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!

Data Processing

Batch processing and Stream processing are the two approaches used in processing data, each suited for different types of data and processing requirements.

Batch Processing -

Involves historical data collected over time and processing this huge volume of data in groups or batches at a scheduled interval.

Stream/ Real-time Processing -

Involves continuous processing of data records as they become available in real-time. The results are to be calculated from the continuously flowing data instantly.

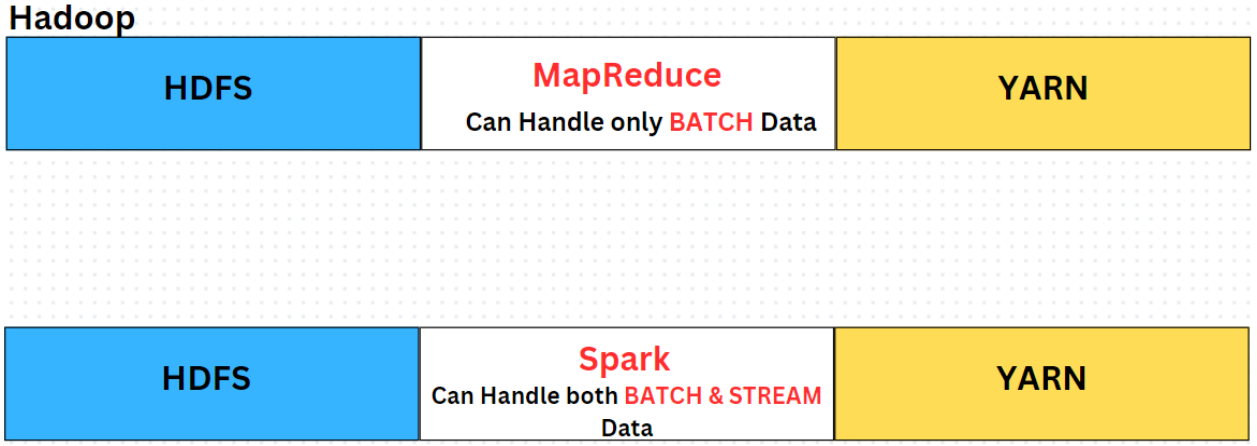
Batch Vs Stream Processing

Batch Processing	Stream Processing
• In Batch Processing the data is collected over time and stored until a predefined batch-size or time-window is reached. Once the batch-size is reached, it is processed as a whole. • Batch processing is meant for large volumes of data that might require minutes, hours or even days to process. • Batch Processing is used in scenarios where the data latency is not critical like - the Reporting / Analytics.	• In Stream Processing, data is processed incrementally as it arrives. • Stream processing is meant for handling continuously flowing data. It is designed to minimize processing delays and provide timely insights. • Stream processing is commonly used in scenarios where immediate data insights or actions are required, such as fraud detection, real-time monitoring

MapReduce Vs Apache Spark

MapReduce cannot handle real-time streaming data. This is where Apache Spark comes into picture that can handle both Batch and streaming data.

The Structured Streaming module present in Apache Spark is used to handle the real-time streaming data.



Two ways of handling Streaming Data in Spark

1. RDD - Lower level constructs. This approach of handling streaming data is also known as Spark Streaming APIs.
2. Dataframe / SparkSQL - Higher level constructs. This approach is widely used to handle the streaming data and is termed as Spark structured streaming.

Example Use-Case for Spark Structured Streaming

Suppose the requirement is to find out the trending Hashtags, then -

Producer - Is an application that generates data continuously. Ex : Twitter

Consumer - Is an application, like the Spark Structured Streaming application that consumes the data for processing.

STEP1 : Creating a dummy Producer that can generate a continuous stream of data mimicking the real-time scenario. The data needs to be generated on a particular socket. (Created in a terminal in External Lab)

Socket = IP + Port (Localhost + 9998)

nc -lk 9998

STEP2 : Start a Consumer (Spark Structured Streaming application) in another terminal

Note:

Whenever some data is entered in the producer terminal, the required business logic is applied on this data on a real-time basis and the results start reflecting in the consumer terminal instantly.

