Structured Streaming

Created by Mahidhar

Reference : Spark Structured Streaming

Problems with Classical Spark Streaming

1. Processing with event_time and late data
   - DStream exposes batch time but it is hard to incorporate event time

2. Interacting with batch and Stream
   - DStream/RDD have similar API but then also require translation

3. About End-End Guarantees (Exactly-once)
   - Data Consistency in the storage while being updated

What is Structured Streaming

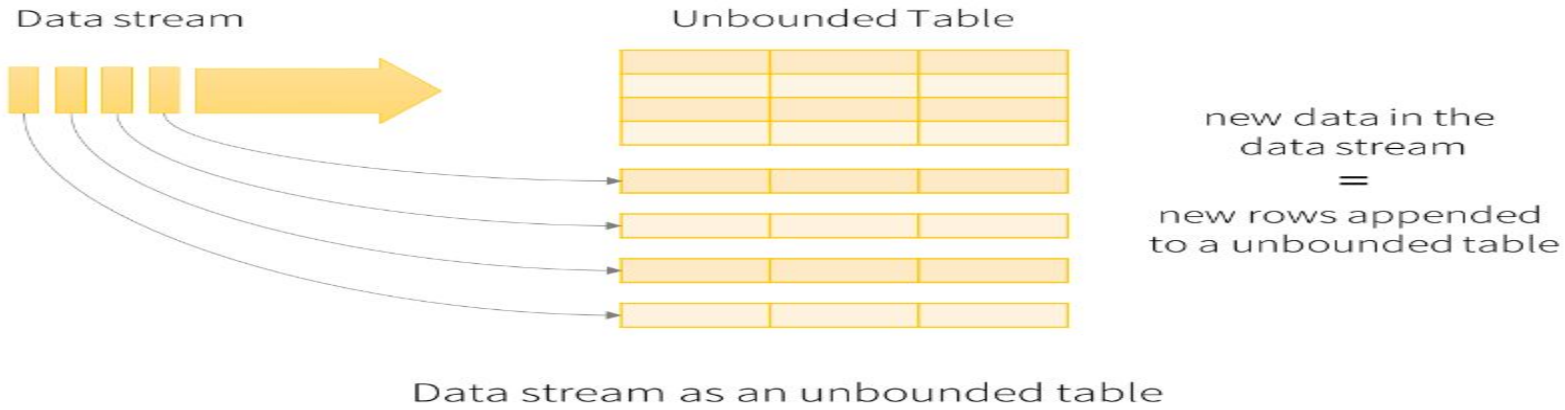
- Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine.

- The Spark SQL engine will take care of running it incrementally and continuously and updating the final result as streaming data continues to arrive.

- A unified API for working with both Batch and Streaming Data.

Programming Model

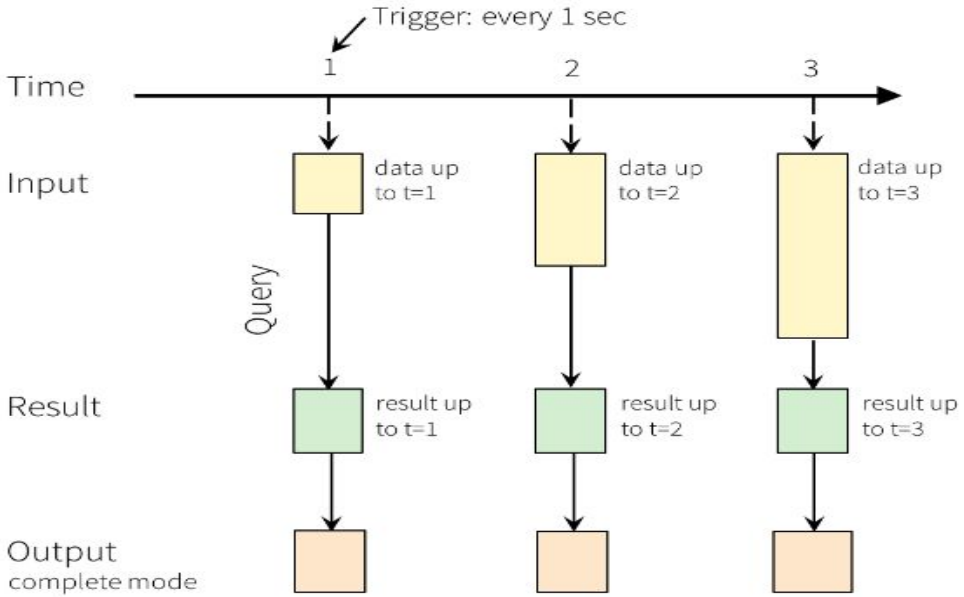
- Key idea is to treat live data as a table that is continuously appended
- Query is performed as a incremental query



