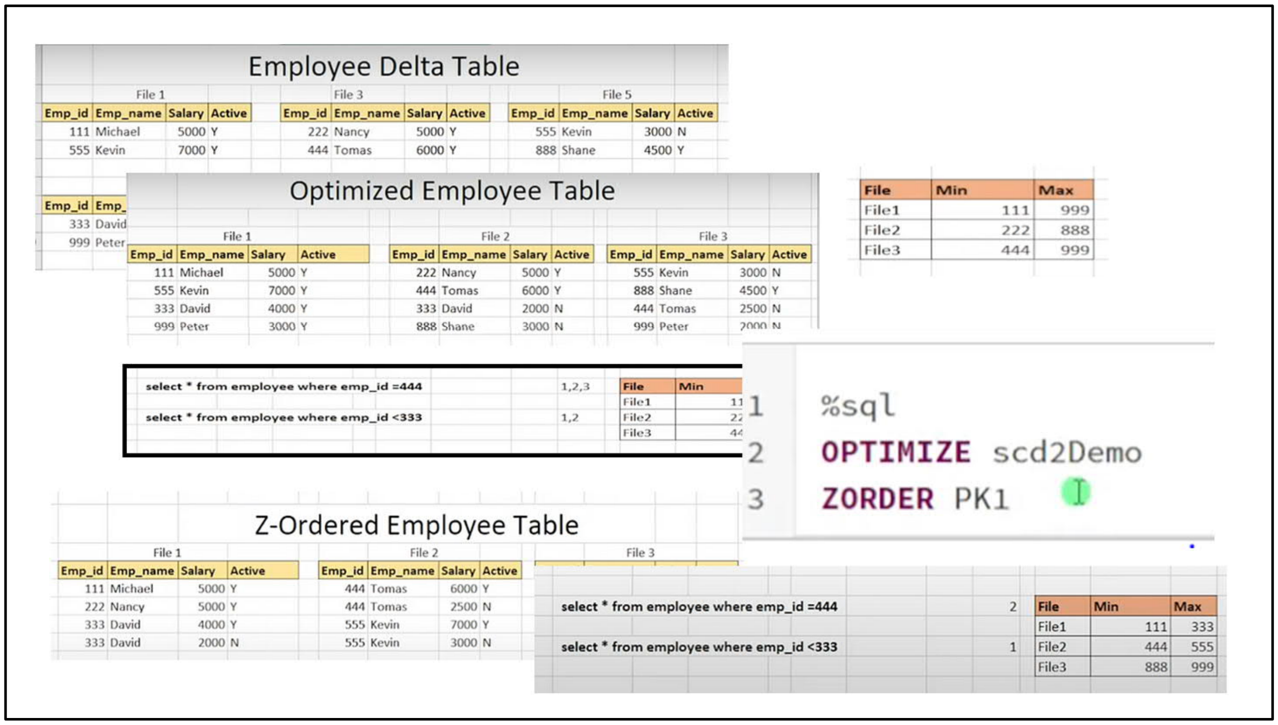


Optimize performances with file management

Z-Ordering

- If you expect a column to be commonly used in query predicates and if that column has high cardinality, then use ZORDER BY
- You can specify multiple columns for ZORDER BY as a comma-separated list. However, the effectiveness of the locality drops with each additional column

https://databricks.com/blog/2018/07/31/processing-petabytes-of-data-in-seconds-with-databricks-delta.html



Employee Delta Table

File 1				File 3				File 5			
Emp_id	Emp_name	Salary	Active	Emp_id	Emp_name	Salary	Active	Emp_id	Emp_name	Salary	Active
111	Michael	5000	Y	222	Nancy	5000	Y	555	Kevin	3000	N
555	Kevin	7000	Y	444	Tomas	6000	Y	888	Shane	4500	Y

Emp_id	Emp_
333	David
999	Peter

File	Min	Max
File1	111	999
File2	222	888
File3	444	999

Optimized Employee Table

File 1				File 2				File 3			
Emp_id	Emp_name	Salary	Active	Emp_id	Emp_name	Salary	Active	Emp_id	Emp_name	Salary	Active
111	Michael	5000	Y	222	Nancy	5000	Y	555	Kevin	3000	N
555	Kevin	7000	Y	444	Tomas	6000	Y	888	Shane	4500	Y
333	David	4000	Y	333	David	2000	N	444	Tomas	2500	N
999	Peter	3000	Y	888	Shane	3000	N	999	Peter	2000	N

		File	Min
select * from employee where emp_id =444	1,2,3	File1	11
select * from employee where emp_id <333	1,2	File2	22
		File3	44

```
%sql
OPTIMIZE scd2Demo
ZORDER PK1
```