Challenges of Stream Processing

In the case of Batch processing, the data was captured in a file and this file was processed in regular intervals. The intervals could vary from several hours / minutes / seconds.

As the frequency of the batch data processing increases, i.e., the time reduces to minutes/ seconds, then it would be equivalent to Stream data Processing and this could get challenging due to following reasons -

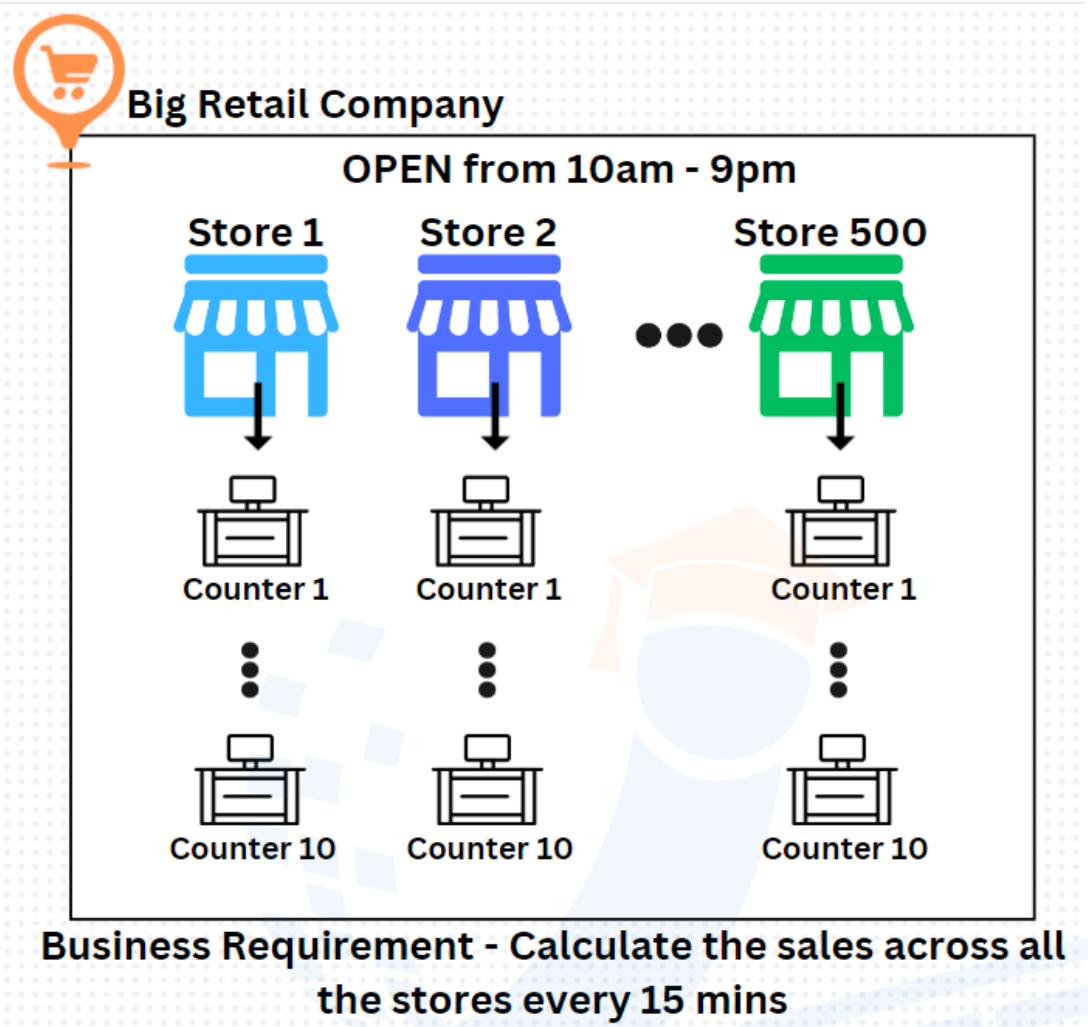
When a job is scheduled, there will be certain activities performed like the job startup tasks, job end tasks, job cleanup tasks, job processing. All of these activities will require time and may reduce the performance as every small job would require more time for other activities like startup, cleanup, etc when compared to the actual processing.