Creating streaming DataFrames

- Streaming DataFrames can be created through the DataStreamReader interface (Scala/Java/Python docs) returned by SparkSession.readStream()
- Input sources are :
  - **File source** - Reads files written in a directory as a stream of data. Supported file formats are text, csv, json, orc, parquet.
  - **Kafka source** - Reads data from Kafka. It's compatible with Kafka broker versions 0.10.0 or higher.
  - **Socket source (for testing)** - Reads UTF8 text data from a socket connection. The listening server socket is at the driver. Note that this should be used only for testing as this does not provide end-to-end fault-tolerance guarantees.
  - **Rate source (for testing)** - Generates data at the specified number of rows per second, each output row contains a timestamp and value. Where timestamp is a Timestamp type containing the time of message dispatch, and value is of Long type containing the message count, starting from 0 as the first row. This source is intended for testing and benchmarking.

Query



Programming Model for Structured Streaming

A query on the input will generate the “Result Table”. Every trigger interval (say, every 1 second), 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.

Triggers

- The trigger settings of a streaming query defines the timing of streaming data processing, whether the query is going to executed as micro-batch query with a fixed batch interval or as a continuous processing query
- If no trigger setting is explicitly specified, then by default, the query will be executed in micro-batch mode, where micro-batches will be generated as soon as the previous micro-batch has completed processing.
- **Fixed interval micro-batches:** The query will be executed with micro-batches mode, where micro-batches will be kicked off at the user-specified intervals.
  - If the previous micro-batch completes within the interval, then the engine will wait until the interval is over before kicking off the next micro-batch.
  - If the previous micro-batch takes longer than the interval to complete (i.e. if an interval boundary is missed), then the next micro-batch will start as soon as the previous one completes (i.e., it will not wait for the next interval boundary).
  - If no new data is available, then no micro-batch will be kicked off.

contd..

One-time micro-batch

The query will execute *only one* micro-batch to process all the available data and then stop on its own. This is useful in scenarios you want to periodically spin up a cluster, process everything that is available since the last period, and then shutdown the cluster. In some case, this may lead to significant cost savings.

Continuous with fixed checkpoint interval*(experimental)*

The query will be executed in the new low-latency, continuous processing mode.

Output-mode

The “Output” is defined as what gets written out to the external storage. The output can be defined in a different mode:

- *Complete Mode* - The entire updated Result Table will be written to the external storage.
- *Append Mode* - Only the new rows appended in the Result Table. This is applicable only on the queries where existing rows in the Result Table are not expected to change.

contd..

- *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.

Output Sinks

- There are a few types of built-in output sinks.