Z-Ordered Employee Table

File 1				File 2				File 3
Emp_id	Emp_name	Salary	Active	Emp_id	Emp_name	Salary	Active	
111	Michael	5000	Y	444	Tomas	6000	Y	
222	Nancy	5000	Y	444	Tomas	2500	N	
333	David	4000	Y	555	Kevin	7000	Y	
333	David	2000	N	555	Kevin	3000	N	

		File	Min	Max
select * from employee where emp_id =444	2	File1	111	333
select * from employee where emp_id <333	1	File2	444	555
		File3	888	999

https://medium.com/@amarkrgupta96/unlocking-performance-optimize-vacuum-and-z-ordering-in-databricks-delta-tables

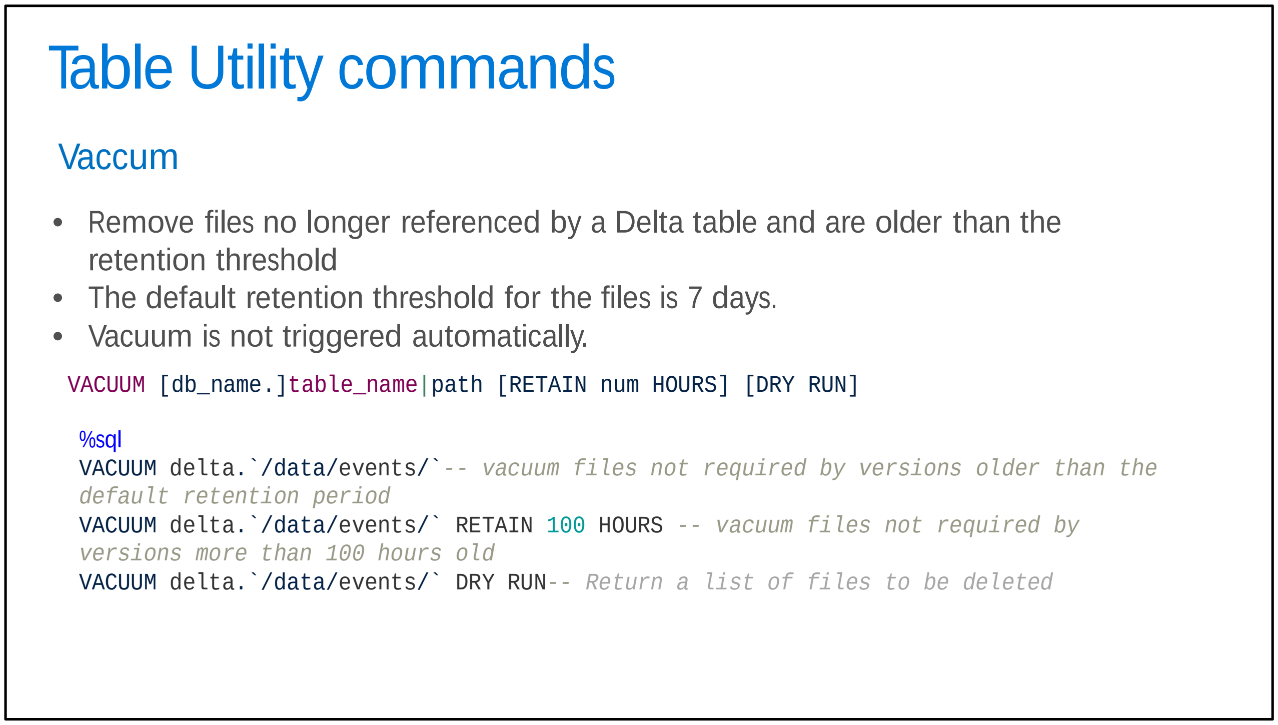


Table Utility commands

Vaccum

- Remove files no longer referenced by a Delta table and are older than the retention threshold
- The default retention threshold for the files is 7 days.
- Vacuum is not triggered automatically.

```
VACUUM [db_name.]table_name|path [RETAIN num HOURS] [DRY RUN]

 %sql
 VACUUM delta.`/data/events/`-- vacuum files not required by versions older than the
 default retention period
 VACUUM delta.`/data/events/` RETAIN 100 HOURS -- vacuum files not required by
 versions more than 100 hours old
 VACUUM delta.`/data/events/` DRY RUN-- Return a list of files to be deleted
```

