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.

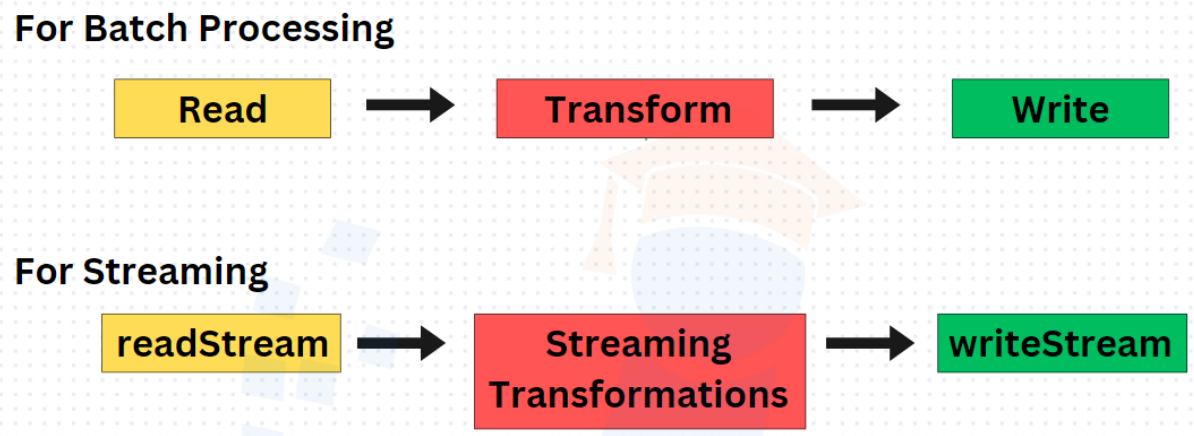
- If you've obtained these slides for free on a website that is not the course's website, please reach out to legal@trendytech.in . Thanks!

- All the Best and Happy Learning!

Different Output Modes

1. Append
2. Update
3. Complete

3 Important Processes :



Note : Spark internally is a microbatch engine. It doesn't offer true streaming but rather works on small batches to give the same effect as streaming.

Categories of Streaming Transformations

Stateless	Stateful
• Previous state is not maintained. • Example - Transformations like - Select, Filter, etc doesn't require the previous state to be maintained. • No Intermediate results are captured.	• Previous state needs to be maintained as aggregations and grouping are performed. • Example - Transformations like - groupBy, join, windowing functions. • Intermediate results are captured in the State Store (memory of the executor). • Drawbacks - Additional overhead of maintaining the state in memory. Leads to out of memory error / performance issue if the state store grows exponentially. • State store holds the aggregated results of previous micro batches in the executor memory.

Amazon Use-case :

Requirement - Get the order details (customer_id, orders_placed, products_purchased, amount_spent) of all the customers of Amazon from the time they have registered with Amazon. (say there are 300+ million active customers)

- In this case, the State Store has to maintain 300+ million records to provide all the order details.
- When the time is **unbounded** (i.e., right from the time the customers registered) then it would be a heavy operation. Since the data will be more in this scenario, state-store cannot be used and would require a custom solution.
- It would be easier to handle if **time-bound** (i.e., for an interval - weekly, monthly, quarterly), in which case it would be feasible to maintain the data in the state store.

Example Scenario : Consider the Orders and Customers datasets.

Create a new notebook in the databricks community edition account and attach a cluster to execute the code.

Two approaches :

1. **Without custom function for aggregations (uses state store)**

A state store will be created at the moment an aggregation is performed.

2. **With custom function for aggregations (doesn't use state store)**

Note:

- In the first scenario, since the aggregation logic is present outside the custom function, a state-store gets created and it can be visualized in the 'Spark UI' under the 'Structured Streaming' tab.
- In the second scenario, since the aggregation logic is added as part of the custom function and there are no aggregations performed before the writeStream, a state-store is not created and this can be checked in the 'Spark UI' under the 'Structured Streaming' tab.

Triggers :

Spark is internally a microbatch engine. At every interval, as and when the data arrives, a small microbatch gets created and this gives an illusion of a real-time streaming. However, internally it is getting processed in the form of micro-batches.

