## ➢ Spark :

**-Software Needed:**

1. Databricks Community Edition / Azure Databricks Free Trial

2. Microsoft Azure free trial account

**Installation:**

Github Login (Optional) for login

***Note:**

To execute code you need Notebook Labs, you can download from the below links:

https://github.com/wagh-pradeep/databricks.git

----------Day 2----------

**--Q.1 cloud resources in azure**

Azure offers a wide range of cloud resources, including:

1. Virtual Machines: Create and manage virtual machines running on Windows or Linux.

2. Storage: Create and manage storage solutions, including blobs, files, queues, and Tables.

3. Networking: Create and manage virtual networks, load balancers, and VPN gateways.

4. Containers: Create and manage Kubernetes clusters and Docker containers.

5. Security: Create and manage identity and access management, security policies and secrets, and more.

6. Databases: Create and manage Azure SQL, Azure Database for MySQL, Azure Database for PostgreSQL, and CosmosDB.

7. Analytics: Create and manage data pipelines, data lakes, and analytics services.

8. AI/ML: Create and manage AI apps and services, including Cognitive Services and Machine Learning.

9. Serverless: Create and manage functions, Logic Apps, Event Grid, and other serverless services.

**--Q.2 Container in azure**

A container in Azure is a type of virtualization technology that can be used to deploy and manage applications.

Containers are self-contained, lightweight, and portable units of software that can be quickly and easily deployed in Azure.

Containers provide a way to package and isolate applications, making it easier to manage and deploy applications across multiple environments. They can also be used to deploy microservices and distributed applications.

**--Q.3 What is gen2 data lake in azure**

Azure Data Lake Gen2 is a set of capabilities in Azure Data Lake Storage that combines the best of both the blob storage and file system worlds. It extends the existing capabilities of Azure Data Lake Storage Gen1, a hyper-scale repository for big data analytics workloads, with a hierarchical namespace, and an access control list (ACL)-based authorization model allowing you to manage access to your data. With Azure Data Lake Gen2, you can store data of any size, shape, and speed, and do all types of processing and analytics across platforms and languages.

**--Q.4 What is access storage tiers in azure**

Access Storage Tiers in Azure are storage tiers that are used to store data in a cost-effective manner. It consists of two levels of storage including the Hot Access Tier and the Cool Access Tier.

The Hot Access Tier is used for data that needs to be accessed frequently and is more expensive than the Cool Access Tier. The Cool Access Tier is used for data that is infrequently accessed and is more economical to store. Both tiers can be used to store data in Azure Blob Storage, Azure Table Storage, and Azure Queue Storage.

**--Q.5 What is blob types in azure**

Azure Blob storage is Microsoft's object storage solution for the cloud. Blobs can be any type of text or binary data, such as a document, media file, or application installer.

Blob storage is also referred to as object storage. Azure Blob storage is divided into three types: block blobs, append blobs, and page blobs.

1. Block blobs are the most commonly used type of blob and are ideal for storing text and binary files, such as documents and media files.

2. Append blobs are similar to block blobs in that they are made up of blocks, but they are optimized for append operations, making them ideal for scenarios such as logging data.

3. Page blobs are a collection of 512-byte pages optimized for random read/write operations and are most commonly used for storing VHD files.

**--Q.6 Types of storage containers in azure**

1. Blob Storage: Blob Storage is used to store large amounts of unstructured data such as images, videos, audio, documents, backups, and logs.

2. File Storage: File Storage is used to store data in the form of files and folders, like a traditional file system.

3. Queue Storage: Queue Storage is used to store messages in a queue for asynchronous processing.

4. Table Storage: Table Storage is used to store structured NoSQL data in the form of tables.

5. Disk Storage: Disk Storage is used to store data as virtual hard disks and can be used as the storage layer for virtual machines.

6. Archive Storage: Archive Storage is used to store rarely accessed data with lower storage costs.

7. StorSimple: StorSimple is used to store data in the cloud while also caching frequently used data on-premises.

----------Day 3----------

**--Q.7 Explore azure data factory:**

Azure Data Factory is a cloud-based data integration service that allows you to create data-driven workflows to orchestrate and automate data movement and data transformation.

With Azure Data Factory, you can create and schedule data-driven pipelines to move and transform data from a variety of sources, such as on-premises and cloud-based data stores, to a destination data store in the cloud.

**Using Azure Data Factory, you can:**

• Automate the movement and transformation of data from a variety of sources, including on-premises, cloud-based, and streaming data.

• Use the data integration capabilities of Azure Data Factory to load data into Azure Data Lake Storage, Azure SQL Database, Azure Synapse Analytics, and other data stores.

• Create and schedule data-driven pipelines to move and transform data from a variety of sources.

• Connect to Azure Machine Learning and Azure Cognitive Services to build intelligent data pipelines.

• Monitor and manage your data pipelines, with advanced features like pipeline triggers and activity logging.

• Create visual representations of your data pipelines using the Azure Data Factory visual tools.

• Securely integrate data from multiple sources and transform it into the desired format for your business needs

**Data Factory Components:**

1. Pipelines
2. Change data capture (preview)
3. Datasets
4. Data flows
5. Power Query

----------Day 4----------

**--Q. How to store data in container:**

ANS: Create Resource group >> Storage Account >> Container >> Upload data file.

**--Q. How to create ADF:**

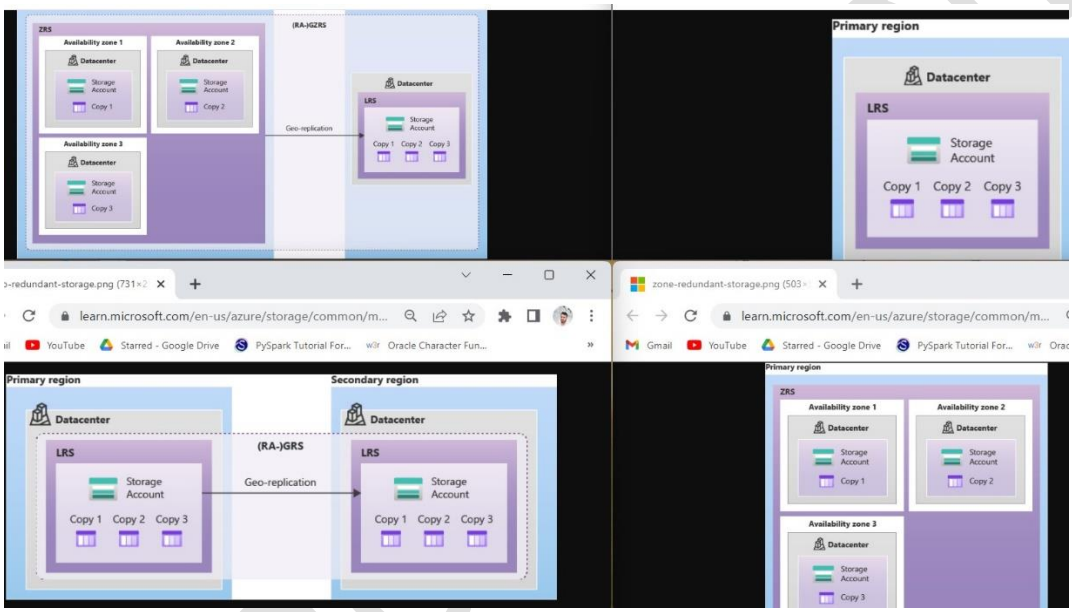
ANS: Create ADF >> Create Datasets >> Create Pipeline >> use file copy option.

**--ADF component:**

1. Integration runtime
2. Linked service
3. Pipeline
4. Storage account

**--Data Redundancy:**

Azure Storage provides data redundancy to ensure your data is highly available and durable. This is accomplished through a combination of built-in replication, which stores copies of data in multiple Azure data centers, as well as the ability to replicate data to another storage account in a different region. This helps protect against single points of failure and increases the durability and availability of your data. Azure Storage also offers the ability to snapshot and restore data, which provides an additional layer of data protection.




**>> Primary Storage**

**1. Locally-Redundant Storage (LRS):** LRS replicates your data three times within a single storage Account and single datacenter, providing a low-cost redundancy option for your data.

**2. Zone-Redundant Storage (ZRS):** ZRS replicates your data across three storage clusters in a single region, providing higher availability for your data than LRS.

**>> Secondary Storage**

**3. Geo-Redundant Storage (GRS):** GRS replicates your data across two regions, providing higher availability for your data than both LRS and ZRS. It is combination of LRS + LRS.

**4. Read-Access Geo-Redundant Storage (RA-GRS):** RA-GRS provides the same benefits as GRS, with the added benefit of providing read access to data in the secondary region.

**5. Geo-Zone-Redundant Storage (GZRS):** GZRS provides the same benefits as GRS, with the added benefit of replicating your data across three or more Azure Availability Zones in the primary region. Combination of LRS + ZRS.

----------Day 5----------

**--Dataflow:**

Azure Data Factory (ADF) is an Azure cloud service that allows you to create data-driven workflows for orchestrating and automating data movement and data transformation.

ADF provides a visual interface for designing and monitoring the pipelines that move and transform your data. ADF pipelines can be triggered to run on a schedule, or on-demand, and can be monitored to ensure that data is transferred correctly. ADF also provides powerful connectors to a variety of data sources, such as Azure Blob Storage, Azure SQL Database, Cosmos DB, and Azure Data Lake Store.

ADF is a powerful tool for automating complex data flows in the cloud.

----------Day 6----------

**--ADF**

**--Q.1. Read => Pii Data & NPii Data**

ANS: PII, or Personally Identifiable Information, is information collected by organizations and used to identify an individual. This includes data such as name, address, phone number, Social Security number, bank account information, and more.

PII is used by organizations to provide services, such as banking, and is regulated by laws like the General Data Protection Regulation (GDPR).

In Azure, PII is protected and managed in accordance with these laws and regulations.

**NPii Data in azure:**

The PI System provides a powerful set of cloud services for collecting, analyzing and sharing data from the PI System. It enables the secure storage and sharing of PI System data in the cloud, enabling customers to access and analyze the data from anywhere.

This data can be used to power applications, build custom dashboards and create reports. Additionally, the PI System provides the ability to collect data from other sources and store it in the cloud, as well as to connect to other cloud services such as Microsoft Azure and Amazon Web Services.

**--Q.2. Data Masking (All data types)**

ANS: Data masking is the process of obscuring sensitive data in a database by replacing the original data with a fictitious value or random character string. This is done to protect data from unauthorized access and to ensure data privacy.

It is often used when transferring data to a third party, or when sensitive data needs to be shared with a partner organization.

Data masking can be used to help protect a company's data from fraudulent activities and malicious attacks.

**--Q.3. #Columns to Mask -> Hands on**

1. Id => Number Masking Logic (1->9, 2->8, 3->7, 4->6, 5->5, 6->4..)

2. Scrape_id => Number Masking (same as Above)

3. Name => String Masking Logic (a->z, A->Z,b->y B->Y,.....)

Hint: Write Cases

SOLUTION:

Use derived column option and apply any one of the following way.

**Static masking:**

create derived column and add datatypes as sha2(256,column_name)

**Dynamic Masking:**

data flow >>  Derived column >> use translate(id, 1, 9) like that

switch >> cases

----------Day 7----------

**--Data Flow Operation:**

1. Rank()
2. Filter()
3. Window()
4. derived column
5. Joins
6. Aggregate columns

----------Day 8----------

**--Masking:** create derived column and add datatypes as sha2(256,column_name) or we can use translation for dynamic masking

---------- Day 9 ----------

## --DataBricks Started (Community Edition):

**--Q. What is DBFS?**

ANS: Databricks File System (DBFS) is a distributed file system installed on Databricks clusters. DBFS allows you to access data in DBFS using the file APIs available in your language of choice, such as local file APIs, HDFS APIs, or S3 APIs. DBFS provides a consistent view of data across nodes and clusters, allowing you to easily access and manage data.

**--Q. Max file size we can store on DBFS?**

ANS: 2GB max size to store data, it is possible to extend this limit by increasing the size of the cluster. To do this, you can increase the number of workers in the cluster or switch to a larger instance type.

**--Q. Why we do not use DBFS in production?**

ANS: Databricks File System (DBFS) is not recommended for production use because it is not designed to be used as a production-grade file system. DBFS is designed for development and testing purposes and does not have the features and scalability needed for production use cases.

**--Q. Why we use Databricks over Data factory**

ANS: Processing time will depend upon size of data and complexity of logic and size of compute engine generally, size of compute engine we usually keep small in learning environment but configure in production to increase performance but increasing size of cluster causes more cost.

In databricks we are getting big cluster with less cost than dataframe, through both uses spark at backend.

**--HDInsightCluster:** Is services which provides open source application in azure i.e

**--Cost calculator**: To see estimated cost for our resources w.r.t. time of use or usage Azure Data Bricks. We will use Azure as an storage and databricks for processing computing clusture using some authentications.

**--MapReduce:**

reads data line by line and hence n number of i/o where there

**--Spark:**

Take file into ram and perform operation and then write it (in memory processing) Hadoop was written in JAVA, Spark written in Scala but Scala is written in java, hence we required JVM for running environment.

**--SC:**

sc is a object of Spark Context class.

**--Transformation:**

In Apache Spark, a transformation is an operation on a RDD (Resilient Distributed Dataset) that produces a new RDD. Transformations are lazy operations, meaning they are only computed when an action is invoked.

Examples of transformations include map(), filter(), and groupBy().

**--Actions:**

In Apache Spark, an action is an operation on an RDD (Resilient Distributed Dataset) that produces a result. Actions are eager operations, meaning they are computed immediately.

Examples of actions include count(), collect(), and saveAsTextFile()

---------- Day 10 ----------

**--What is RDD:**

The Resilient Distributed Datasets are the group of data items that can be stored in-memory on worker nodes. Here,

- Resilient: Restore the data on failure.
- Distributed: Data is distributed among different nodes.
- Dataset: Group of data.

**Properties of RDD**

1. Immutable
2. Distributed (partitioned)
3. resilient (fault tolerance)
4. Ability to be cached and the ability to recover from failures

**Ways to create RDD**

1. from flat file
2. from collection datatypes
3. existing RDD

**--Functions:**

1. textFile()
2. collect()
3. saveAsTextFile() - for write

**--NoteBook 1_PRADEEP_RDD:**

**--Create RDD**

RDD1 = sc.textFile("filepath")

**--Retrieve data from RDD**

RDD1.collect()

**--Retrieve first row from RDD**

RDD1.take(1)

**--Retrieve two row from RDD**

RDD1.take(2)

**--How to check partitions in RDD**

RDD1.getNumPartitions()

**--How to save text file in RDD**

RDD1.saveAsTextFile("path_to_save_file")

**--How to list files and directory in spark**

%fs ls <Path>

**--NOTE:** Number of partitions are depends on number of core of CPU(cluster). maximum partition allowed are 200

**--To know the lean age graph on RDD**

RDD1.toDebugString()

**--2nd way to create RDD**

rdd = sc.parallelize(range(1, 4)).map(lambda x: (x, "a" * x))

**#MapReduce Job In RDD:**

**1. Create one input RDD**

inputRDD = sc.textFile("/FileStore/tables/sample_txt.txt")

**2. Split words by using delimiter split and flatmap function**

```
wordsRDD = inputRDD.flatMap(lambda line: line.split(" "))
wordsRDD.collect()
```

**3. Add kayvalue to the split data by using map function**

```
key_valRDD = wordsRDD.map(lambda word: (word, 1))
key_valRDD.collect()
```

**4. Reduce words by using key**

```
wordcount_RDD = key_valRDD.reduceByKey(lambda a,b:a +b)
wordcount_RDD.collect()
```

**5. Save output file as txt**

```
key_valRDD.saveAsTextFile("/FileStore/PRADEEP_datasets/key_val_RDD")
```

**6. Listout the content of saved file**

```
dbutils.fs.ls("/FileStore/PRADEEP_datasets/key_val_RDD")
```

**7. Display head of first partitioned file**

```
%fs head /FileStore/PRADEEP_datasets/key_val_RDD/part-00000
```

**8. Display head of second partitioned file**

```
%fs head /FileStore/PRADEEP_datasets/key_val_RDD/part-00001
```

**9. Sort and display wordcount data**

sorted_word_count = wordcount_RDD.sortByKey()

**--DAG (Directed acyclic graph)**

Directed Acyclic Graph is a finite direct graph that performs a sequence of computations on data. Each node is an RDD partition, and the edge is a transformation on top of data.

A Directed Acyclic Graph (DAG) is a type of graph used to represent a collection of directed relationships between nodes. A DAG graph consists of nodes (or vertices) that are connected by edges (or arcs). The edges in a DAG graph point in one direction and do not form a cycle, meaning that no node can be reached from itself by following the edges in the graph. DAG graphs are often used to represent complex systems, such as computer networks, workflows, or scheduling algorithms.

**--DBFS:**

The Databricks File System (DBFS) is a virtual file system that allows you to treat cloud object storage as though it were local files and directories on the cluster.

Run file system commands on DBFS using the magic command: %fs, %run, %md

1. open for all

2. size will limited (2GB max size to store data)

**--Data Profiling:**

Data profiling in Azure Databicks is the process of analyzing and validating the data in a data set. This process can help identify any issues or discrepancies in the data that may affect the quality of the data.

The data profiling process involves looking at the data in its entirety to identify trends, patterns, outliers, missing values, and other irregularities. Once these issues are identified, the data can be further examined to determine the root cause and any corrective actions that may be necessary.

Data profiling can help improve the accuracy and quality of data sets, as well as ensure that any data stored in the system is reliable and trustworthy.

**--Databricks:**

Databricks is a cloud-based platform for data engineering, data science, and machine learning. It provides a unified platform for data preparation, real-time analytics, and artificial intelligence. The platform is designed to simplify the process of building and deploying data-driven applications and machine learning models in the cloud.

**--Catalyst Optimizer:**

The Datariccks Azure platform provides a powerful and intuitive interface for optimizing and managing data-driven workloads. It includes a suite of tools, such as the Catalytic Optimizer, that enable users to optimize their data and workloads to get the most out of their cloud resources.

The Catalytic Optimizer is a smart, automated service that optimizes workloads for efficiency and cost savings, allowing users to achieve the maximum performance and cost savings with minimal effort.

The Catalytic Optimizer automatically tunes workloads to ensure they are running as efficiently as possible, helping users save money and time by optimizing their data and workloads.

**--AQE:**

Adaptive query execution is a feature of Apache Spark that optimizes query performance by dynamically adjusting the execution plan based on the data and resources available. This feature allows Spark to adapt to changing workloads and data, making it possible to optimize query performance in real time.

**AQE Application**

AQE applies to all queries that are:

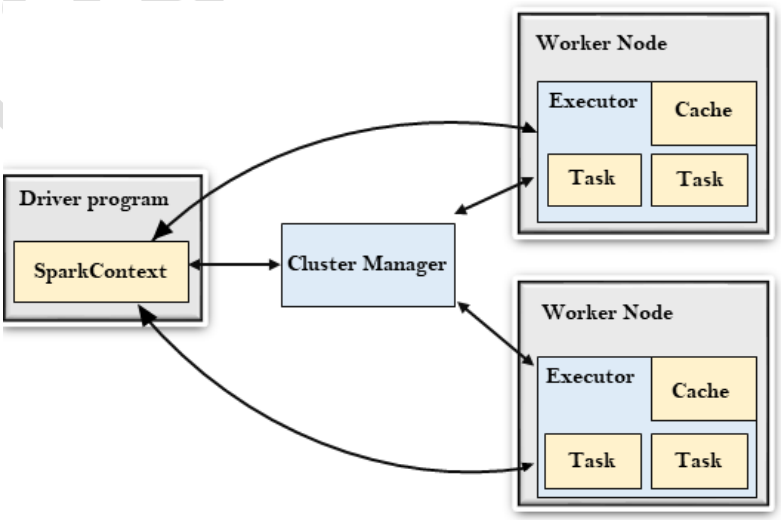
- Non-streaming
- Contain at least one exchange (usually when there's a join, aggregate, or window), one sub-query, or both.

**AQE Capabilities**

In Databricks Runtime 7.3 LTS and above, AQE is enabled by default. It has 4 major features:

- Dynamically changes sort merge join into broadcast hash join.
- Dynamically coalesces partitions (combine small partitions into reasonably sized partitions) after shuffle exchange. Very small tasks have worse I/O throughput and tend to suffer more from scheduling overhead and task setup overhead. Combining small tasks saves resources and improves cluster throughput.
- Dynamically handles skew in sort merge join and shuffle hash join by splitting (and replicating if needed) skewed tasks into roughly evenly sized tasks.
- Dynamically detects and propagates empty relations.

***Spark Job Architecture:**



**1. Driver Program:**

The Driver Program is a process that runs the main() function of the application and creates the **SparkContext** object. The purpose of **SparkContext** is to coordinate the spark applications, running as independent sets of processes on a cluster.

To run on a cluster, the **SparkContext** connects to a different type of cluster managers and then perform the following tasks: -

- It acquires executors on nodes in the cluster.
- Then, it sends your application code to the executors. Here, the application code can be defined by JAR or Python files passed to the SparkContext.
- At last, the SparkContext sends tasks to the executors to run.

**2. Cluster Manager:**

- The role of the cluster manager is to allocate resources across applications. The Spark is capable enough of running on a large number of clusters.
- It consists of various types of cluster managers such as Hadoop YARN, Apache Mesos and Standalone Scheduler.
- Here, the Standalone Scheduler is a standalone spark cluster manager that facilitates to install Spark on an empty set of machines.

**3. Worker Node:**

- The worker node is a slave node
- Its role is to run the application code in the cluster.

**4. Executor:**

- An executor is a process launched for an application on a worker node.
- It runs tasks and keeps data in memory or disk storage across them.
- It read and write data to the external sources.

- Every application contains its executor.

**5. Task:**

- A unit of work that will be sent to one executor.

---------- **Day 11 DataBricks** ----------

***IMP Note: **For performing and understanding code you need following notebooks in your databricks edition, if you don't have notebooks please contact me, I will provide you.**

**Contact me on: contact@pradeepworld.com, pradeep.wagh01@gmail.com**

**--Data Frame Started:**

**--Notebook:** ASP 1.2 - Databricks Platform

**Databricks Platform**

Demonstrate basic functionality and identify terms related to working in the Databricks workspace.

**--Objectives:**

1. Execute code in multiple languages
2. Create documentation cells
3. Access DBFS (Databricks File System)
4. Create database and table
5. Query table and plot results
6. Add notebook parameters with widgets

**--Databricks Notebook Utilities:**

- Magic commands: %python, %scala, %sql, %r, %sh, %md
- DBUtils: dbutils.fs (%fs), dbutils.notebooks (%run), dbutils.widgets
- Visualization: display, displayHTML

**--Setup**

Run classroom setup to mount Databricks training datasets and create your own database for BedBricks.

