Disclaimer: These slides are copyrighted and strictly for personal use only

- This document is reserved for people enrolled into the [Ultimate Big Data Masters Program (Cloud Focused) by Sumit Sir]

- **Please do not share this document,** it is intended for personal use and exam preparation only, thank you.

- If you've obtained these slides for free on a website that is not the course's website, please reach out to legal@trendytech.in . Thanks!

- All the Best and Happy Learning!

Higher Level APIs in Apache Spark

1. **Dataframes**
2. **Spark SQL**

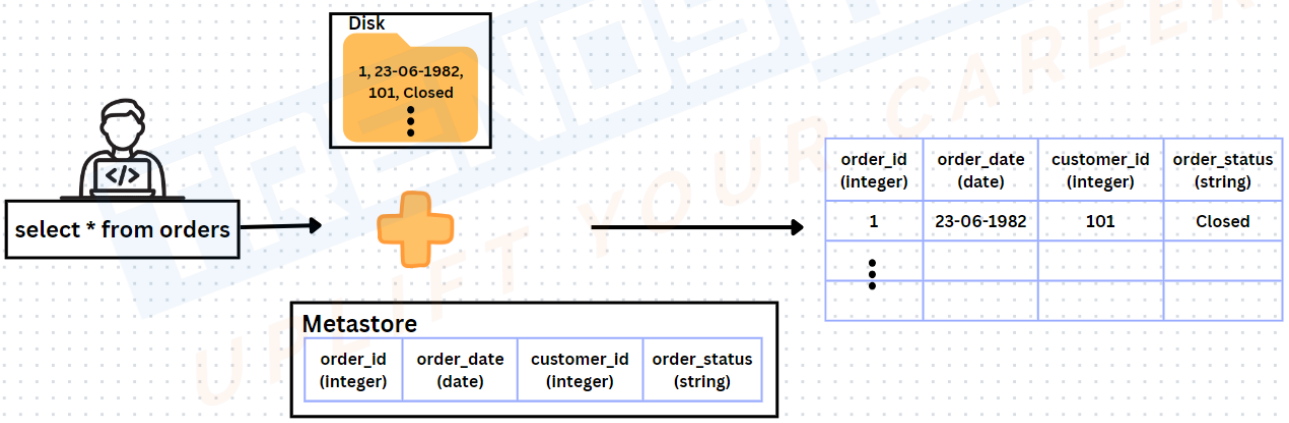
(Datasets is another language specific High Level API which is functional in case of Scala and Java and not with Python)

Previously we have understood about RDDs and its functionality -

RDDs consists of raw data distributed across partitions without any associated structure (schema/metadata)

In case of Normal Databases, where the data is stored in the form of tables, there are 2 things to it -

1. Actual Data
2. Schema / Metadata



In the case of Apache Spark,

1. **Spark SQL/ Spark Table** - Is Persistent over different sessions. Data files are stored on the Disk (any storage like Datalake / HDFS / S3) and the schema / metadata is in a Metastore (Database)

2. **Dataframes** - are in the form of RDDs with some structure / schema that is not Persistent as it is available only in that particular session. Data is stored in memory and Metadata is saved in a temporary metadata catalog.