To answer the following questions : What is the Size of the Micro-batch? When is the Micro-batch triggered?

It is required to have an understanding of different types of Triggers.

Types of Triggers -

1. **Unspecified (Default)**

   In this case, once the first micro-batch (say there are 2 files - File1 & File2 in the first micro-batch) processing is complete, the second micro-batch will be triggered provided there is some data that needs to be processed. The subsequent micro-batch gets triggered only when there are some files that need to be processed.

   Second micro-batch gets triggered when the File3 arrives.

   Suppose File4 has arrived and File3 is still in process, then the third micro-batch will be triggered soon after the 2nd micro-batch processing is complete.

2. **Fixed Interval**

   Each micro-batch will start after a certain specified interval, provided there is some data that needs to be processed.

   Syntax :

   .trigger(processingTime = “<time-interval-secs>”)

   Example

   For instance, Let's consider that the fixed interval is set to 5mins.

   Scenario1 - If the 1st micro-batch completes its processing in 2 mins, it will wait for 3 more minutes to meet the fixed interval time set. Soon after mins, if a new file has arrived that needs to be processed, then the 2nd micro-batch gets triggered. If there are no files to be processed, then the 2nd micro-batch doesn’t get triggered.

Scenario2 - If the 1st micro-batch completes its processing in 8mins which has exceeded the specified fixed time interval set, then it will immediately trigger the second micro-batch provided there is some data to process. It won't wait until 10 mins to trigger the next micro-batch as the current processing has exceeded the time limit.

Note: Triggers are applicable for writeStream.

3. **Available Now**

   In this approach, the trigger automatically stops soon after the processing of the micro-batch.

   Syntax :

   .trigger(availableNow = True)

   Example - Say a microbatch has 3 files File1, File2 and File3. Then, the Available Now Trigger will process all the files associated with the micro-batch and then it will stop.

Note :

- It seems that Available now is similar to Fixed Interval. However, in a fixed interval trigger, the resources are held up even when it is idle. In Available Now, resources will be released soon after the processing is complete.
- Available Now, seems similar to batch processing as there is a need to schedule the jobs for the next processing. However, Available Now, automatically handles incremental processing which is not the case with Batch processing.
- How to trigger a new micro-batch is an important strategy and it depends on the business requirement.

Fault Tolerance and Exactly Once Guarantee :

A streaming application is meant to run continuously. However, due to certain scenarios like :

- When Exceptions occur
- When Maintenance activities needs to be executed

In these cases, the streaming jobs might stop.

Fault tolerance means that our application should be able to restart gracefully even after failures due to the above mentioned reasons.

To be able to restart gracefully from failures, it requires that **Exactly One Semantics is Maintained.**

To maintain exactly one semantics, the following needs to be performed :

- No input records should be missed.
- No duplicate output records should be created.

Spark Structured Streaming provides Fault tolerance :

By maintaining the state of the micro-batch in the check-point location.

Checkpoint location helps in achieving fault tolerance. It mainly consists of the following informations :

1. **Read positions**, providing details of the files that are processed.
2. **State Information**

Spark Structured Streaming provides ample support for exactly one semantics through the following:

1. Restart the application with the same check-point location.

Say the application failed while processing the 3rd micro-batch due to some exception. The commit logs would consist of the information of the previous 2 commits which would be helpful in restarting the application from the same point where the exception occurred.

2. Use a Replayable Source.

Let's assume that the first 2 micro-batches have processed successfully and the 3rd microbatch consists of 100 records and failed due to an exception after processing 30 records. These 30 records which were processed completely should be available when restarting the application from the point of failure. This can be achieved by using Replayable Sources.

Note :

- **Socket source** is not a Replayable Source and we cannot get the old data back again after the exception occurred and the data was lost.
- **Kafka and File Sources** are Replayable sources.

3. Use Deterministic Computation.

A Deterministic computation is one which yields the same results at any point in time. Therefore, to achieve exactly one semantics, it is required to use deterministic computation.

Ex : Sqrt(4) gives the same result whenever it is executed.

However, dollar_to_INR(10) is a non-deterministic function as it would vary as per the current exchange-rates.