https://docs.databricks.com/spark/latest/spark-sql/language-manual/vacuum.html

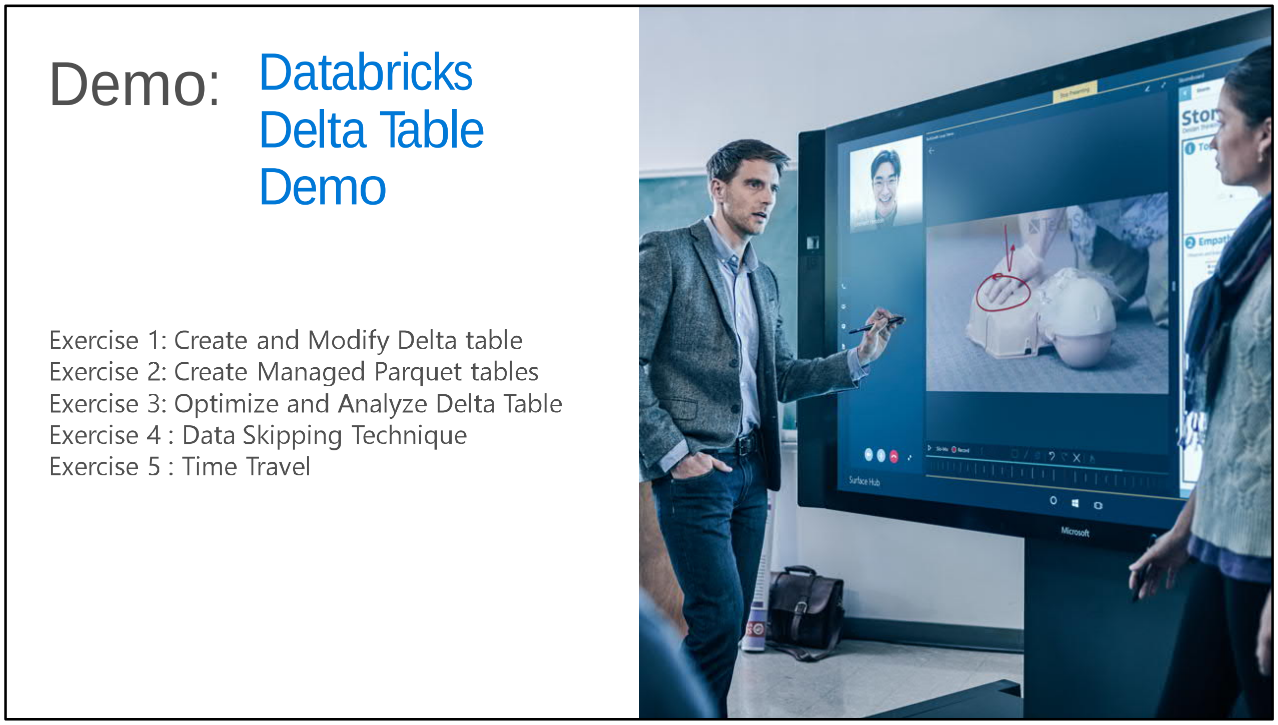
By default, retain hours for vacuum should be greater than 168 hour, which can be altered to a lower number with following setting :

spark.databricks.delta.retentionDurationCheck.enabled to false

Table Utility commands - Contd

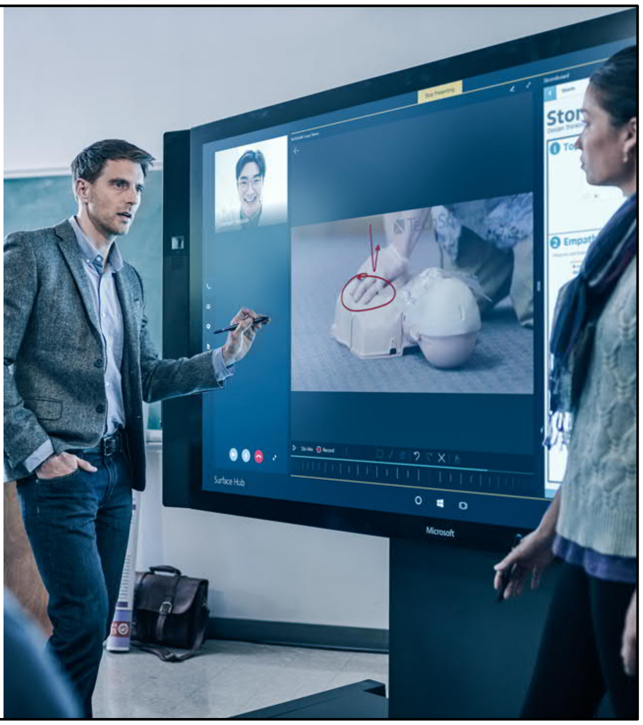
```
DESCRIBE HISTORY '/data/events/' -- get the full history of the table
DESCRIBE HISTORY '/data/events/' LIMIT 1 -- get the last operation only

DESCRIBE DETAIL <DATABASENAME.TABLENAME> -- Show table details

SHOW DATABASES   -- List Databases
SHOW TABLES   -- List tables
SHOW CREATE TABLE <DATABASENAME.TABLENAME> -- Show table creation script

RESTORE TABLE <DATABASENAME.TABLENAME> TO VERSION AS OF <version>

RESTORE TABLE '/data/target/' TO TIMESTAMP AS OF <timestamp>
```

