



# 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*.

# 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
```

# 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

# 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
```



# Datasets

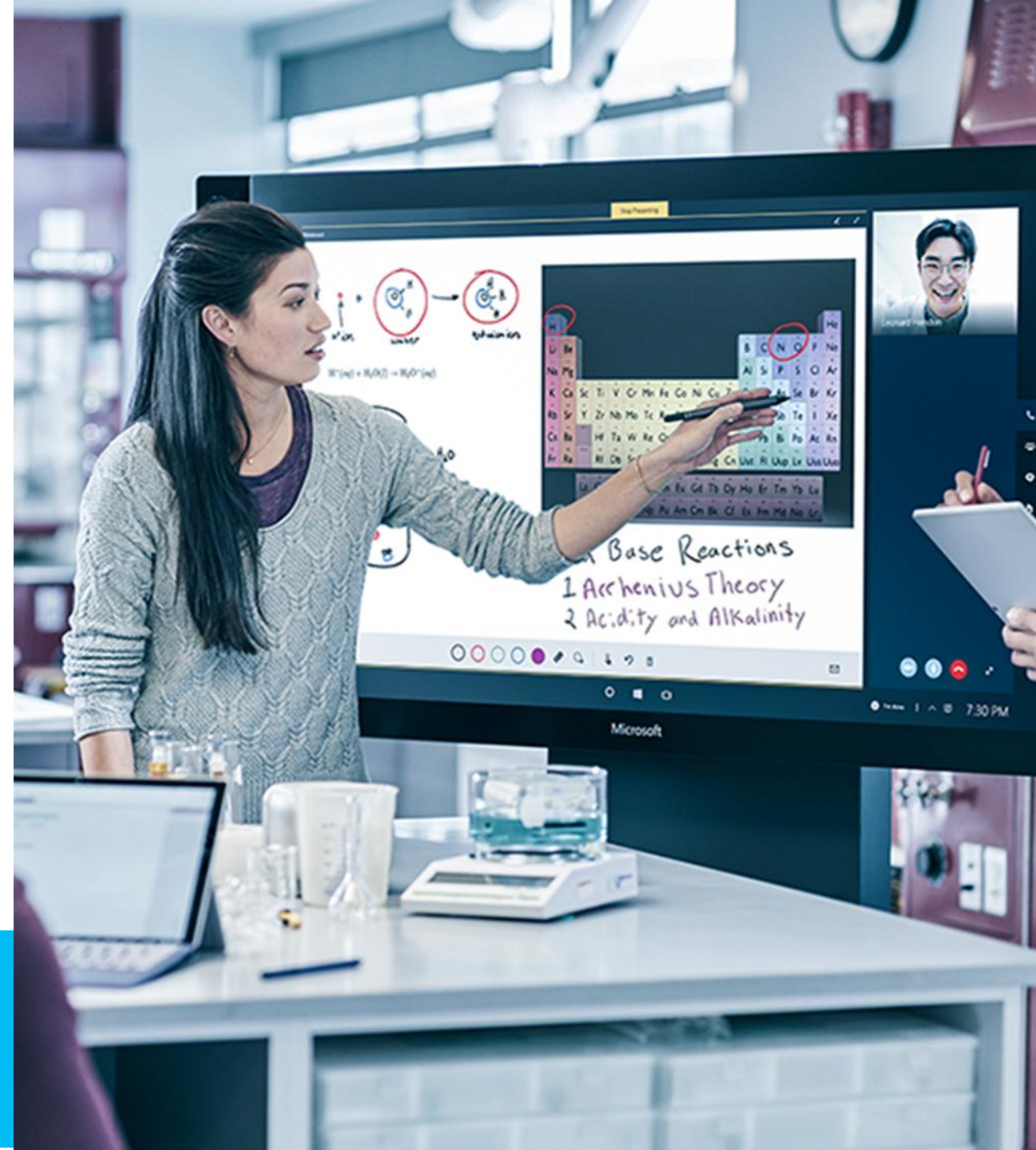
- The Apache Spark Dataset API provides a type-safe, object-oriented programming interface
- **DataFrame** is an alias for an untyped **Dataset [Row]**
- Datasets provide compile-time type safety
- The Dataset API also offers high-level domain-specific language operations
- Datasets are only available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