In the case of Streaming, the frequency of processing is increased, which implies more jobs are processed at small time intervals.

Example Use-Case :

Consider a **Big Retail Chain Company** consisting of 500 stores all across. Every store has 10 billing counters. So, there are 5000 billing counters open from 10am - 9pm. The business requirement is to calculate the total sales across all the stores for every 15 mins.

The data is landing as new files in a folder within the Datalake (Like : HDFS / ADLS GEN2 / AWS S3)



Consider that the following amount of sales have been achieved

10:00am - 10:15am -> 2 Million (Batch 1)

10:15am - 10:30am -> 5 Million (Batch 2)

10:30am - 10:45am -> 9 Million (Batch 3)

We can schedule batch jobs for every 15 mins to calculate the sales. However, the challenges faced in this approach are -

1. **Back Pressure :** Due to several reasons, the transactions that occurred between 10:00am to 10:15am might not have been processed even by 10:30am due to several reasons. That is, when Batch 2 was supposed to be processed, Batch 1 is still running leading to back pressure.
2. **Late Arriving Records :** Let's suppose a record that was generated at an event time of 10:08am arrives late at 10:20am. This could be

because of several reasons like Network Issues, etc. It took more than the expected time to arrive at the Datalake.

Additional challenges that needs to be handled when not using a Framework -

- In the case of streaming jobs, time is a very critical factor as the results have to be calculated instantly. In such scenarios, Instead of recalculating the previous results each time, an **Incremental Approach** would be more beneficial.
- What if one of the batch fails due to some reasons, how to handle a batch job failure.
- Fault Tolerance
- How to manage the intermediate results/state, in the case of aggregations.
- How to process the data as and when it arrives. I.e., how to schedule the jobs for continuous listening.

Spark Structured Streaming handles all of the above challenges allowing you to focus only on the important Business Logic -

- It performs automatic looping between the batches.
- An approach of storing intermediate results.
- Automatically combines the current batch results to the previous batch results to provide aggregate output.
- Provides fault tolerance by restarting from the same point in case of failure.

Word Count Program Code

Can be practised on several environments

1. Local (using an IDE like Visual Studio Code)
2. External Lab
3. Databricks Community Edition

3 Major Steps in a Spark Streaming Application

Initially, a Spark Session needs to be created.

1. **READ the Data**
2. **Processing Logic**
3. **WRITE the Data to the Sink**

Note : The Processing Logic is very similar to the Batch Processing. However, the Read and Write is different as we are handling dynamic data in case of Stream Processing.

#1 Read the Data

```
lines = spark \
.readStream \
.format("socket") \
.option("host", "localhost") \
.option("port", 9999) \
.load()
```

(In real-time projects, reading the data through socket is not the best approach. However, for demonstration purposes, we are using sockets.)

#2 Processing Logic

```
words = lines.select(explode(split(lines.value, " ")).alias("word"))
wordsCounts = words.groupBy("word").count()
```

(The above logic performs the following

1. Splits the Lines based on spaces and creates an array of words.
2. Every word is present in a separate new line with the use of explode function that returns a new row for each element present in the array
3. Renaming the default column name to word using alias.)

#3 Write to the Sink

```
lines \
.writeStream \
.outputMode("append") \
.format("console") \
.option("checkpointLocation", "checkpointdir1") \
.start()

query.awaitTermination()
```

Steps to execute:

- Start a Netcat utility to render the data to a particular socket.

  nc -lk 9990

- Run the above Spark Application.
- Processing happens in micro-batches and the data is captured in a column called **value**. (A structure is associated as it is Structured Streaming. A dataframe consisting of a value column that captures the data.)
- We can monitor the spark jobs and check the performance of the spark job to check for any further scope of optimizations.
- Initially we can see that there are 2 stages created and due to which 200 shuffle partitions get created. These 200 partitions would consume a lot of time for coordination among the tasks.
- We can reduce the shuffle partitions in the main file - "application_main.py" during the creation of spark session by adding the following :

  .config("spark.sql.shuffle.partitions", 3)

  This will make the application to deliver the results at a much faster rate.

