Lecture Notes

Real-Time Data Processing Using Spark Structured Streaming

Introduction to Spark Streaming

What is Streaming?

Before you get started with Spark Streaming itself, you need to first understand the meaning of the term Data Streams.

A stream of data is nothing but a continuous inflow of data, for example, a live video or Amazon's log data. There is no discrete start or end to a data stream as it keeps flowing forever. Since this stream is continuous, it is also usually high-volume data, which means the size of the incoming data would be large. Data streams are usually processed in near real-time.

Streams are used for the following important reasons:

1. They help capture real-time reactions, which can be used for various analytical purposes.
2. Since data is processed in near-real-time, they can be used for fraud detection or for initiating prompt responses as per the needs of the situation.
3. They can be used for scaling up or scaling down hardware as per incoming load.

Streaming data architecture is a framework that is built for ingesting and processing streams of data. It comprises multiple components, such as:

- Stream Consumer,
- Data Persistence,
- Processing/Transformations and
- Analytics/BI.

Differences Between Streaming and Micro-Batching

In batch processing, data is divided into different batches based on a time window, which could be hourly, daily, weekly or monthly. Since we can have large time windows, it has the capability of producing high volumes of data. However, this could be a potential cause of substantial latency for the BI layer. It would be a waste of time if we wanted data to be processed instantly. So, what happens if we keep reducing the batch size?

We could set the time window to minutes, or even seconds, which would enable us to achieve near-real-time data availability, which is exactly what Spark Streaming does. It runs on micro-batches, with the trigger set to 0, to ensure data is read continuously. This makes it practically the same as streaming.

What is Spark Streaming?

Spark Streaming is an API of Apache Spark that is built specifically to process real-time streaming data. Spark offers both low-level and high-level APIs. The low-level APIs include RDDs and DStreams (collections on RDDs), while the high-level APIs consist of DataFrames, DataSets and SQL. Using these APIs, programmers can utilise applications such as Streaming, ML and Graphx as per their needs.

Spark Streaming has a declarative API, which means it shields the programmer from the internal complexities of running the job. These are taken care of by Spark and YARN. The user need not worry about job allocation or resource usage.

The Spark Structured Streaming API is similar to the Spark batch processing model, and, in addition, it also offers the DataFrames and SQL APIs. Various Spark optimisations are picked up automatically, and it also offers the Exactly Once guarantee, which means the same batch of data will not be processed more than once; it would be discarded. It also processes late-arriving data more effectively than the DStream API and, hence, it is preferred.

Structured Streaming - Basic

What is Structured Streaming?

Structured Streaming is a scalable and fault-tolerant stream-processing engine, which is built on the Spark SQL engine. This means you can express your streaming computation in the same way as you would express a batch computation on static data. Since Structured Streaming is built over the Spark SQL engine, it offers several advantages.

Below are the reasons we use Structured Streaming:

- It is a high-level API, which means it has more features for our accessibility and usage.
- It offers ease of development.
- It is compatible with other Spark APIs.
- Spark optimisations are built-in.

The key fundamentals of Structured Streaming are as follows:

- Spark principles stay in place
    - **Lazy evaluation -** Lazy evaluation in Spark means execution will not start until an action is triggered.
    - **Transformations -** Functions need to be applied to input data in accordance with the requirements to get the desired outputs.

    - **Actions -** The output would be presented only when some action is performed.
- Inputs
    - **Streaming systems -** Kafka, Flume
    - **File systems -** S3
    - Sockets
- Outputs
    - Databases
    - Input systems

The general code flow of a Spark Streaming application is as follows:

- Create a SparkSession
    - This is the entry point for a structured streaming job
- Read from source
    - Socket, Kafka, File, etc.
    - Read a stream
    - Return a DataFrame
- Perform transformations
    - Create one or more DataFrames
- **Start -** Action
- AwaitTermination
    - Wait for the stream to finish

Trigger and Output Modes

The trigger settings of a streaming query define the timing of streaming data processing, specifically, whether the query will be executed as a micro-batch query with a fixed batch interval or as a continuous processing query. The output modes of a streaming query define what DataFrames are to be written to the output sink.

The various output modes in Spark Structured Streaming include the following:

· **Append -** Only new records are added to the sink.

· **Update -** Only modified records are added to the sink.

· **Complete -** All records are added to the sink.

There are, however, a few restrictions on output modes; for example, there can be no aggregations with update mode or append mode without watermarks. Triggers are a means to decide when new data gets processed in a stream. The default setting would be to do it every time a new micro-batch arrives. You could also do it continuously at each record level or just once for a single micro-batch.

Transformations and Aggregations

There are a few key APIs to work with Spark Structured Streaming. These include the following:

- **PySpark.SQL -** Access the SQL functionalities of Spark
- **Select -** Select columns names
- **SelecExpr -** Any SQL-like statement can be written as a string

Now, let's take a look at transformations:

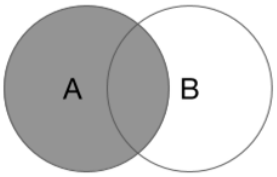
- **Filter/Where -** To filter out certain elements from an RDD that do not meet the criteria defined in the lambda expression
- **As/Aliasing -** To make the output more readable by giving a different name, a.k.a. aliasing
- **GroupBy -** Shuffle and group data appropriately
- **Aggregations -** Avg, Sum, Min, Max

Joins With Streams

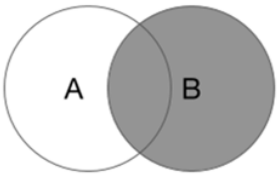
Joins in streams are quite similar to joins in SQL. You only need to assume that each streaming DataFrame is a table. So, stream DataFrames can be joined with other stream or static DataFrames in the same way.