Sink	Output Mode	Fault Tolerance
FIle	Append	Yes
Kafka	All	Yes
ForEach	All	Depends
Console	All	No
Memory Sink	Append , Complete	No, but restarted query will recreate the full table for complete mode

WordCount -illustration

Input Table - lines (Data Frame)

```
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()
```

Result table - Word Counts (Data Frame)

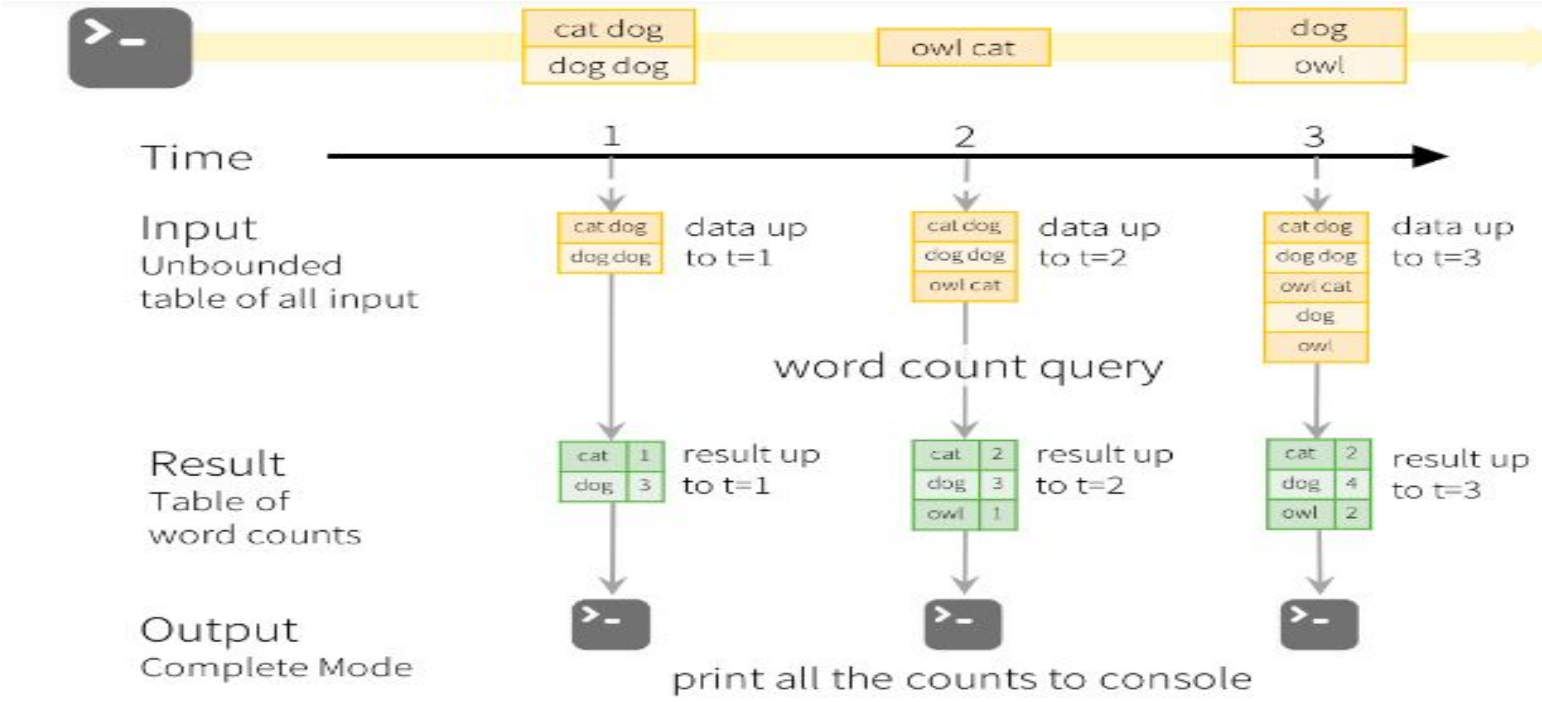
```
query = wordCounts \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

query.awaitTermination()
```

Operations on streaming DataFrames

- All kinds of operations on streaming DataFrames/Datasets – ranging from untyped, SQL-like operations (e.g. select, where, groupBy), to RDD-like operations (e.g. map, filter, flatMap)
- To identify whether a DataFrame/Dataset has streaming data or not by use df.isStreaming( )
- There are few unsupported operations like :
  - Multiple streaming aggregations (i.e. a chain of aggregations on a streaming DF) are not yet supported on streaming Datasets.
  - Limit and take first N rows are not supported on streaming Datasets.
  - Distinct operations on streaming Datasets are not supported.
  - Sorting operations are supported on streaming Datasets only after an aggregation and in Complete Output Mode.
  - Few types of outer joins on streaming Datasets are not supported.