- Since we see that 200 partitions is too high for the data that is being processed, we can reduce the number of partitions to improve the performance of the spark application.

Executing Word Count Program in the External Lab

1. **Using Notebook**

   - Create a new notebook and select the PySpark 3 in the kernel options.

   - Create a Spark Session and start executing the code.

   Note: However, there is no terminal available for the Notebook. Therefore, while using Socket and Console for reading and writing the data, we cannot choose Notebook. For such scenarios, we need to go with the terminal approach

2. **Using Terminal**

   - Create 2 terminals

   Terminal 1 : Pyspark terminal

   Execute the following on the terminal

```
pyspark3
#Import the required functions
from pyspark.sql.functions import *
#Read Data
lines = spark \
.readStream \
.format("socket") \
.option("host", "localhost") \
.option("port", 9999) \
.load()
```

```
#Processing Logic
words = lines.select(explode(split(lines.value, “ ”)).alias(“word”))
wordsCounts = words.groupBy(“word”).count()
#Write Data
lines \
.writeStream \
.outputMode(“append”) \
.format(“console”) \
.option(“checkpointLocation”, “checkpointdir1”) \
.start()
```

- When the lines are entered in Terminal 2, the Processing logic in terminal1 will be executed and the results will be visible.

Terminal 2 : Normal Terminal

```
nc -lk 9999
```

- We can type in the lines in this terminal

Data Stream Reader is used to create Spark Streaming Dataframe.

Input Sources

Some of the built-in sources.

1. **File Source** - This is one of the very commonly used input sources. Reads files written in a directory as a stream of data. Files will be processed in the order of file modification time. Usually by default, the files that have arrived first will be processed first. If Latest First is set, then the order of processing will be reversed, the file that has arrived Latest will be processed first.

   Spark streaming application will be monitoring the specified directory location. If any new file arrives, it will instantly trigger the job.

   Supported File Formats - Text, CSV, JSON, ORC, PARQUET

2. **KAFKA Source -** Reads the data from Kafka Topic.
3. **Socket Source (for Testing) -** The listening server socket is on the server itself therefore used majorly for testing as it does not provide end-to-end fault tolerance guarantees.

   Checkpoint Directory

   - As soon as the streaming query execution starts, a Checkpoint location is created. It is used for the following purpose -

   a. Fault tolerance - If a job fails after a certain micro-batch, the state/logs of the job are maintained in the checkpoint directory in order to resume the job from the same point where it failed, avoiding re-iteration from the beginning.

   b.

   - The Streaming query will keep monitoring the checkpoint directory location for new data.
   - Once it finds new Data / Files, a new micro-batch will be triggered.
   - Before the micro-batch starts, the streaming query will update the checkpoint that the new file is going to be processed.
   - Once the micro-batch is processed and complete, the checkpoint will be updated
   - Streaming query keeps on monitoring the checkpoint directory for new files and the above steps repeat.
   - These logs will be helpful for fault tolerance.

Reading the data from a JSON File Source

```
lines = spark \
.readStream \
.format("json") \
.schema(orders_schema) \
.option("path", " data/inputdir") \
.load()
#Processing Logic
orders_df = createOrReplaceTempView("orders")
completed_orders =spark.sql("select * from orders where order_status = "COMPLETE")
#Writing to the Sink
query = completed_orders \
.writeStream \
.format("csv") \
.outputMode("append") \
.option("path", "data/outputdir") \
.option("checkpointLocation", "checkpointdir1") \
.start()

query.awaitTermination()
```

Note:

