Spark Streaming

Introduction

- Streaming transmits data as a continuous flow, which allows the recipients to start processing almost immediately. The data could be coming from multiple sources. The data could be going to multiple sinks.
- A stream represents a continuous sequence of events that goes from producers to consumers, where an event is defined as a key-value pair.
- Common examples: Audio, Video (for example Netflix, iTunes, Hulu, YouTube etc)
- More interesting examples!
  - The analysis of GPS car data can allow cities to optimize traffic flows based on real-time traffic information.
  - Uber uses Spark streaming to detect and visualize popular Uber locations
  - Smart cities will be using about 1.39 billion connected cars, IoT sensors, and devices by 2020. The analysis of location and behavior patterns within cities will allow optimization of traffic, better planning decisions, and smarter advertising
  - Real time data in manufacturing can be used for anomaly detection

Streaming in Spark

- Spark provides two ways of processing stream data:
- Spark Streaming: a separate library in Spark to process continuously flowing streaming data. It provides us with the `DStream` API, which is powered by Spark RDDs. DStreams provide us data divided into chunks as RDDs received from the source of streaming to be processed and, after processing, sends it to the destination.
- Structured Streaming: Built on the Spark SQL library, Structured Streaming is is based on `DataFrame` and `Dataset` APIs. Hence, with this library, we can easily apply any SQL query (using the `DataFrame` API) or Scala/Java operations (using `Dataset` API) on streaming data.
- We can express our streaming computation the same way we would express a batch computation on static data. The Spark SQL engine will take care of running it incrementally and continuously and updating the final result as streaming data continues to arrive.

More on Structured Streaming in Spark

- The system ensures end-to-end exactly-once fault-tolerance guarantees through checkpointing and Write-Ahead Logs.
- Internally, by default, Structured Streaming queries are processed using a micro-batch processing engine, which processes data streams as a series of small batch jobs thereby achieving end-to-end latencies as low as 100 milliseconds and exactly-once fault-tolerance guarantees.
- Since Spark 2.3, a new low-latency processing mode called Continuous Processing has been introduced that can achieve end-to-end latencies as low as 1 millisecond with at-least-once guarantees.

Example: Streaming Wordcount (1)

- Let's say that we want to maintain a running word count of text data received from a data server listening on a TCP socket. This would be an example of a streaming application.
- First we setup a streaming DataFrame that represents text data received from a server listening on `localhost:9999`. Here `spark` is the usual SparkSession object.

```
Dataset<Row> lines = spark
  .readStream()
  .format("socket")
  .option("host", "localhost")
  .option("port", 9999)
  .load();
```

- Next we split the line into words and then we count them into another streaming DataFrame, which represents tthe he running word counts of the stream.

```
// Split the lines into words
Dataset<String> words = lines
  .as(Encoders.STRING())
  .flatMap((FlatMapFunction<String, String>) x -> Arrays.
    asList(x.split(" ")).iterator(), Encoders.STRING());

// Generate running word count
Dataset<Row> wordCounts = words.groupBy("value").count();
```

Example: Streaming Wordcount (2)

- Now the query on the streaming data is set up. So we now start receiving data and computing the counts.

```
StreamingQuery query = wordCounts.writeStream()
  .outputMode("complete")
  .format("console")
  .start();

query.awaitTermination();
```

- After this code is executed, the streaming computation will have started in the background.
- The query object is a handle to that active streaming query, and we have decided to wait for the termination of the query using `awaitTermination()` to prevent the process from exiting while the query is active.
- See streaming/JavaStructuredNetworkWordCount.java.

Programming Model (1)

- The key idea is that the live data stream is treated as a table that is being continuously appended. We will express our streaming computation as standard batch-like query as on a static table, and Spark runs it as an incremental query on the unbounded input table.
- Every trigger interval, new rows get appended to the Input Table, which eventually updates the Result Table. See the next slide for various trigger options.
- The output can be defined in a different mode (note that each mode is applicable to certain types of queries):
  - Complete mode: The entire updated Result Table will be written to the external sink.
  - Append mode (*default*): Only the new rows appended in the Result Table since the last trigger will be written to the external sink.
  - Update mode: Only the rows that were updated in the Result Table since the last trigger will be written to the external sink.
  - Use the method `outputMode("option")` on the result of `writeStream` to change the default.

Programming Model (2): Triggers

- Default trigger runs the micro-batch as soon as it can.

```
StreamingQuery query = wordCounts.writeStream()
    .outputMode("complete")
    .format("console").start();
```

- Processing time trigger with 5 seconds micro-batch interval

```
StreamingQuery query = wordCounts.writeStream()
    .outputMode("complete")
    .format("console")
    .trigger(Trigger.ProcessingTime(5000)).start();
```

- One-time trigger

```
StreamingQuery query = wordCounts.writeStream()
    .outputMode("complete")
    .format("console")
    .trigger(Trigger.Once()).start();
```

- Continuous trigger: A trigger that continuously processes streaming data, asynchronously checkpointing at the specified interval.

```
StreamingQuery query = wordCounts.writeStream()
    .outputMode("complete")
    .format("console")
    .trigger(Trigger.Continuous(1000)).start();
```

Programming Model (3)

- Handling Event-time and Late Data: support for watermarking which allows the user to specify the threshold of late data, and allows the engine to accordingly clean up old state.
- Fault Tolerance Semantics: Delivers end-to-end exactly-once semantics by designing the sinks and the execution engine to reliably track the exact progress of the processing so that it can handle any kind of failure by restarting and/or reprocessing. Uses *replayable* sources, such as Kafka, and *idempotent* sinks.
  - Apache Kafka is used for building real-time data pipelines and streaming apps. It is horizontally scalable, fault-tolerant, wicked fast, and runs in production in thousands of companies.
- Input sources: Files (text, CSV, JSON, ORC, Parquet), Kafka, Sockets (for testing), Rate source (for testing/benchmarking)

```
StructType userSchema = new StructType().add("name", "
    string").add("age", "integer");
Dataset<Row> csvDF = spark
  .readStream()
  .option("sep", ";")
  .schema(userSchema)
  .csv("/path/to/directory");
```

Output Sinks

- File sink: Stores the output to a directory.

```
writeStream
    .format("parquet")        // can be "orc", "json", "
    csv", etc.
    .option("path", "path/to/destination/dir")
    .start()
```

- Kafka sink: Stores the output to one or more topics in Kafka.

```
writeStream
    .format("kafka")
    .option("kafka.bootstrap.servers", "host1:port1,
    host2:port2")
    .option("topic", "updates")
    .start()
```

- Foreach sink: Runs arbitrary computation on the records in the output.

```
writeStream
    .foreach(...)
    .start()
```

- Console sink (for demo/debugging): Prints the output to the console/stdout every time there is a trigger.

```
writeStream
    .format("console")
    .start()
```

- Memory sink (for demo/debugging): The output is stored in memory as an in-memory table.

```
writeStream
    .format("memory")
    .queryName("tableName")
    .start()
```