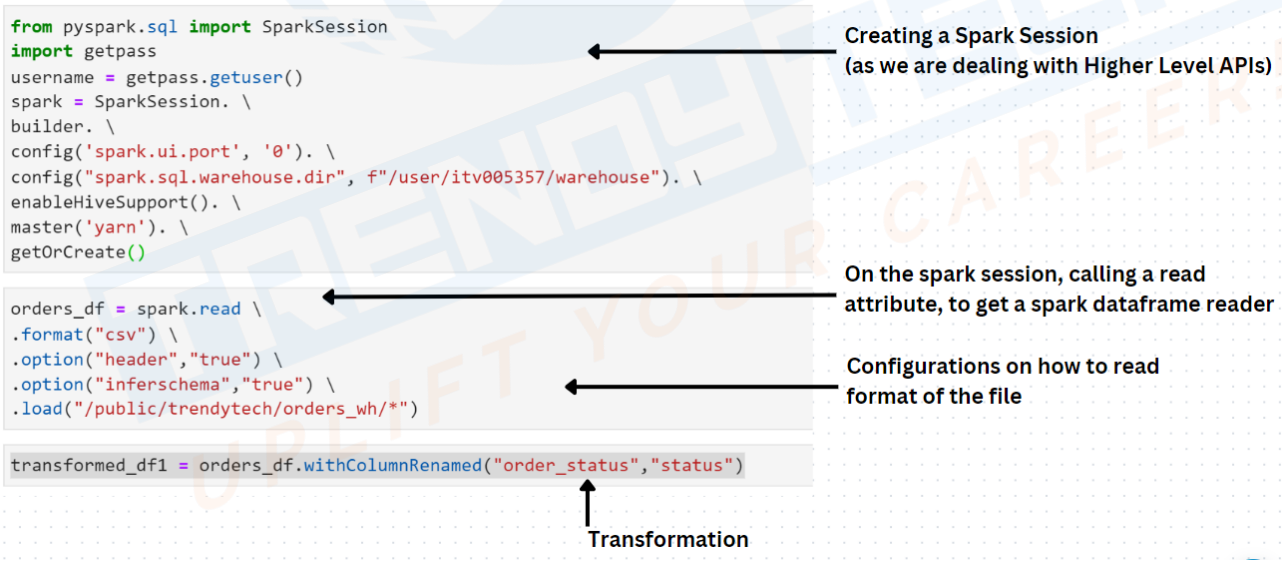
(Note: Why are Higher Level APIs more performant ?

As the system, that is the spark engine is provided with more context to handle the data effectively with the help of metadata information.)

Working of Dataframes

1. Load the data file and create a Spark Dataframe
2. Apply Transformations
3. Write the results back to Storage.

(Spark Session is an entry point to the Spark Cluster in case of Higher Level APIs. Spark Context is for lower level RDDs)



Note : Inferschema configuration attribute should be avoided as the inference of the data types may not always be correct and it takes some additional execution time to infer the schema.

Dataframe Reader

Standard way to create a Dataframe -

```
df = spark.read \
.format("csv") \
.option("header","true") \
.option("inferschema","true") \
.load("<file-path>")
```

Alternate shortcuts to create a Dataframe for different file formats :

csv

```
df = spark.read \
.csv(“<file-path>”, header = “true”, inferSchema = “true”)
```

json

```
df = spark.read \
.json(“<file-path>”)
```

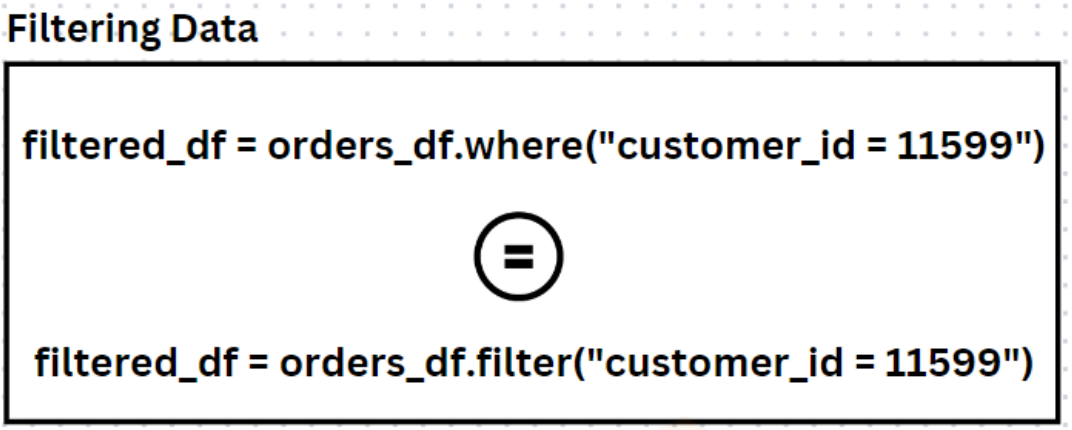
parquet (column based file format with embedded schema and is most compatible with Spark)

```
df = spark.read \
.parquet(“<file-path>”)
```