- When the file is moved to the data/inputdir, the job gets triggered and the data will be processed instantly. A JSON source file when moved to the data/inputdir, it will be processed and converted to CSV form and will be written to the data/outputdir
- **Output Modes** : Complete, Update, Append

- **Complete Mode** : All the changes made due to the addition of a new file along the previous results will be reflected in the output. **The entire updated result table will be written to the external storage.**
- **Update Mode** : UPSERT(Update+Insert) Only the updates and inserts made as per the new file added will reflect in the output. **Only the rows that are updated or newly inserted will be written to the external storage.**
- **Append Mode** : Append mode doesn't work on Aggregation operations. **Only the new records added will be shown.**

State Store :

State store is a key-value store which provides both the read and write operations.

In the case of Structured Streaming we use State Store to handle the Stateful operations across the batches.

Spark Streaming maintains the state across the batches in-memory State Store to eventually calculate the total aggregated results of all the batches. The same is persisted in the checkpoint directory to avoid any loss of data.

(**Stateful transformation** is where the state needs to be saved unlike Stateless Transformation. Ex: groupBy, in this case, the data or intermediate results needs to be saved across multiple micro batches in-order to get the final aggregated data.)

Note:

- From Spark 3.1 onwards, it is possible to read the data from and write the data to the tables.
- DataStreamReader.table() is used to read tables as streaming dataframes.
- DataStreamWriter.toTable() is used to write streaming dataframes as tables.

Generating Input data (JSON)

1. Create Spark Session
2. Define the Schema and Create Dataframes for Orders, Order_item and Customers datasets are present in the external lab
3. Join the above three dataframes.
4. Save the results of the join operation in a Dataframe - joined_df.
5. Perform groupBy on joined_df by selecting the required columns.
6. Since we need a list, use collect_list function to get the list of order items.
7. collect_list is an aggregation just like min,max,etc.
8. We need a list of struct-type, therefore using struct function -

   agg(collect_list(struct("order_item_id",.....

   (gives a list or array of Structs)

9. Save the above Dataframe created in JSON format

```
result_df \
.repartition(1) \
.write \
.format("json") \
.mode("overwrite") \
.option("path", "/user/…/data_json_orders") \
.save()
```

Batch Processing Use Case

Reading the file from a storage (like HDFS / ADLS Gen2, etc) and processing it to get the desired results (Ex : Number of orders placed by a particular customer, Total Products Purchased by a particular customer in an order, etc) in a tabular view using Batch Processing.

1. Create Spark Session
2. Define the Schema of the input data - orders_schema (use array<struct<...>> to define the Line_items

3. Read the Data

```
orders_df = spark.read \
.format("json") \
.schema("orders_schema") \
.option("path", "/user/...") \
.load()
```

4. We need to work on Spark SQL, therefore create a table on top of the input data.

```
orders_df.createOrReplaceTempView("orders")
spark.sql("select * from orders").show()
```

5. Since there are multiple Line items, in order to get these line items to different rows, we need to use the Explode Function to flatten the line items.

```
exploded_orders = spark.sql("""select order_id, customer_id, city, state,
pincode, explode(line_items) lines from orders""")
exploded_orders.createOrReplaceTempView("exploded_orders")
spark.sql("select * from exploded_orders").show()
flattened_orders = spark.sql("""select order_id, customer_id, city, state,
pincode, lines.order_item_id as item_id, lines.order_item_product_id as
product_id, lines.order_item_quantity as quantity,
lines.order_item_product_price as price, lines.order_item_subtotal as
subtotal from exploded_orders""")
flattened_orders.createOrReplaceTempView("orders_flattened")
aggregated_orders = spark.sql("""select customer_id,
count(distinct(order_id) as orders_placed, count(item_id) as
products_purchased, sum(subtotal) as amount_spent from
orders_flattened group by customer_id""")
aggregated_orders.createOrReplaceTempView("orders_aggregated")
spark.sql("select * from orders_aggregated).show()
```