Use the **%run** magic command to run another notebook within a notebook

**--Execute code in multiple languages**

Run default language of notebook

**--Print using python**

%python

print("Run python")

**--Print using scala**

%scala

println("Run scala")

**--Print using SQL**

%sql

select "Run SQL"

**--Print using r**

```
%r
print("Run R", quote=FALSE)
```

**--Print using Linux**

```
%sh
echo "Hello"
```

**--Note:** %fs is shorthand for the DBUtils module: dbutils.fs

**--To know help**

```
%fs help
```

**--Run file system commands on DBFS using DBUtils directly**

```
dbutils.fs.ls("/databricks-datasets")
```

**--Create table**

**Run Databricks SQL Commands to create a table named events using BedBricks event files on DBFS.**

```
%sql
CREATE TABLE IF NOT EXISTS events USING parquet OPTIONS (path
"/mnt/training/ecommerce/events/events.parquet");
```

**--Print 1st row**

```
%sql
select * from events limit 1
```

**--To See the database name printed below**

print(databaseName)

**#Query table and plot results**

**Use SQL to query the events table**

**--Display data from table**

%sql

SELECT * FROM events

**--Display sum of purchase_revenue_in_usd from ecommerce column group by traffic_source**

%sql

SELECT traffic_source, SUM(ecommerce.purchase_revenue_in_usd) AS total_revenue

FROM events

GROUP BY traffic_source

**#Add notebook parameters with widgets**

**Use widgets to add input parameters to your notebook.**

**1. Create a text input widget using SQL.**

%sql

CREATE WIDGET TEXT state DEFAULT "CA"  **--it stores CA in argument**

**2. Access the current value of the widget using the function getArgument**

**-Display data from events table where state from geo column**

```
%sql

SELECT *

FROM events

WHERE geo.state = getArgument("state")  --argument value is 'CA'
```

**3. Remove the text widget**

```
%sql

REMOVE WIDGET state
```

**# To create widgets in Python, Scala, and R, use the DBUtils module: dbutils.widgets**

```
dbutils.widgets.text("name", "Brickster", "Name")

dbutils.widgets.multiselect("colors", "orange", ["red", "orange", "black", "blue"], "Traffic Sources")
```

**# Access the current value of the widget using the dbutils.widgets function get**

```
name = dbutils.widgets.get("name")

colors = dbutils.widgets.get("colors").split(",")


html = "<div>Hi {}! Select your color preference.</div>".format(name)

for c in colors:

  html += """<label for="{}" style="color:{}"><input type="radio"> {}</label><br>""".format(c, c, c)
```

displayHTML(html)

**O/P:** Hi Brickster! Select your color preference.

orange

**# Remove all widgets**

dbutils.widgets.removeAll()

**Assignment 1 (Question & Answer)**

**--Note: you will find notes at the last of notebook**

**--1. List data files in DBFS using magic commands**

**Use a magic command to display files located in the DBFS directory: /mnt/training/ecommerce**

**Hint:** You should see four items: events, products, sales, users

ANS: %fs ls /mnt/training/ecommerce

**--2. List data files in DBFS using dbutils**

**Use dbutils to get the files at the directory above and save it to the variable files**

**Use the Databricks display() function to display the contents in files**

**Hint:** You should see four items: events, items, sales, users

ANS: files = dbutils.fs.ls("/mnt/training/ecommerce")

display(files)

**--3. Create tables below from files in DBFS**

**Create users table using files at location "/mnt/training/ecommerce/users/users.parquet"**

**Create sales table using files at location "/mnt/training/ecommerce/sales/sales.parquet"**

**Create products table using files at location "/mnt/training/ecommerce/products/products.parquet"**

**(We created events table earlier using files at location "/mnt/training/ecommerce/events/events.parquet")**

ANS:

```
%sql

create table if not exists users using parquet options (path
"/mnt/training/ecommerce/users/users.parquet");

create table if not exists sales using parquet options (path
"/mnt/training/ecommerce/sales/sales.parquet");

create table if not exists products using parquet options (path
"/mnt/training/ecommerce/products/products.parquet");
```

**# 4. Execute SQL to explore BedBricks datasets**

**Run SQL queries on the products, sales, and events tables to answer the following questions.**

**-- Q1: What products are available for purchase at BedBricks?**

**The products dataset contains the ID, name, and price of products on the BedBricks retail site.**

**----------|**

**field | type | description |**

**----------|**

**item_id | string | unique item identifier |**

**name | string | item name in plain text |**

**price | double | price of item |**

**----------|**

**Execute a SQL query that selects all from the products table.**

ANS:

%sql

select * from products

**-- Q2: What is the average purchase revenue for a transaction at BedBricks?**

**The sales dataset contains order information representing successfully processed sales.**

**Most fields correspond directly with fields from the clickstream data associated with a sale finalization event.**

**Execute a SQL query that computes the average purchase_revenue_in_usd from the sales table.**

**Hint:** The result should be 1042.79

ANS:

%sql

select floor(avg(purchase_revenue_in_usd),2) from sales;

**-- Q3: What types of events are recorded on the BedBricks website?**

**The events dataset contains two weeks' worth of parsed JSON records, created by consuming updates to an operational database.**

**Records are received whenever: (1) a new user visits the site, (2) a user provides their email for the first time.**

**Execute a SQL query that selects distinct values in event_name from the events table**

**Hint:** You should see 23 distinct event_name values.

ANS:

%sql

select distinct(event_name) from events

**-- How to clean up notebook**

da.cleanup()

**---------- Day 12 ----------**

**# Notebook ASP 1.3 - Spark SQL**

**-- Multiple Interfaces:** Spark SQL is a module for structured data processing with multiple interfaces.

**We can interact with Spark SQL in two ways:**

**1. Executing SQL queries**

**2. Working with the DataFrame API.**

**# 1. Method 1: Executing SQL queries**

**--Explain plan of below sql query**

```
%sql
explain SELECT name, price
FROM products
WHERE price < 2000
ORDER BY price
```

**# 2. Method 2: Working with the DataFrame API**

**We can also express Spark SQL queries using the DataFrame API.**

**The following cell returns a DataFrame containing the same results as those retrieved above.**

**--Create a dataframe and display name and price from product table where price is less than 200 and order by price.**

```
products_DF = (spark.table("products")
  .select("name", "price")
  .where("price < 200")
  .orderBy("price"))

display(products_DF)
```

**-- Display name and price from product table where price is less than 200 and order by price.**

```
display(spark.table("products")
  .select("name", "price")
  .where("price < 200")
  .orderBy("price"))
```

**-- Note:** Resilient Distributed Datasets (RDDs) are the low-level representation of datasets processed by a Spark cluster. In early versions of Spark, you had to write code manipulating RDDs directly. In modern versions of Spark you should instead use the higher-level DataFrame APIs, which Spark automatically compiles into low-level RDD operations.

**# SparkSession:** The SparkSession class is the single entry point to all functionality in Spark using the DataFrame API. In Databricks notebooks, the SparkSession is created for you, stored in a variable called **spark**.

**-- Spark :** to store a spark session in spark variable

**-- Create Dataframe on product table**

```
productsDF = spark.table("products")
display(productsDF)
```

**# Below are several additional methods we can use to create DataFrames. All of these can be found in the documentation for SparkSession.**

**-- SparkSession Methods:**

| Method | Description |
|---|---|
| 1. sql | Returns a DataFrame representing the result of the given query |
| 2. table | Returns the specified table as a DataFrame |
| 3. read | Returns a DataFrameReader that can be used to read data in as a DataFrame |
| 4. range | Create a DataFrame with a column containing elements in a range from start to end (exclusive) with step value and number of partitions. |
| 5. createDataFrame | Creates a DataFrame from a list of tuples, primarily used for testing. |

**-- Let's use a SparkSession method to run SQL.**

```
resultDF = spark.sql("""
SELECT name, price
FROM products
WHERE price < 200
ORDER BY price
""")

display(resultDF)
```

**# DataFrames**

Recall that expressing our query using methods in the DataFrame API returns results in a DataFrame. Let's store this in the variable budgetDF.

**-- A DataFrame is a distributed collection of data grouped into named columns.**

```
budgetDF = (spark.table("products")
  .select("name", "price")
  .where("price < 200")
  .orderBy("price"))

display(budgetDF)
```

**-- The schema defines the column names and types of a dataframe. Access a dataframe's schema using the schema attribute.**

1. budgetDF.printSchema()
2. budgetDF.schema

**-- View a nicer output for this schema using the printSchema() method.**

**budgetDF.printschema()**

**# Transformations:** When we created budgetDF, we used a series of DataFrame transformation methods e.g. select, where, orderBy.

```
productsDF
  .select("name", "price")
  .where("price < 200")
  .orderBy("price")
```

Transformations operate on and return DataFrames, allowing us to chain transformation methods together to construct new DataFrames. However, these operations can't execute on their own, as transformation methods are lazily evaluated.

**Running the following cell does not trigger any computation.**

**# Actions:** Conversely, DataFrame actions are methods that trigger computation.

Actions are needed to trigger the execution of any DataFrame transformations.

The show action causes the following cell to execute transformations.

```
(productsDF
  .select("name", "price")
  .where("price < 200")
  .orderBy("price")
  .show())
```

**-- Below are several examples of DataFrame actions.**

**DataFrame Action Methods**

| Method | Description |
|---|---|
| 1. show | Displays the top n rows of DataFrame in a tabular form. |
| 2. count | Returns the number of rows in the DataFrame. |
| 3. describe, summary | Computes basic statistics for numeric and string columns. |
| 4. first, head | Returns the the first row. |
| 5. collect | Returns an array that contains all rows in this DataFrame. |
| 6. take | Returns an array of the first n rows in the DataFrame. |

**--To count rows**

budgetDF.count()

**--To collect array of all rows from DF**

budgetDF.collect()

**# Convert between DataFrames and SQL**

createOrReplaceTempView creates a temporary view based on the DataFrame. The lifetime of the temporary view is tied to the SparkSession that was used to create the DataFrame.

**-- Global View**

budgetDF.createOrReplaceGlobalTempView("budget_global")

display(spark.sql("SELECT * FROM budget_global"))

**-- Temp View**

budgetDF.createOrReplaceTempView("budget")

display(spark.sql("SELECT * FROM budget"))

**# Spark SQL Lab Assignment 2**

**-- Tasks_To_Do_Answer_Is_Below**

1. Create a DataFrame from the events table

2. Display the DataFrame and inspect its schema

3. Apply transformations to filter and sort macOS events

4. Count results and take the first 5 rows

5. Create the same DataFrame using a SQL query

**-- Methods**

**1. SparkSession:** sql, table

**2. DataFrame transformations:** select, where, orderBy

**3. DataFrame actions:** select, count, take

**4. Other DataFrame methods:** printSchema, schema, createOrReplaceTempView

**----------Assignment 2 - Questions & Answers ----------**

**1. Create a DataFrame from the events table Use SparkSession to create a DataFrame from the events table**

**ANS:**

```
eventsDF = spark.table("events")

display(eventsDF)
```

**2. Display DataFrame and inspect schema Use methods above to inspect DataFrame contents and schema**

**ANS:**

**-- To display:**

```
display(eventsDF)
```

**-- To inspect schema:**

```
eventsDF.printSchema()
```

**3. Apply transformations to filter and sort macOS events Filter for rows where device is macOS Sort rows by event_timestamp**

**Hint: Use single and double quotes in your filter SQL expression**

```
macDF = (eventsDF
    .sort("event_timestamp")
    .where("device = 'macOS'"))
display(macDF)
```

**4. Count results and take first 5 rows Use DataFrame actions to count and take rows**

```
numRows = macDF.count() #to display count
display(numRows)

rows = macDF.take(5)  #to take first 5 rows
display(rows)
```

**5. Create the same DataFrame using SQL query Use SparkSession to run a SQL query on the events table Use SQL commands to write the same filter and sort query used earlier**

```
macSQLDF = spark.sql("select * from events")
display(macSQLDF)

macSQLDF = spark.sql("select * from
events").where("device='macOS'").sort("event_timestamp")
display(macSQLDF)
```

**-- Pyhton Notebook : ASP 1.4 - Reader & Writer**

**# DataFrameReader Interface used to load a DataFrame from external storage systems**

spark.read.parquet("path/to/files")

DataFrameReader is accessible through the SparkSession attribute read. This class includes methods to load DataFrames from different external storage systems.

**# Read from CSV files**

**-- Read from CSV with the DataFrameReader's csv method and the following options:**

**Tab separator, use first line as header, infer schema**

%fs head /mnt/training/ecommerce/users/users-500k.csv/part-00000-tid-6798248775191304424-0020915c-5cd2-4aae-8903-2d586d002073-2359-1-c000.csv

**-- Read CSV file and display schema**

usersCsvPath = "/mnt/training/ecommerce/users/users-500k.csv"

**-- Spark's Python API also allows you to specify the DataFrameReader options as parameters to the csv method**

```
usersDF = (spark
        .read
        .option("sep", "\t")
        .option("header", True)
        .option("inferSchema", True)
```

```
        .csv(usersCsvPath)
       )

usersDF.printSchema()

-- Count of rows
usersDF.count()

# Manually define the schema by creating a StructType with column names and data types
from pyspark.sql.types import LongType, StringType, StructType, StructField

userDefinedSchema = StructType([
   StructField("user_id", StringType(), True),
   StructField("user_first_touch_timestamp", LongType(), True),
   StructField("email", StringType(), True)
])

print(type(userDefinedSchema))

# Define the schema using data definition language (DDL) syntax.
DDLSchema = "user_id string, user_first_touch_timestamp long, email string"

usersDF = (spark
        .read
```

```
        .option("sep", "\t")
        .option("header", True)
        .schema(DDLSchema)
        .csv(usersCsvPath)
       )
```

**-- Read from JSON files**

**Read from JSON with DataFrameReader's json method and the infer schema option**

**-- Display Content of file**

```
%fs head /mnt/training/ecommerce/events/events-500k.json/part-00000-tid-309888144738233288-fab86c62-ff9f-4176-98c6-587f95ee9066-2365-1-c000.json
```

```
eventsJsonPath = "/mnt/training/ecommerce/events/events-500k.json"
```

```
eventsDF = (spark
        .read
        .option("inferSchema", True)
        .json(eventsJsonPath)
       )
```

```
eventsDF.printSchema()
```

**-- Read data faster by creating a StructType with the schema names and data types**

**from pyspark.sql.types import ArrayType, DoubleType, IntegerType, LongType, StringType, StructType, StructField**

```
userDefinedSchema = StructType([
  StructField("device", StringType(), True),
  StructField("ecommerce", StructType([
    StructField("purchaseRevenue", DoubleType(), True),
    StructField("total_item_quantity", LongType(), True),
    StructField("unique_items", LongType(), True)
  ]), True),
  StructField("event_name", StringType(), True),
  StructField("event_previous_timestamp", LongType(), True),
  StructField("event_timestamp", LongType(), True),
  StructField("geo", StructType([
    StructField("city", StringType(), True),
    StructField("state", StringType(), True)
  ]), True),
  StructField("items", ArrayType(
    StructType([
      StructField("coupon", StringType(), True),
      StructField("item_id", StringType(), True),
      StructField("item_name", StringType(), True),
      StructField("item_revenue_in_usd", DoubleType(), True),
```

```
        StructField("price_in_usd", DoubleType(), True),
        StructField("quantity", LongType(), True)
      ])
  ), True),
  StructField("traffic_source", StringType(), True),
  StructField("user_first_touch_timestamp", LongType(), True),
  StructField("user_id", StringType(), True)
])
```

**-- Using this schema now we can create DataFrame**

```
eventsDF = (spark
        .read
        .schema(userDefinedSchema)
        .json(eventsJsonPath)
       )
```

**-- You can use the StructType Scala method toDDL to have a DDL-formatted string created for you.**

**In a Python notebook, create a Scala cell to create the string to copy and paste.**

```
%scala
spark.read.parquet("/mnt/training/ecommerce/events/events.parquet").schema.toDDL
```

```
DDLSchema = "`device` STRING,`ecommerce`
STRUCT<`purchase_revenue_in_usd`: DOUBLE, `total_item_quantity`: BIGINT,
`unique_items`: BIGINT>,`event_name` STRING,`event_previous_timestamp`
BIGINT,`event_timestamp` BIGINT,`geo` STRUCT<`city`: STRING, `state`:
STRING>,`items` ARRAY<STRUCT<`coupon`: STRING, `item_id`: STRING,
`item_name`: STRING, `item_revenue_in_usd`: DOUBLE, `price_in_usd`: DOUBLE,
`quantity`: BIGINT>>,`traffic_source` STRING,`user_first_touch_timestamp`
BIGINT,`user_id` STRING"


eventsDF = (spark
        .read
        .schema(DDLSchema)
        .json(eventsJsonPath)
       )
```

**# DataFrameWriter Interface used to write a DataFrame to external storage systems**

```
(df.write
  .option("compression", "snappy")
  .mode("overwrite")
  .parquet(outPath)
)
```

DataFrameWriter is accessible through the SparkSession attribute write. This class includes methods to write DataFrames to different external storage systems.

**# Write DataFrames to files**

**-- Write usersDF to parquet with DataFrameWriter's parquet method and the following configurations:**

**Snappy compression, overwrite mode**

```
usersOutputPath = workingDir + "/users.parquet"


(usersDF
 .write
 .option("compression", "snappy")
 .mode("overwrite")
 .parquet(usersOutputPath)
)

display(
   dbutils.fs.ls(usersOutputPath)
)
```

**# As with DataFrameReader, Spark's Python API also allows you to specify the DataFrameWriter options as parameters to the parquet method**

**--Overwrite a file with snappy compression**

```
(usersDF
 .write
 .parquet(usersOutputPath, compression="snappy", mode="overwrite")
)
```

**# Write DataFrames to tables:**

**Write eventsDF to a table using the DataFrameWriter method saveAsTable**

**Note:** This creates a global table, unlike the local view created by the DataFrame method createOrReplaceTempView.

**-- Print dataframe schema**

eventsDF.printSchema()

**-- Print current database name**

print(databaseName)

**# Delta Lake:** In almost all cases, the best practice is to use Delta Lake format, especially whenever the data will be referenced from a Databricks workspace. Delta Lake is an open source technology designed to work with Spark to bring reliability to data lakes.

**--Delta Lake's Key Features:**

1. ACID transactions
2. Scalable metadata handling
3. unified streaming and batch processing
4. Time travel (data versioning)
5. Schema enforcement and evolution
6. Audit history
7. Parquet format
8. Compatible with Apache Spark API

**-- Save File**

```
eventsOutputPath = workingDir + "/delta/events"


(eventsDF
 .write
 .format("delta")
 .mode("overwrite")
 .save(eventsOutputPath)
)
```

**# Ingesting Data Lab:**

**Read in CSV files containing products data.**

**Tasks**

**1. Read with infer schema:**

**Ans:**

```
productsCsvPath = "/mnt/training/ecommerce/products/products.csv"


productsDF = (spark.read
        .option("header",True)
        .option("inferschema",True)
        .csv(productsCsvPath)
        )


productsDF.printSchema()
```

**2. Read with user-defined schema**

**Ans:**

```
from pyspark.sql.types import ArrayType, DoubleType, IntegerType, LongType, StringType, StructType, StructField

userDefinedSchema = StructType([
                StructField("item_id",IntegerType(),True),
                StructField("name",StringType(),True),
                StructField("price",LongType(),True),
                ])


productsCsvPath = "/mnt/training/ecommerce/products/products.csv"
productsDF2 = (spark.read
            .option("sep", "\t")
            .option("header", True)
            .schema(userDefinedSchema)
            .csv(productsCsvPath)
           )
```

**3. Read with schema as DDL formatted string**

**Ans:**

```
DDLSchema = "`item_id`DOUBLE,`name`STRING, `price`BIGINT"


productsDF3 =(spark.read
            .option("sep", "\t")
            .option("header", True)
```

```
        .schema(DDLSchema)
        .csv(productsCsvPath)
       )
```

**4. Write using Delta format**

**Ans:**

Write productsDF to the filepath provided in the variable productsOutputPath

```
productsOutputPath = workingDir + "/delta/products"
(productsDF3.write
 .format("delta")
 .mode("overwrite")
 .save(productsOutputPath)
)
```

---------- Day 13 ----------

**Notebook : ASP 1.5 - DataFrame & Column**

**# DataFrame & Column**

**-- Objectives**

1. Construct columns
2. Subset columns
3. Add or replace columns
4. Subset rows
5. Sort rows

**-- Methods**

**-DataFrame:** select, selectExpr, drop, withColumn, withColumnRenamed, filter, distinct, limit, sort

**-Column:** alias, isin, cast, isNotNull, desc, operators

**--Let's use the BedBricks events dataset.**

eventsDF = spark.read.parquet(eventsPath)

display(eventsDF)

**# Column Expressions**

A Column is a logical construction that will be computed based on the data in a DataFrame using an expression.

Construct a new Column based on existing columns in a DataFrame

from pyspark.sql.functions import col

eventsDF.device

eventsDF["device"]

col("device")

col("ecommerce")

**# Column Operators and Methods**

| Method | Description |
|---|---|
| *, + , <, >= | Math and comparison operators |
| | |
| ==, != | Equality and inequality tests (Scala operators are === and =!=) |
| | |

| | |
|---|---|
| alias | Gives the column an alias |
| | |
| cast, astype | Casts the column to a different data type |
| | |
| isNull, isNotNull, isNan | Is null, is not null, is NaN |
| | |
| asc, desc | Returns a sort expression based on ascending/descending order of the column |

**-- Create complex expressions with existing columns, operators, and methods.**

```
col("ecommerce.purchase_revenue_in_usd") +
col("ecommerce.total_item_quantity")
```

```
col("event_timestamp").desc()
```

```
(col("ecommerce.purchase_revenue_in_usd") * 100).cast("int")
```

**# Subset columns**

Use DataFrame transformations to subset columns

**-- select()**

Selects a list of columns or column based expressions

```
devicesDF = eventsDF.select("user_id", "device")
```

```
display(devicesDF)
```

```
from pyspark.sql.functions import col
locationsDF = eventsDF.select(
   "user_id",
   col("geo.city").alias("city"),
   col("geo.state").alias("state")
)
display(locationsDF)
```

**# selectExpr()**

Selects a list of SQL expressions

```
appleDF = eventsDF.selectExpr("user_id", "device in ('macOS', 'iOS') as apple_user")
display(appleDF)
```

**# drop()**

Returns a new DataFrame after dropping the given column, specified as a string or Column object

Use strings to specify multiple columns

```
-- Drop column and return a new DF
anonymousDF = eventsDF.drop("user_id", "geo", "device")
display(anonymousDF)
```

```
-- Drop ecommerce column
noSalesDF = eventsDF.drop(col("ecommerce"))
```

display(noSalesDF)

**# Add or replace columns**

Use DataFrame transformations to add or replace columns

**-- withColumn()**

Returns a new DataFrame by adding a column or replacing an existing column that has the same name.

mobileDF = eventsDF.withColumn("mobile", col("device").isin("iOS", "Android"))

display(mobileDF)

--

purchaseQuantityDF = eventsDF.withColumn("purchase_quantity", col("ecommerce.total_item_quantity").cast("int"))

purchaseQuantityDF.printSchema()

display(purchaseQuantityDF)

**# withColumnRenamed()** : Returns a new DataFrame with a column renamed.

locationDF = eventsDF.withColumnRenamed("geo", "location")

display(locationDF)

**# Subset Rows:** Use DataFrame transformations to subset rows

**-- filter():** Filters rows using the given SQL expression or column based condition.

**-- By using filter display ecommerce column**

```
purchasesDF = eventsDF.filter("ecommerce.total_item_quantity > 0")

display(purchasesDF)
```

**-- Using filter display not null value from column**

```
revenueDF =
eventsDF.filter(col("ecommerce.purchase_revenue_in_usd").isNotNull())

display(revenueDF)
```

**-- DropDuplicates()** : Returns a new DataFrame with duplicate rows removed, optionally considering only a subset of columns.

Alias : distinct

```
eventsDF.distinct()
```

**--Drop duplicates**

```
distinctUsersDF = eventsDF.dropDuplicates(["user_id"])

display(distinctUsersDF)

distinctUsersDF.count()
```

**-- limit()** : Returns a new DataFrame by taking the first n rows.

```
limitDF = eventsDF.limit(100)

display(limitDF)
```

**-- Sort rows :** Use DataFrame transformations to sort rows sort() Returns a new DataFrame sorted by the given columns or expressions.

Alias: orderBy

**-- Sort by timestamp**

```
increaseTimestampsDF = eventsDF.sort("event_timestamp")

display(increaseTimestampsDF)
```

**-- Sort by timestamp descending order**

```
decreaseTimestampsDF = eventsDF.sort(col("event_timestamp").desc())

display(decreaseTimestampsDF)
```

**-- Sort by two columns**

```
increaseSessionsDF = eventsDF.orderBy(["user_first_touch_timestamp",
"event_timestamp"])

display(increaseSessionsDF)
```

**-- Sort by 2 columns but first is descending**

```
decreaseSessionsDF = eventsDF.sort(col("user_first_touch_timestamp").desc(),
col("event_timestamp"))

display(decreaseSessionsDF)
```

----------Assignment 4 - Questions & Answers----------

**--Q.1. Extract purchase revenue for each event**

Add new column revenue by extracting ecommerce.purchase_revenue_in_usd

**ANS:**

```
from pyspark.sql.functions import col


revenueDF =
(eventsDF.withColumn('revenue',col("ecommerce.purchase_revenue_in_usd").ca
st("float"))
        .sort(col("revenue").desc())
        )
display(revenueDF)
```

**--Q.2. Filter events where revenue is not null**

Filter for records where revenue is not null

**ANS**:

```
purchasesDF = revenueDF.filter(col("revenue").isNotNull())
display(purchasesDF)
```

**--Q.3. Check what types of events have revenue**

Find unique event_name values in purchasesDF in one of two ways:

Select "event_name" and get distinct records

Drop duplicate records based on the "event_name" only

Hint : There's only one event associated with revenues

**ANS :**

```
distinctDF = purchasesDF.dropDuplicates(["event_name"])
display(distinctDF)
```

**--Q.4. Drop unneeded column**

Since there's only one event type, drop event_name from purchasesDF.