orc

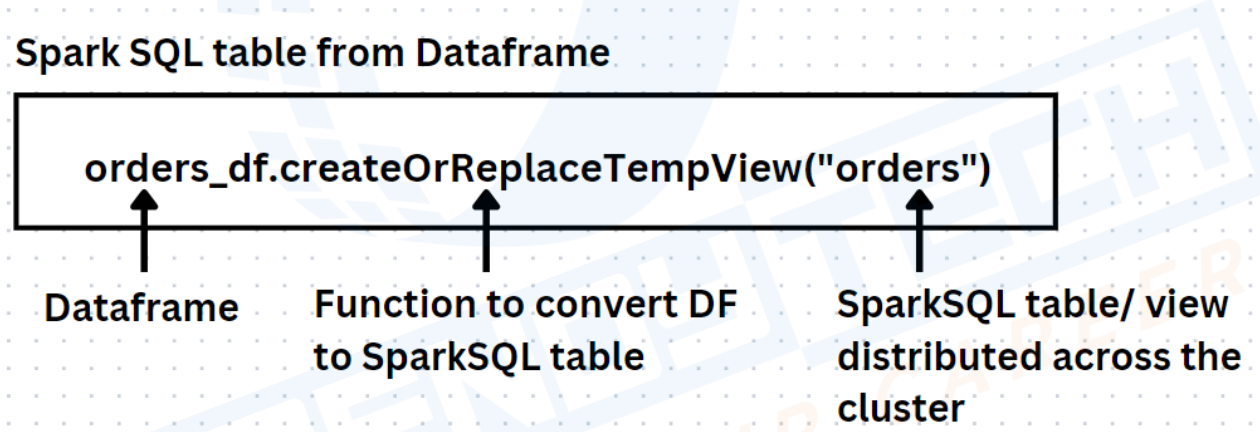
```
df = spark.read \
.orc(“<file-path>”)
```

Some transformations



Dataframes and Spark SQL table

Dataframes and Spark SQL tables are interconvertible.



Now, we can execute normal SQL queries on the SparkSQL table view created from a Dataframe,

```
filtered_df = spark.sql("select * from orders where order_status = 'Closed'")
```

In order to be able to work in a normal SQL style (much convenient for SQL developers), a SparkSQL table view is created from a Dataframe.

Converting from SparkSQL table to a Dataframe

```
orders_df = spark.read.table("orders")
```

```
orders_df.show()
```

Other alternative to create a SparkSQL table from Dataframe

createOrReplaceTempView - creates a table. If the table exists, then it replaces the existing table without throwing any error.

createTempView - creates a table. If the table already exists, then it errors out, stating the table already exists.

createGlobalTempView - The table view will be visible across other applications as well. If the table already exists, then it errors out, stating the table already exists.

createOrReplaceGlobalTempView - replaces any existing table view with the newly created table.

Creating a Spark Table

If a table is created without selecting the Database in which it has to be created, it will be created under the Default Database.

- To create Database

spark.sql("create database if not exists <Database-name-with-path>")

- To view the database

spark.sql("show databases")

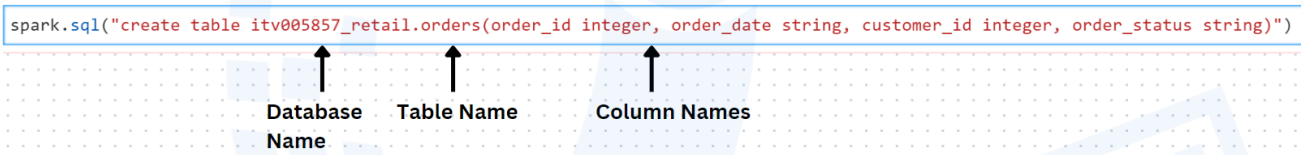
- To view the databases with a certain pattern in their name

spark.sql("show databases").filter("namespace like '<pattern>%'").show()

