DDAM, 2019
CONCEPTS ON SPARK STREAMING

Docente: Patrizio Dazzi

OVERVIEW

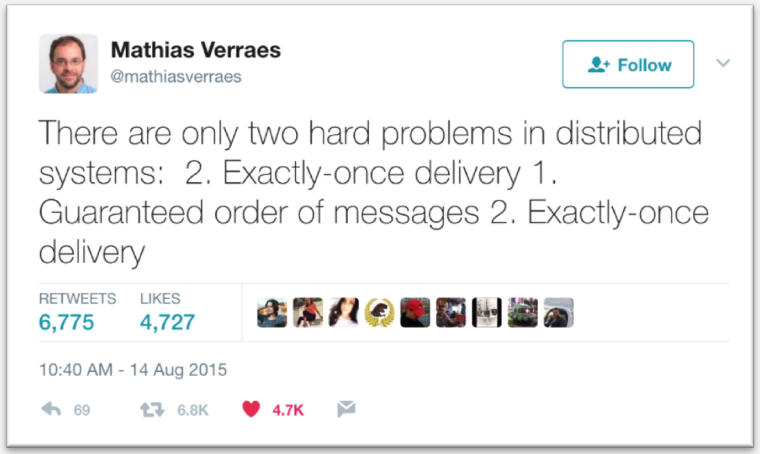
- Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine.
- Streaming computation represented in the same way of a batch computation on static data.
  - The Spark SQL engine will take care of running it incrementally and continuously and updating the final result as streaming data continues to arrive.
- You can use the Dataset/DataFrame API in Scala, Java, Python or R to express streaming aggregations, event-time windows, stream-to-batch joins, etc.
- The computation is executed on the same optimized Spark SQL engine.

PROPERTIES

- The system ensures end-to-end exactly-once fault-tolerance guarantees through checkpointing and Write-Ahead Logs
- In short, Structured Streaming provides:
  - fast,
  - scalable,
  - fault-tolerant,
  - end-to-end exactly-once

stream processing without the user having to reason about streaming.

EXACTLY-ONCE SEMANTICS



- When we say "exactly-once semantics," what we mean is that:
  - each incoming event affects the final results exactly once
  - Even in case of a machine or software failure, there's no duplicate data and no data that goes unprocessed.

SPARK STREAMING

- 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 and exactly-once fault-tolerance guarantees.
- However, recently Spark introduced a new low-latency processing mode called Continuous Processing, which can achieve end-to-end latencies as low as 1 millisecond with at-least-once guarantees.
- Without changing the Dataset/DataFrame operations in your queries, you will be able to choose the mode based on your application requirements.

SCENARIO

- You want to maintain a running word count of text data received from a data server listening on a TCP socket. Do you know what a socket is ?
- Let's see how you can express this using Structured Streaming.

LET'S START…

- First, we have to import the necessary classes and create a local SparkSession, the starting point of all functionalities related to Spark.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode
from pyspark.sql.functions import split

spark = SparkSession \
    .builder \
    .appName("StructuredNetworkWordCount") \
    .getOrCreate()
```

SPARK STREAMING (1)

- Next, let's create a streaming DataFrame that represents text data received from a server listening on localhost:9999, and transform the DataFrame to calculate word counts.

```
# Create DataFrame representing the stream of input lines from connection to localhost:9999
lines = spark \
    .readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

# Split the lines into words
words = lines.select(
   explode(
       split(lines.value, " ")
   ).alias("word")
)

# Generate running word count
wordCounts = words.groupBy("word").count()
```

SPARK STREAMING (2)

- **lines** in DataFrame represents an unbounded *table* containing the streaming text data.
- This *table* contains one column of strings named "value", and each line in the streaming text data becomes a row in the table.
- Note, that this is not currently receiving any data as we are just setting up the transformation, and have not yet started it.
- Next, we have used two built-in SQL functions - split and explode, to split each line into multiple rows with a word each.
- In addition, we use the function alias to name the new column as "word".
- Finally, we have defined the wordCounts DataFrame by grouping by the unique values in the Dataset and counting them.
- Note that this is a streaming DataFrame which represents the running word counts of the stream.

SPARK STREAMING (3)

- We have now set up the query on the streaming data.
- All that is left is to actually start receiving data and computing the counts.
- To do this, we set it up to print the complete set of counts (specified by outputMode("complete")) to the console every time they are updated.
- And then start the streaming computation using start().