https://docs.delta.io/latest/delta-utility.html

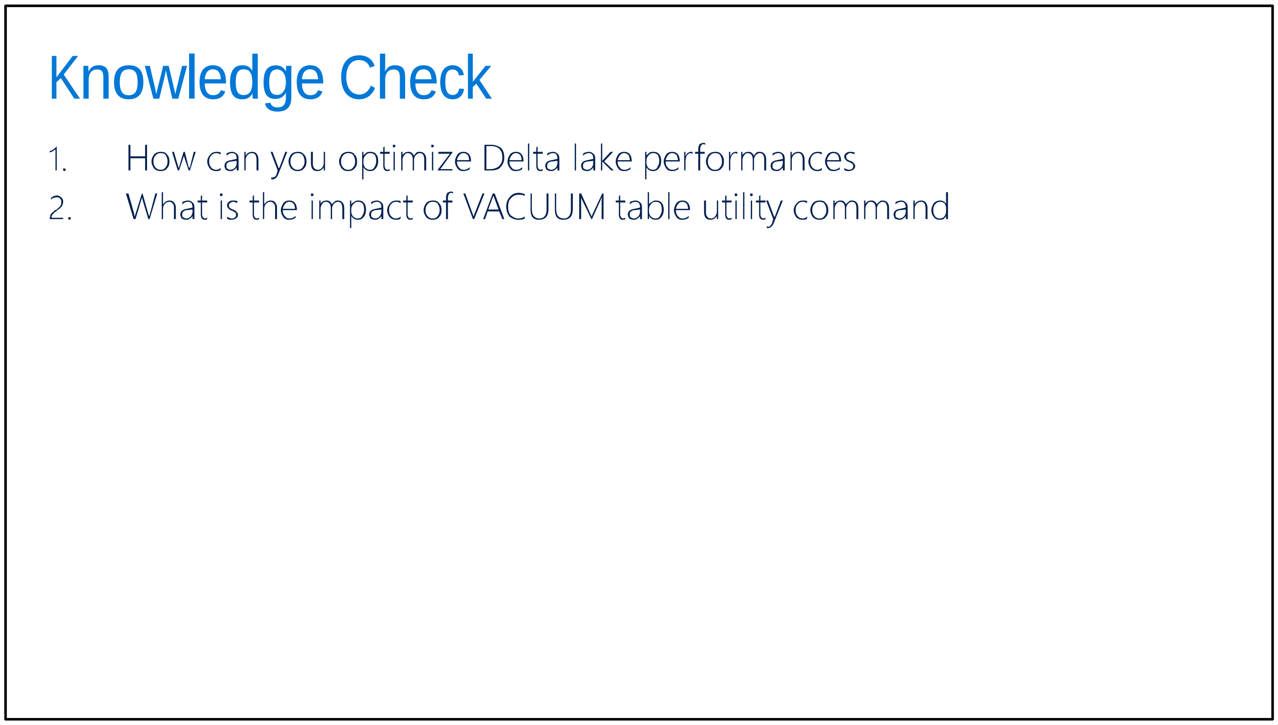


Demo: Databricks Delta Table Demo

Exercise 1: Create and Modify Delta table
Exercise 2: Create Managed Parquet tables
Exercise 3: Optimize and Analyze Delta Table
Exercise 4 : Data Skipping Technique
Exercise 5 : Time Travel



Refer to M03_L02_Demo02.ipynb file for Databricks notebook code. Remember to import the notebook to your workspace and read the directions before executing the code



Knowledge Check

1. How can you optimize Delta lake performances
2. What is the impact of VACUUM table utility command

1. How can you optimize Delta lake performances?

- By using Bin-Packing and Zordering

2. What is the impact of VACUUM table utility command

Recursively vacuum directories associated with the Delta table and remove files that are no longer in the latest state of the transaction log for the table and are older than a retention threshold

Module Summary

- In this module we covered the following:
  - Delta Lake is a storage layer that brings scalable, ACID transactions to Apache Spark and other big-data engines
  - Data Skipping Technique of Delta Table
  - Optimize and Analyze Delta Table
  - Time Traveler capability

.