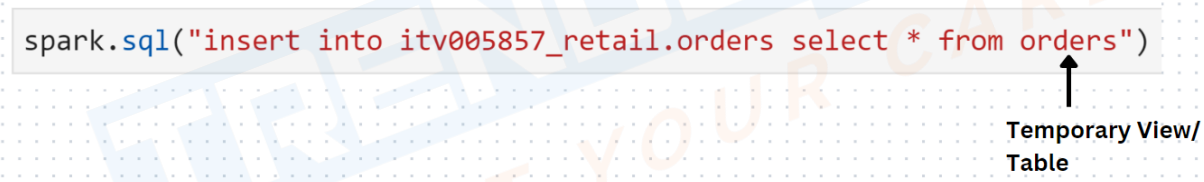
- To view the tables

spark.sql("show tables").show()

To create spark sql table

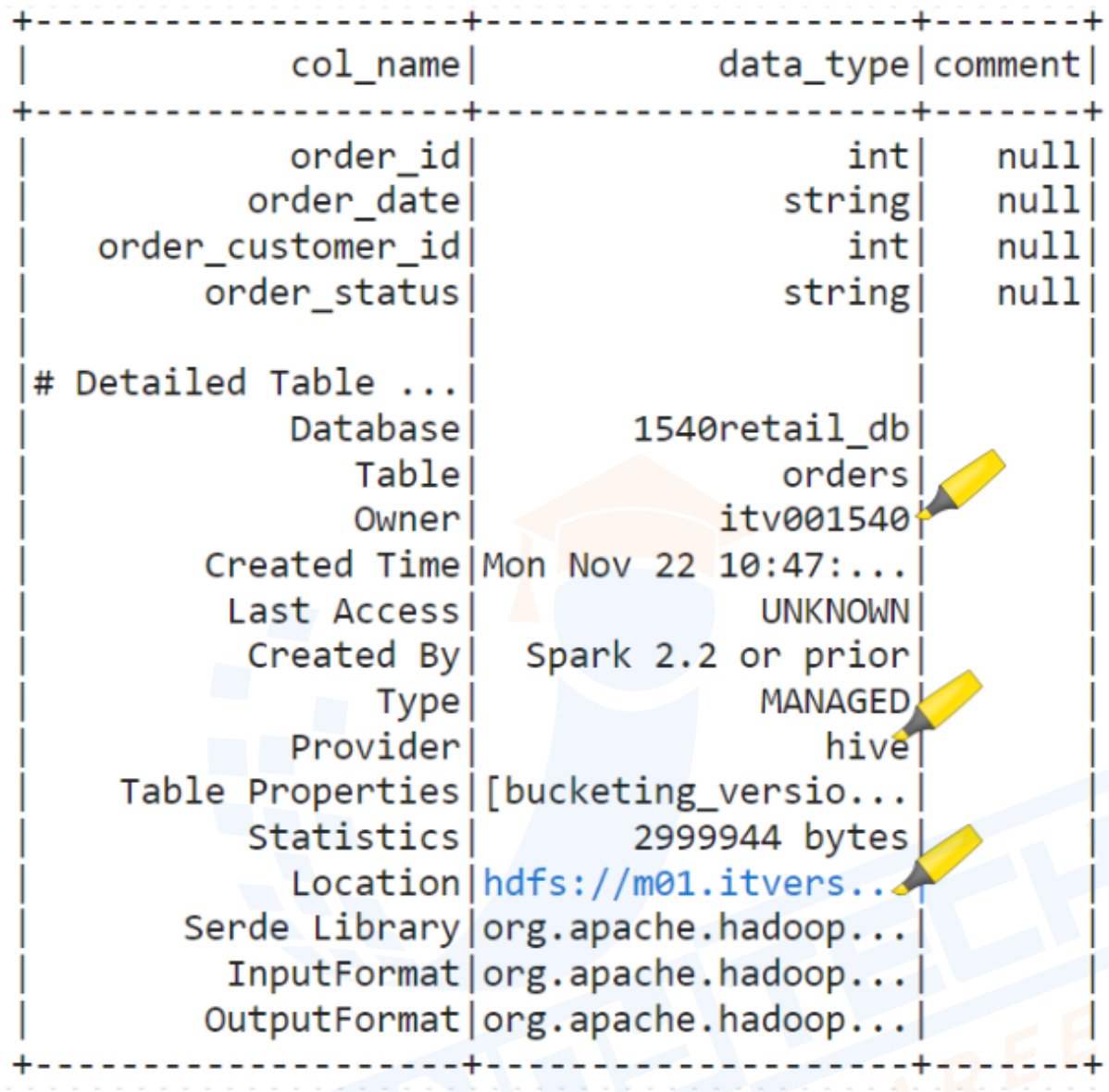


Scenario : We have a temporary table(created using createTempView) with some data and we need to persist this data to a persistent table in the Database



To view the extended description of the table created, use the following command -

```
spark.sql("describe extended itv005857_retail.orders").show()
```



col_name	data_type	comment
order_id	int	null
order_date	string	null
order_customer_id	int	null
order_status	string	null
# Detailed Table ...		
Database	1540retail_db	
Table	orders	
Owner	itv001540	
Created Time	Mon Nov 22 10:47:...	
Last Access	UNKNOWN	
Created By	Spark 2.2 or prior	
Type	MANAGED	
Provider	hive	
Table Properties	[bucketing_versio...	
Statistics	2999944 bytes	
Location	hdfs://m01.itvers...	
Serde Library	org.apache.hadoop...	
InputFormat	org.apache.hadoop...	
OutputFormat	org.apache.hadoop...	

The above table is a Managed Table. We know that the table consists of Data and Metadata