```
 # Start running the query that prints the running counts to the console
query = wordCounts \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

query.awaitTermination()
```

SPARK STREAMING (4)

- 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.

```
$ nc -lk 9999
```

```
$ ./bin/spark-submit examples/src/main/python/sql/streaming/structured_network_wordcount.py localhost 9999
```

SPARK STREAMING (5)

```
# TERMINAL 1:
# Running Netcat

$ nc -lk 9999
apache spark
apache hadoop
...
```

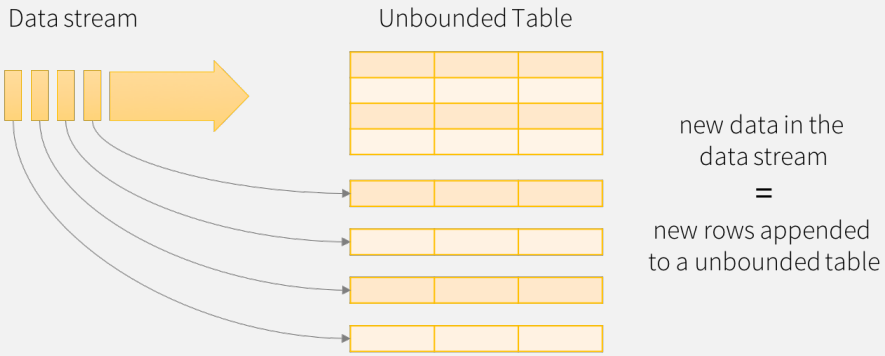
```
# TERMINAL 2: RUNNING structured_network_wordcount.py

$ ./bin/spark-submit examples/src/main/python/sql/streaming/structured_network_wordcount.py localhost 9999

-------------------------------------------
Batch: 0
-------------------------------------------
+------+-----+
| value|count|
+------+-----+
|apache|    1|
| spark|    1|
+------+-----+