# Demo: Data Exploration with Azure Databricks



M03 L01 Lab 01  
30 mins

# Lab: Data Exploration with Azure Databricks



M03 L01 Lab 01  
30 mins

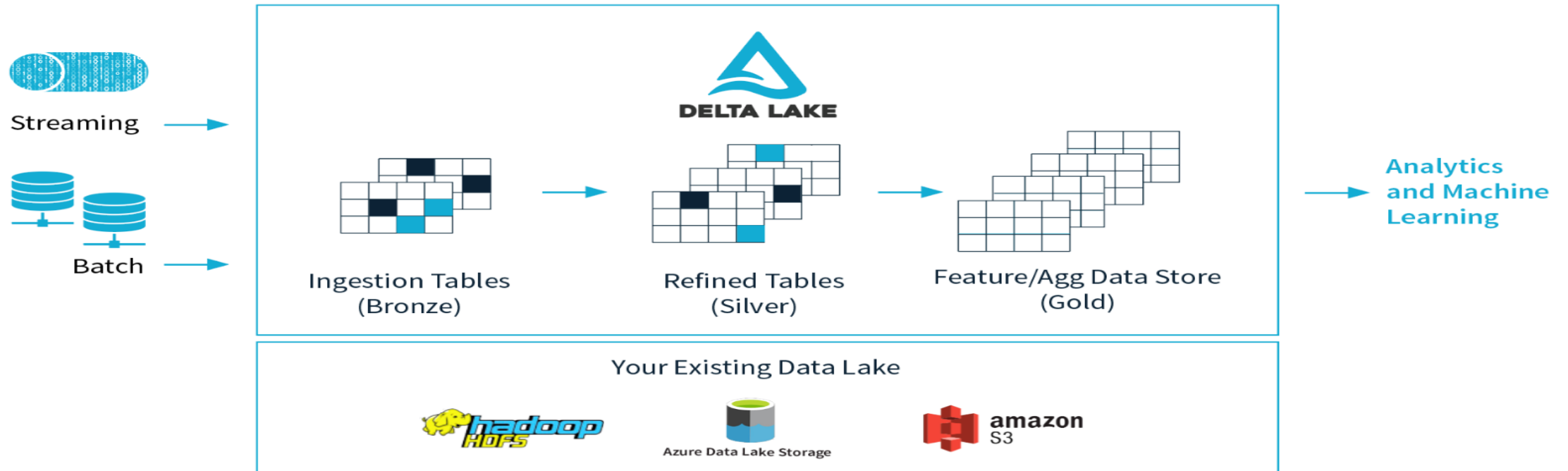
# Lesson 2: Introduction to Delta Lake

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





# 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

# 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



# 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.

Partition Folder layout  
Example(mnth is a partition  
column)

\_delta\_log directory

Location: testload / bikeSharingpartition / \_delta\_log



Search blobs by prefix (case-sensitive)

Name
<input type="checkbox"/>  [..]
<input type="checkbox"/>  __tmp_path_dir
<input type="checkbox"/>  00000000000000000000.crc
<input type="checkbox"/>  00000000000000000000.json

Partition Directory(partition column : mnth)

Location: testload / bikeSharingpartition / mnth=1















Search blobs by prefix (case-sensitive)

Name
<input type="checkbox"/>  [..]
<input type="checkbox"/>  part-00000-fbec9d52-28d2-4b89-a11c-5d0db15e9ffb.c000.snappy.parquet