1. **Metadata is in the metastore - we can view this metadata using the describe table command**
2. **Data / actual data is stored in the directory (as highlighted in the above diagram as - Location)**

Note :

When a Managed table is dropped, both the data and metadata are deleted

Types of Tables

1. Managed Table
2. External Table

Managed Table

- Create an empty table
- Load the data from a temporary view / table
- When this table is dropped, both the data and metadata is lost

Ex :

spark.sql("create table itv005857_retail.orders(order_id integer, order_date string, customer_id integer, order_status string) using csv")

External Table

- The data is already present in a specific location and a structure has to be created to get a tabular view
- When this table is dropped, only the metadata is lost.

Ex :

spark.sql("create table itv005857_retail.orders(order_id integer, order_date string, customer_id integer, order_status string) using csv location '/public/trendytech/retail_db/orders' ")

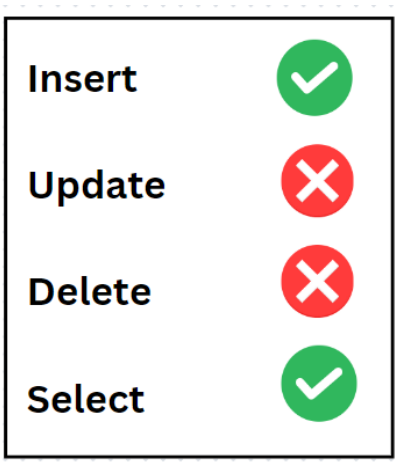
Managed Table Vs External Table

-When the data is owned by a single user then Managed tables could be used.