**ANS:**

```
finalDF = purchasesDF.drop("event_name")

display(finalDF)
```

**--Q.5. Chain all the steps above excluding step 3**

**ANS**:

```
finalDF =
(eventsDF.withColumn('revenue',col("ecommerce.purchase_revenue_in_usd").cast("float")
        .filter(col("revenue").isNotNull())
        .dropDuplicates(["event_name"])
        .drop("event_name")
        )

display(finalDF)
```

---------- Day 14 ----------

**Notebook: ASP 2.1 - Aggregation**

**# Aggregation**

**-- Objectives**

1. Group data by specified columns

2. Apply grouped data methods to aggregate data

3. Apply built-in functions to aggregate data

**-- Methods**

1. DataFrame: groupBy

2. Grouped Data: agg, avg, count, max, sum

3. Built-In Functions: approx_count_distinct, avg, sum

**-- Let's use the BedBricks events dataset.**

```
df = spark.read.parquet(eventsPath)
display(df)
```

**# groupBy**

Use the DataFrame groupBy method to create a grouped data object.

This grouped data object is called RelationalGroupedDataset in Scala and GroupedData in Python.

**-- Group by event_name column and create new DF**

```
df1 = df.groupBy("event_name")
```

**-- Type of DF**

```
print(type(df1))
```

**-- Group by on two columns**

```
df.groupBy("geo.state", "geo.city")
```

**# Grouped data methods**

Various aggregation methods are available on the GroupedData object.

| Method | Description |
|---|---|
| agg | Compute aggregates by specifying a series of aggregate columns |
| avg | Compute the mean value for each numeric columns for each group |
| count | Count the number of rows for each group |
| max | Compute the max value for each numeric columns for each group |
| mean | Compute the average value for each numeric columns for each group |
| min | Compute the min value for each numeric column for each group |
| pivot | Pivots a column of the current DataFrame and performs the specified aggregation |
| sum | Compute the sum for each numeric columns for each group |

**-- Group by event name & count records**

```
eventCountsDF = df.groupBy("event_name").count()

display(eventCountsDF)
```

**-- Type of DF**

```
print(type(eventCountsDF))
```

**-- Here, we're getting the average purchase revenue for each state**

```
avgStatePurchasesDF =
df.groupBy("geo.state").avg("ecommerce.purchase_revenue_in_usd")

display(avgStatePurchasesDF)
```

**-- Total quantity and sum of the purchase revenue for each combination of state and city.**

```
cityPurchaseQuantitiesDF = df.groupBy("geo.state",
"geo.city").sum("ecommerce.total_item_quantity",
"ecommerce.purchase_revenue_in_usd")

display(cityPurchaseQuantitiesDF)
```

**# Built-In Functions**

In addition to DataFrame and Column transformation methods, there are a ton of helpful functions in Spark's built-in SQL functions module.

In Scala, this is org.apache.spark.sql.functions, and pyspark.sql.functions in Python. Functions from this module must be imported into your code.

**# Aggregate Functions**

Here are some of the built-in functions available for aggregation.

| Method | Description |
|---|---|
| approx_count_distinct | Returns the approximate number of distinct items in a group |
| avg | Returns the average of the values in a group |
| collect_list | Returns a list of objects with duplicates |
| corr | Returns the Pearson Correlation Coefficient for two columns |
| max | Compute the max value for each numeric columns for each group |
| mean | Compute the average value for each numeric columns for each group |
| stddev_samp | Returns the sample standard deviation of the expression in a group |

| | |
|---|---|
| sumDistinct | Returns the sum of distinct values in the expression |
| var_pop | Returns the population variance of the values in a group |

Use the grouped data method agg to apply built-in aggregate functions, This allows you to apply other transformations on the resulting columns, such as alias.

**# Write dataframe to JSON file (Nested JSON) => complex data types.**

```
statePurchasesDF.write.json("/FileStore/tables/PRADEEP_json_sample_out")
```

**-- Read data**

```
%fs head /FileStore/tables/PRADEEP_json_sample_out/part-00000-tid-4021800894627806559-8915f387-7b92-4629-9b3f-c646ca03932c-130-1-c000.json
```

**-- Apply multiple aggregate functions on grouped data**

```
from pyspark.sql.functions import avg, approx_count_distinct

stateAggregatesDF = (df
                     .groupBy("geo.state")
                    .agg(avg("ecommerce.total_item_quantity").alias("avg_quantity"),
                         approx_count_distinct("user_id").alias("distinct_users"))
                    )

display(stateAggregatesDF)
```

**# Math Functions**

**Here are some of the built-in functions for math operations.**

| Method | Description |
|---|---|
| ceil | Computes the ceiling of the given column. |
| cos | Computes the cosine of the given value. |
| log | Computes the natural logarithm of the given value. |
| round | Returns the value of the column e rounded to 0 decimal places with HALF_UP round mode. |
| sqrt | Computes the square root of the specified float value. |

```
from pyspark.sql.functions import cos, sqrt


display(
  spark.range(10)  # Create a DataFrame with a single column called "id" with a range of integer values
  .withColumn("sqrt", sqrt("id"))
  .withColumn("cos", cos("id"))
)
```

**# Revenue by Traffic Lab**

Get the 3 traffic sources generating the highest total revenue.

**# Aggregate revenue by traffic source**

--Get top 3 traffic sources by total revenue

Clean revenue columns to have two decimal places

**-- Methods**

DataFrame: groupBy, sort, limit

Column: alias, desc, cast, operators

Built-in Functions: avg, sum

**# Revenue by Traffic Lab**

Get the 3 traffic sources generating the highest total revenue.

**# Setup**

Run the cell below to create the starting DataFrame df.

```
from pyspark.sql.functions import col

# Purchase events logged on the BedBricks website
df = (spark.read.parquet(eventsPath)
    .withColumn("revenue", col("ecommerce.purchase_revenue_in_usd"))
    .filter(col("revenue").isNotNull())
    .drop("event_name")
   )
display(df)
```

**----------Assignment 5 Question & Answers----------**

**Q.1. Aggregate revenue by traffic source**

- Group by traffic_source
- Get sum of revenue as total_rev
- Get average of revenue as avg_rev

Remember to import any necessary built-in functions.

**ANS:**

```
from pyspark.sql.functions import sum

trafficDF = (df.groupBy("traffic_source")
        .agg(sum("revenue").alias("total_rev"),avg("revenue").alias("avg_rev"))
        )
display(trafficDF)
```

**Q.2. Get top three traffic sources by total revenue**

- Sort by total_rev in descending order
- Limit to first three rows

**ANS:**

```
topTrafficDF = (trafficDF.sort("total_rev", ascending=False)
          .limit(3)
      )
display(topTrafficDF)
```

**Q.3 Limit revenue columns to two decimal places**

- Modify columns avg_rev and total_rev to contain numbers with two decimal places
- Use withColumn() with the same names to replace these columns

- To limit to two decimal places, multiply each column by 100, cast to long, and then divide by 100

**ANS:**

```
finalDF = (topTrafficDF.withColumn("avg_rev",
(col("avg_rev")*100).cast("long")/100)
                .withColumn("total_rev", (col("total_rev")*100).cast("long")/100)
)
display(finalDF)
```

**Q.4. Bonus: Rewrite using a built-in math function**

Find a built-in math function that rounds to a specified number of decimal places

**ANS:**

```
from pyspark.sql.functions import round
bonusDF = topTrafficDF.withColumn("avg_rev_rounded", round("avg_rev",2))
                .withColumn("total_rev_rounded", round("total_rev",2))
display(bonusDF)
```

**Q.5 Chain all the steps above**

**ANS:**

```
from pyspark.sql.functions import round
chainDF = (df.groupBy("traffic_source")
        .agg(sum("revenue").alias("total_rev"),avg("revenue").alias("avg_rev"))
        .sort("total_rev", ascending=False)
        .limit(3)
        .withColumn("avg_rev", (col("avg_rev")*100).cast("long")/100)
```

```
        .withColumn("total_rev", (col("total_rev")*100).cast("long")/100)
        .withColumn("avg_rev_rounded", round("avg_rev",2))
        .withColumn("total_rev_rounded", round("total_rev",2))
      )
display(chainDF)
```

---

**NoteBook : ASP 2.2 - Datetimes**

**# Datetime Functions**

**--Objectives**

- Cast to timestamp
- Format datetimes
- Extract from timestamp
- Convert to date
- Manipulate datetimes

**--Methods**

- Column: cast
- Built-In Functions: date_format, to_date, date_add, year, month, dayofweek, minute, second

--Let's use a subset of the BedBricks events dataset to practice working with date timestamp.

```
from pyspark.sql.functions import col
```

```
df = spark.read.parquet(eventsPath).select("user_id",
col("event_timestamp").alias("timestamp"))

display(df)
```

**# Built-In Functions: Date Time Functions**

Here are a few built-in functions to manipulate dates and times in Spark.

| Method | Description |
|---|---|
| add_months | Returns the date that is numMonths after startDate |
| current_timestamp | Returns the current timestamp at the start of query evaluation as a timestamp column |
| date_format | Converts a date/timestamp/string to a value of string in the format specified by the date format given by the second argument. |
| Dayofweek | Extracts the day of the month as an integer from a given date/timestamp/string |
| from_unixtime | Converts the number of seconds from unix epoch (1970-01-01 00:00:00 UTC) to a string representing the timestamp of that moment in the current system time zone in the yyyy-MM-dd HH:mm:ss format |
| Minute | Extracts the minutes as an integer from a given date/timestamp/string. |
| unix_timestamp | Converts time string with given pattern to Unix timestamp (in seconds) |

**# Cast to Timestamp**

--cast() : Casts column to a different data type, specified using string representation or DataType.

```
timestampDF = df.withColumn("timestamp",(col("timestamp") / 1e6).cast("timestamp"))

display(timestampDF)
```

```
--import TimestampType

from pyspark.sql.types import TimestampType

timestampDF = df.withColumn("timestamp", (col("timestamp") / 1e6).cast(TimestampType()))

display(timestampDF)
```

**# Datetime Patterns for Formatting and Parsing**

There are several common scenarios for datetime usage in Spark:

- CSV/JSON datasources use the pattern string for parsing and formatting datetime content.
- Datetime functions related to convert StringType to/from DateType or TimestampType e.g. unix_timestamp, date_format, from_unixtime, to_date, to_timestamp, etc.
- Spark uses pattern letters for date and timestamp parsing and formatting. A subset of these patterns are shown below.

| Symbol | Meaning | Presentation | Examples |
|---|---|---|---|
| G | Era | Text | AD; Anno Domini |
| Y | year | Year | 2020; 20 |
| D | day-of-year | number(3) | 189 |
| M/L | month-of-year | month | 7; 07; Jul; July |
| d | day-of-month | number(3) | 28 |
| Q/q | quarter-of-year | number/text | 3; 03; Q3; 3rd quarter |
| E | day-of-week | text | Tue; Tuesday |

Spark's handling of dates and timestamps changed in version 3.0, and the patterns used for parsing and formatting these values changed as well.

**# Format date**

--date_format()

Converts a date/timestamp/string to a string formatted with the given date time pattern.

```
from pyspark.sql.functions import date_format

formattedDF = (timestampDF

        .withColumn("date string", date_format("timestamp", "MMMM dd, yyyy"))

            .withColumn("time string", date_format("timestamp", "HH:mm:ss.SSSSSS"))

            )

display(formattedDF)
```

**# Extract datetime attribute from timestamp**

--year

Extracts the year as an integer from a given date/timestamp/string.

Similar methods: month, dayofweek, minute, second, etc.

```
-- Extract month, dayofweek, minute, second, etc.
from pyspark.sql.functions import year, month, dayofweek, minute, second
datetimeDF = (timestampDF
        .withColumn("year", year(col("timestamp")))
        .withColumn("month", month(col("timestamp")))
        .withColumn("dayofweek", dayofweek(col("timestamp")))
        .withColumn("minute", minute(col("timestamp")))
        .withColumn("second", second(col("timestamp")))
       )
display(datetimeDF)
```

**# Convert to Date**

--to_date : Converts the column into DateType by casting rules to DateType.

```
from pyspark.sql.functions import to_date
dateDF = timestampDF.withColumn("date", to_date(col("timestamp")))
display(dateDF)
```

**# Manipulate Datetimes**

--date_add : Returns the date that is the given number of days after start

**--Add 2 days in given date**

```
from pyspark.sql.functions import date_add

plus2DF = timestampDF.withColumn("plus_two_days",
date_add(col("timestamp"), 2))

display(plus2DF)
```

**# Active Users Lab:** Plot daily active users and average active users by day of week.

1. Extract timestamp and date of events
2. Get daily active users
3. Get average number of active users by day of week
4. Sort day of week in correct order

**# Setup:** Run the cell below to create the starting DataFrame of user IDs and timestamps of events logged on the BedBricks website.

```
df = (spark
    .read
    .parquet(eventsPath)
    .select("user_id", col("event_timestamp").alias("ts"))
   )
display(df)
```

**----------Assignment 6 Questions & Answers----------**

**Q.1. Extract timestamp and date of events**

1. Convert ts from microseconds to seconds by dividing by 1 million and cast to timestamp

2. Add date column by converting ts to date

**ANS:**

```
from pyspark.sql.functions import to_date

datetimeDF = (df

        .withColumn("ts", (col("ts") / 1e6).cast(("timestamp")))

        .withColumn("date",to_date(col("ts")))

)

display(datetimeDF)
```

**Q.2. Get daily active users**

- Group by date
- Aggregate approximate count of distinct user_id and alias to "active_users"
- Recall built-in function to get approximate count distinct
- Sort by date
- Plot as line graph

**ANS:**

```
from pyspark.sql.functions import approx_count_distinct,col

activeUsersDF = (datetimeDF

        .groupBy("date")

        .agg(approx_count_distinct("user_id").alias("active_users"))

        .sort(col("date"))

)
```

```
display(activeUsersDF)
```

**Q.3. Get average number of active users by day of week**

- Add day column by extracting day of week from date using a datetime pattern string
- Group by day
- Aggregate average of active_users and alias to "avg_users"

**ANS:**

```
from pyspark.sql.functions import avg,date_format,col
activeDowDF = (activeUsersDF
        .withColumn("day", date_format(col("date"),"E"))
        .groupBy("day")
        .agg(avg("active_users").alias("avg_users"))
)
display(activeDowDF)
```

---------- Day 15 ----------

**NoteBook : ASP 2.3 - Complex Types**

**# Complex Types**

Explore built-in functions for working with collections and strings.

**--Objectives**

Apply collection functions to process arrays

Union DataFrames together

**--Methods**

DataFrame: unionByName

Built-In Functions:

Aggregate: collect_set

Collection: array_contains, element_at, explode

String: split

--In this demo, we're going to use the sales data set.

df = spark.read.parquet(salesPath)

display(df)

**# String Functions**

Here are some of the built-in functions available for manipulating strings.

| Method | Description |
|---|---|
| Translate | Translate any character in the src by a character in replaceString |
| regexp_replace | Replace all substrings of the specified string value that match regexp with rep |
| regexp_extract | Extract a specific group matched by a Java regex, from the specified string column |
| Ltrim | Removes the leading space characters from the specified string column |
| Lower | Converts a string column to lowercase |
| Split | Splits str around matches of the given pattern |

**# Collection Functions**

Here are some of the built-in functions available for working with arrays.

| Method | Description |
|---|---|
| array_contains | Returns null if the array is null, true if the array contains value, and false otherwise. |
| element_at | Returns element of array at given index. Array elements are numbered starting with 1. |
| Explode | Creates a new row for each element in the given array or map column. |
| collect_set | Returns a set of objects with duplicate elements eliminated. |

**# Aggregate Functions**

Here are some of the built-in aggregate functions available for creating arrays, typically from GroupedData.

| Method | Description |
|---|---|
| collect_list | Returns an array consisting of all values within the group. |
| collect_set | Returns an array consisting of all unique values within the group. |

**# User Purchases**

List all size and quality options purchased by each buyer.

1. Extract item details from purchases
2. Extract size and quality options from mattress purchases
3. Extract size and quality options from pillow purchases
4. Combine data for mattress and pillows
5. List all size and quality options bought by each user

**Q.1. Extract item details from purchases**

- Explode the items field in df with the results replacing the existing items field
- Select the email and item.item_name fields
- Split the words in item_name into an array and alias the column to "details"
- Assign the resulting DataFrame to detailsDF

**ANS:**

```
from pyspark.sql.functions import *

detailsDF = (df
        .withColumn("items", explode("items"))
        .select("email", "items.item_name")
        .withColumn("details", split(col("item_name"), " "))
       )

display(detailsDF)
```

**--Write a DF into file**

```
detailsDF.write.json("/FileStore/tables/PRADEEP_pyspark")
```

**--List file**

```
%fs ls /FileStore/tables/PRADEEP_pyspark
```

**Q.2. Extract size and quality options from mattress purchases**

- Filter detailsDF for records where details contains "Mattress"
- Add a size column by extracting the element at position 2
- Add a quality column by extracting the element at position 1
- Save the result as mattressDF.

**ANS:**

```
mattressDF = (detailsDF
```

```
        .filter(array_contains(col("details"), "Mattress"))
        .withColumn("size", element_at(col("details"), 2))
        .withColumn("quality", element_at(col("details"), 1))
       )
display(mattressDF)
```

**Q.3. Extract size and quality options from pillow purchases**

- Filter detailsDF for records where details contains "Pillow"
- Add a size column by extracting the element at position 1
- Add a quality column by extracting the element at position 2
- Save result as pillowDF.

**Note:** the positions of size and quality are switched for mattresses and pillows.

**ANS:**

```
pillowDF = (detailsDF
        .filter(array_contains(col("details"), "Pillow"))
        .withColumn("size", element_at(col("details"), 1))
        .withColumn("quality", element_at(col("details"), 2))
       )
display(pillowDF)
```

**Q.4. Combine data for mattress and pillows**

- Perform a union on mattressDF and pillowDF by column names
- Drop the details column
- Save the result as unionDF.

**Warning:** The DataFrame union method resolves columns by position, as in standard SQL.

You should use it only if the two DataFrames have exactly the same schema, including the column order.

In contrast, the DataFrame unionByName method resolves columns by name.

**ANS:**

```
unionDF = mattressDF.unionByName(pillowDF).drop("details")

display(unionDF)
```

**Q.5. List all size and quality options bought by each user**

- Group rows in unionDF by email
- Collect the set of all items in size for each user and alias the column to "size options"
- Collect the set of all items in quality for each user and alias the column to "quality options"
- Save the result as optionsDF.

**ANS:**

```
optionsDF = (unionDF
         .groupBy("email")
         .agg(collect_set("size").alias("size options"),
              collect_set("quality").alias("quality options"))
        )
display(optionsDF)
```

**--Save in json**

```
optionsDF.write.format("json").save("/FileStore/tables/PRADEEP_pune_json")
```

**--NoteBook: ASP 2.4 - Additional Functions**

**# Additional Functions**

**--Objectives**

-Apply built-in functions to generate data for new columns

-Apply DataFrame NA functions to handle null values

-Join DataFrames

**--Methods**

-DataFrameNaFunctions: fill

-Built-In Functions:

Aggregate: collect_set

Collection: explode

Non-aggregate and miscellaneous: col, lit

**# DataFrameNaFunctions:** DataFrameNaFunctions is a DataFrame submodule with methods for handling null values. Obtain an instance of DataFrameNaFunctions by accessing the na attribute of a DataFrame.

| **Method** | **Description** |
|---|---|
| drop | Returns a new DataFrame omitting rows with any, all, or a specified number of null values, considering an optional subset of columns |
| Fill | Replace null values with the specified value for an optional subset of columns |

| | |
|---|---|
| Replace | Returns a new DataFrame replacing a value with another value, considering an optional subset of columns |

**# Non-aggregate and Miscellaneous Functions**

Here are a few additional non-aggregate and miscellaneous built-in functions.

| Method | Description |
|---|---|
| col / column | Returns a Column based on the given column name. |
| Lit | Creates a Column of literal value |
| Isnull | Return true iff the column is null |
| Rand | Generate a random column with independent and identically distributed (i.i.d.) samples uniformly distributed in [0.0, 1.0) |

**# Joining DataFrames:** The DataFrame join method joins two DataFrames based on a given join expression.

Several different types of joins are supported. For example:

1. Inner join based on equal values of a shared column called 'name' (i.e., an equi join)

df1.join(df2, 'name')

2. Inner join based on equal values of the shared columns called 'name' and 'age'

df1.join(df2, ['name', 'age'])

3. Full outer join based on equal values of a shared column called 'name'

df1.join(df2, 'name', 'outer')

4. Left outer join based on an explicit column expression

df1.join(df2, df1['customer_name'] == df2['account_name'], 'left_outer')

**# Abandoned Carts Lab**

Get abandoned cart items for email without purchases.

Get emails of converted users from transactions

-Join emails with user IDs

-Get cart item history for each user

-Join cart item history with emails

-Filter for emails with abandoned cart items

**--Methods**

-DataFrame: join

-Built-In Functions: collect_set, explode, lit

-DataFrameNaFunctions: fill

**# Setup**

Run the cells below to create DataFrames salesDF, usersDF, and eventsDF.

**%run ./Includes/Classroom-Setup**

**# sale transactions at BedBricks**

salesDF = spark.read.parquet(salesPath)

```
display(salesDF)
```

**# user IDs and emails at BedBricks**

```
usersDF = spark.read.parquet(usersPath)
display(usersDF)
```

**# events logged on the BedBricks website**

```
eventsDF = spark.read.parquet(eventsPath)
display(eventsDF)
```

**----------Assignment 7 Question & Answers----------**

**Q.1-A: Get emails of converted users from transactions**

- Select the email column in salesDF and remove duplicates
- Add a new column converted with the value True for all rows
- Save the result as convertedUsersDF.

**ANS:**

```
from pyspark.sql.functions import *
convertedUsersDF = (salesDF.drop_duplicates(["email"])
                    .withColumn("converted",(lit("true").cast("boolean")))
                    .select("email","converted")
)
display(convertedUsersDF)
```

**Q.2-A: Join emails with user IDs**

- Perform an outer join on convertedUsersDF and usersDF with the email field
- Filter for users where email is not null
- Fill null values in converted as False
- Save the result as conversionsDF.

**ANS:**

```
conversionsDF = (usersDF.join(convertedUsersDF,"email", "outer")
        .filter(col("email").isNotNull())
        .fillna(value="False",subset="converted")
)
display(conversionsDF)
```

**Q.3-A: Get cart item history for each user**

- Explode the items field in eventsDF with the results replacing the existing items field
- Group by user_id
- Collect a set of all items.item_id objects for each user and alias the column to "cart"
- Save the result as cartsDF.

**ANS:**

```
cartsDF = (eventsDF.withColumn("items", explode(col("items")))
        .groupBy(col("user_id"))
        .agg(collect_set("items.item_id").alias("cart"))
)
display(cartsDF)
```

**Q.4-A: Join cart item history with emails**

- Perform a left join on conversionsDF and cartsDF on the user_id field
- Save result as emailCartsDF.

**ANS:**

```
emailCartsDF = conversionsDF.join(cartsDF,"user_id","left_outer")

display(emailCartsDF)
```

**Q.5-A: Filter for emails with abandoned cart items**

- Filter emailCartsDF for users where converted is False
- Filter for users with non-null carts
- Save result as abandonedItemsDF.

**ANS:**

```
abandonedCartsDF = (emailCartsDF
                    .filter("converted == False")
                    .filter(col("cart").isNotNull())
)

display(abandonedCartsDF)
```

**Q.6-A: Bonus Activity**

Plot number of abandoned cart items by product

**ANS:**

```
abandonedItemsDF = (abandonedCartsDF
                    .select(col("cart","items")
                    .groupBy("cart")
                   )
```

```
display(abandonedItemsDF)
```

----------Day 16 ----------

**NoteBook : ASP 2.5 - UDFs**

**# User-Defined Functions**

**--Objectives**

-Define a function

-Create and apply a UDF

-Register the UDF to use in SQL

-Create and register a UDF with Python decorator syntax

-Create and apply a Pandas (vectorized) UDF

**--Methods**

-UDF Registration (spark.udf): register

-Built-In Functions: udf

-Python UDF Decorator: @udf

-Pandas UDF Decorator: @pandas_udf

**--Define a function:** Define a function (on the driver) to get the first letter of a string from the email field.

```
def firstLetterFunction(email):
  return email[0]

firstLetterFunction("annagray@kaufman.com")
```

**--Create and apply UDF:** Register the function as a UDF. This serializes the function and sends it to executors to be able to transform DataFrame records.

```
firstLetterUDF = udf(firstLetterFunction)
```

**--Show function type**

```
print(type(firstLetterUDF))
```

**--Apply the UDF on the email column in exist data.**

```
from pyspark.sql.functions import col

display(salesDF.select(firstLetterUDF(col("email"))))
```

**--Count**

```
salesDF.count()
```

**# Register UDF to use in SQL**

Register the UDF using spark.udf.register to also make it available for use in the SQL namespace.

```
salesDF.createOrReplaceTempView("sales")

firstLetterUDF = spark.udf.register("sql_udf", firstLetterFunction)
```

**# You can still apply the UDF from Python**

```
display(salesDF.select(firstLetterUDF(col("email"))))
```

**# Apply UDF on sql**

%sql

**--You can now also apply the UDF from SQL**

SELECT sql_udf(email), email AS firstLetter FROM sales

**# Use Decorator Syntax (Python Only)**

1. Alternatively, you can define and register a UDF using Python decorator syntax. The @udf decorator parameter is the Column datatype the function returns.