Demo of Structured Streaming Word Count

Word Count --Contd..



Event Time and Late Data

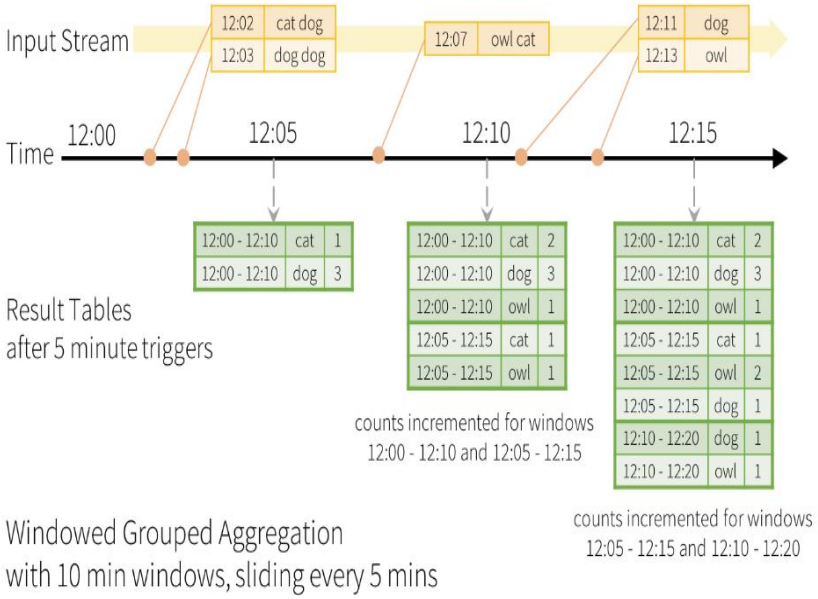
- Event-time is the time embedded in the data itself.
- This event-time is very naturally expressed in this model – each event from the devices is a row in the table, and event-time is a column value in the row.
- This allows window-based aggregations (e.g. number of events every minute) to be just a special type of grouping and aggregation on the event-time column – each time window is a group and each row can belong to multiple windows/groups.
- Therefore, such event-time-window-based aggregation queries can be defined consistently on both a static dataset (e.g. from collected device events logs) as well as on a data stream, making the life of the user much easier.

Contd..

- Furthermore, this model naturally handles data that has arrived later than expected based on its event-time.
- Since Spark is updating the Result Table, it has full control over updating old aggregates when there is late data, as well as cleaning up old aggregates to limit the size of intermediate state data.
- Spark supports watermarking which allows the user to specify the threshold of late data, and allows the engine to accordingly clean up old state.

Window Operations on Event Time

- Aggregations over a sliding event-time window -- very similar to grouped aggregations.
- In a grouped aggregation, aggregate values (e.g. counts) are maintained for each unique value in the user-specified grouping column.
- In case of window-based aggregations, aggregate values are maintained for each window the event-time of a row falls into.