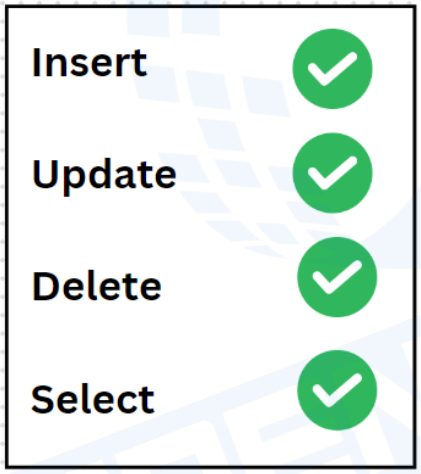
-However, if multiple users are accessing the data kept at a centralized location, then it is best to use external tables. With this, the data remains intact as the users just reuse the data by applying the schema on the existing data but cannot delete the data.

DML Operations

In case of open source **Apache Spark** :



However, In case of **Databricks** :



Working with Spark SQL API Vs Dataframe API

Use cases to understand the working of these higher level APIs

1. **Top 15 customers who placed most number of orders**

   DataFrame way -

   result = ordersdf.groupBy("customer_id").count().sort("count", ascending = false).limit(15)

SparkSQL way -

result = spark.sql("select customer_id, count(order_id) as count from orders group by customer_id order by count desc limit 15")

2. Find the number of orders under each order status

DataFrame way -

result = ordersdf.groupBy("order_status").count()

[Note: count in this case when used with groupBy is a transformation as there is still scope for parallel processing]

SparkSQL way -

result = spark.sql("select order_status, count(order_id) as count from orders group by order_status")

3. Number of active customers (customers who have placed atleast one order)

DataFrame way -

results = ordersdf.select("customer_id").distinct().count()

[Note: count in this case when used directly would be an action and the result is captured in a local variable.]

SparkSQL way -

results = spark.sql("select count(distinct(customer_id)) as active_customers from orders")

4. Customers with most number of orders

DataFrame way -

results = ordersdf.filter("order_status = 'CLOSED'".groupBy("customer_id").count().sort("count", ascending = false)

SparkSQL way -

results = spark.sql("select customer_id, count(order_id) as count from orders where order_status = 'CLOSED' group by customer_id order by count desc")

Examples of Action	Examples of Transformation
count	groupBy.count
show	orderBy
head	filter
tail	distinct
take	join
collect	

Utility Function - Neither a transformation nor an action
printSchema
cache
createOrReplaceTempView

Spark Optimizations / Performance Tuning

Types of Optimizations :

1. **Application Code Level Optimization**

   Ex- Usage of Cache, Using reduceByKey instead of groupByKey

2. **Cluster Level Optimization / Resource level Optimization**

   Containers / Executors

Resource Level Optimization -

In order for a job to run efficiently, the right amount of resources should be allocated.

Resources include -

Memory (RAM)

CPU cores (Compute)

Scenario :

Consider a 10 node cluster (i.e., 10 worker nodes). Each machine has the following resources

16 CPU cores & 64GB RAM

Executor / Container / JVM - Is a container of resources (CPU & RAM). A single node can have more than one executor.

What is the ideal number of executors that a node can have for efficient processing?

Strategies for creating Containers

Thin Executors	Fat Executors
-Idea : More executors created holding minimal resources -total 16 executors -each executor holding 1 CPU core and 4GB RAM -Drawback : 1. No Multithreading 2. Shared variables leads to many copies	-Idea : Give maximum resources to each executor -total 1 executor -each executor holding 16 CPU core and 64GB RAM -Drawback : 1. HDFS throughput suffers 2. Takes a lot of time for garbage collection

Right / Balanced strategy for creating Containers

We have 16 cores and 64GB RAM

1 core goes for background activity
1GB RAM is allocated to the Operating System

Therefore, left with 15 cores and 63GB RAM
Requirement :
1.Multithreading within executor
2.HDFS throughput shouldn't suffer

5 cores and 21GB RAM / executor is considered ideal

Overhead / off heap memory = max(384MB, 7% of executor memory)

~ 1.5 GB (Not part of containers)

Therefore, 21GB - 1.5GB = 19GB

=> Each worker node can have ~ 3 executors

Each executor holds - 5 CPU cores & 19GB RAM