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!

Autoloader

Autoloader is used to handle Streaming Data / workloads (Ex: New files are coming in every 1 minute, which needs a real-time approach to handle such live streaming data)

When we have Spark Structured Streaming, that can gracefully handle streaming data, so what is the need of Autoloader?

The Autoloader offers extra functionalities not available in Spark. It acts as a wrapper leveraging Spark Structured Streaming internally and also provides additional benefits that are given by Spark.

Example Scenario : 'copy into'

Create a delta table named Orders -

```
%sql
create table orders(
order_id int,
order_date string,
customer_id int,
order_status string
) using delta
```

Upload the orders.csv under the data folder in DBFS.

```
%sql
describe detail orders
```

(Shows that it is a delta table. The meta-data is present in user/hive warehouse and contains 0 records)

Listing the files in the DBFS

%fs

ls /FileStore/data (Shows the orders.csv that was previously uploaded)

Copy this data to the newly created delta table - orders

%sql

copy into orders

from (select order_id :: int, order_date, customer_id :: int, order_status from 'dbfs:/FileStore/data/*')

fileformat = csv

format_options('header' = 'true')

To check the data in the table

%sql

select * from orders (displays the delta table data)

- 'copy into' is an idempotent operation, it keeps the track of already loaded files and if the data is already loaded, it won't reload again.
- 'copy into' works well when the schema is static but doesn't support schema evolution as it doesn't capture the changes made to the schema.
    - Scenario 1 : When a new column is added, this modification is not captured
    - Scenario 2 : When the Datatype of the value changes, then the value is replaced with Null, thereby losing the data.
- Since 'copy into' is a retriable idempotent operation, there are chances of the job failing while re-executing in case the table already exists. To avoid this, in place of

  %sql

  create table orders

  USE

  create table if not exists orders

- ‘copy into’ is a best fit when the data to be handled is comparatively less (up-to thousands of files). It cannot handle billions of files.
- ‘copy into’ is well suited for Batch Data, scheduled jobs that run at periodic intervals.

Example Scenario : ‘Autoloader’

Autoloader is best suited to handle Streaming data files present on Cloud. Autoloader is an extension to Spark Structured Streaming.

```
/Declaring Variables
landing_zone = “dbfs:/FileStore/retail_data”
orders_data = landing_zone + “/orders_data”
checkpoint_path = landing_zone + “/orders_checkpoint”

orders_df = spark.readStream \
.format(“cloudFiles”) \
.option(“cloudFiles.format”, “csv”)
.option(“cloudFiles.inferSchema”, “true”)
.option(“cloudFiles.inferColumnTypes”, “true”)
.option(“cloudFiles.schemaLocation”, checkpoint_path)
.load(orders_data)

orders_df.display()
```

Key Points:

- Streaming data arrives very frequently - Ex: every 1 minute and as soon as the data arrives in the landing zone, there should be respective jobs triggered and executed to process the data.

- The landing zone can be any cloud storage like ADLS Gen2 / Amazon S3..

(Actual data present at : 'dbfs:/FileStore/retail_data/orders_data')

- Check point path ('dbfs:/FileStore/retail_data/orders_checkpoint' : stores the progress in the checkpointed location) is important to restart the failed job just from the point where it failed. Say there were 3 files processed and the job failed while processing the 4th file. Then, while restarting the job, it should resume with processing the 4th file and not from the beginning (first file)

- .format("cloudFiles") indicates Autoloader comes into picture, without this option, it is just normal Spark Structured Streaming.

- .option("cloudFiles.inferColumnTypes", "true") To determine the datatypes of Columns.

- .option("cloudFiles.schemaLocation", checkpoint_path) to support schema evolution

- Either you manually define the schema or ensure that there is at-least one data file present so that the schema can be inferred.