4. Use an Idempotent Sink.

Consider the same scenario where the 3rd micro-batch with 100 records fails after processing 30 records.

When the 3rd micro-batch is re-started, these 30 records are re-processed. It implies that we have 2 outputs for these 30 records. However, the previous output needs to be discarded or overwritten by the newly processed 30 records. This is known as an idempotent sink and is important to achieve exactly one semantics.

Types of Aggregations :

1. **Continuous Aggregations (Un-bounded)**
   - Continuous aggregations are not time bound. There is no time-range / window specified.
   - Example : A Retail Store's Customer Reward Points (don't expire) data maintenance. This data needs to be maintained right from the beginning i.e., from the time the customer account was created.
   - The history data keeps on growing in the state-store over time and the state-store cannot be cleaned up in such cases.
   - Therefore, it would be better to implement a custom solution wherein the state-store is not created.

2. **Time Bound Aggregations (Windowing Aggregations)**
   - Window aggregations are Time bound and the time interval can be specified in this case.

- Example : A Retail Store's Customer Reward Points (Ex: expires in one month) data maintenance. This data needs to be maintained for the specified time interval.
- The State-store can be cleaned up after the time-interval / window has expired.
- There are 2 types of Window Aggregations :

1. **Tumbling Window**

   - A series of fixed size, non-overlapping time intervals.

   - Windows do not overlap, they are contiguous and consecutive.

   - Tumbling windows are useful when you want to perform aggregations over distinct, non-overlapping periods, like counting events per minute.

   - Example :

   10 - 10:15

   10:15 - 10:30

   10:30 - 10:45 and so on… With an interval of 15 mins

2. **Sliding Window**

   - A series of overlapping time intervals of fixed size, with a specified slide interval.

   - All windows have the same fixed length.

   - Windows can overlap, depending on the slide interval.

   - Sliding windows are useful when you want to perform continuous aggregations with overlapping periods, like calculating a moving average.

   - Example :

   10 - 10:15

   10:05 - 10:20

   10:10 - 10:25

Aggregation windows are based on event time and not on trigger time.

Event Time - Is the time when the event occurs (Like - record is created).

Trigger Time - Is the time when the event reaches the Spark framework for processing.

(There would be a considerable difference between Event time and the Trigger time in some cases because of Network delays and these records are termed as late arriving records)

Business Requirement : To find the amount of Sales happening every 15 mins.

Solution :

- In this scenario we will be using a Tumbling Window (Fixed size window of 15 mins)
- Spark Code to get the required results
    1. Create Spark Session
    2. Schema for orders table.
    3. Use Socket as Source and Console as output for reading and writing the data (For Demo purpose only, not for real-time production environment)
    4. Delete any previously created checkpoint directories.
    5. Use Netcat utility for providing the input.
    6. Use JSON utility to parse the String input provided into JSON.
    7. Use spark readStream to read the data.
    8. Perform the required aggregations

       Example :

       ```
       window_agg_df = refined_orders_df \
       .groupBy(window(col("order_date"), "15 minutes")) \
       .agg(sum("amount").alias("total_invoice"))
       ```

       (here we are using a window interval of 15 mins to perform the aggregation - sum(amount) to get the total sales)

    9. While writing the results to the output dataframe using wrtiteStream, we cannot use **Append mode** as we are performing aggregations and aggregations require the previous data as well. For aggregations, **Update mode** needs to be used.

Note :

- The records arriving within the specified window are aggregated together. Example - all the records that are arriving between 11:00 to 11:15, are associated to one window and will be aggregated in the same window.

- This approach is beneficial to handle late arriving records. The windows data is maintained in the State-store right from the beginning and the state-store is not cleaned at regular intervals. This can lead to out-of-memory issues as the state-store grows over time and cannot be accommodated in the executor memory.

Watermark Feature in Streaming

Watermark concept is used to handle late arriving records and also accommodate regular State-store cleaning.

- An expiry is set to the records. Spark can ignore the records that are exceeding the expiry duration. This time limit on the record's arrival is termed as **Watermark Boundary**.
- How to set the right watermark duration. Based on the business accuracy expectation requirement, the watermark can be set to accommodate the maximum data that falls within the business accuracy requirement.