Joins are of the following different types:

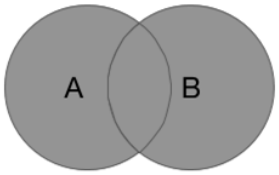
- **Inner Joins -** Will return records that have matching values in both tables
- **Outer Joins**
  - **Left -** Will return all the records from the left DataFrame and matching records from the right DataFrame
  - **Right -** Will return all the records from the right DataFrame and matching records from the left DataFrame
  - **Full -** Will return all the records when there is a match in either the left or the right DataFrame



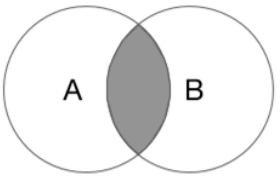
Left outer join



Right outer join



Full outer join



Inner join

There are, however, a few restrictions on outer joins in streams. These include the following:

Stream - Stream outer joins can be performed only using watermarks.

Stream - A static right outer or full outer join is not permitted.

Static - A stream left outer or full outer join is not permitted.

The reason behind the above restrictions is that you cannot have the entire data of a static DataFrame for a join as it could be too huge a load on the system, owing to the volume of the data.

Stream - A stream full outer join is not permitted, since streams are unbounded and a full outer join would mean putting everything the stream has ever received. This would again mean a huge load on the system.

Because of the computational constraints mentioned above, only append mode is supported for Stream–Stream joins in Spark Structured Streaming.

Advanced Structured Streaming

Window Operations

In applications that process real-time events, it is common to perform some set-based computations (aggregation) or other operations on subsets of events that fall within some period of time. Since the concept of time is a fundamental necessity of complex event-processing systems, it is important to have a simple method to work with the time component of the query logic in the system.

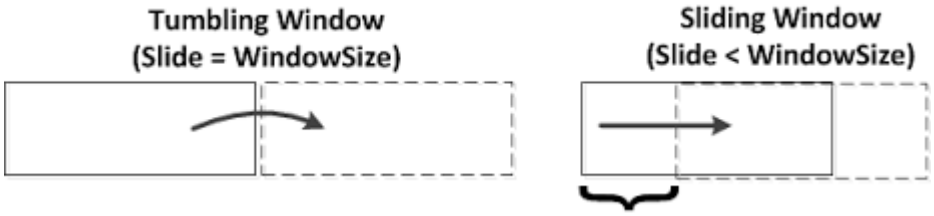
Event time is the time when a record is generated at the source. It is generally represented as a column in the source data set, and is different from the processing time.

Processing time is the time when a record arrives at the Spark processing layer. The differences between event time and processing time arise due to various reasons, such as publishing failures, lags in distributed systems, network delays and other such latencies.

Next, you learnt about windows in Spark and understood how they are similar to the concept of windows in SQL. A window is nothing but a collection of records over a specific time period. Windows are of the following two types:

1. **Tumbling window:** No two windows overlap.
2. **Sliding window:** Windows may or may not overlap.

Sliding duration in the tumbling window is equal to window duration, which ensures that no two windows are overlapping. In the sliding window, window duration is always a multiple of sliding duration, thereby causing an overlap.



Window functions are values that are computed over a window.

Handling Late-Arriving Data

The real world contains many dependencies, which leads to frequent substantial latencies. In the case of streaming data, there must be a mechanism in place to handle late-arriving data effectively so as to get proper results.

Late-arriving data occurs due to a delay between event time and processing time. Stream-processing systems need a way to handle such data. The system cannot keep waiting forever for data to arrive because of resource constraints. This is where we use watermarks.

Watermarks define the time period after which Spark will start dropping the records of astream. It is a scalable way of managing late-arriving data. With every incoming batch, Spark checks the maximum event time that it has already received and applies a watermark on that event time.

While choosing a watermark, we must bear in mind that there should be enough time to accommodate failovers, network lags, etc. However, it should not be so long that it puts the system under excess load. The actual watermark would depend on available memory and incoming traffic per minute.

Spark Integration - Apache Kafka

Kafka Integration

Kafka is a potential messaging and integration platform for Spark Streaming. It acts as the central hub for real-time streams of data, which are processed in Spark Streaming using complex algorithms. Once data is processed, Spark Streaming either publishes the results into another Kafka topic or stores them in HDFS, databases or dashboards.

Kafka is a state-of-the-art messaging system. It is highly scalable, thus making it popular in the industry. It follows the publisher–subscriber or the pub–sub model. In this model, a publisher can give its inputs to a topic and multiple subscribers can access that topic. A topic can be thought of as equivalent to a table in a database system. This integrates well with Spark Streaming for further processing of data streams.