2. You will no longer be able to call the local Python function (i.e., firstLetterUDF("annagray@kaufman.com") will not work).

**Note:** This example also uses Python type hints, which were introduced in Python 3.5.

Type hints are not required for this example, but instead serve as documentation" to help developers use the function correctly. They are used in this example to emphasize that the UDF processes one record at a time, taking a single str argument and returning a str value.

**# Our input/output is a string**

@udf("string")

def firstLetterUDF(email: str) -> str:

    return email[0]

**# And let's use our decorator UDF here.**

from pyspark.sql.functions import col

**salesDF** = spark.read.parquet("/mnt/training/ecommerce/sales/sales.parquet")

display(salesDF.select(firstLetterUDF(col("email"))))

---------- **Assignment 8 Questions & Answers** ----------

**--Q.1. Define UDF to label day of week**

Use the labelDayOfWeek function provided below to create the UDF labelDowUDF

```
def labelDayOfWeek(day: str) -> str:
  dow = {"Mon": "1", "Tue": "2", "Wed": "3", "Thu": "4",
       "Fri": "5", "Sat": "6", "Sun": "7"}
  return dow.get(day) + "-" + day
```

**ANS:**

labelDowUDF = udf(labelDayOfWeek)

**--Q.2. Apply UDF to label and sort by day of week**

-Update the day column by applying the UDF and replacing this column

-Sort by day

-Plot as a bar graph

**ANS:**

finalDF = df.select("day",labelDowUDF(col("day")).alias("updated_day"))

display(finalDF)

**--Homework:**

**1. Write a email validation function and separate username and domain and mask username:**

**ANS:**

# After email validation -> if email is valid, mask username & domain part, if not vali keep as it is. & Split username & domains to store in new columns of username & domain.

**# create a DF from salesPath**

```
df = spark.read.parquet(salesPath)
display(df)
```

**# create function to verify email**

```
def verify_email(email):
  import re
  regex = '^\w+([\.-]?\w+)@\w+([\.-]?\w+)(\.\w{2,3})+$'
    if(re.search(regex,email)):
    return True
  else:
    return False
```

**# Convert python function to UDF function**

```
verify_email = udf(verify_email)
```

**# used column to verify email**

```
from pyspark.sql.functions import col
```

```
# result of email validation store into new DF as df1
df1 = df.select("email",verify_email(col("email"))
      )
display(df1)

# define a udf for split username
def split_username(email):
   if '@' in email:
      return email.split('@')[0]
   else:
      return email

# define a udf for split domain
def split_domain(email):
   if '@' in email:
      return email.split('@')[1]
   else:
      return email

# register the udf
split_username_udf = udf(split_username)
split_domain_udf = udf(split_domain)

# apply the udf
```

```
df_result = (df1.withColumn('Username', split_username_udf("email"))
        .withColumn("Domain",split_domain_udf("email"))
        )
display(df_result)
```

**# Import functions and perform masking based on condition**

```
from pyspark.sql.functions import lit, col, when
masked_df = df_result.select(
   col("email"),
   col("Username"),
   col("Domain"),
   col("verify_email(email)"),
   when(col("verify_email(email)") == lit("true"),
lit("**")).otherwise(col("Username")).alias("masked_username")
)
```

**# Final_Output**

```
display(masked_df)
```

**2.Phone validation with country & masking on first digit after country code with 8**

**ANS:**

```
pdf = (spark.read
        .option("sep", ",")
        .option("header", True)
```

```
        .option("inferSchema", True)
        .csv("/FileStore/tables/demo_data-1.csv")
       )
display(pdf)


import re
def verify_phone(phone):
  # Define the regular expression pattern for phone numbers
  phone_regex = r'^\+?\d{1,3}[- ]?\d{3,4}[- ]?\d{4}$'

  # Use the re.match() function to check if the phone number matches the pattern
  if re.match(phone_regex, phone):
    return True
  else:
    return False

#Convert python function to UDF function
verify_phone = udf(verify_phone)

df1 = (pdf
       .select("phone", verify_phone(col("phone")).alias("valid_phone"))
       .filter("valid_phone == 'true'")
      )
#display(df1)
```

```
from pyspark.sql.functions import substring, concat

# Use substring() to extract the first digit and concat()

df1 = df1.withColumn('Masked_column', concat(lit('8'),substring('phone', 2, 100)))

# Display the masked DataFrame

display(df1)
```

**-- What is sql namespace**

In computer programming, a namespace is a collection of symbols that are used to organize code in a manageable way. In the context of SQL, namespaces are used to help organize and distinguish objects like tables, views, functions, and stored procedures. They can also help to ensure that similar objects have unique names.

**-- Sql tablespace**

A tablespace is a logical container in a relational database that is used to group related logical structures together. In SQL, tablespaces are used to store tables and indexes. Tablespaces can also be used to manage the physical placement of database files on disk.

---------- Day 17 ----------

**--NoteBook : ASP 3.3 - Partitioning**

**# Partitioning**

**--Objectives**

- Get partitions and cores
- Repartition DataFrames
- Configure default shuffle partitions

**--Methods**

-DataFrame: repartition, coalesce, rdd.getNumPartitions

-SparkConf: get, set

-SparkSession: spark.sparkContext.defaultParallelism

--SparkConf Parameters

-spark.sql.shuffle.partitions, spark.sql.adaptive.enabled

**--Classroom setup**

%run ./Includes/Classroom-Setup

**--Get partitions and cores:** Use the rdd method getNumPartitions to get the number of DataFrame partitions.

```
df = spark.read.parquet(eventsPath)
df.rdd.getNumPartitions()
```

**--Write as table**

```
df.coalesce(1).write.mode("overwrite")saveAsTable("default.events")
```

**--Read Table**

```
eventsDF = spark.read.table("default.events")
display(eventsDF)
```

**--Select device & ecommerce column**

```
eventsDF1 = spark.sql("select device, ecommerce from default.events")
```

```
eventsDF1.count() #count in eventsDF

spark.sql("select count(*) from default.events").show() #count in spark.sql
```

**#Views in Spark/PySpark**

```
eventsDF.createTempView("events_view")
```

**#Local View -> Available to user who created on the cluster.**

```
eventsDF.createOrReplaceTempView("events_view")
```

**#Gloab View -> Available to all users on Cluster.**

```
eventsDF.createOrReplaceGlobalTempView("events_view_global")
```

**--To know type of events**

```
print(type("events_view"))
```

**--select all from view**

```
spark.sql("select * from events_view").show()
```

**--Read file and show partitions**

```
df1 = spark.read.parquet("/FileStore/tables/PRADEEP_part_demo/part-00000-tid-5992040760267038782-dc61ca8a-b554-47b8-8e09-24f59a517b66-57-1-c000.snappy.parquet")

df1.rdd.getNumPartitions()
```

**--Note:** Under Utilization of Cluster / Resources. Default Parallalism = default slots (parallel threads) available to process data => here it is 8.

**--To check default paralism**

print(spark.sparkContext.defaultParallelism) or

print(sc.defaultParallelism)

**# repartition**

Returns a new DataFrame that has exactly n partitions.

- Wide transformation
- Pro: Evenly balances partition sizes
- Con: Requires shuffling all data

**--Make repartition 8**

repartitionedDF = df.repartition(8)

**--To check partition**

repartitionedDF.rdd.getNumPartitions()

**# coalesce**

Returns a new DataFrame that has exactly n partitions, when fewer partitions are requested.

If a larger number of partitions is requested, it will stay at the current number of partitions.

-Narrow transformation, some partitions are effectively concatenated

-Pro: Requires no shuffling

**-Cons:**

Is not able to increase partitions

Can result in uneven partition sizes

**- create 8 partition using colasce**

coalesceDF = df.coalesce(8) Note: it shows original (as it is) because we cannot create more partition than exists using coalesce.

**--To check partition**

coalesceDF.rdd.getNumPartitions()

**# Configure default shuffle partitions**

Use the SparkSession's conf attribute to get and set dynamic Spark configuration properties.

The spark.sql.shuffle.partitions property determines the number of partitions that result from a shuffle.

Let's check its default value:

spark.conf.get("spark.sql.shuffle.partitions")

**--Assuming that the data set isn't too large, you could configure the default number of shuffle.**

partitions to match the number of cores:

spark.conf.set("spark.sql.shuffle.partitions",
spark.sparkContext.defaultParallelism)

print(spark.conf.get("spark.sql.shuffle.partitions"))

**# Partitioning Guidelines**

- Make the number of partitions a multiple of the number of cores
- Target a partition size of ~200MB
- Size default shuffle partitions by dividing largest shuffle stage input by the target partition size (e.g., 4TB / 200MB = 20,000 shuffle partition count)

**--Enable AQE: #AQE is a optimize partitions**

spark.conf.get("spark.sql.adaptive.enabled")

**--Disable AQE**

spark.conf.get("spark.sql.adaptive.enabled","False")

---

**-- NoteBook: ASP 3.4 - Review**

DataFrames and Transformations Review

De-Duping Data Lab

In this exercise, we're doing ETL on a file we've received from a customer. That file contains data about people, including:

1.first, middle and last names

2.gender

3.birth date

4.Social Security number

5.salary

But, as is unfortunately common in data we get from this customer, the file contains some duplicate records. Worse:

In some of the records, the names are mixed case (e.g., "Carol"), while in others, they are uppercase (e.g., "CAROL").

The Social Security numbers aren't consistent either. Some of them are hyphenated (e.g., "992-83-4829"), while others are missing hyphens ("992834829").

If all of the name fields match -- if you disregard character case -- then the birth dates and salaries are guaranteed to match as well, and the Social Security Numbers would match if they were somehow put in the same format.

Your job is to remove the duplicate records. The specific requirements of your job are:

-Remove duplicates. It doesn't matter which record you keep; it only matters that you keep one of them.

-Preserve the data format of the columns. For example, if you write the first name column in all lowercase, you haven't met this requirement.

Hint: The initial dataset contains 103,000 records. The de-duplicated result has 100,000 records.

Next, write the results in Delta format as a single data file to the directory given by the variable deltaDestDir.

Hint: Remember the relationship between the number of partitions in a DataFrame and the number of files written.

**--Methods**

- DataFrameReader
- DataFrame
- Built-In Functions
- DataFrameWriter

--It's helpful to look at the file first, so you can check the format. dbutils.fs.head() (or just %fs head) is a big help here.

%fs head dbfs:/mnt/training/dataframes/people-with-dups.txt

**--ETL ANSWER:**

```
sourceFile = "dbfs:/mnt/training/dataframes/people-with-dups.txt"
destFile = workingDir + "/people.parquet"

# In case it already exists
dbutils.fs.rm(destFile, True)

# Complete your work here...
from pyspark.sql.functions import lower, when, length, concat_ws
from pyspark.sql import SparkSession
```

```
# create a SparkSession
spark = SparkSession.builder.appName("DuplicateRemover").getOrCreate()

# load the data into DF
df = spark.read.option("delimiter", ":").csv(sourceFile, header=True)

# normalize the case of the name fields by lowercase in two ways
# first_way:
 # df = df.withColumn("firstName", lower(df["firstName"]))
 # df = df.withColumn("middleName", lower(df["middleName"]))
 # df = df.withColumn("lastName", lower(df["lastName"]))

# second_way:
df = (df.withColumn("firstName", lower(df["firstName"]))
      .withColumn("middleName", lower(df["middleName"]))
      .withColumn("lastName", lower(df["lastName"])))

# normalize the (ssn) Social Security numbers
df = df.withColumn("ssn", when(length(df["ssn"]) == 9,
                        concat_ws("-",
                              df["ssn"].substr(1, 3),
                              df["ssn"].substr(4, 2),
                              df["ssn"].substr(6, 4)))
```

```
                .otherwise(df["ssn"]))
```

**# drop duplicates**

```
df = df.dropDuplicates(["firstName", "middleName", "lastName", "gender", "birthDate", "salary", "ssn"])

display(df)
```

**# check your result by counting rows it should be 1,00,000**

```
df1 = df.count()

display(df1)
```

**# write the results in Delta format as a single data file**

```
(df.write.format("delta")
  .mode("overwrite")
  .option("compression", "snappy")
  .option("maxRecordsPerFile", 1000000)
  .option("singleFile", "true")
  .save(destFile))
```

---------- Day 18 ----------

**--HomeWork:** read sales data in table and partition based on order id and name should be order_id follwed by order_id

**ANS:**

```
import os
```

```
from datetime import datetime
# Get current date and time
current_time = datetime.now().strftime("%Y-%m-%d_%H%M%S")
# Path to files
path = "/FileStore/part4/"
# Get all files in the directory
files = dbutils.fs.ls(path)

# Loop over the files
for f in files:
  # Get the directory of each file
  dirs = f.name

  # If the directory starts with 'user_id'
  if dirs.startswith("user_id"):
    # Get the new directory
    new_dir = path + dirs

    # Get all files in the new directory
    outputFile = dbutils.fs.ls(new_dir)

    # Loop over the files
    for i in outputFile:
      # Get the file name
```

```
file1 = i.name

# If the file ends with '.parquet'
if file1.endswith(".parquet"):
  # Get the source directory of the file
  srcDir = new_dir + file1

  # Get the destination directory of the file
  destDir = new_dir + current_time + ".parquet"

  # Move the file
  dbutils.fs.mv(srcDir,destDir)
```

---

**--NoteBook: DE 0.1 - Spark SQL**

Spark SQL

Demonstrate fundamental concepts in Spark SQL using the DataFrame API.

**--Objectives**

1.Run a SQL query

2.Create a DataFrame from a table

3.Write the same query using DataFrame transformations

4.Trigger computation with DataFrame actions

5.Convert between DataFrames and SQL

**--Methods**

-SparkSession: sql, table

-DataFrame:

Transformations: select, where, orderBy

Actions: show, count, take

Other methods: printSchema, schema, createOrReplaceTempView

%run ../Includes/Classroom-Setup-00.1

**--Multiple Interfaces**

Spark SQL is a module for structured data processing with multiple interfaces.

We can interact with Spark SQL in two ways:

1. Executing SQL queries
2. Working with the DataFrame API.

**--Method 1:** Executing SQL queries

This is a basic SQL query.

```
%sql
SELECT name, price
FROM products
WHERE price < 200
ORDER BY price
```

**--Method 2:** Working with the DataFrame API

We can also express Spark SQL queries using the DataFrame API. The following cell returns a DataFrame containing the same results as those retrieved above.

```
display(spark
      .table("products")
      .select("name", "price")
      .where("price < 200")
      .orderBy("price")
     )
```

**--Spark API Documentation**

To learn how we work with DataFrames in Spark SQL, let's first look at the Spark API documentation. The main Spark documentation page includes links to API docs and helpful guides for each version of Spark.

The Scala API and Python API are most commonly used, and it's often helpful to reference the documentation for both languages. Scala docs tend to be more comprehensive, and Python docs tend to have more code examples.

Navigating Docs for the Spark SQL Module

Find the Spark SQL module by navigating to org.apache.spark.sql in the Scala API or pyspark.sql in the Python API. The first class we'll explore in this module is the SparkSession class. You can find this by entering "SparkSession" in the search bar.

**--SparkSession**

The SparkSession class is the single entry point to all functionality in Spark using the DataFrame API.

In Databricks notebooks, the SparkSession is created for you, stored in a variable called spark.

--Below are several additional methods we can use to create DataFrames. All of these can be found in the documentation for SparkSession.

--SparkSession Methods

| Method | Description |
|---|---|
| sql | Returns a DataFrame representing the result of the given query |
| table | Returns the specified table as a DataFrame |
| read | Returns a DataFrameReader that can be used to read data in as a DataFrame |
| range | Create a DataFrame with a column containing elements in a range from start to end (exclusive) with step value and number of partitions |
| createDataFrame | Creates a DataFrame from a list of tuples, primarily used for testing |

--Let's use a SparkSession method to run SQL.

```
result_df = spark.sql("""
SELECT name, price
FROM products
WHERE price < 200
ORDER BY price
""")
```

```
display(result_df)
```

**--DataFrames**

Recall that expressing our query using methods in the DataFrame API returns results in a DataFrame. Let's store this in the variable budget_df.

A DataFrame is a distributed collection of data grouped into named columns.

```
budget_df = (spark
        .table("products")
        .select("name", "price")
        .where("price < 200")
        .orderBy("price")
       )
```

The schema defines the column names and types of a dataframe. Access a dataframe's schema using the schema attribute.

1. budget_df.schema
2. budget_df.printSchema()

**--Transformations**

When we created budget_df, we used a series of DataFrame transformation methods e.g. select, where, orderBy.

```
products_df
  .select("name", "price")
  .where("price < 200")
```

```
 .orderBy("price")
```

Transformations operate on and return DataFrames, allowing us to chain transformation methods together to construct new DataFrames. However, these operations can't execute on their own, as transformation methods are lazily evaluated.

Running the following cell does not trigger any computation.

**--Actions**

Conversely, DataFrame actions are methods that trigger computation. Actions are needed to trigger the execution of any DataFrame transformations.

The show action causes the following cell to execute transformations.

```
(products_df
 .select("name", "price")
 .where("price < 200")
 .orderBy("price")
 .show())
```

Below are several examples of DataFrame actions.

DataFrame Action Methods

| Method | Description |
|---|---|
| show | Displays the top n rows of DataFrame in a tabular form |
| Count | Returns the number of rows in the DataFrame |
| describe, summary | Computes basic statistics for numeric and string columns |
| first, head | Returns the the first row |

| | |
|---|---|
| Collect | Returns an array that contains all rows in this DataFrame |
| Take | Returns an array of the first n rows in the DataFrame |

**--Convert between DataFrames and SQL**

createOrReplaceTempView creates a temporary view based on the DataFrame. The lifetime of the temporary view is tied to the SparkSession that was used to create the DataFrame.

```
budget_df.createOrReplaceTempView("budget")

display(spark.sql("SELECT * FROM budget"))
```

---

**--NoteBook : DE 0.2L - Spark SQL Lab**

**--Spark SQL Lab**

**Tasks**

1. Create a DataFrame from the events table
2. Display the DataFrame and inspect its schema
3. Apply transformations to filter and sort macOS events
4. Count results and take the first 5 rows
5. Create the same DataFrame using a SQL query

**--Methods**

1. SparkSession: sql, table
2. DataFrame transformations: select, where, orderBy
3. DataFrame actions: select, count, take
4. Other DataFrame methods: printSchema, schema, createOrReplaceTempView

```
%run ../Includes/Classroom-Setup-00.2L
```

**1. Create a DataFrame from the events table**

Use SparkSession to create a DataFrame from the events table

```
events_df = spark.table("events")
```

**2. Display DataFrame and inspect schema**

Use methods above to inspect DataFrame contents and schema

```
events_df.schema
```

**3. Apply transformations to filter and sort macOS events**

- Filter for rows where device is macOS
- Sort rows by event_timestamp
- Hint Use single and double quotes in your filter SQL expression

```
mac_df = (events_df
        .select("device","event_timestamp")
        .where("device == 'macOS'")
      .orderBy("event_timestamp")
       )
display(mac_df)
```

**#second_way_through_table**

```
mac_df = spark.sql("""
SELECT device,event_timestamp FROM events
```

```
WHERE device = 'macOS'
ORDER BY event_timestamp asc
""")

display(mac_df)
```

**4. Count results and take first 5 rows**

Use DataFrame actions to count and take rows

```
num_rows = mac_df.take(5)
rows = mac_df.count()
display(rows)
```

**5. Create the same DataFrame using SQL query**

Use SparkSession to run a SQL query on the events table

Use SQL commands to write the same filter and sort query used earlier

```
mac_sql_df = spark.sql("""
        select device,event_timestamp from events
        where device="macOS"
        order by event_timestamp
        """)

display(mac_sql_df.take(5))
```

---

**--NoteBook : DE 0.3 - DataFrame & Column**

**--DataFrame & Column**

**-Objectives**

1. Construct columns
2. Subset columns
3. Add or replace columns
4. Subset rows
5. Sort rows

**-Methods**

1. DataFrame: select, selectExpr, drop, withColumn, withColumnRenamed, filter, distinct, limit, sort
2. Column: alias, isin, cast, isNotNull, desc, operators

```
%run ../Includes/Classroom-Setup-00.3
```

```
events_df = spark.table("events")
display(events_df)
```

**--Column Expressions**

A Column is a logical construction that will be computed based on the data in a DataFrame using an expression

Construct a new Column based on existing columns in a DataFrame

```
from pyspark.sql.functions import col
```

```
print(events_df.device)

print(events_df["device"])

print(col("device"))
```

**--Column Operators and Methods**

| Method | Description |
|---|---|
| *, + , <, >= | Math and comparison operators |
| ==, != | Equality and inequality tests (Scala operators are === and =!=) |
| Alias | Gives the column an alias |
| cast, astype | Casts the column to a different data type |
| isNull, isNotNull, isNan | Is null, is not null, is NaN |
| asc, desc | Returns a sort expression based on ascending/descending order of the column |

**--Create complex expressions with existing columns, operators, and methods.**

```
col("ecommerce.purchase_revenue_in_usd") +
col("ecommerce.total_item_quantity")

col("event_timestamp").desc()

(col("ecommerce.purchase_revenue_in_usd") * 100).cast("int")
```

**--Here's an example of using these column expressions in the context of a DataFrame**.

```
rev_df = (events_df
    .filter(col("ecommerce.purchase_revenue_in_usd").isNotNull())
```

```
        .withColumn("purchase_revenue",
(col("ecommerce.purchase_revenue_in_usd") * 100).cast("int"))
        .withColumn("avg_purchase_revenue",
col("ecommerce.purchase_revenue_in_usd") /
col("ecommerce.total_item_quantity"))
        .sort(col("avg_purchase_revenue").desc())
       )

display(rev_df)
```

--DataFrame Transformation Methods

| **Method** | **Description** |
|---|---|
| select | Returns a new DataFrame by computing given expression for each element |
| drop | Returns a new DataFrame with a column dropped |
| withColumnRenamed | Returns a new DataFrame with a column renamed |
| withColumn | Returns a new DataFrame by adding a column or replacing the existing column that has the same name |
| filter, where | Filters rows using the given condition |
| sort, orderBy | Returns a new DataFrame sorted by the given expressions |
| dropDuplicates, distinct | Returns a new DataFrame with duplicate rows removed |
| Limit | Returns a new DataFrame by taking the first n rows |

| | |
|---|---|
| groupBy | Groups the DataFrame using the specified columns, so we can run aggregation on them |

**--Subset columns**

Use DataFrame transformations to subset columns

**--select()**

Selects a list of columns or column based expressions

```
devices_df = events_df.select("user_id", "device")
display(devices_df)
```

```
from pyspark.sql.functions import col
locations_df = events_df.select(
   "user_id",
   col("geo.city").alias("city"),
   col("geo.state").alias("state")
)
display(locations_df)
```

**--selectExpr()**

Selects a list of SQL expressions

```
apple_df = events_df.selectExpr("user_id", "device in ('macOS', 'iOS') as apple_user")
display(apple_df)
```

**--drop()**

Returns a new DataFrame after dropping the given column, specified as a string or Column object.

Use strings to specify multiple columns

anonymous_df = events_df.drop("user_id", "geo", "device")

display(anonymous_df)

no_sales_df = events_df.drop(col("ecommerce"))

display(no_sales_df)

**--Add or replace columns**

Use DataFrame transformations to add or replace columns

**--withColumn()**

Returns a new DataFrame by adding a column or replacing an existing column that has the same name.

mobile_df = events_df.withColumn("mobile", col("device").isin("iOS", "Android"))

display(mobile_df)

purchase_quantity_df = events_df.withColumn("purchase_quantity", col("ecommerce.total_item_quantity").cast("int"))

purchase_quantity_df.printSchema()

**--withColumnRenamed()**

Returns a new DataFrame with a column renamed.

```
location_df = events_df.withColumnRenamed("geo", "location")
display(location_df)
```

**--Subset Rows**

Use DataFrame transformations to subset rows

**--filter()**

Filters rows using the given SQL expression or column based condition.