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





# 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
<p>Create a table</p> <pre>CREATE TABLE bikeSharing USING DELTA LOCATION '/tmp/bikeSharing/'</pre>	<p>Create a table</p> <pre>df = spark.read.format("delta").load("/tmp/delta-table") df.write.format("delta").save("/tmp/delta-table")</pre>
<p>Partition Data</p> <pre>CREATE TABLE bikeSharingpartition USING DELTA PARTITIONED BY (mnth) LOCATION '/tmp/bikeSharing/'</pre>	<p>Partition Data</p> <pre>df = spark.read.format("delta").load("/tmp/delta-table") df.write.format("delta").partitionBy("date").save("/tmp/delta-table")</pre>
<p>Read Data</p> <pre>SELECT * FROM bikeSharing</pre>	<p>Read Data</p> <pre>df = spark.read.format("delta").load("/tmp/delta-table") df.show()</pre>

# 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'
```

# 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

This screenshot shows the Databricks Catalog Explorer interface. The left sidebar displays the navigation menu with options like Workspace, Recents, Catalog, Workflows, Compute, and Marketplace. The main panel shows the 'Catalog Explorer' for the 'hive\_metastore' database, specifically the 'dblab' schema. Under the 'About this schema' section, it indicates the owner is 'Not set'. Below this, a search bar for tables is shown with '5 tables' listed. The table list includes 'bikesharing', 'bikesharingday', 'bikesharingdayoptdel', 'bikesharingdayoptpar', and 'bikesharingpartition'.

This screenshot shows the Databricks Catalog Explorer interface, specifically the 'History' tab for the 'bikesharing' table. The left sidebar is the same as the previous screenshot. The main panel shows the 'History' tab with a table of operations. The table has columns for Version, Timestamp, User Id, Username, Operation, and Operation Parameters. The first row shows a 'CREATE TABLE AS SELECT' operation performed by 'admin@mngenvmcap569188...' on 'Mar 14, 2025, 03:08 PM'.

Version	Timestamp	User Id	Username	Operation	Operation Parameters
0	Mar 14, 2025, 03:08 PM	7663973307676980	admin@mngenvmcap569188...	CREATE TABLE AS SELECT	[...] // 4 items

# Table Batch Reads and Writes

## Update a Table

### Spark SQL

#### Append

```
INSERT INTO bikesharing SELECT * FROM bikesharing_temp
```

#### Overwrite

```
INSERT OVERWRITE TABLE bikesharing SELECT * FROM  
bikesharing_temp
```

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"
```

### Dataframe API in Pyspark

#### Append

```
df.write.format("delta").mode("append").save("/tmp/d  
elta-table")
```

#### Overwrite

```
df.write.format("delta").mode("overwrite").save("/tm  
p/delta-table")
```

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")
```

# Table Batch Reads and Writes

## Update a Table – Fix incorrect updates

Spark SQL	Dataframe API in Pyspark
<p>Fix the cnt column and increment it by one where date falls between "2011-01-01" <b>and</b> "2011-01-01"</p> <pre><b>UPDATE</b> bikesharing <b>SET</b> cnt = cnt+1 <b>where</b> dteday <b>between</b> "2011-01-01" <b>and</b> "2011-01-01"</pre>	<p>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:</p> <pre>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 */</pre>

# Table Batch Reads and Writes

## Update a Table - Upsert

Spark SQL	Dataframe API in Pyspark
<p>Upsert (Merge)</p> <pre><b>MERGE INTO</b> my_table target <b>USING</b> my_table <b>TIMESTAMP AS OF</b> date_sub(<b>current_date</b>(), 1) <b>source</b> <b>ON source</b>.userId = target.userId <b>WHEN MATCHED THEN UPDATE SET</b> *</pre>	<p>Upsert (Merge)</p> <p>You can upsert data from a DataFrame into a Delta table using the merge operation</p> <pre>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()</pre>



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

# 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

# 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:

# 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'
```

# 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](#).

# 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)
```

# 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

# 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*



# 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>
```