  Example : Say that a particular business requires 99.9% accuracy and 99.9% of the records are arriving within 30 mins as per the previous data analysis, then the watermark can be set to 30 mins. The remaining 0.1% records that are arriving late can be ignored.

- Syntax to define Watermark

  While creating a dataframe, the watermark option can be included.

  .withWatermark("<column_name>", "<duration>")

  Example : Let's consider the previous example of calculating total sales.

```
window_agg_df = redefined_orders_df \
.withWatermark("order_date", "30 minutes") \
.groupBy(window(col("order_date"), "15 minutes")) \
.agg(sum("amount").alias("total_invoice")
```

- Watermark is important to clean the state-store regularly to avoid out-of-memory issues.
- Events occurring within the watermark boundary will be updated.
- Events occurring outside of the watermark boundary may or may not be updated.
- In some scenarios, even if the watermark boundary has exceeded i.e., the record arrives late beyond the watermark boundary, it will still be updated in the respective window provided the window still exists and is not deleted during the regular state-store cleanup.

The output modes that can be used along with Watermark

1. Complete mode - It states that the records are maintained right from the beginning. With this feature of complete mode, it will not allow the state-store clean activity. Therefore, Complete mode is not the right choice to be used with Watermark.
2. Append mode - updates are not possible in append mode. Ideally, this mode doesn't support aggregations. However, if the watermark boundary is set to a certain limit and there are no changes made after this point, then the Append mode allows windowing aggregations.

   The window information will be displayed only when the particular widow expires. The limitation is that we need to wait for the watermark boundary duration to view the window information.

3. Update mode - is mostly used along with the watermark feature

Sliding Window

A series of overlapping time intervals of fixed size, with a specified slide interval.

Ex : 10 - 10:15

10:05 - 10:20

10:10 - 10:25

We need to specify the slide interval in the window option while creating the dataframe :

```
window_agg_df = refined_orders_df \
.groupBy(window(col("order_date"), "15 minutes","5 minutes")) \
.agg(sum("amount").alias("total_invoice"))
```

Streaming Joins

Structured streaming supports 2 kinds of joins

1. **Streaming dataframe to Static dataframe**
2. **Streaming dataframe to another Streaming dataframe**

Streaming dataframe to Static dataframe

- Mainly used to perform **stream enrichments** where we perform quick lookups to add further enhancements (Ex: add more columns) to the data.
- Example : Banking Domain

  ATM transaction : Whenever a card is swiped, a transaction will be generated.

  Say the Transactions data **(Streaming data)** has some limited data but you would like to enhance this table by adding more relevant data associated with that particular transaction (like : member_id / country) which is present in another table called Member_table **(Static Data)**. We need to join a Streaming Data with the Static data to achieve the desired requirement.

  Syntax for joining the Streaming dataframe with Static dataframe

  ```
  join_expr = streaming_df.<column_name> == static_df.<column_name>
  join_type = "<join-type>"
  enriched_dataframe = streaming_df.join(static_df, join_expr, join_type).drop(static_df["join_column"])
  ```

Left Outer Join

All the matching records + All the non-matching records from the left table padded with nulls on the right table columns.

Scenario 1:

- **On the Left is a Static Dataframe**
- **On the Right is a Stream Data**

Will the Left Outer join be achievable in the above scenario?

It is not possible because, say we have a record in the left table but there is no matching stream record on the Right Stream Data at the moment. However, the matching stream data might arrive in the future. This is unpredictable, therefore left outer join is not possible in the above scenario.

Scenario 2:

- **On the Left is a Stream Data**
- **On the Right is a Static Dataframe**

Will the Left Outer join be achievable in the above scenario?

In this scenario, it is definitely possible to implement the left outer join as it doesn't lead to any unpredictable situations. We are sure that the records in the left stream data are either present in the right static table or not.

Note :

- **For the Left Outer Join to work :**

  Left side should have Stream data and the Right side should have a Static Table.

- **For the Right Outer Join to work :**

  Right side should have Stream data and the Left side should have a Static Table.