Alias: where

```
purchases_df = events_df.filter("ecommerce.total_item_quantity > 0")
display(purchases_df)
```

```
revenue_df = events_df.filter(col("ecommerce.purchase_revenue_in_usd").isNotNull())
display(revenue_df)
```

```
android_df = events_df.filter((col("traffic_source") != "direct") & (col("device") == "Android"))
display(android_df)
```

**--dropDuplicates()**

Returns a new DataFrame with duplicate rows removed, optionally considering only a subset of columns.

Alias: distinct

```
display(events_df.distinct())

distinct_users_df = events_df.dropDuplicates(["user_id"])

display(distinct_users_df)
```

**--limit()**

Returns a new DataFrame by taking the first n rows.

```
limit_df = events_df.limit(100)

display(limit_df)
```

**--Sort rows**

Use DataFrame transformations to sort rows

sort()

Returns a new DataFrame sorted by the given columns or expressions.

Alias: orderBy

```
increase_timestamps_df = events_df.sort("event_timestamp")

display(increase_timestamps_df)

decrease_timestamp_df = events_df.sort(col("event_timestamp").desc())

display(decrease_timestamp_df)

increase_sessions_df = events_df.orderBy(["user_first_touch_timestamp",
"event_timestamp"])
```

```
display(increase_sessions_df)


decrease_sessions_df = events_df.sort(col("user_first_touch_timestamp").desc(),
col("event_timestamp"))


----------------------------------------
```

**--NoteBook : DE 0.4L - Purchase Revenues Lab**

**--Purchase Revenues Lab**

Prepare dataset of events with purchase revenue.

**Tasks**

1. Extract purchase revenue for each event
2. Filter events where revenue is not null
3. Check what types of events have revenue
4. Drop unneeded column

**--Methods**

1. DataFrame: select, drop, withColumn, filter, dropDuplicates
2. Column: isNotNull

```
%run ../Includes/Classroom-Setup-00.4L


events_df = spark.table("events")

display(events_df)
```

**1. Extract purchase revenue for each event**

Add new column revenue by extracting ecommerce.purchase_revenue_in_usd

```
from pyspark.sql.functions import col
revenue_df =
(events_df.withColumn("revenue",col("ecommerce.purchase_revenue_in_usd"))
        )
display(revenue_df)
```

**2. Filter events where revenue is not null**

Filter for records where revenue is not null

```
purchases_df = revenue_df.filter(col("revenue").isNotNull())
display(purchases_df)
```

**3. Check what types of events have revenue**

Find unique event_name values in purchases_df in one of two ways:

Select "event_name" and get distinct records

Drop duplicate records based on the "event_name" only

Hint There's only one event associated with revenues

```
distinct_df = purchases_df.select("event_name").distinct()
display(distinct_df)
```

**4. Drop unneeded column**

Since there's only one event type, drop event_name from purchases_df.

```
final_df = purchases_df.drop("event_name")
display(final_df)
```

**5. Chain all the steps above excluding step 3**

```
from pyspark.sql.functions import col
final_df = (events_df
        .withColumn("revenue",col("ecommerce.purchase_revenue_in_usd"))
        .filter(col("revenue").isNotNull())
        .drop("event_name")
       )

display(final_df)
```

---

**--NoteBook : DE 0.5 - Aggregation**

Aggregation

**--Objectives**

- Group data by specified columns
- Apply grouped data methods to aggregate data
- Apply built-in functions to aggregate data

**--Methods**

- DataFrame: groupBy
- Grouped Data: agg, avg, count, max, sum
- Built-In Functions: approx_count_distinct, avg, sum

```
%run ../Includes/Classroom-Setup-00.5

df = spark.table("events")
```

display(df)

**-groupBy**

Use the DataFrame groupBy method to create a grouped data object.

This grouped data object is called RelationalGroupedDataset in Scala and GroupedData in Python.

df.groupBy("event_name")

df.groupBy("geo.state", "geo.city")

**--Grouped data methods**

Various aggregation methods are available on the GroupedData object.

| Method | Description |
|---|---|
| Agg | Compute aggregates by specifying a series of aggregate columns |
| avg | Compute the mean value for each numeric columns for each group |
| Count | Count the number of rows for each group |
| max | Compute the max value for each numeric columns for each group |
| Mean | Compute the average value for each numeric columns for each group |
| Min | Compute the min value for each numeric column for each group |
| Pivot | Pivots a column of the current DataFrame and performs the specified aggregation |
| Sum | Compute the sum for each numeric columns for each group |

```
event_counts_df = df.groupBy("event_name").count()

display(event_counts_df)
```

**--Here, we're getting the average purchase revenue for each.**

```
avg_state_purchases_df =
df.groupBy("geo.state").avg("ecommerce.purchase_revenue_in_usd")

display(avg_state_purchases_df)
```

**--And here the total quantity and sum of the purchase revenue for each combination of state and city.**

```
city_purchase_quantities_df = df.groupBy("geo.state",
"geo.city").sum("ecommerce.total_item_quantity",
"ecommerce.purchase_revenue_in_usd")

display(city_purchase_quantities_df)
```

**--Built-In Functions**

In addition to DataFrame and Column transformation methods, there are a ton of helpful functions in Spark's built-in SQL functions module.

In Scala, this is org.apache.spark.sql.functions, and pyspark.sql.functions in Python. Functions from this module must be imported into your code.

**--Aggregate Functions**

Here are some of the built-in functions available for aggregation.

| Method | Description |
|---|---|
| approx_count_distinct | Returns the approximate number of distinct items in a group |

| | |
|---|---|
| Avg | Returns the average of the values in a group |
| collect_list | Returns a list of objects with duplicates |
| corr | Returns the Pearson Correlation Coefficient for two columns |
| Max | Compute the max value for each numeric columns for each group |
| Mean | Compute the average value for each numeric columns for each group |
| stddev_samp | Returns the sample standard deviation of the expression in a group |
| sumDistinct | Returns the sum of distinct values in the expression |
| var_pop | Returns the population variance of the values in a group |

Use the grouped data method agg to apply built-in aggregate functions

This allows you to apply other transformations on the resulting columns, such as alias.

```
from pyspark.sql.functions import sum

state_purchases_df =
df.groupBy("geo.state").agg(sum("ecommerce.total_item_quantity").alias("total_
purchases"))

display(state_purchases_df
```

**--Apply multiple aggregate functions on grouped data**

```
from pyspark.sql.functions import avg, approx_count_distinct


state_aggregates_df = (df
```

```
    .groupBy("geo.state")
    .agg(avg("ecommerce.total_item_quantity").alias("avg_quantity"),
        approx_count_distinct("user_id").alias("distinct_users"))
)

display(state_aggregates_df)
```

**--Math Functions**

Here are some of the built-in functions for math operations.

| Method | Description |
|---|---|
| Ceil | Computes the ceiling of the given column. |
| Cos | Computes the cosine of the given value. |
| Log | Computes the natural logarithm of the given value. |
| round | Returns the value of the column e rounded to 0 decimal places with HALF_UP round mode. |
| Sqrt | Computes the square root of the specified float value. |

```
from pyspark.sql.functions import cos, sqrt

display(spark.range(10)  # Create a DataFrame with a single column called "id" with a range of integer values
    .withColumn("sqrt", sqrt("id"))
    .withColumn("cos", cos("id"))
)
```

---

**--NoteBook : DE 0.6L - Revenue by Traffic Lab**

Revenue by Traffic Lab

Get the 3 traffic sources generating the highest total revenue.

- Aggregate revenue by traffic source
- Get top 3 traffic sources by total revenue
- Clean revenue columns to have two decimal places

**--Methods**

- DataFrame: groupBy, sort, limit
- Column: alias, desc, cast, operators
- Built-in Functions: avg, sum

```
%run ../Includes/Classroom-Setup-00.6L
```

**--Setup**

Run the cell below to create the starting DataFrame df.

```
from pyspark.sql.functions import col
# Purchase events logged on the BedBricks website
df = (spark.table("events")
    .withColumn("revenue", col("ecommerce.purchase_revenue_in_usd"))
    .filter(col("revenue").isNotNull())
```

```
    .drop("event_name")
   )

display(df)
```

**--Q.1. Aggregate revenue by traffic source**

**Group by traffic_source**

**Get sum of revenue as total_rev. Round this to the tens decimal place (e.g. nnnnn.n).**

**Get average of revenue as avg_rev**

**Remember to import any necessary built-in functions.**

```
from pyspark.sql.functions import round, sum, avg
traffic_df = (df.groupBy("traffic_source")
          .agg(round(sum("revenue"),1).alias("total_rev"),
             round(avg("revenue"),4).alias("avg_rev"))
         )

display(traffic_df)
```

**--Q.2. Get top three traffic sources by total revenue**

**Sort by total_rev in descending order**

**Limit to first three rows**

```
from pyspark.sql.functions import col
```

```
top_traffic_df = (traffic_df.sort(col("total_rev").desc()).limit(3)
                  )
display(top_traffic_df)
```

**--Q.3. Limit revenue columns to two decimal places**

**Modify columns avg_rev and total_rev to contain numbers with two decimal places**

**Use withColumn() with the same names to replace these columns**

**To limit to two decimal places, multiply each column by 100, cast to long, and then divide by 100**

```
from pyspark.sql.functions import col, round
from pyspark.sql.types import LongType

final_df = top_traffic_df.withColumn("total_rev", col("total_rev") * 100)\
                         .withColumn("avg_rev", col("avg_rev") * 100)\
                         .withColumn("total_rev", col("total_rev").cast(LongType()) / 100)\
                         .withColumn("avg_rev", col("avg_rev").cast(LongType()) / 100)

display(final_df)
```

**--Q.4. Bonus: Rewrite using a built-in math function**

Find a built-in math function that rounds to a specified number of decimal places

```
from pyspark.sql.functions import col, round
```

```
bonus_df = top_traffic_df.withColumn("total_rev", round(col("total_rev"), 2))\
                         .withColumn("avg_rev", round(col("avg_rev"), 2))

display(bonus_df)
```

**--Q.5 Chain all the steps above**

```
# TODO
chain_df = (df
            .groupBy("traffic_source")
            .agg(round(sum("revenue"),1).alias("total_rev"),
            round(avg("revenue"),4).alias("avg_rev"))
            .sort(col("total_rev").desc()).limit(3)
            .withColumn("total_rev", col("total_rev") * 100)
            .withColumn("avg_rev", col("avg_rev") * 100)
            .withColumn("total_rev", col("total_rev").cast(LongType()) / 100)
            .withColumn("avg_rev", col("avg_rev").cast(LongType()) / 100)
            .withColumn("total_rev", round(col("total_rev"), 2))
            .withColumn("avg_rev", round(col("avg_rev"),2))
        )

display(chain_df)
```

---

**--NoteBook : DE 2.1 - Querying Files Directly**

Extracting Data Directly From Files with Spark SQL

In this notebook, you'll learn to extract data directly from files using Spark SQL on Databricks.

A number of file formats support this option, but it is most useful for self-describing data formats (such as Parquet and JSON).

**--Learning Objectives**

By the end of this lesson, you should be able to:

- Use Spark SQL to directly query data files
- Layer views and CTEs to make referencing data files easier
- Leverage text and binaryFile methods to review raw file contents

**--Run Setup**

The setup script will create the data and declare necessary values for the rest of this notebook to execute.

%run ../Includes/Classroom-Setup-02.1

**--Data Overview**

In this example, we'll work with a sample of raw Kafka data written as JSON files.

Each file contains all records consumed during a 5-second interval, stored with the full Kafka schema as a multiple-record JSON file.

| field |
|---|

| **type** |
| --- |
| **description** |
| |
| key |
| BINARY |
| The user_id field is used as the key; this is a unique alphanumeric field that corresponds to session/cookie information. |
| |
| value |
| BINARY |
| This is the full data payload (to be discussed later), sent as JSON |
| |
| topic |
| STRING |
| While the Kafka service hosts multiple topics, only those records from the clickstream topic are included here. |
| |
| partition |
| INTEGER |
| Our current Kafka implementation uses only 2 partitions (0 and 1). |
| |
| offset |
| LONG |
| This is a unique value, monotonically increasing for each partition. |
| |
| timestamp |
| LONG |
| This timestamp is recorded as milliseconds since epoch, and represents the time at which the producer appends a record to a partition |

**--Note that our source directory contains many JSON files.**

%python

print(DA.paths.kafka_events)

files = dbutils.fs.ls(DA.paths.kafka_events)

display(files)

Here, we'll be using relative file paths to data that's been written to the DBFS root.

Most workflows will require users to access data from external cloud storage locations.

In most companies, a workspace administrator will be responsible for configuring access to these storage locations.

Instructions for configuring and accessing these locations can be found in the cloud-vendor specific self-paced courses titled "Cloud Architecture & Systems Integrations".

**--Query a Single File**

To query the data contained in a single file, execute the query with the following pattern:

SELECT * FROM file_format.`/path/to/file`

Make special note of the use of back-ticks (not single quotes) around the path.

SELECT * FROM json.`${DA.paths.kafka_events}/001.json`

SELECT count(*) FROM json.`${DA.paths.kafka_events}/000.json`

**--Query a Directory of Files**

Assuming all of the files in a directory have the same format and schema, all files can be queried simultaneously by specifying the directory path rather than an individual file.

```
SELECT * FROM json.`${DA.paths.kafka_events}`

SELECT count(*) FROM json.`${DA.paths.kafka_events}`
```

**--Create References to Files**

This ability to directly query files and directories means that additional Spark logic can be chained to queries against files.

When we create a view from a query against a path, we can reference this view in later queries.

```
CREATE OR REPLACE VIEW event_view

AS SELECT * FROM json.`${DA.paths.kafka_events}`
```

--As long as a user has permission to access the view and the underlying storage location, that user will be able to use this view definition to query the underlying data. This applies to different users in the workspace, different notebooks, and different clusters.

```
SELECT count(*) FROM event_view
```

**--Create Temporary References to Files**

Temporary views similarly alias queries to a name that's easier to reference in later queries.

```
CREATE OR REPLACE TEMP VIEW events_temp_view

AS SELECT * FROM json.`${DA.paths.kafka_events}`
```

**--Temporary views exists only for the current SparkSession. On Databricks, this means they are isolated to the current notebook, job, or DBSQL query.**

SELECT count(*) FROM events_temp_view

**--Apply CTEs for Reference within a Query**

Common table expressions (CTEs) are perfect when you want a short-lived, human-readable reference to the results of a query.

WITH cte_json

AS (SELECT * FROM json.`${DA.paths.kafka_events}`)

SELECT count(*) FROM cte_json

**--Extract Text Files as Raw Strings**

When working with text-based files (which include JSON, CSV, TSV, and TXT formats), you can use the text format to load each line of the file as a row with one string column named value. This can be useful when data sources are prone to corruption and custom text parsing functions will be used to extract values from text fields.

SELECT * FROM text.`${DA.paths.kafka_events}`

**--Extract the Raw Bytes and Metadata of a File**

Some workflows may require working with entire files, such as when dealing with images or unstructured data. Using binaryFile to query a directory will provide file metadata alongside the binary representation of the file contents.

Specifically, the fields created will indicate the path, modificationTime, length, and content.

SELECT * FROM binaryFile.`${DA.paths.kafka_events}`

---

**--NoteBook : DE 2.2 - Providing Options for External Sources**

go through NoteBook : DE 4.2 - Providing Options for External Sources

**--NoteBook : DE 4.2 - Providing Options for External Sources**

**--Providing Options for External Sources**

While directly querying files works well for self-describing formats, many data sources require additional configurations or schema declaration to properly ingest records.

In this lesson, we will create tables using external data sources. While these tables will not yet be stored in the Delta Lake format (and therefore not be optimized for the Lakehouse), this technique helps to facilitate extracting data from diverse external systems.

**Learning Objectives**

By the end of this lesson, you should be able to:

1. Use Spark SQL to configure options for extracting data from external sources
2. Create tables against external data sources for various file formats
3. Describe default behavior when querying tables defined against external sources

**--Run Setup**

The setup script will create the data and declare necessary values for the rest of this notebook to execute.

%run ../Includes/Classroom-Setup-4.2

**--When Direct Queries Don't Work**

While views can be used to persist direct queries against files between sessions, this approach has limited utility.

CSV files are one of the most common file formats, but a direct query against these files rarely returns the desired results.

SELECT * FROM csv.`${da.paths.working_dir}/sales-csv`

**-We can see from the above that:**

The header row is being extracted as a table row

All columns are being loaded as a single column

The file is pipe-delimited (|)

The final column appears to contain nested data that is being truncated

**--Registering Tables on External Data with Read Options**

While Spark will extract some self-describing data sources efficiently using default settings, many formats will require declaration of schema or other options.

While there are many additional configurations you can set while creating tables against external sources, the syntax below demonstrates the essentials required to extract data from most formats.

```
CREATE TABLE table_identifier (col_name1 col_type1, ...)
USING data_source
OPTIONS (key1 = val1, key2 = val2, ...)
LOCATION = path
```

Note that options are passed with keys as unquoted text and values in quotes. Spark supports many data sources with custom options, and additional systems may have unofficial support through external libraries.

**NOTE:** Depending on your workspace settings, you may need administrator assistance to load libraries and configure the requisite security settings for some data sources.

-The cell below demonstrates using Spark SQL DDL to create a table against an external CSV source, specifying:

- The column names and types
- The file format
- The delimiter used to separate fields
- The presence of a header
- The path to where this data is stored

```
CREATE TABLE sales_csv
  (order_id LONG, email STRING, transactions_timestamp LONG, total_item_quantity INTEGER, purchase_revenue_in_usd DOUBLE, unique_items INTEGER, items STRING)
USING CSV
OPTIONS (
  header = "true",
  delimiter = "|"
```

```
)
LOCATION "${da.paths.working_dir}/sales-csv"
```

-Note that no data has moved during table declaration. Similar to when we directly queried our files and created a view, we are still just pointing to files stored in an external location.

Run the following cell to confirm that data is now being loaded correctly.

```
SELECT * FROM sales_csv
SELECT COUNT(*) FROM sales_csv
```

--All the metadata and options passed during table declaration will be persisted to the metastore, ensuring that data in the location will always be read with these options.

NOTE: When working with CSVs as a data source, it's important to ensure that column order does not change if additional data files will be added to the source directory. Because the data format does not have strong schema enforcement, Spark will load columns and apply column names and data types in the order specified during table declaration.

Running DESCRIBE EXTENDED on a table will show all of the metadata associated with the table definition.

```
DESCRIBE EXTENDED sales_csv
describe history sales_csv
```

**--Limits of Tables with External Data Sources**

If you've taken other courses on Databricks or reviewed any of our company literature, you may have heard about Delta Lake and the Lakehouse. Note that whenever we're defining tables or queries against external data sources, we cannot expect the performance guarantees associated with Delta Lake and Lakehouse.

For example: while Delta Lake tables will guarantee that you always query the most recent version of your source data, tables registered against other data sources may represent older cached versions.

The cell below executes some logic that we can think of as just representing an external system directly updating the files underlying our table.

```
%python
(spark.table("sales_csv")
    .write.mode("append")
    .format("csv")
    .save(f"{DA.paths.working_dir}/sales-csv"))
```

**--If we look at the current count of records in our table, the number we see will not reflect these newly inserted rows.**

```
SELECT COUNT(*) FROM sales_csv
```

--At the time we previously queried this data source, Spark automatically cached the underlying data in local storage. This ensures that on subsequent queries, Spark will provide the optimal performance by just querying this local cache.

Our external data source is not configured to tell Spark that it should refresh this data.

We can manually refresh the cache of our data by running the REFRESH TABLE command.

```
REFRESH TABLE sales_csv
```

--Note that refreshing our table will invalidate our cache, meaning that we'll need to rescan our original data source and pull all data back into memory.

For very large datasets, this may take a significant amount of time.

```
SELECT COUNT(*) FROM sales_csv
```

**--Extracting Data from SQL Databases**

SQL databases are an extremely common data source, and Databricks has a standard JDBC driver for connecting with many flavors of SQL.

**The general syntax for creating these connections is:**

```
CREATE TABLE
USING JDBC
OPTIONS (
   url = "jdbc:{databaseServerType}://{jdbcHostname}:{jdbcPort}",
```

```
  dbtable = "{jdbcDatabase}.table",
  user = "{jdbcUsername}",
  password = "{jdbcPassword}"
)
```

In the code sample below, we'll connect with SQLite.

**NOTE:** SQLite uses a local file to store a database, and doesn't require a port, username, or password.

WARNING: The backend-configuration of the JDBC server assume you are running this notebook on a single-node cluster. If you are running on a cluster with multiple workers, the client running in the executors will not be able to connect to the driver.

```
DROP TABLE IF EXISTS users_jdbc;
CREATE TABLE users_jdbc
USING JDBC
OPTIONS (
 url = "jdbc:sqlite:/${da.username}_ecommerce.db",
 dbtable = "users"
)
```

**--Now we can query this table as if it were defined locally.**

```
SELECT * FROM users_jdbc
```

--Looking at the table metadata reveals that we have captured the schema information from the external system. Storage properties (which would include

the username and password associated with the connection) are automatically redacted.

DESCRIBE EXTENDED users_jdbc

While the table is listed as MANAGED, listing the contents of the specified location confirms that no data is being persisted locally.

%python

jdbc_users_path = f"{DA.paths.user_db}/users_jdbc/"

print(jdbc_users_path)

files = dbutils.fs.ls(jdbc_users_path)

print(f"Found {len(files)} files")

--Note that some SQL systems such as data warehouses will have custom drivers. Spark will interact with various external databases differently, but the two basic approaches can be summarized as either:

Moving the entire source table(s) to Databricks and then executing logic on the currently active cluster

Pushing down the query to the external SQL database and only transferring the results back to Databricks

In either case, working with very large datasets in external SQL databases can incur significant overhead because of either:

Network transfer latency associated with moving all data over the public internet

Execution of query logic in source systems not optimized for big data queries

---------- Day 19 ----------

**--NoteBook : DE 2.3L - Extract Data Lab:**

--Extract Data Lab

In this lab, you will extract raw data from JSON files.

--Learning Objectives

By the end of this lab, you should be able to:

Register an external table to extract data from JSON files

--Run Setup

Run the following cell to configure variables and datasets for this lesson.

%run ../Includes/Classroom-Setup-02.3L

-Overview of the Data

We will work with a sample of raw Kafka data written as JSON files.

Each file contains all records consumed during a 5-second interval, stored with the full Kafka schema as a multiple-record JSON file.

The schema for the table:

| field | type | description |
|---|---|---|

| | | |
|---|---|---|
| key | BINARY | The user_id field is used as the key; this is a unique alphanumeric field that corresponds to session/cookie information |
| offset | LONG | This is a unique value, monotonically increasing for each partition |
| partition | INTEGER | Our current Kafka implementation uses only 2 partitions (0 and 1) |
| timestamp | LONG | This timestamp is recorded as milliseconds since epoch, and represents the time at which the producer appends a record to a partition |
| topic | STRING | While the Kafka service hosts multiple topics, only those records from the clickstream topic are included here |
| value | BINARY | This is the full data payload (to be discussed later), sent as JSON |

**--Extract Raw Events From JSON Files**

To load this data into Delta properly, we first need to extract the JSON data using the correct schema.

Create an external table against JSON files located at the filepath provided below. Name this table events_json and declare the schema above.

```
create table if not exists events_json (key binary, offset long, partition integer, timestamp long, topic string, value binary)
using json
options (
header = "True"
)
location "${DA.paths.kafka_events}"
```

select * from events_json;

**--NOTE:** We'll use Python to run checks occasionally throughout the lab. The following cell will return an error with a message on what needs to change if you have not followed instructions. No output from cell execution means that you have completed this step.

---

**--NoteBook:** DE 2.4 - Cleaning Data

**--Cleaning Data**

As we inspect and clean our data, we'll need to construct various column expressions and queries to express transformations to apply on our dataset.

Column expressions are constructed from existing columns, operators, and built-in functions. They can be used in SELECT statements to express transformations that create new columns.

Many standard SQL query commands (e.g. DISTINCT, WHERE, GROUP BY, etc.) are available in Spark SQL to express transformations.

In this notebook, we'll review a few concepts that might differ from other systems you're used to, as well as calling out a few useful functions for common operations.

We'll pay special attention to behaviors around NULL values, as well as formatting strings and datetime fields.

**--Learning Objectives**

By the end of this lesson, you should be able to:

a. Summarize datasets and describe null behaviors
b. Retrieve and remove duplicates
c. Validate datasets for expected counts, missing values, and duplicate records
d. Apply common transformations to clean and transform data

**--Run Setup**

The setup script will create the data and declare necessary values for the rest of this notebook to execute.