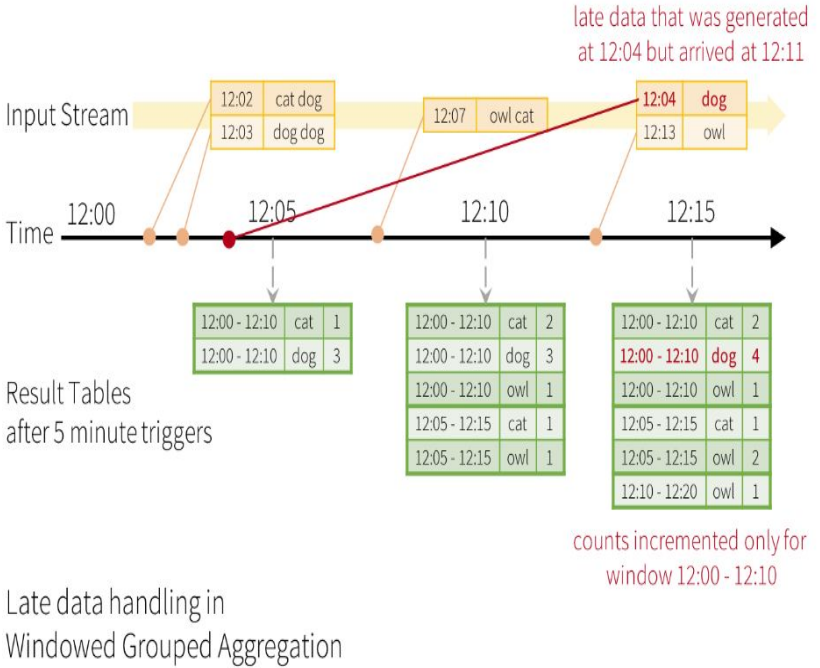


Handling Late Data and Watermarking

Watermarking - lets the engine automatically track the current event time in the data and attempt to clean up old state accordingly

- The engine will maintain state and allow late data to update the state until (max event time seen by the engine - late threshold > T

```
windowedCounts = words \
    .withWatermark("timestamp", "10 minutes") \
    .groupBy(
        window(words.timestamp, "10 minutes", "5 minutes"),
        words.word) \
    .count()
```



Fault Tolerance Semantics

- The semantics of streaming systems are often captured in terms of how many times each record can be processed by the system. There are three types of guarantees that a system can provide under all possible operating conditions

1. *At most once*: Each record will be either processed once or not processed at all.
2. *At least once*: Each record will be processed one or more times. This is stronger than *at-most once* as it ensure that no data will be lost. But there may be duplicates.
3. *Exactly once*: Each record will be processed exactly once - no data will be lost and no data will be processed multiple times. This is obviously the strongest guarantee of the three.

Fault Recovery and Storage System Requirements

Structured Streaming keeps its results valid even if machines fail. To do this, it places two requirements on the input sources and output sinks:

1. **Input sources must be *replayable*, so that recent data can be re-read if the job crashes**. For example, message buses like Apache Kafka are replayable, as is the file system input source. Only a few minutes' worth of data needs to be retained; Structured Streaming will maintain its own internal state after that.
2. **Output sinks must support *transactional updates*, so that the system can make a set of records appear atomically.** The current version of Structured Streaming implements this for file sinks.

Contd ...

- Structured Streaming Sources are by design replayable and generates the same data when the correct offset in WAL is recovered by planner.

  Ex: Kafka
- Intermediate data is maintained in versioned key-value maps in spark workers , backed by HDFS.
- Planner makes sure that correct version is used after failure.
- Sink are designed to be idempotent and handles re-execution to avoid double commiting the output

Recovering from Failures with Checkpointing

- In case of a failure or intentional shutdown, you can recover the previous progress and state of a previous query, and continue where it left off.
- This is done using checkpointing and write ahead logs.
- You can configure a query with a checkpoint location, and the query will save all the progress information (i.e. range of offsets processed in each trigger) and the running aggregates (e.g. word counts in the quick example) to the checkpoint location.
- This checkpoint location has to be a path in an HDFS compatible file system, and can be set as an option in the DataStreamWriter when starting a query.

Using replayable sources and idempotent sinks, Structured Streaming can ensure ***end-to-end exactly-once semantics*** *under any failure.*