-------------------------------------------
Batch: 1
-------------------------------------------
+------+-----+
| value|count|
+------+-----+
|apache|    2|
| spark|    1|
|hadoop|    1|
+------+-----+
...
```

PROGRAMMING MODEL

- The key idea in Structured Streaming is to treat a live data stream as a table that is being continuously appended.
- This leads to a new stream processing model that is very similar to a batch processing model.
- You will express your 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.

APPROACH

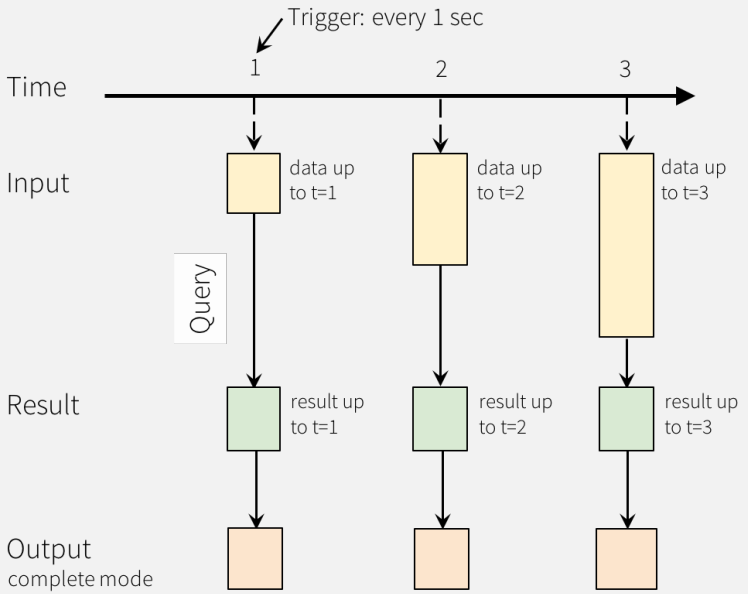
Consider the input data stream as the “Input Table”. Every data item that is arriving on the stream is like a new row being appended to the Input Table.



Data stream as an unbounded table

TRIGGER

- A query on the input will generate the "Result Table".
- Every trigger interval, new rows get appended to the Input Table, which eventually updates the Result Table.
- Whenever the result table gets updated, we would want to write the changed result rows to an external sink.



Programming Model for Structured Streaming

OUTPUT MODES

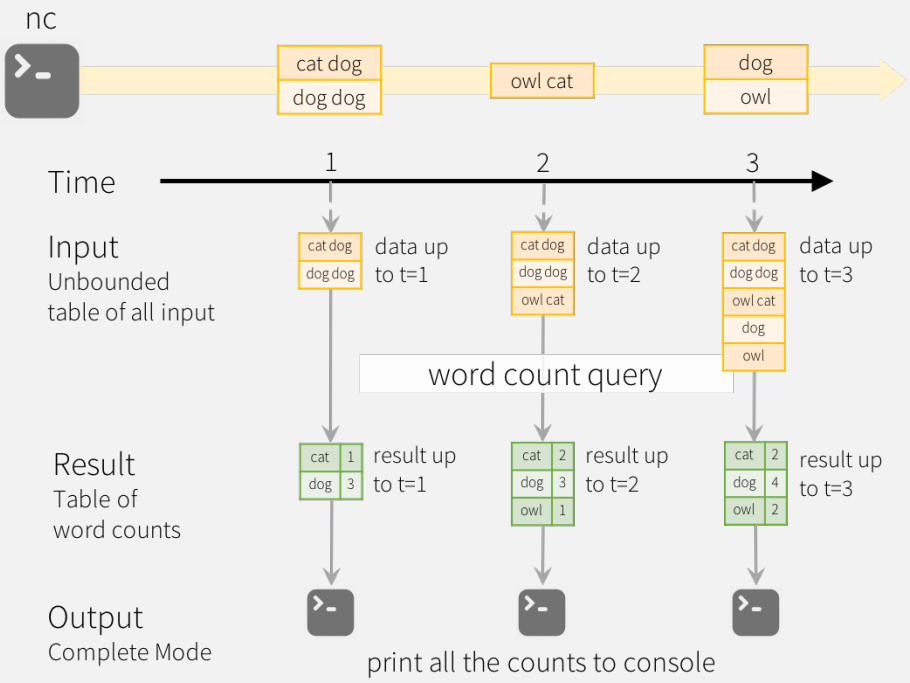
- Complete Mode - The entire updated Result Table will be written to the external storage. It is up to the storage connector to decide how to handle writing of the entire table.

- Append Mode - Only the new rows appended in the Result Table since the last trigger will be written to the external storage. This is applicable only on the queries where existing rows in the Result Table are not expected to change.

- Update Mode - Only the rows that were updated in the Result Table since the last trigger will be written to the external storage.
  - Note that this is different from the Complete Mode in that this mode only outputs the rows that have changed since the last trigger.
  - If the query doesn't contain aggregations, it will be equivalent to Append mode.

- The first lines DataFrame is the input table, and the final wordCounts DataFrame is the result table.
- Note that the query on streaming lines DataFrame to generate wordCounts is exactly the same as it would be a static DataFrame. However, when this query is started, Spark will continuously check for new data from the socket connection.
- If there is new data, Spark will run an "incremental" query that combines the previous running counts with the new data to compute updated counts, as shown below.

EXAMPLE DEPICTED



Model of the Quick Example

DATA MATERIALIZATION

- Note that Structured Streaming does not materialize the entire table.
- It reads the latest available data from the streaming data source, processes it incrementally to update the result, and then discards the source data.
- It only keeps around the minimal intermediate state data as required to update the result (e.g. intermediate counts in the earlier example).
- This model is significantly different from many other stream processing engines.
- Many streaming systems require the user to maintain running aggregations themselves, thus having to reason about fault-tolerance, and data consistency (at-least-once, or at-most-once, or exactly-once).
- In this model, Spark is responsible for updating the Result Table when there is new data, thus relieving the users from reasoning about it.