%run ../Includes/Classroom-Setup-02.4

**--Display count of all**

SELECT count(*), count(user_id), count(user_first_touch_timestamp), count(email), count(updated) FROM users_dirty

**--Inspect Missing Data**

Based on the counts above, it looks like there are at least a handful of null values in all of our fields.

**NOTE:**

1. Null values behave incorrectly in some math functions, including count().

2. count(col) skips NULL values when counting specific columns or expressions.
3. count(*) is a special case that counts the total number of rows (including rows that are only NULL values).

We can count null values in a field by filtering for records where that field is null, using either:

count_if(col IS NULL) or count(*) with a filter for where col IS NULL.

Both statements below correctly count records with missing emails.

**--count_if email is null**

SELECT count_if(email IS NULL) FROM users_dirty;

**--count email is null**

SELECT count(*) FROM users_dirty WHERE email IS NULL;

**--Count**

%python

from pyspark.sql.functions import col

usersDF = spark.read.table("users_dirty")

usersDF.selectExpr("count_if(email IS NULL)")

usersDF.where(col("email").isNull()).count()

**--Deduplicate Rows**

We can use DISTINCT * to remove true duplicate records where entire rows contain the same values.

**--Distinct**

SELECT DISTINCT(*) FROM users_dirty

--distinct() & count()

%python

usersDF.distinct().count()

**--Deduplicate Rows Based on Specific Columns**

The code below uses GROUP BY to remove duplicate records based on user_id and user_first_touch_timestamp column values. (Recall that these fields are both generated when a given user is first encountered, thus forming unique tuples.)

Here, we are using the aggregate function max as a hack to:

Keep values from the email and updated columns in the result of our group by
Capture non-null emails when multiple records are present

**--Create temp view and count rows**

CREATE OR REPLACE TEMP VIEW deduped_users AS

SELECT user_id, user_first_touch_timestamp, max(email) AS email, max(updated) AS updated

FROM users_dirty

WHERE user_id IS NOT NULL

```
GROUP BY user_id, user_first_touch_timestamp;


SELECT count(*) FROM deduped_users


--Display data from deduped_user
SELECT * FROM deduped_users


--Sort data using max function and display count
%python
from pyspark.sql.functions import max
dedupedDF = (usersDF
   .where(col("user_id").isNotNull())
   .groupBy("user_id", "user_first_touch_timestamp")
   .agg(max("email").alias("email"),
       max("updated").alias("updated"))
   )


dedupedDF.count()
display(dedupedDF)


--Let's confirm that we have the expected count of remaining records after deduplicating based on distinct user_id and user_first_touch_timestamp values.
SELECT COUNT(DISTINCT(user_id, user_first_touch_timestamp))
FROM users_dirty
WHERE user_id IS NOT NULL
```

**--Count using df**

```
%python
(usersDF
   .dropDuplicates(["user_id", "user_first_touch_timestamp"])
   .filter(col("user_id").isNotNull())
   .count())
```

**--Validate Datasets**

Based on our manual review above, we've visually confirmed that our counts are as expected. We can also programmatically perform validation using simple filters and WHERE clauses. Validate that the user_id for each row is unique.

**--Find Duplicates**

```
SELECT max(row_count) <= 1 no_duplicate_ids FROM (
 SELECT user_id, count(*) AS row_count
 FROM deduped_users
 GROUP BY user_id)
```

**--Find Duplicates**

```
%python
from pyspark.sql.functions import count

display(dedupedDF
   .groupby("user_id")
```

```
  .agg(count("*").alias("row_count"))
  .select((max("row_count") <= 1).alias("no_duplicate_ids")))
```

**--Confirm that each email is associated with at most one user_id.**

```
SELECT max(user_id_count) <= 1 at_most_one_id FROM (
 SELECT email, count(user_id) AS user_id_count
 FROM deduped_users
 WHERE email IS NOT NULL
 GROUP BY email)

 ---
%python

display(dedupedDF
  .where(col("email").isNotNull())
  .groupby("email")
  .agg(count("user_id").alias("user_id_count"))
  .select((max("user_id_count") <= 1).alias("at_most_one_id")))
```

**--Date Format and Regex**

Now that we've removed null fields and eliminated duplicates, we may wish to extract further value out of the data.

The code below:

a. Correctly scales and casts the user_first_touch_timestamp to a valid timestamp
b. Extracts the calendar data and clock time for this timestamp in human readable format
c. Uses regexp_extract to extract the domains from the email column using regex

```
SELECT *,
  date_format(first_touch, "MMM d, yyyy") AS first_touch_date,
  date_format(first_touch, "HH:mm:ss") AS first_touch_time,
  regexp_extract(email,"(?<=@).+", 0) AS email_domain
FROM (
  SELECT *,
    CAST(user_first_touch_timestamp / 1e6 AS timestamp) AS first_touch
  FROM deduped_users
)
```

**--Second Way:**

```
%python
from pyspark.sql.functions import date_format, regexp_extract

display(dedupedDF
    .withColumn("first_touch", (col("user_first_touch_timestamp") /
1e6).cast("timestamp"))
    .withColumn("first_touch_date", date_format("first_touch", "MMM d, yyyy"))
    .withColumn("first_touch_time", date_format("first_touch", "HH:mm:ss"))
```

```
    .withColumn("email_domain", regexp_extract("email", "(?<=@).+", 0))
)
```

---

**--NoteBook:** DE 2.5 - Complex Transformations

**--Transforming Complex Types**

Querying tabular data stored in the data lakehouse with Spark SQL is easy, efficient, and fast.

This gets more complicated as the data structure becomes less regular, when many tables need to be used in a single query, or when the shape of data needs to be changed dramatically. This notebook introduces a number of functions present in Spark SQL to help engineers complete even the most complicated transformations.

**--Learning Objectives**

By the end of this lesson, you should be able to:

a. Use . and : syntax to query nested data
b. Parse JSON strings into structs
c. Flatten and unpack arrays and structs
d. Combine datasets using joins
e. Reshape data using pivot tables

**--Run Setup**

The setup script will create the data and declare necessary values for the rest of this notebook to execute.

```
%run ../Includes/Classroom-Setup-02.5
```

**--Data Overview**

The events_raw table was registered against data representing a Kafka payload. In most cases, Kafka data will be binary-encoded JSON values.

Let's cast the key and value as strings to view these values in a human-readable format.

**--Create temp view with sparksql**

```
CREATE OR REPLACE TEMP VIEW events_strings AS
SELECT string(key), string(value) FROM events_raw;

SELECT * FROM events_strings
```

**--Second way with df**

```
%python
from pyspark.sql.functions import col

events_stringsDF = (spark
   .table("events_raw")
   .select(col("key").cast("string"),
         col("value").cast("string"))
   )
display(events_stringsDF)
```

**--Work with Nested Data**

The code cell below queries the converted strings to view an example JSON object without null fields (we'll need this for the next section).

**NOTE:** Spark SQL has built-in functionality to directly interact with nested data stored as JSON strings or struct types.

a. Use : syntax in queries to access subfields in JSON strings
b. Use . syntax in queries to access subfields in struct types

**--Select column with sub value in json file by using :**

```
SELECT * FROM events_strings WHERE value:event_name = "finalize" ORDER BY key LIMIT 1
```

**--Second way pyspark**

```
%python
display(events_stringsDF
  .where("value:event_name = 'finalize'")
  .orderBy("key")
  .limit(1)
)
```

**--Let's use the JSON string example above to derive the schema, then parse the entire JSON column into struct types.**

schema_of_json() returns the schema derived from an example JSON string.

from_json() parses a column containing a JSON string into a struct type using the specified schema.

**--After we unpack the JSON string to a struct type, let's unpack and flatten all struct fields into columns.**

* unpacking can be used to flattens structs; col_name.* pulls out the subfields of col_name into their own columns.

```
CREATE OR REPLACE TEMP VIEW parsed_events AS SELECT json.* FROM (

SELECT from_json(value,
schema_of_json('{"device":"Linux","ecommerce":{"purchase_revenue_in_usd":1075.5,"total_item_quantity":1,"unique_items":1},"event_name":"finalize","event_previous_timestamp":1593879231210816,"event_timestamp":1593879335779563,"geo":{"city":"Houston","state":"TX"},"items":[{"coupon":"NEWBED10","item_id":"M_STAN_K","item_name":"Standard King Mattress","item_revenue_in_usd":1075.5,"price_in_usd":1195.0,"quantity":1}],"traffic_source":"email","user_first_touch_timestamp":1593454417513109,"user_id":"UA000000106116176"}')) AS json

FROM events_strings);


SELECT * FROM parsed_events
```

**--Same in pyspark**

```
%python

from pyspark.sql.functions import from_json, schema_of_json


json_string = """
```

{"device":"Linux","ecommerce":{"purchase_revenue_in_usd":1047.6,"total_item_quantity":2,"unique_items":2},"event_name":"finalize","event_previous_timestamp":1593879787820475,"event_timestamp":1593879948830076,"geo":{"city":"Huntington
Park","state":"CA"},"items":[{"coupon":"NEWBED10","item_id":"M_STAN_Q","item_name":"Standard Queen
Mattress","item_revenue_in_usd":940.5,"price_in_usd":1045.0,"quantity":1},{"coupon":"NEWBED10","item_id":"P_DOWN_S","item_name":"Standard Down
Pillow","item_revenue_in_usd":107.10000000000001,"price_in_usd":119.0,"quantity":1}],"traffic_source":"email","user_first_touch_timestamp":1593583891412316,"user_id":"UA000000106459577"}

"""

```
parsed_eventsDF = (events_stringsDF
    .select(from_json("value", schema_of_json(json_string)).alias("json"))
    .select("json.*")
)

display(parsed_eventsDF)
```

**--Manipulate Arrays**

Spark SQL has a number of functions for manipulating array data, including the following:

a. explode() separates the elements of an array into multiple rows; this creates a new row for each element.
b. size() provides a count for the number of elements in an array for each row.

The code below explodes the items field (an array of structs) into multiple rows and shows events containing arrays with 3 or more items.

```
CREATE OR REPLACE TEMP VIEW exploded_events AS
SELECT *, explode(items) AS item
FROM parsed_events;

SELECT * FROM exploded_events WHERE size(items) > 0
```

**--Second Way using pyspark**

```
%python
from pyspark.sql.functions import explode, size

exploded_eventsDF = (parsed_eventsDF
   .withColumn("item", explode("items"))
)

display(exploded_eventsDF.where(size("items") > 2))
```

**--The code below combines array transformations to create a table that shows the unique collection of actions and the items in a user's cart.**

a. collect_set() collects unique values for a field, including fields within arrays.
b. flatten() combines multiple arrays into a single array.
c. array_distinct() removes duplicate elements from an array.

```
SELECT user_id,
  collect_set(event_name) AS event_history,
  array_distinct(flatten(collect_set(items.item_id))) AS cart_history
```

```
FROM exploded_events
GROUP BY user_id
```

**--Secons Way using pyspark**

```
%python
from pyspark.sql.functions import array_distinct, collect_set, flatten

display(exploded_eventsDF
   .groupby("user_id")
   .agg(collect_set("event_name").alias("event_history"),
        array_distinct(flatten(collect_set("items.item_id"))).alias("cart_history"))
)
```

**--Join Tables**

Spark SQL supports standard JOIN operations (inner, outer, left, right, anti, cross, semi). Here we join the exploded events dataset with a lookup table to grab the standard printed item name.

```
CREATE OR REPLACE TEMP VIEW item_purchases AS
SELECT *
FROM (SELECT *, explode(items) AS item FROM sales) a
INNER JOIN item_lookup b
ON a.item.item_id = b.item_id;

SELECT * FROM item_purchases
```

**--Second Way using pyspark**

```
%python
exploded_salesDF = (spark
   .table("sales")
   .withColumn("item", explode("items"))
)

itemsDF = spark.table("item_lookup")

item_purchasesDF = (exploded_salesDF
   .join(itemsDF, exploded_salesDF.item.item_id == itemsDF.item_id)
)

display(item_purchasesDF)
```

**--Pivot Tables**

We can use PIVOT to view data from different perspectives by rotating unique values in a specified pivot column into multiple columns based on an aggregate function.

The PIVOT clause follows the table name or subquery specified in a FROM clause, which is the input for the pivot table.

Unique values in the pivot column are grouped and aggregated using the provided aggregate expression, creating a separate column for each unique value in the resulting pivot table.

The following code cell uses PIVOT to flatten out the item purchase information contained in several fields derived from the sales dataset. This flattened data format can be useful for dashboarding, but also useful for applying machine learning algorithms for inference or prediction.

```
SELECT *
FROM item_purchases
PIVOT (
  sum(item.quantity) FOR item_id IN (
    'P_FOAM_K',
    'M_STAN_Q',
    'P_FOAM_S',
    'M_PREM_Q',
    'M_STAN_F',
    'M_STAN_T',
    'M_PREM_K',
    'M_PREM_F',
    'M_STAN_K',
    'M_PREM_T',
    'P_DOWN_S',
    'P_DOWN_K')
)
```

**--Second way using pyspark**

%python

```
transactionsDF = (item_purchasesDF
   .groupBy("order_id",
      "email",
      "transaction_timestamp",
      "total_item_quantity",
      "purchase_revenue_in_usd",
      "unique_items",
      "items",
      "item",
      "name",
      "price")
   .pivot("item_id")
   .sum("item.quantity")
)
display(transactionsDF)
```

---------- Day 20 ----------

**--NoteBook:** DE 2.6L - Reshape Data Lab

**--Reshaping Data Lab**

In this lab, you will create a clickpaths table that aggregates the number of times each user took a particular action in events and then join this information with a flattened view of transactions to create a record of each user's actions and final purchases.

The clickpaths table should contain all the fields from transactions, as well as a count of every event_name from events in its own column. This table should contain a single row for each user that completed a purchase.

**--Learning Objectives**

By the end of this lab, you should be able to:

Pivot and join tables to create clickpaths for each user

**--Run Setup**

The setup script will create the data and declare necessary values for the rest of this notebook to execute.

%run ../Includes/Classroom-Setup-02.6L

--We'll use Python to run checks occasionally throughout the lab. The helper functions below will return an error with a message on what needs to change if you have not followed instructions. No output means that you have completed this step.

**--Pivot events to get event counts for each user**

Let's start by pivoting the events table to get counts for each event_name.

We want to aggregate the number of times each user performed a specific event, specified in the event_name column. To do this, group by user_id and pivot on event_name to provide a count of every event type in its own column, resulting in the schema below. Note that user_id is renamed to user in the target schema.

field type

user STRING

cart BIGINT

pillows BIGINT

login BIGINT

main BIGINT

careers BIGINT

guest BIGINT

faq BIGINT

down BIGINT

warranty BIGINT

finalize BIGINT

register BIGINT

shipping_info BIGINT

checkout BIGINT

mattresses BIGINT

add_item BIGINT

press BIGINT

email_coupon BIGINT

cc_info BIGINT

foam BIGINT

reviews BIGINT

original BIGINT

delivery BIGINT

premium BIGINT

A list of the event names are provided in the TODO cells below.

```
-- TODO
CREATE OR REPLACE VIEW events_pivot as
select * from (
        select user_id as USER, event_name
        from events
        group by user_id, event_name
        )

pivot(
 count(event_name)
 for event_name in
     ("cart", "pillows", "login", "main", "careers", "guest", "faq", "down",
"warranty", "finalize",
"register", "shipping_info", "checkout", "mattresses", "add_item", "press",
"email_coupon",
"cc_info", "foam", "reviews", "original", "delivery", "premium")
);

select * from events_pivot;
```

---

**--NoteBook:** DE 2.7B - Python UDFs

-Python User-Defined Functions

**--Objectives:**

a. Define a function
b. Create and apply a UDF
c. Register the UDF to use in SQL
d. Create and register a UDF with Python decorator syntax
e. Create and apply a Pandas (vectorized) UDF

**--Methods:**

a. UDF Registration (spark.udf): register
b. Built-In Functions: udf
c. Python UDF Decorator: @udf
d. Pandas UDF Decorator: @pandas_udf

**--User-Defined Function (UDF)**

**-A custom column transformation function**

a. Can’t be optimized by Catalyst Optimizer
b. Function is serialized and sent to executors
c. Row data is deserialized from Spark's native binary format to pass to the UDF, and the results are serialized back into Spark's native format
d. For Python UDFs, additional interprocess communication overhead between the executor and a Python interpreter running on each worker node

**--Create DF**

```
sales_df = spark.table("sales")
```

```
display(sales_df)
```

**--Define a function**

Define a function (on the driver) to get the first letter of a string from the email field.

```
def first_letter_function(email):
  return email[0]

first_letter_function("annagray@kaufman.com")
```

**--Create and apply UDF**

Register the function as a UDF. This serializes the function and sends it to executors to be able to transform DataFrame records.

```
first_letter_udf = udf(first_letter_function)
```

**--Apply the UDF on the email column.**

```
from pyspark.sql.functions import col

display(sales_df.select(first_letter_udf(col("email"))))
```

**--Register UDF to use in SQL**

Register the UDF using spark.udf.register to also make it available for use in the SQL namespace.

```
sales_df.createOrReplaceTempView("sales")
```

```
first_letter_udf = spark.udf.register("sql_udf", first_letter_function)

# You can still apply the UDF from Python
display(sales_df.select(first_letter_udf(col("email"))))

%sql
-- You can now also apply the UDF from SQL
SELECT sql_udf(email) AS first_letter FROM sales
```

**--Use Decorator Syntax (Python Only)**

Alternatively, you can define and register a UDF using Python decorator syntax. The @udf decorator parameter is the Column datatype the function returns.

You will no longer be able to call the local Python function (i.e., first_letter_udf("annagray@kaufman.com") will not work).

**Note:** This example also uses Python type hints, which were introduced in Python 3.5. Type hints are not required for this example, but instead serve as "documentation" to help developers use the function correctly. They are used in this example to emphasize that the UDF processes one record at a time, taking a single str argument and returning a str value.

```
# Our input/output is a string
@udf("string")
def first_letter_udf(email: str) -> str:
  return email[0]
```

**--And let's use our decorator UDF here.**

```
from pyspark.sql.functions import col


sales_df = spark.table("sales")

display(sales_df.select(first_letter_udf(col("email"))))
```

**--Pandas/Vectorized UDFs**

Pandas UDFs are available in Python to improve the efficiency of UDFs. Pandas UDFs utilize Apache Arrow to speed up computation.

Blog post

Documentation

**The user-defined functions are executed using:**

1. Apache Arrow, an in-memory columnar data format that is used in Spark to efficiently transfer data between JVM and Python processes with near-zero (de)serialization cost
2. Pandas inside the function, to work with Pandas instances and APIs
3. Warning as of Spark 3.0, you should always define your Pandas UDF using Python type hints.

```
import pandas as pd

from pyspark.sql.functions import pandas_udf


# We have a string input/output

@pandas_udf("string")

def vectorized_udf(email: pd.Series) -> pd.Series:
```

```
    return email.str[0]

# Alternatively
def vectorized_udf(email: pd.Series) -> pd.Series:
    return email.str[0]
 vectorized_udf = pandas_udf(vectorized_udf, "string")

display(sales_df)
display(sales_df.select(vectorized_udf(col("email"))))
```

**--We can also register these Pandas UDFs to the SQL namespace.**

```
spark.udf.register("sql_vectorized_udf", vectorized_udf)

%sql
-- Use the Pandas UDF from SQL
SELECT sql_vectorized_udf(email) AS firstLetter FROM sales
```

**---------- Day 21 ----------**

**--NoteBook:** DE 2.7A - SQL UDFs

**-SQL UDFs and Control Flow**

**Learning Objectives**

**--By the end of this lesson, you should be able to:**

a. Define and registering SQL UDFs
b. Describe the security model used for sharing SQL UDFs

c. Use CASE / WHEN statements in SQL code
d. Leverage CASE / WHEN statements in SQL UDFs for custom control flow

**--Run Setup**

Run the following cell to setup your environment.

**%run ../Includes/Classroom-Setup-02.7A**

**--User-Defined Functions**

User Defined Functions (UDFs) in Spark SQL allow you to register custom SQL logic as functions in a database, making these methods reusable anywhere SQL can be run on Databricks. These functions are registered natively in SQL and maintain all of the optimizations of Spark when applying custom logic to large datasets.

At minimum, creating a SQL UDF requires a function name, optional parameters, the type to be returned, and some custom logic.

Below, a simple function named sale_announcement takes an item_name and item_price as parameters. It returns a string that announces a sale for an item at 80% of its original price.

**--Create Function in SQL**

```
CREATE OR REPLACE FUNCTION sale_announcement(item_name STRING,
item_price INT)

RETURNS STRING

RETURN concat("The ", item_name, " is on sale for $", round(item_price * 0.8, 0));
```

SELECT *, sale_announcement(name, price) AS message FROM item_lookup

**Note:** that this function is applied to all values of the column in a parallel fashion within the Spark processing engine. SQL UDFs are an efficient way to define custom logic that is optimized for execution on Databricks.

**--Scoping and Permissions of SQL UDFs**

SQL user-defined functions:

a. Persist between execution environments (which can include notebooks, DBSQL queries, and jobs).
b. Exist as objects in the metastore and are governed by the same Table ACLs as databases, tables, or views.
c. Require USAGE and SELECT permissions to use the SQL UDF

We can use DESCRIBE FUNCTION to see where a function was registered and basic information about expected inputs and what is returned (and even more information with DESCRIBE FUNCTION EXTENDED).

**--DESCRIBE FUNCTION**

DESCRIBE FUNCTION EXTENDED sale_announcement

**Note:** that the Body field at the bottom of the function description shows the SQL logic used in the function itself.

**--Simple Control Flow Functions**

Combining SQL UDFs with control flow in the form of CASE / WHEN clauses provides optimized execution for control flows within SQL workloads. The standard SQL syntactic construct CASE / WHEN allows the evaluation of multiple conditional statements with alternative outcomes based on table contents.

Here, we demonstrate wrapping this control flow logic in a function that will be reusable anywhere we can execute SQL.

**--Function by using CASE WHEN**

```
CREATE OR REPLACE FUNCTION item_preference(name STRING, price INT)
RETURNS STRING
RETURN CASE
  WHEN name = "Standard Queen Mattress" THEN "This is my default mattress"
  WHEN name = "Premium Queen Mattress" THEN "This is my favorite mattress"
  WHEN price > 100 THEN concat("I'd wait until the ", name, " is on sale for $", round(price * 0.8, 0))
  ELSE concat("I don't need a ", name)
END;

SELECT *, item_preference(name, price) FROM item_lookup
```

--While the examples provided here are simple, these same basic principles can be used to add custom computations and logic for native execution in Spark SQL.

Especially for enterprises that might be migrating users from systems with many defined procedures or custom-defined formulas, SQL UDFs can allow a handful of users to define the complex logic needed for common reporting and analytic queries.

---

**--NoteBook:** DE 2.99 - OPTIONAL Higher Order Functions

**--Higher Order Functions in Spark SQL**

Higher order functions in Spark SQL allow you to transform complex data types, such as array or map type objects, while preserving their original structures. Examples include:

a. FILTER() filters an array using the given lambda function.
b. EXIST() tests whether a statement is true for one or more elements in an array.
c. TRANSFORM() uses the given lambda function to transform all elements in an array.
d. REDUCE() takes two lambda functions to reduce the elements of an array to a single value by merging the elements into a buffer, and the apply a finishing function on the final buffer.

**Learning Objectives:**

By the end of this lesson, you should be able to Use higher order functions to work with arrays

**--Run Setup**

**%run ../Includes/Classroom-Setup-02.99**

**--Filter**

We can use the FILTER function to create a new column that excludes values from each array based on a provided condition.

Let's use this to remove products in the items column that are not king-sized from all records in our sales dataset.

FILTER (items, i -> i.item_id LIKE "%K") AS king_items

In the statement above:

a. FILTER : the name of the higher-order function
b. items : the name of our input array
c. i : the name of the iterator variable. You choose this name and then use it in the lambda function. It iterates over the array, cycling each value into the function one at a time.
   ➔ : Indicates the start of a function
d. i.item_id LIKE "%K" : This is the function. Each value is checked to see if it ends with the capital letter K. If it is, it gets filtered into the new column, king_items

**NOTE:** You may write a filter that produces a lot of empty arrays in the created column. When that happens, it can be useful to use a WHERE clause to show only non-empty array values in the returned column.

```
SELECT * FROM (
 SELECT
  order_id,
  FILTER (items, i -> i.item_id LIKE "%K") AS king_items
 FROM sales)
WHERE size(king_items) > 0
```

**--Transform**

The TRANSFORM() higher order function can be particularly useful when you want to apply an existing function to each element in an array.