- **Scenario 1** : Whenever a file is uploaded by changing its schema (by adding a new column), the job fails (with **UnknownFieldException**) in case of Autoloader. In this scenario, the job needs to be re-run again and the schema gets updated automatically on re-run. Whenever a new column is added to an existing data file / **Schema Evolves**, the job will fail and on re-running, the evolved schema gets reflected.

- **Scenario 2** : **Datatype Mismatch** - The mismatched data will be captured in the **rescued_data** column for future reference.

Writing the above data into a Table :

```
orders_df.writeStream \
.format("delta") \
.option("checkpointLocation", checkpoint_path)
.outputMode("append") \
.toTable("orderdelta")
```

Viewing the data

```
%sql
show tables
```

```
%sql
select * from orderdelta
select count(*) from orderdelta
describe orderdelta
```

NOTE

- 'copy into' is meant for batch data and autoloader is specially designed to handle streaming data that arrives in cloud storages.
- Autoloader incrementally and efficiently processes new data files as they arrive in the cloud storage.
- Advantages :
    - There is no file state management required.
    - Easily scales to millions and billions of files.
    - Autoloader is easy to use.
    - Autoloader can detect schema drifts. Supports schema evolution and inference.
    - In case of Datatype mismatch, the otherwise lost data is maintained in the rescued_data column for future references.

- Checkpoint Optimization : maintains the checkpoint information as key-values pairs in rocks-db. The checkpoint optimization is much more efficient in case of Autoloader because of the information getting stored effectively as key-value pairs in rocks-db.
- Autoloader optimizes file listing :
    - performs Incremental listing rather than complete listing.
    - It used cloud APIs to get a list of files.
    - Also, it uses file notification service to get an instant update on the newly arriving files.
    - It uses fewer APIs (effective and cost efficient)
    - Autoloader can handle the data present in cloud storages like : AWS S3, Azure Datalake Storage Gen2, Blob Storage, Google Cloud Storage, Databricks File System.

- **'copy into' vs Autoloader**

  copy into :

    - Meant to handle batch data
    - Use Case : few thousand files
    - Doesn't support Schema Evolution.
    - On DataType Mismatch the data gets lost

  Autoloader

    - Meant to handle Streaming data present in cloud storage.
    - Use Case : Million / Billions of files
    - Supports Schema Evolution but requires the job to be restarted.
    - On DataType Mismatch, the data is captured under the rescued_data column.

How Schema Inference works

- If you have large volumes of data, the schema is inferred based on the first few initially arrived data.

  Ex : If there are 1000 files or 50 GB of data, then the schema is inferred by just reading the first 10 files or 1GB of Data.

- The properties that are used modify the default values of schema inference are

  cloudFiles.schemaInference. sampleSize.numBytes

  cloudFiles.schemaInference. sampleSize.numFiles

- In case there is no data file present from which the schema can be inferred, then we need to manually define the schema. This is done to ensure that the job doesn't fail in case there is no data file initially to start with.

```
orders_schema = StructType([StructField("order_id", IntegerType()), StructField("order_date", TimestampType(), StructField("customer_id" IntegerType(), StructFile("order_status", StringType())])


orders_df = spark.readStream \
.format("cloudFiles") \
.option("cloudFiles.format", "csv") \
.schema(orders_schema) \
.option("cloudFiles.inferColumnTypes", "true") \
.option("cloudFiles.schemaLocation", checkpoint_path) \
.option("cloudFiles.schemaHints", "order_date String") \
.option("header", True) \
.load(orders_data)


orders_df.display()
```

- With Schema Hints, we can superimpose the desired schema - datatype on the specific column.

  **.option(“cloudFiles.schemaHints”, “order_date String”) **

Additional Information :

- Say you want to capture two additional details of a record :
  1. The file from which the record was loaded.
  2. The time at which the record was loaded(Timestamp)

  Adding a new column using .withColumn()

  **.withColumn(“file_name”, input_file_name()) **

  **.withColumn(“time_of_load”, current_timestamp()) **