6. Write the results to a particular path in a specific format.

```
aggregated_orders.repartition(1) \
.write \
.format("csv") \
.mode("overwrite") \
.option("header", True) \
.option("path", "/user/…/result") \
.save()
```

Streaming Use Case - Complete Mode

Above Batch processing code converted to Streaming code. The demonstration will be in Databricks as we can use Delta Format to illustrate the Complete mode. (Complete mode is not supported in Parquet / CSV formats)

1. Start a Databricks cluster with minimal configurations (You can use the Community edition of Databricks)
2. Create a New Notebook and execute the spark command to check if the spark session is active.
3. Define the Schema of the input data - orders_schema (use array<struct<...>> to define the Line_items
4. Read the Data from the specified DBFS path

```
orders_df = spark \
.readStream \
.format("json") \
.schema("orders_schema") \
.option("path", "/user/…") \
.load()
```

5. We need to work on Spark SQL, therefore create a table on top of the input data.

orders_df.createOrReplaceTempView(“orders”)

spark.sql(“select * from orders”).show()

6. Since there are multiple Line items, in order to get these line items to different rows, we need to use the Explode Function to flatten the line items.

   exploded_orders = spark.sql(“““select order_id, customer_id, city, state, pincode, explode(line_items) lines from orders”””)

   exploded_orders.createOrReplaceTempView(“exploded_orders”)

   spark.sql(“select * from exploded_orders”).show()

   flattened_orders = spark.sql(“““select order_id, customer_id, city, state, pincode, lines.order_item_id as item_id, lines.order_item_product_id as product_id, lines.order_item_quantity as quantity, lines.order_item_product_price as price, lines.order_item_subtotal as subtotal from exploded_orders”””)

   flattened_orders.createOrReplaceTempView(“orders_flattened”)

   aggregated_orders = spark.sql(“““select customer_id, approx_count_distinct(order_id) as orders_placed, count(item_id) as products_purchased, sum(subtotal) as amount_spent from orders_flattened group by customer_id”””)

   aggregated_orders.createOrReplaceTempView(“orders_aggregated”)

   spark.sql(“select * from orders_aggregated).show()

7. Write the results to a particular path in a specific format.

   StreamingQuery = aggregated_orders \

   .writeStream \

   .format(“delta”) \

   .outputMode(“complete”) \

   .option(“header”, True) \

   .option(“checkpointLocation”, “checkpointdir101”) \

   .toTable(“orders_results101”) Is an action

8. Add the file to the specified DBFS path. If the file is not present in the input path, then the execution of the query will be waiting for the data to be added to the location to proceed with the execution.

   spark.sql("select * from orders_results101).show()

Update Mode

Delta format doesn't support toTable with Update Mode.

Logic of Update mode - If an existing record is updated the changes will be made to that particular record. If the record to be updated doesn't exist, then a new entry is added for this record. It is an upsert operation (update/Insert). Microbatch is implemented in the case of update mode.

There are 2 tables coming into picture in this case :

1. **Target table is the final result table (t)**
2. **Source table is the micro batch table (s)**

Logic :

if t.customer_id == s.customer_id

When it is a match, Update all the columns

else

If it is not matched, then insert a new record

Writing the results to the specified location

```
def myFunction(orders_result, batch_id)

    orders_result.createOrReplaceTempView("orders_result")

    merge_statement = """merge into orders_final_result t using
orders_result s

on t.customer_id == s.customer_id

when matched then
```

```
update set t.products_purchased = s.products_purchased,
t.orders_placed = s.order_placed, t.amount_spent = s.amount_spent

when not matched then

insert *

"""

orders_result._jdf.sparkSession().sql(merge_statement)


StreamingQuery = aggregated_orders \

.writeStream \

.format("delta") \

.outputMode("complete") \

.option("header", True) \

.option("checkpointLocation", "checkpointdir101") \

.foreachBatch(myFunction) \

.start()
```

orders_final_result has to be created before starting the execution

```
spark.sql("create table orders_final_result (customer_id long,
orders_placed long, products_purchased long, amount_spent float)")
```

To view the final results

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