Let's apply this to create a new array column called item_revenues by transforming the elements contained in the items array column.

In the query below: items is the name of our input array, i is the name of the iterator variable (you choose this name and then use it in the lambda function; it iterates over the array, cycling each value into the function one at a time), and -> indicates the start of a function.

```
SELECT *,
  TRANSFORM (
    items, i -> CAST(i.item_revenue_in_usd * 100 AS INT)
  ) AS item_revenues
FROM sales
```

--The lambda function we specified above takes the item_revenue_in_usd subfield of each value, multiplies that by 100, casts to integer, and includes the result in the new array column, item_revenues

**--Exists Lab**

Here, you'll use the higher order function EXISTS with data from the sales table to create boolean columns mattress and pillow that indicate whether the item purchased was a mattress or pillow product.

For example, if item_name from the items column ends with the string "Mattress", the column value for mattress should be true and the value for pillow should be false. Here are a few examples of items and the resulting values.

| items | mattress | pillow |
|---|---|---|
| [{..., "item_id": "M_PREM_K", "item_name": "Premium King Mattress", ...}] | true | false |

[{..., "item_id": "P_FOAM_S", "item_name": "Standard Foam Pillow", ...}] false
true

[{..., "item_id": "M_STAN_F", "item_name": "Standard Full Mattress", ...}] true
false

See documentation for the exists function.

You can use the condition expression item_name LIKE "%Mattress" to check whether the string item_name ends with the word "Mattress".

**-- TODO**

CREATE OR REPLACE TABLE sales_product_flags AS

SELECT *,

EXISTS(items, i -> i.item_name LIKE "%Mattress") as mattress,

EXISTS(items, i -> i.item_name LIKE "%Pillow") AS pillow

FROM sales;

**#display**

select * from sales_product_flags;

--The helper function below will return an error with a message on what needs to change if you have not followed instructions. No output means that you have completed this step.

%python

def check_table_results(table_name, num_rows, column_names):

assert spark.table(table_name), f"Table named **`{table_name}`** does not exist"

assert set(spark.table(table_name).columns) == set(column_names), "Please name the columns as shown in the schema above"

assert spark.table(table_name).count() == num_rows, f"The table should have {num_rows} records"

**--Run the cell below to confirm the table was created correctly.**

%python

check_table_results("sales_product_flags", 10510, ['items', 'mattress', 'pillow'])

product_counts = spark.sql("SELECT sum(CAST(mattress AS INT)) num_mattress, sum(CAST(pillow AS INT)) num_pillow FROM sales_product_flags").first().asDict()

assert product_counts == {'num_mattress': 9986, 'num_pillow': 1384}, "There should be 9986 rows where mattress is true, and 1384 where pillow is true"

---------- **Day 22** ----------

**--NoteBook: DE 3.1** - Schemas and Tables

**--Schemas and Tables on Databricks**

In this demonstration, you will create and explore schemas and tables.

**--Learning Objectives**

By the end of this lesson, you should be able to:

- Use Spark SQL DDL to define schemas and tables
- Describe how the LOCATION keyword impacts the default storage directory

**--Resources:**

- Schemas and Tables - Databricks Docs
- Managed and Unmanaged Tables

Creating a Table with the UI

Create a Local Table

Saving to Persistent Tables

**--Lesson Setup**

The following script clears out previous runs of this demo and configures some Hive variables that will be used in our SQL queries.

%run ../Includes/Classroom-Setup-03.1

**--Schemas**

Let's start by creating two schemas:

One with no LOCATION specified

One with LOCATION specified

**--How To Create Schema**

create schema demoschemaext LOCATION '/FileStore/tables/demo_db';

**--How to show databases**

show databases;

**--How to use existing database**

use pradeepworld0101_umhc_dbacademy_delp

**--Create schema**

```
CREATE SCHEMA IF NOT EXISTS ${da.schema_name}_default_location;

CREATE SCHEMA IF NOT EXISTS ${da.schema_name}_custom_location LOCATION
'${da.paths.working_dir}/${da.schema_name}_custom_location.db';
```

**--Note**: that the location of the first schema is in the default location under dbfs:/user/hive/warehouse/ and that the schema directory is the name of the schema with the .db extension

**--Describe Schema:**

```
DESCRIBE SCHEMA EXTENDED ${da.schema_name}_default_location;
```

**--Note:** that the location of the second schema is in the directory specified after the LOCATION keyword.

**--Describe Extended Schema**

```
DESCRIBE SCHEMA EXTENDED ${da.schema_name}_custom_location;
```

**--We will create a table in the schema with default location and insert data.**

**--Note** that the schema must be provided because there is no data from which to infer the schema.

```
USE ${da.schema_name}_default_location;

CREATE OR REPLACE TABLE managed_table_in_db_with_default_location (width
INT, length INT, height INT);

INSERT INTO managed_table_in_db_with_default_location

VALUES (3, 2, 1);
```

```
SELECT * FROM managed_table_in_db_with_default_location;
```

**--We can look at the extended table description to find the location (you'll need to scroll down in the results).**

```
DESCRIBE DETAIL managed_table_in_db_with_default_location;
```

**--By default, managed tables in a schema without the location specified will be created in the** dbfs:/user/hive/warehouse/<schema_name>.db/ directory.

We can see that, as expected, the data and metadata for our Delta Table are stored in that location.

```
%python
table_name = "managed_table_in_db_with_default_location"
tbl_location = spark.sql(f"DESCRIBE DETAIL {table_name}").first().location
print(tbl_location)

files = dbutils.fs.ls(tbl_location)
display(files)
```

**--Drop the table.**

```
DROP TABLE managed_table_in_db_with_default_location;
```

**--Note** the table's directory and its log and data files are deleted. Only the schema directory remains.

```
%python
```

```
schema_custom_location = spark.sql(f"DESCRIBE SCHEMA
{DA.schema_name}_custom_location").collect()[3].database_description_value

print(schema_custom_location)


dbutils.fs.ls(schema_custom_location)
```

**--Tables**

We will create an external (unmanaged) table from sample data.

The data we are going to use are in CSV format. We want to create a Delta table with a LOCATION provided in the directory of our choice.

```
USE ${da.schema_name}_default_location;


CREATE OR REPLACE TEMPORARY VIEW temp_delays

USING CSV OPTIONS (

  path = '${da.paths.datasets}/flights/departuredelays.csv',

  header = "true",

  mode = "FAILFAST" -- abort file parsing with a RuntimeException if any
malformed lines are encountered

);

CREATE OR REPLACE TABLE external_table LOCATION
'${da.paths.working_dir}/external_table' AS

  SELECT * FROM temp_delays;


SELECT * FROM external_table;
```

**--Count rows**

SELECT COUNT(*) FROM external_table;

**--Let's note the location of the table's data in this lesson's working directory.**

DESCRIBE TABLE EXTENDED external_table;

**--drop the table.**

DROP TABLE external_table;

**--The table definition no longer exists in the metastore, but the underlying data remain intact.**

%python

tbl_path = f"{DA.paths.working_dir}/external_table"

files = dbutils.fs.ls(tbl_path)

display(files)

**--Drop both schemas.**

DROP SCHEMA ${da.schema_name}_default_location CASCADE;

DROP SCHEMA ${da.schema_name}_custom_location CASCADE;

---------- **Day 23** ----------

**--Components of the Databricks Lakehouse**

The primary components of the Databricks Lakehouse are:

1. **Delta tables:**

ACID transactions

Data versioning

ETL

Indexing

2. **Unity Catalog:**

Data governance

Data sharing

Data auditing

- **Delta lake:**

Delta Lake is a data lake storage layer that provides ACID transactions, data versioning, and data auditing capabilities, enabling data engineers and data scientists to build reliable data pipelines and applications on top of big data platforms such as Apache Spark and Azure Databricks.

Delta Lake is an open source storage layer that brings reliability to data lakes. It is a storage layer on top of cloud storage that uses Apache Spark APIs to provide ACID transactions, data versioning, and audit history.

- **Delta log :**

Delta log is a type of file or database that only stores changes made to a system since the last checkpoint. It is used to track changes and diagnose problems.

Delta log is a type of log file or database that stores only the changes that have been made to a system since the last checkpoint. It is useful for tracking the history of changes made to a system in order to quickly identify and diagnose problems. Delta logs are often used in distributed systems to keep track of changes across multiple nodes.

- **Merge & Upsert :**

Merge and upsert are two distinct operations in databases. Merge is an operation that combines two datasets into one, while upsert is an operation that either inserts a new record into a table or updates an existing record in the same table.

Merge is used when combining data from multiple sources, while upsert is used when inserting or updating a single record.

---------- **Day 24** ----------

**--NoteBook: DE 3.2** - Version and Optimize Delta Tables

**--Versioning, Optimization, Vacuuming in Delta Lake**

Now that you feel comfortable performing basic data tasks with Delta Lake, we can discuss a few features unique to Delta Lake.

**Note:** that while some of the keywords used here aren't part of standard ANSI SQL, all Delta Lake operations can be run on Databricks using SQL

**--Learning Objectives**

By the end of this lesson, you should be able to:

Use OPTIMIZE to compact small files

Use ZORDER to index tables

Describe the directory structure of Delta Lake files

Review a history of table transactions

Query and roll back to previous table version

Clean up stale data files with VACUUM

**--Resources**

Delta Optimize - Databricks Docs

Delta Vacuum - Databricks Docs

**--Run Setup**

The first thing we're going to do is run a setup script. It will define a username, userhome, and database that is scoped to each user.

%run ../Includes/Classroom-Setup-03.2

**--Creating a Delta Table with History**

The cell below condenses all the transactions from the previous lesson into a single cell. (Except for the DROP TABLE!)

As you're waiting for this query to run, see if you can identify the total number of transactions being executed

**--Create Table**

```
CREATE TABLE if not exists students
  (id INT, name STRING, value DOUBLE);
```

**--Insert into students table**

```
INSERT INTO students VALUES (1, "Yve", 1.0);
INSERT INTO students VALUES (2, "Omar", 2.5);
INSERT INTO students VALUES (3, "Elia", 3.3);
```

**--insert multiple values in students table**

```
INSERT INTO students
VALUES
 (4, "Ted", 4.7),
 (5, "Tiffany", 5.5),
 (6, "Vini", 6.3);
```

**--Update students table record value + 1 where name starts with T**

```
UPDATE students
SET value = value + 1
WHERE name LIKE "T%";
```

**--Delete From students table where value is greater than 6**

```
DELETE FROM students
WHERE value > 6;
```

**--Create a temp view**

```
CREATE OR REPLACE TEMP VIEW updates(id, name, value, type) AS VALUES
 (2, "Omar", 15.2, "update"),
 (3, "", null, "delete"),
 (7, "Blue", 7.7, "insert"),
 (11, "Diya", 8.8, "update");
```

**--Merge table and view on certain condition**

```
MERGE INTO students b
USING updates u
ON b.id=u.id
WHEN MATCHED AND u.type = "update"
 THEN UPDATE SET *
WHEN MATCHED AND u.type = "delete"
 THEN DELETE
WHEN NOT MATCHED AND u.type = "insert"
 THEN INSERT *;
```

**--Examine Table Details**

Databricks uses a Hive metastore by default to register databases, tables, and views.

Using DESCRIBE EXTENDED allows us to see important metadata about our table

```
DESCRIBE EXTENDED students
```

**--DESCRIBE DETAIL is another command that allows us to explore table metadata.**

```
DESCRIBE DETAIL students
```

**--Note the Location field.**

While we've so far been thinking about our table as just a relational entity within a database, a Delta Lake table is actually backed by a collection of files stored in cloud object storage.

**--Explore Delta Lake Files**

We can see the files backing our Delta Lake table by using a Databricks Utilities function.

**NOTE:** It's not important right now to know everything about these files to work with Delta Lake, but it will help you gain a greater appreciation for how the technology is implemented.

```
%python
display(dbutils.fs.ls(f"{DA.paths.user_db}/students"))
```

**--Note that our directory contains a number of Parquet data files and a directory named _delta_log.**

Records in Delta Lake tables are stored as data in Parquet files.Transactions to Delta Lake tables are recorded in the _delta_log.

We can peek inside the _delta_log to see more.

```
%python
display(dbutils.fs.ls(f"{DA.paths.user_db}/students/_delta_log"))
```

**--Each transaction results in a new JSON file being written to the Delta Lake transaction log. Here, we can see that there are 8 total transactions against this table (Delta Lake is 0 indexed).**

**--Reasoning about Data Files**

We just saw a lot of data files for what is obviously a very small table.

DESCRIBE DETAIL allows us to see some other details about our Delta table, including the number of files.

```
DESCRIBE DETAIL students
```

--Here we see that our table currently contains 4 data files in its present version. So what are all those other Parquet files doing in our table directory?

Rather than overwriting or immediately deleting files containing changed data, Delta Lake uses the transaction log to indicate whether or not files are valid in a current version of the table.

Here, we'll look at the transaction log corresponding the MERGE statement above, where records were inserted, updated, and deleted.

```
%python

display(spark.sql(f"SELECT * FROM
json.`{DA.paths.user_db}/students/_delta_log/00000000000000000007.json`"))
```

--The add column contains a list of all the new files written to our table; the remove column indicates those files that no longer should be included in our table.

When we query a Delta Lake table, the query engine uses the transaction logs to resolve all the files that are valid in the current version, and ignores all other data files.

**--Compacting Small Files and Indexing**

Small files can occur for a variety of reasons; in our case, we performed a number of operations where only one or several records were inserted.

Files will be combined toward an optimal size (scaled based on the size of the table) by using the OPTIMIZE command.

OPTIMIZE will replace existing data files by combining records and rewriting the results.

When executing OPTIMIZE, users can optionally specify one or several fields for ZORDER indexing. While the specific math of Z-order is unimportant, it speeds up data retrieval when filtering on provided fields by colocating data with similar values within data files.

OPTIMIZE students

ZORDER BY id

**--Given how small our data is, ZORDER does not provide any benefit, but we can see all of the metrics that result from this operation.**

**--Reviewing Delta Lake Transactions**

**Because all changes to the Delta Lake table are stored in the transaction log, we can easily review the table history.**

DESCRIBE HISTORY students

--As expected, OPTIMIZE created another version of our table, meaning that version 8 is our most current version.

Remember all of those extra data files that had been marked as removed in our transaction log? These provide us with the ability to query previous versions of our table.

These time travel queries can be performed by specifying either the integer version or a timestamp.

**NOTE:** In most cases, you'll use a timestamp to recreate data at a time of interest. For our demo we'll use version, as this is deterministic (whereas you may be running this demo at any time in the future).

--get previous version of table students

SELECT *

FROM students VERSION AS OF 3

-- What's important to note about time travel is that we're not recreating a previous state of the table by undoing transactions against our current version; rather, we're just querying all those data files that were indicated as valid as of the specified version.

**--Rollback Versions**

Suppose you're typing up query to manually delete some records from a table and you accidentally execute this query in the following state.

DELETE FROM students

**--Note** that when we see a -1 for number of rows affected by a delete, this means an entire directory of data has been removed.

Let's confirm this below.

SELECT * FROM students

**--Deleting all the records in your table is probably not a desired outcome. Luckily, we can simply rollback this commit.**

RESTORE TABLE students TO VERSION AS OF 8

**--Note** that a RESTORE command is recorded as a transaction; you won't be able to completely hide the fact that you accidentally deleted all the records in the table, but you will be able to undo the operation and bring your table back to a desired state.

**--Cleaning Up Stale Files**

1. Databricks will automatically clean up stale files in Delta Lake tables.

2. While Delta Lake versioning and time travel are great for querying recent versions and rolling back queries, keeping the data files for all versions of large production tables around indefinitely is very expensive (and can lead to compliance issues if PII is present).

3. If you wish to manually purge old data files, this can be performed with the VACUUM operation.

4. Uncomment the following cell and execute it with a retention of 0 HOURS to keep only the current version:

--VACUUM students RETAIN 0 HOURS

--By default, VACUUM will prevent you from deleting files less than 7 days old, just to ensure that no long-running operations are still referencing any of the files to be deleted. If you run VACUUM on a Delta table, you lose the ability time travel back to a version older than the specified data retention period. In our demos, you may see Databricks executing code that specifies a retention of 0 HOURS. This is simply to demonstrate the feature and is not typically done in production.

In the following cell, we:

Turn off a check to prevent premature deletion of data files

Make sure that logging of VACUUM commands is enabled

se the DRY RUN version of vacuum to print out all records to be deleted

SET spark.databricks.delta.retentionDurationCheck.enabled = false;

SET spark.databricks.delta.vacuum.logging.enabled = true;

VACUUM students RETAIN 0 HOURS DRY RUN

**--By running VACUUM and deleting the 10 files above, we will permanently remove access to versions of the table that require these files to materialize.**

VACUUM students RETAIN 0 HOURS

**--Check the table directory to show that files have been successfully deleted.**

%python

display(dbutils.fs.ls(f"{DA.paths.user_db}/students"))

**--Run the following cell to delete the tables and files associated with this lesson.**

%python

DA.cleanup()

---

**--Assignment**

**--NoteBook:** DE 3.3L - Manipulate Delta Tables Lab

**--Manipulating Delta Tables Lab**

This notebook provides a hands-on review of some of the more esoteric features Delta Lake brings to the data lakehouse.

**--Learning Objectives**

By the end of this lab, you should be able to:

Review table history

Query previous table versions and rollback a table to a specific version

Perform file compaction and Z-order indexing

Preview files marked for permanent deletion and commit these deletes

**--Setup**

Run the following script to setup necessary variables and clear out past runs of this notebook. Note that re-executing this cell will allow you to start the lab over.

%run ../Includes/Classroom-Setup-03.3L

**--Recreate the History of your Bean Collection**

This lab picks up where the last lab left off. The cell below condenses all the operations from the last lab into a single cell (other than the final DROP TABLE statement).

For quick reference, the schema of the beans table created is:

| Field Name | Field type |
|---|---|
| name | STRING |
| color | STRING |
| grams | FLOAT |
| delicious | BOOLEAN |

CREATE TABLE beans

(name STRING, color STRING, grams FLOAT, delicious BOOLEAN);

```
INSERT INTO beans VALUES
("black", "black", 500, true),
("lentils", "brown", 1000, true),
("jelly", "rainbow", 42.5, false);

INSERT INTO beans VALUES
('pinto', 'brown', 1.5, true),
('green', 'green', 178.3, true),
('beanbag chair', 'white', 40000, false);

UPDATE beans
SET delicious = true
WHERE name = "jelly";

UPDATE beans
SET grams = 1500
WHERE name = 'pinto';

DELETE FROM beans
WHERE delicious = false;

CREATE OR REPLACE TEMP VIEW new_beans(name, color, grams, delicious) AS
VALUES
('black', 'black', 60.5, true),
('lentils', 'green', 500, true),
```

```
('kidney', 'red', 387.2, true),
('castor', 'brown', 25, false);

MERGE INTO beans a
USING new_beans b
ON a.name=b.name AND a.color = b.color
WHEN MATCHED THEN
  UPDATE SET grams = a.grams + b.grams
WHEN NOT MATCHED AND b.delicious = true THEN
  INSERT *;
```

**--Review the Table History**

Delta Lake's transaction log stores information about each transaction that modifies a table's contents or settings.

**Review the history of the beans table below.**

**-- TODO**

```
Describe extended beans;
```

**--If all the previous operations were completed as described you should see 7 versions of the table (NOTE: Delta Lake versioning starts with 0, so the max version number will be 6).**

**The operations should be as follows:**

**version operation**

0 CREATE TABLE

1 WRITE

2 WRITE

3 UPDATE

4 UPDATE

5 DELETE

6 MERGE

The operationsParameters column will let you review predicates used for updates, deletes, and merges. The operationMetrics column indicates how many rows and files are added in each operation.

Spend some time reviewing the Delta Lake history to understand which table version matches with a given transaction.

**NOTE:** The version column designates the state of a table once a given transaction completes. The readVersion column indicates the version of the table an operation executed against. In this simple demo (with no concurrent transactions), this relationship should always increment by 1.

**--How to check versioning**

Describe History beans

**--Query a Specific Version**

After reviewing the table history, you decide you want to view the state of your table after your very first data was inserted.

**--Run the query below to see this.**

SELECT * FROM beans VERSION AS OF 1

**--And now review the current state of your data.**

SELECT * FROM beans

**--You want to review the weights of your beans before you deleted any records.**

Fill in the statement below to register a temporary view of the version just before data was deleted, then run the following cell to query the view.

**--TODO**

CREATE OR REPLACE TEMP VIEW pre_delete_vw AS

select * from beans version as of 4;

**--Display View**

SELECT * FROM pre_delete_vw

**--Run the cell below to check that you have captured the correct version.**

%python

assert spark.table("pre_delete_vw"), "Make sure you have registered the temporary view with the provided name `pre_delete_vw`"

assert spark.table("pre_delete_vw").count() == 6, "Make sure you're querying a version of the table with 6 records"

assert spark.table("pre_delete_vw").selectExpr("int(sum(grams))").first()[0] == 43220, "Make sure you query the version of the table after updates were applied"

**--Restore a Previous Version**

Apparently there was a misunderstanding; the beans your friend gave you that you merged into your collection were not intended for you to keep.

Revert your table to the version before this MERGE statement completed.

**--TODO**

restore table beans to version as of 5

**--History after restore**

DESCRIBE HISTORY beans

**--To check it is correct or not**

%python

last_tx = spark.conf.get("spark.databricks.delta.lastCommitVersionInSession")

assert spark.sql(f"DESCRIBE HISTORY beans").select("operation").first()[0] == "RESTORE", "Make sure you reverted your table with the `RESTORE` keyword"

assert spark.table("beans").count() == 5, "Make sure you reverted to the version after deleting records but before merging"

**--File Compaction**

Looking at the transaction metrics during your reversion, you are surprised you have some many files for such a small collection of data.

While indexing on a table of this size is unlikely to improve performance, you decide to add a Z-order index on the name field in anticipation of your bean collection growing exponentially over time.

Use the cell below to perform file compaction and Z-order indexing.

**--TODO**

optimize beans

zorder by name

DESCRIBE DETAIL beans

**--Run the cell below to check that you've successfully optimized and indexed your table.**

%python

last_tx = spark.sql("DESCRIBE HISTORY beans").first()

assert last_tx["operation"] == "OPTIMIZE", "Make sure you used the `OPTIMIZE` command to perform file compaction"

assert last_tx["operationParameters"]["zOrderBy"] == '["name"]', "Use `ZORDER BY name` with your optimize command to index your table"

**--Cleaning Up Stale Data Files**

You know that while all your data now resides in 1 data file, the data files from previous versions of your table are still being stored alongside this. You wish to remove these files and remove access to previous versions of the table by running VACUUM on the table.

Executing VACUUM performs garbage cleanup on the table directory. By default, a retention threshold of 7 days will be enforced.

The cell below modifies some Spark configurations. The first command overrides the retention threshold check to allow us to demonstrate permanent removal of data.

**NOTE:** Vacuuming a production table with a short retention can lead to data corruption and/or failure of long-running queries. This is for demonstration purposes only and extreme caution should be used when disabling this setting.

The second command sets spark.databricks.delta.vacuum.logging.enabled to true to ensure that the VACUUM operation is recorded in the transaction log.

**NOTE:** Because of slight differences in storage protocols on various clouds, logging VACUUM commands is not on by default for some clouds as of DBR 9.1.

```
SET spark.databricks.delta.retentionDurationCheck.enabled = false;

SET spark.databricks.delta.vacuum.logging.enabled = true;
```

**--Before permanently deleting data files, review them manually using the DRY RUN option.**

```
VACUUM beans RETAIN 0 HOURS DRY RUN
```

**--All data files not in the current version of the table will be shown in the preview above.**

Run the command again without DRY RUN to permanently delete these files.

NOTE: All previous versions of the table will no longer be accessible.

```
VACUUM beans RETAIN 0 HOURS
```

--Because VACUUM can be such a destructive act for important datasets, it's always a good idea to turn the retention duration check back on. Run the cell below to reactive this setting.

SET spark.databricks.delta.retentionDurationCheck.enabled = true

**--Note** that the table history will indicate the user that completed the VACUUM operation, the number of files deleted, and log that the retention check was disabled during this operation.

DESCRIBE HISTORY beans

--Query your table again to confirm you still have access to the current version.

SELECT * FROM beans

--Because Delta Cache stores copies of files queried in the current session on storage volumes deployed to your currently active cluster, you may still be able to temporarily access previous table versions (though systems should not be designed to expect this behavior).

Restarting the cluster will ensure that these cached data files are permanently purged.