# 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



# Lab:

## Databricks Delta Table Demo

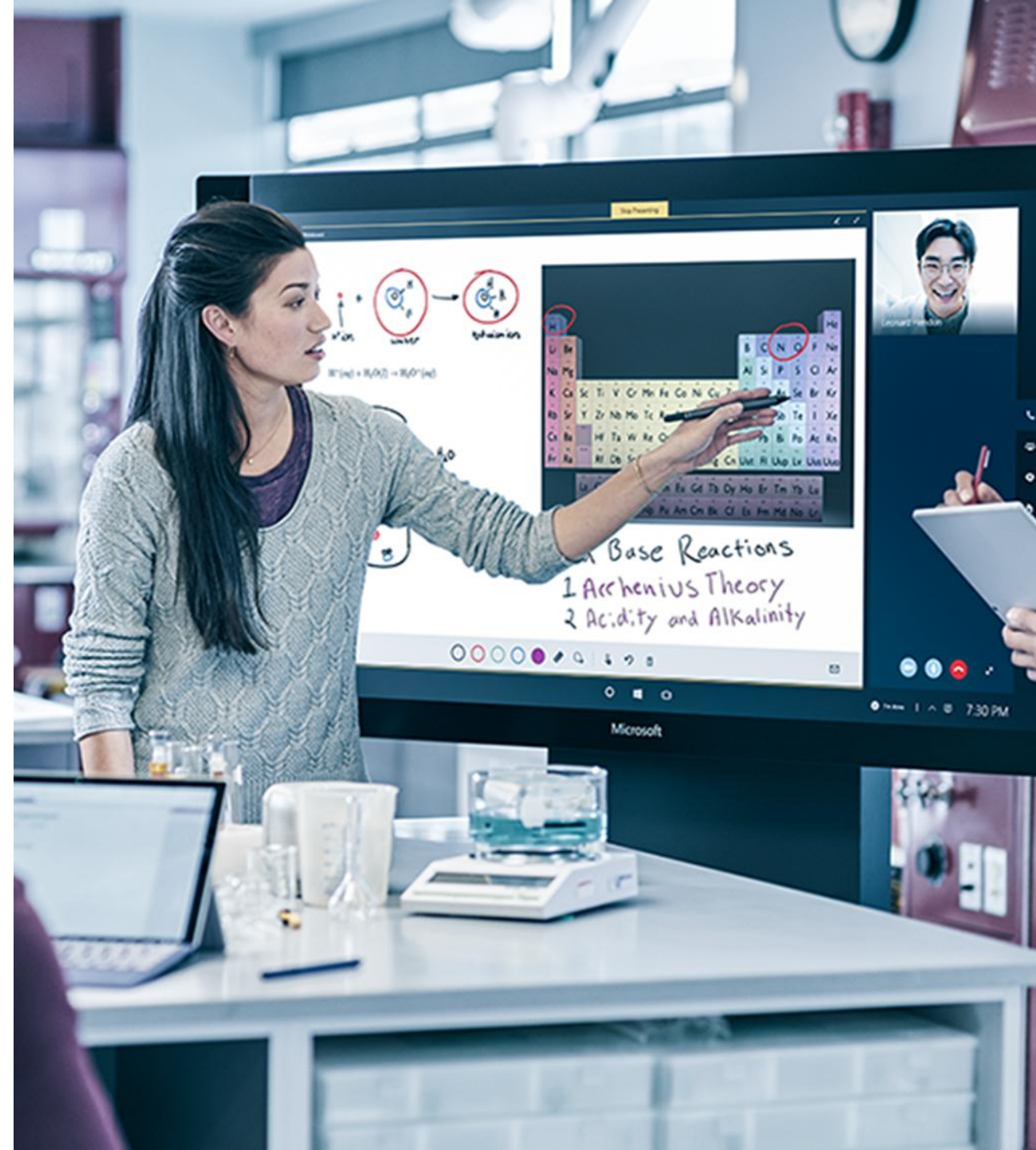
Exercise 1: Create and Modify Delta table

Exercise 2: Create Managed Parquet tables

Exercise 3: Optimize and Analyze Delta Table

Exercise 4 : Data Skipping Technique

Exercise 5 : Time Travel



# Knowledge Check

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

# 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