You can see an example of this by uncommenting and running the following cell that may, or may not, fail (depending on the state of the cache).

SELECT * FROM beans@v1

**--By completing this lab, you should now feel comfortable:**

Completing standard Delta Lake table creation and data manipulation commands

Reviewing table metadata including table history

Leverage Delta Lake versioning for snapshot queries and rollbacks

Compacting small files and indexing tables

Using VACUUM to review files marked for deletion and committing these deletes

**--Run the following cell to delete the tables and files associated with this lesson.**

%python

DA.cleanup()

---------- **Day 25** ----------

**--NoteBook: DE 3.4** - Set Up Delta Tables

**--Setting Up Delta Tables**

After extracting data from external data sources, load data into the Lakehouse to ensure that all of the benefits of the Databricks platform can be fully leveraged.

While different organizations may have varying policies for how data is initially loaded into Databricks, we typically recommend that early tables represent a mostly raw version of the data, and that validation and enrichment occur in later stages. This pattern ensures that even if data doesn't match expectations with regards to data types or column names, no data will be dropped, meaning that programmatic or manual intervention can still salvage data in a partially corrupted or invalid state.

This lesson will focus primarily on the pattern used to create most tables, CREATE TABLE _ AS SELECT (CTAS) statements.

**--Learning Objectives**

By the end of this lesson, you should be able to:

Use CTAS statements to create Delta Lake tables

Create new tables from existing views or tables

Enrich loaded data with additional metadata

Declare table schema with generated columns and descriptive comments

Set advanced options to control data location, quality enforcement, and partitioning

**--Run Setup**

The setup script will create the data and declare necessary values for the rest of this notebook to execute.

%run ../Includes/Classroom-Setup-03.4

**--Create Table as Select (CTAS)**

CREATE TABLE AS SELECT statements create and populate Delta tables using data retrieved from an input query.

CREATE OR REPLACE TABLE sales AS

SELECT * FROM parquet.`${DA.paths.datasets}/ecommerce/raw/sales-historical`;

DESCRIBE EXTENDED sales;

--CTAS statements automatically infer schema information from query results and do not support manual schema declaration.

This means that CTAS statements are useful for external data ingestion from sources with well-defined schema, such as Parquet files and tables.

CTAS statements also do not support specifying additional file options.

We can see how this would present significant limitations when trying to ingest data from CSV files.

```
CREATE OR REPLACE TABLE sales_unparsed AS
SELECT * FROM csv.`${da.paths.datasets}/ecommerce/raw/sales-csv`;

SELECT * FROM sales_unparsed;

--To correctly ingest this data to a Delta Lake table, we'll need to use a reference to the files that allows us to specify options.
In the previous lesson, we showed doing this by registering an external table. Here, we'll slightly evolve this syntax to specify the options to a temporary view, and then use this temp view as the source for a CTAS statement to successfully register the Delta table.

CREATE OR REPLACE TEMP VIEW sales_tmp_vw
  (order_id LONG, email STRING, transactions_timestamp LONG, total_item_quantity INTEGER, purchase_revenue_in_usd DOUBLE, unique_items INTEGER, items STRING)
USING CSV
OPTIONS (
  path = "${da.paths.datasets}/ecommerce/raw/sales-csv",
  header = "true",
  delimiter = "|"
);
```

```
CREATE TABLE sales_delta AS
SELECT * FROM sales_tmp_vw;

SELECT * FROM sales_delta
```

**--Filtering and Renaming Columns from Existing Tables**

Simple transformations like changing column names or omitting columns from target tables can be easily accomplished during table creation.

The following statement creates a new table containing a subset of columns from the sales table.

Here, we'll presume that we're intentionally leaving out information that potentially identifies the user or that provides itemized purchase details. We'll also rename our fields with the assumption that a downstream system has different naming conventions than our source data.

```
CREATE OR REPLACE TABLE purchases AS
SELECT order_id AS id, transaction_timestamp, purchase_revenue_in_usd AS price
FROM sales;

SELECT * FROM purchases
```

**--Note that we could have accomplished this same goal with a view, as shown below.**

```
CREATE OR REPLACE VIEW purchases_vw AS
```

```
SELECT order_id AS id, transaction_timestamp, purchase_revenue_in_usd AS
price

FROM sales;

FROM sales;


SELECT * FROM purchases_vw
```

**--Declare Schema with Generated Columns**

As noted previously, CTAS statements do not support schema declaration. We note above that the timestamp column appears to be some variant of a Unix timestamp, which may not be the most useful for our analysts to derive insights. This is a situation where generated columns would be beneficial.

Generated columns are a special type of column whose values are automatically generated based on a user-specified function over other columns in the Delta table (introduced in DBR 8.3).

The code below demonstrates creating a new table while:

- Specifying column names and types
- Adding a generated column to calculate the date
- Providing a descriptive column comment for the generated column

```
CREATE OR REPLACE TABLE purchase_dates (
  id STRING,
  transaction_timestamp STRING,
  price STRING,
  date DATE GENERATED ALWAYS AS (
```

```
  cast(cast(transaction_timestamp/1e6 AS TIMESTAMP) AS DATE))
  COMMENT "generated based on `transactions_timestamp` column")
```

--Because date is a generated column, if we write to purchase_dates without providing values for the date column, Delta Lake automatically computes them.

NOTE: The cell below configures a setting to allow for generating columns when using a Delta Lake MERGE statement. We'll see more on this syntax later in the course.

```
SET spark.databricks.delta.schema.autoMerge.enabled=true;
```

```
MERGE INTO purchase_dates a
USING purchases b
ON a.id = b.id
WHEN NOT MATCHED THEN
  INSERT *
```

--We can see below that all dates were computed correctly as data was inserted, although neither our source data or insert query specified the values in this field.

As with any Delta Lake source, the query automatically reads the most recent snapshot of the table for any query; you never need to run REFRESH TABLE.

```
SELECT * FROM purchase_dates
```

--It's important to note that if a field that would otherwise be generated is included in an insert to a table, this insert will fail if the value provided does not

exactly match the value that would be derived by the logic used to define the generated column.

We can see this error by uncommenting and running the cell below:

-- INSERT INTO purchase_dates VALUES

-- (1, 600000000, 42.0, "2020-06-18")

**--Add a Table Constraint**

The error message above refers to a CHECK constraint. Generated columns are a special implementation of check constraints.

Because Delta Lake enforces schema on write, Databricks can support standard SQL constraint management clauses to ensure the quality and integrity of data added to a table.

-Databricks currently support two types of constraints:

NOT NULL constraints

CHECK constraints

In both cases, you must ensure that no data violating the constraint is already in the table prior to defining the constraint. Once a constraint has been added to a table, data violating the constraint will result in write failure.

Below, we'll add a CHECK constraint to the date column of our table. Note that CHECK constraints look like standard WHERE clauses you might use to filter a dataset.

ALTER TABLE purchase_dates ADD CONSTRAINT valid_date CHECK (date > '2020-01-01');

**--Table constraints are shown in the TBLPROPERTIES field.**

DESCRIBE EXTENDED purchase_dates

**--Enrich Tables with Additional Options and Metadata**

So far we've only scratched the surface as far as the options for enriching Delta Lake tables.

Below, we show evolving a CTAS statement to include a number of additional configurations and metadata.

Our SELECT clause leverages two built-in Spark SQL commands useful for file ingestion:

current_timestamp() records the timestamp when the logic is executed

input_file_name() records the source data file for each record in the table

We also include logic to create a new date column derived from timestamp data in the source.

**The CREATE TABLE clause contains several options:**

A COMMENT is added to allow for easier discovery of table contents

A LOCATION is specified, which will result in an external (rather than managed) table

The table is PARTITIONED BY a date column; this means that the data from each data will exist within its own directory in the target storage location

**NOTE:** Partitioning is shown here primarily to demonstrate syntax and impact. Most Delta Lake tables (especially small-to-medium sized data) will not benefit from partitioning. Because partitioning physically separates data files, this approach can result in a small files problem and prevent file compaction and

efficient data skipping. The benefits observed in Hive or HDFS do not translate to Delta Lake, and you should consult with an experienced Delta Lake architect before partitioning tables.

As a best practice, you should default to non-partitioned tables for most use cases when working with Delta Lake.

```
CREATE OR REPLACE TABLE users_pii
COMMENT "Contains PII"
LOCATION "${da.paths.working_dir}/tmp/users_pii"
PARTITIONED BY (first_touch_date)
AS
  SELECT *,
    cast(cast(user_first_touch_timestamp/1e6 AS TIMESTAMP) AS DATE) first_touch_date,
    current_timestamp() updated,
    input_file_name() source_file
  FROM parquet.`${da.paths.datasets}/ecommerce/raw/users-historical/`;

SELECT * FROM users_pii;
```

--The metadata fields added to the table provide useful information to understand when records were inserted and from where. This can be especially helpful if troubleshooting problems in the source data becomes necessary.

All of the comments and properties for a given table can be reviewed using DESCRIBE TABLE EXTENDED.

**NOTE:** Delta Lake automatically adds several table properties on table creation.

```
DESCRIBE EXTENDED users_pii
```

```
--Listing the location used for the table reveals that the unique values in the partition column **`first_touch_date`** are used to create data directories.
```

```
%python
files = dbutils.fs.ls(f"{DA.paths.working_dir}/tmp/users_pii")
display(files)
```

**--Cloning Delta Lake Tables**

Delta Lake has two options for efficiently copying Delta Lake tables.

DEEP CLONE fully copies data and metadata from a source table to a target. This copy occurs incrementally, so executing this command again can sync changes from the source to the target location.

```
CREATE OR REPLACE TABLE purchases_clone
DEEP CLONE purchases
```

```
--Because all the data files must be copied over, this can take quite a while for large datasets.
```

If you wish to create a copy of a table quickly to test out applying changes without the risk of modifying the current table, SHALLOW CLONE can be a good option.

Shallow clones just copy the Delta transaction logs, meaning that the data doesn't move.

```
CREATE OR REPLACE TABLE purchases_shallow_clone
SHALLOW CLONE purchases
```

--In either case, data modifications applied to the cloned version of the table will be tracked and stored separately from the source. Cloning is a great way to set up tables for testing SQL code while still in development.

**--Summary**

In this notebook, we focused primarily on DDL and syntax for creating Delta Lake tables. In the next notebook, we'll explore options for writing updates to tables.

---

**--NoteBook: DE 3.5** - Load Data into Delta Lake

**--Loading Data into Delta Lake**

Delta Lake tables provide ACID compliant updates to tables backed by data files in cloud object storage.

In this notebook, we'll explore SQL syntax to process updates with Delta Lake. While many operations are standard SQL, slight variations exist to accommodate Spark and Delta Lake execution.

**--Learning Objectives**

By the end of this lesson, you should be able to:

a. Overwrite data tables using INSERT OVERWRITE
b. Append to a table using INSERT INTO
c. Append, update, and delete from a table using MERGE INTO
d. Ingest data incrementally into tables using COPY INTO

**--Run Setup**

The setup script will create the data and declare necessary values for the rest of this notebook to execute.

%run ../Includes/Classroom-Setup-03.5

**--Complete Overwrites**

We can use overwrites to atomically replace all of the data in a table. There are multiple benefits to overwriting tables instead of deleting and recreating tables:

Overwriting a table is much faster because it doesn't need to list the directory recursively or delete any files.

The old version of the table still exists; can easily retrieve the old data using Time Travel.

It's an atomic operation. Concurrent queries can still read the table while you are deleting the table.

Due to ACID transaction guarantees, if overwriting the table fails, the table will be in its previous state.

Spark SQL provides two easy methods to accomplish complete overwrites.

Some students may have noticed previous lesson on CTAS statements actually used CRAS statements (to avoid potential errors if a cell was run multiple times).

CREATE OR REPLACE TABLE (CRAS) statements fully replace the contents of a table each time they execute.

```
CREATE OR REPLACE TABLE events AS

SELECT * FROM parquet.`${da.paths.datasets}/ecommerce/raw/events-historical`
```

**--Reviewing the table history shows a previous version of this table was replaced.**

```
DESCRIBE HISTORY events
```

--INSERT OVERWRITE provides a nearly identical outcome as above: data in the target table will be replaced by data from the query.

**INSERT OVERWRITE:**

a. Can only overwrite an existing table, not create a new one like our CRAS statement
b. Can overwrite only with new records that match the current table schema -- and thus can be a "safer" technique for overwriting an existing table without disrupting downstream consumers
c. Can overwrite individual partitions

```
INSERT OVERWRITE sales

SELECT * FROM parquet.`${da.paths.datasets}/ecommerce/raw/sales-historical/`
```

--Note that different metrics are displayed than a CRAS statement; the table history also records the operation differently.

```
DESCRIBE HISTORY sales
```

--A primary difference here has to do with how Delta Lake enforces schema on write.

Whereas a CRAS statement will allow us to completely redefine the contents of our target table, INSERT OVERWRITE will fail if we try to change our schema (unless we provide optional settings).

Uncomment and run the cell below to generate an expected error message.

```
-- INSERT OVERWRITE sales

-- SELECT *, current_timestamp() FROM
parquet.`${da.paths.datasets}/ecommerce/raw/sales-historical`
```

**--Append Rows**

We can use INSERT INTO to atomically append new rows to an existing Delta table. This allows for incremental updates to existing tables, which is much more efficient than overwriting each time.

Append new sale records to the sales table using INSERT INTO.

```
INSERT INTO sales

SELECT * FROM parquet.`${da.paths.datasets}/ecommerce/raw/sales-30m`
```

**--Note** that INSERT INTO does not have any built-in guarantees to prevent inserting the same records multiple times. Re-executing the above cell would write the same records to the target table, resulting in duplicate records.

**--Merge Updates**

You can upsert data from a source table, view, or DataFrame into a target Delta table using the MERGE SQL operation. Delta Lake supports inserts, updates and deletes in MERGE, and supports extended syntax beyond the SQL standards to facilitate advanced use cases.

```
MERGE INTO target a
USING source b
ON {merge_condition}
WHEN MATCHED THEN {matched_action}
WHEN NOT MATCHED THEN {not_matched_action}
```

We will use the MERGE operation to update historic users data with updated emails and new users.

```
CREATE OR REPLACE TEMP VIEW users_update AS
SELECT *, current_timestamp() AS updated
FROM parquet.`${da.paths.datasets}/ecommerce/raw/users-30m`;

select * from users_update;
```

**--The main benefits of MERGE:**

a. updates, inserts, and deletes are completed as a single transaction
b. multiple conditionals can be added in addition to matching fields
c. provides extensive options for implementing custom logic

Below, we'll only update records if the current row has a NULL email and the new row does not.

All unmatched records from the new batch will be inserted.

```
MERGE INTO users a
```

```
USING users_update b
ON a.user_id = b.user_id
WHEN MATCHED AND a.email IS NULL AND b.email IS NOT NULL THEN
  UPDATE SET email = b.email, updated = b.updated
WHEN NOT MATCHED THEN INSERT *
```

**--Note** that we explicitly specify the behavior of this function for both the MATCHED and NOT MATCHED conditions; the example demonstrated here is just an example of logic that can be applied, rather than indicative of all MERGE behavior.

**--Insert-Only Merge for Deduplication**

A common ETL use case is to collect logs or other every-appending datasets into a Delta table through a series of append operations.

Many source systems can generate duplicate records. With merge, you can avoid inserting the duplicate records by performing an insert-only merge.

This optimized command uses the same MERGE syntax but only provided a WHEN NOT MATCHED clause.

Below, we use this to confirm that records with the same user_id and event_timestamp aren't already in the events table.

```
MERGE INTO events a
USING events_update b
ON a.user_id = b.user_id AND a.event_timestamp = b.event_timestamp
WHEN NOT MATCHED AND b.traffic_source = 'email' THEN
  INSERT *
```

**--Load Incrementally**

COPY INTO provides SQL engineers an idempotent option to incrementally ingest data from external systems.

**Note that this operation does have some expectations:**

a. Data schema should be consistent
b. Duplicate records should try to be excluded or handled downstream
c. This operation is potentially much cheaper than full table scans for data that grows predictably.

While here we'll show simple execution on a static directory, the real value is in multiple executions over time picking up new files in the source automatically.

COPY INTO sales

FROM "${da.paths.datasets}/ecommerce/raw/sales-30m"

FILEFORMAT = PARQUET

---

**--NoteBook: DE 3.6L** - Load Data Lab

**--Load Data Lab**

In this lab, you will load data into new and existing Delta tables.

**-Learning Objectives**

By the end of this lab, you should be able to:

a. Create an empty Delta table with a provided schema
b. Insert records from an existing table into a Delta table
c. Use a CTAS statement to create a Delta table from files

**--Run Setup**

Run the following cell to configure variables and datasets for this lesson.

%run ../Includes/Classroom-Setup-03.6L

**--Data Overview**

We will work with a sample of raw Kafka data written as JSON files.

Each file contains all records consumed during a 5-second interval, stored with the full Kafka schema as a multiple-record JSON file.

The schema for the table:

| field | type | description |
|---|---|---|
| key | BINARY | The user_id field is used as the key; this is a unique alphanumeric field that corresponds to session/cookie information |
| offset | LONG | This is a unique value, monotonically increasing for each partition |
| partition | INTEGER | Our current Kafka implementation uses only 2 partitions (0 and 1) |
| timestamp | LONG | This timestamp is recorded as milliseconds since epoch, and represents the time at which the producer appends a record to a partition |
| topic | STRING | While the Kafka service hosts multiple topics, only those records from the clickstream topic are included here |
| value | BINARY | This is the full data payload (to be discussed later), sent as JSON |

**--Define Schema for Empty Delta Table**

Create an empty managed Delta table named events_raw using the same schema.

CREATE or replace TABLE events_raw

(key BINARY, offset LONG, partition INTEGER, timestamp LONG, topic STRING, value BINARY);

**--Insert Raw Events into Delta Table**

Once the extracted data and Delta table are ready, insert the JSON records from the events_json table into the new events_raw Delta table.

**--TODO**

INSERT INTO EVENTS_RAW

SELECT * FROM EVENTS_JSON;

**--TODO**

select * from events_raw;

**--Create a Delta Table From Query Results**

In addition to new events data, let's also load a small lookup table that provides product details that we'll use later in the course. Use a CTAS statement to create a managed Delta table named item_lookup that extracts data from the parquet directory provided below.

**--TODO**

Create or REPLACE table item_lookup as

select * from PARQUET.`${da.paths.datasets}/ecommerce/raw/item-lookup`;

---------- **Day 26** ----------

**--NoteBook : ASP 4.1**

**--Spark Component:**

HDFS / DBFS / BLOB / S3 --> Spark Core RDD --> DataFrame, DataSet, SparkSQL

--> Sparl Streaming, Structured Streaming

--EventHUB : used for streaming Data

---------- **Day 27** ----------

**--Azure Databrics :** This NoteBook belongs on Azure Databricks only.

**--Job Scheduling:** we have to create a azure databricks notebook and then write a notebook as follows

```
inputDF = spark.read.text("/FileStore/tables/sample.txt")

from pyspark.sql.functions import *

wcDF = inputDF.withColumn("word", explode(split(col("value"), " "))).groupBy("word").count()

wcDF.write.save("/FileStore/tables/Output/")
```

**--How to run NoteBook or Schedule Notebook on job**

1.Create Job in Compute

2.Select NoteBook to execute

3.Select previous cluster

4. Check SparkUI to check job is executing or not

--SSO --> single signon

**--Method to connect Azure data with databricks**

1. Mount

2. WBS

3. SAS: Shared Access Signature

**--How exactly job execute**

backend of groupBy

write and read boundries

---------- **Day 28** ----------

--egg: to bundle python file together

--Spark job architecture

---------- **Day 29** ----------

**--To store any DF/reused code in memory for future use (it will not erase after end of session)**

1. persist() ==> salesDF.persist()

2. cache() ==> salesDF.cache()

3. salesDF.unpersist()

**--Cluster for streaming data use delta cache accelerated**

**--JOB monitor happens in 3 ways**

1. Driver logs

2. DAG

3. Spark UI

**--Storage Level of all API:**

**--Unity catalog:**

Unity Catalog is a unified governance solution for all data and AI assets including files, tables, machine learning models and dashboards in your lakehouse on any cloud.

1. Centrally manage and govern all data assets
2. Manage fine-grained access controls
3. Unified and secure data search experience
4. Enhanced query performance at any scale
5. Automated and real-time data lineage
6. Secure data sharing across organizations

--validate xml via xsd

---------- **Delta Live Table (Streaming Data)** ----------

**Note:** To run this command you need to download taxi dataset which is available in free datasource of databricks. Or I can add this project in our git repository:

https://github.com/wagh-pradeep/databricks.git

**--Delta Live Table**

**Notebook: 5_OrchestratingJobs_DLT_Script**

**--Declare Bronze Table**

CREATE OR REFRESH STREAMING LIVE TABLE greenTrip_bronze

AS SELECT current_timestamp() receipt_time, input_file_name() source_file, *

FROM
cloud_files("dbfs:/FileStore/tables/nyctaxidata/dltlanding/greendata/green_tripd

```
ata", "csv", map("cloudFiles.schemaHints", "passenger_count
INTEGER,trip_distance DOUBLE, total_amount DOUBLE","header", "true",
"cloudFiles.inferColumnTypes", "true"))



--Taxi Zones File

CREATE OR REFRESH STREAMING LIVE TABLE taxi_zones

AS SELECT * FROM
cloud_files("dbfs:/FileStore/tables/nyctaxidata/dltlanding/yellow_green_commo
ndata/taxizones", "csv", map("header", "true", "cloudFiles.inferColumnTypes",
"true"))


--Declare Silver Tables

-- VendorID, lpep_pickup_datetime, lpep_dropoff_datetime, passenger_count,

-- trip_distance, DOLocationID, total_amount, Borough, Zone,service_zone


CREATE OR REFRESH STREAMING LIVE TABLE greenTrip_taxiZones_Silver

(CONSTRAINT positive_passenger_count EXPECT (passenger_count > 0) ON
VIOLATION DROP ROW,

CONSTRAINT positive_trip_distance EXPECT (trip_distance > 0.0) ON VIOLATION
DROP ROW,

CONSTRAINT positive_total_amount EXPECT (total_amount > 0.0) ON VIOLATION
DROP ROW,

CONSTRAINT pickup_drop_datetime_not_same EXPECT
(pickup_datetime!=dropoff_datetime ) ON VIOLATION DROP ROW

 )

AS SELECT
```

```
a.VendorID,
a.lpep_pickup_datetime pickup_datetime,
a.lpep_dropoff_datetime dropoff_datetime,
CAST(a.passenger_count AS INTEGER) passenger_count,
CAST(a.trip_distance AS DOUBLE) trip_distance,
a.DOLocationID,
CAST(a.total_amount AS DOUBLE) total_amount,
b.Borough,
b.Zone,
b.service_zone
FROM STREAM(live.greenTrip_bronze) a
INNER JOIN STREAM(live.taxi_zones) b
ON a.DOLocationID = b.LocationID;
```

**--Gold Table**

**--Zone wise Passenger count for each vendor**

```
CREATE OR REFRESH LIVE TABLE greenvendor_zone_passenger_count
COMMENT "Total Passenger count for each green taxi vendor for each zone"
AS SELECT VendorID, Zone, sum(passenger_count) total_passenger_count
FROM live.greenTrip_taxiZones_Silver
GROUP BY VendorID, Zone;
```

**--Borough wise total fare collected by each vendor**

```
CREATE OR REFRESH LIVE TABLE greenvendor_borough_total_fare
COMMENT "Total Fare amount for each green taxi vendor for each borough"
AS SELECT VendorID, Borough, sum(total_amount) total_fare_amount
FROM live.greenTrip_taxiZones_Silver
GROUP BY VendorID, Borough;
```

**--Q. How to schedule Delta live tables pipeline for streaming data:**

ANS : workflow >> Delta live table >> Schedule pipeline

**--Worccount program**

```
from pyspark.sql.functions import *
inputDF = spark.read.text("/FileStore/tables/sample.txt")
display(inputDF)
wcDF = inputDF.withColumn("word", explode(split(col("value"), " "))).groupBy("word").count()
wcDF.write.save("/FileStore/tables/Output/")
```

**--Q. How to delete files from databricks:**

ANS:

```
dbutils.fs.rm("/fileSrote/", recurse=True)
```

**--Q. Cloud Features:**

1. Reliable
2. Scaling

1. Horizontal Scaling:
2. Vertical Scaling: increasing RAM or capabilities called vertical scaling

3. Security
    1. encryption
    2. decryption

**--Data Mapping**

Mapping data from on-prem to cloud

1. Check column name
2. Check data types
3. Change column name
4. Check count after mapping