Data 603 – Big Data Platforms

Lecture 7
Structured Streaming (Part 1)

Final Paper & Project Check-Ins

#	Date	Activity	Expected Outcome
4	*2022-02-24*	*Technical paper proposal ready for defense*	*Every student will submit his paper proposal*
5	*2022-03-03*	*Project idea ready*	*Prepare a slide deck for presenting the project idea*
7	**2022-03-17**	**Today**	**Exist**
9	2022-03-31	Submit paper progress (outline minimum)	Every student will prepare and submit a paper progress report (markdown)
10	2022-04-06	Submit project progress report	Every student will prepare and submit a project progress status report (markdown or PDF)

Streaming

- Streaming vs batch
  - Batch: Processing of static block of data
  - Streaming: Processing of continuous stream of data
- Traditional Record at a time model
  - Each record is processed at a time
  - Directed graph of nodes. Each node receives a record at at time, process it and forwards the processed record to the next node in the graph.

Traditional Record-at-a-time Model

- Each record is processed at a time
- Directed graph of nodes.
  - Each node receives a record at at time, process it and forwards the processed record to the next node in the graph.
  - Pro: Can achieve low latencies
  - Cons: No good failure recovery strategy.

Micro-Batch Stream Processing

- Micro-batch stream processing
  - Continuous series of small, map/reduce style batch processing jobs on small chunks of stream data.
- Spark Streaming divides the data from the input stream into micro-batches.
  - Each batch is processed in the Spark cluster in a distributed manner with small deterministic tasks
  - Micro-batches are generated as outputs

Micro-Batch Costs and Benefits

Benefits

- Recovery from failures
  - One or more copies of the tasks can be rescheduled on executors
  - DStream API was built upon Spark's batch RDD; same functional semantics and fault-tolerance model as RDDs

- Deterministic nature of the tasks
  - Ensures that the output data is the same no matter how many times the task is re-executed.
  - Provides end-to-end exactly-once processing guarantee
    - Every input records are processed exactly once.

Cost

- High latency – seconds vs milliseconds

Streaming pipeline characteristics

- Pipelines do not need latencies below few seconds
  - Down stream process may not read output of the stream process
- Larger delays in other parts of the pipeline.
  - Batching at data ingestion layer (e.g. Apache Kafka)

Spark RDD Streaming (DStreams)

- Lack of a single API for batch and stream processing
  - Developers having to explicitly rewrite the code to use different classes when converting batch jobs to streaming job
- Lack of separation between logical and physical plans
  - No scope for automatic optimization
  - Developers will need to hand-optimize the code
  - DStream operations are executed in the same sequence as specified by the developer
- No native support for event-time windows
  - DStream define window operations based only on the time when each record is received by Spark Streaming (processing time)
  - Many cases need to work with event time (when the records were generated).

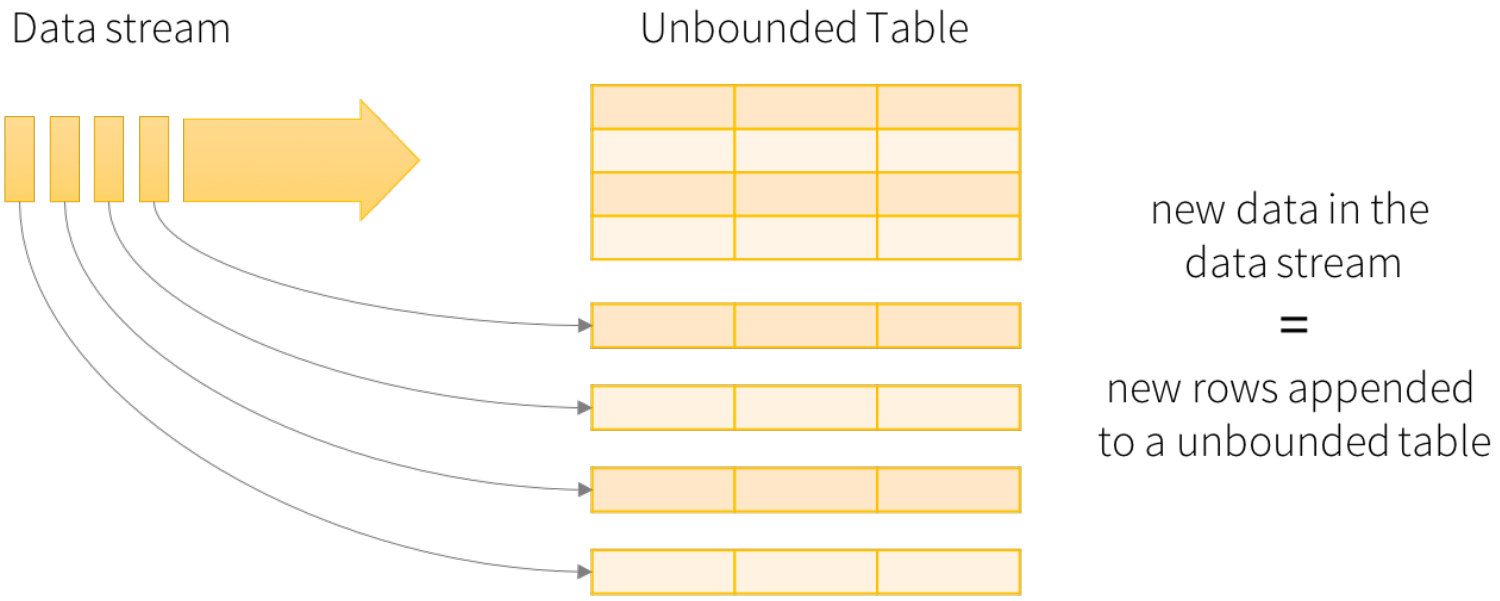
Spark Structured Streaming

- Structured Streaming was designed from scratch
  - Developing streaming process pipeline to be easy as writing batch pipeline.

Structured Streaming – Guiding Principles

- A single unified programming model and interface for batch and stream processing
  - Simple API interface for both batch and streaming workloads
  - Can use SQL or DataFrame queries with the benefits of fault tolerance and optimizations
- Broader definition of stream processing
  - Blurring of the line between batch processing and real-time processing
  - Structured Streaming broadens its applicability from traditional streaming processing to a larger class of applications (continuous periodic processing)

Structured Streaming – Programming Model



Data stream as an unbounded table

Structured Streaming – Programming Model

- Unbounded, continuously appended table.
  - Every new record received in the data stream is a new row being appended to the unbounded input table.
  - Does not retain all the input
  - The output produced during time T is equivalent to having all of the input in a static bounded table and running a batch job on the table.

Structured Streaming – Programming Model

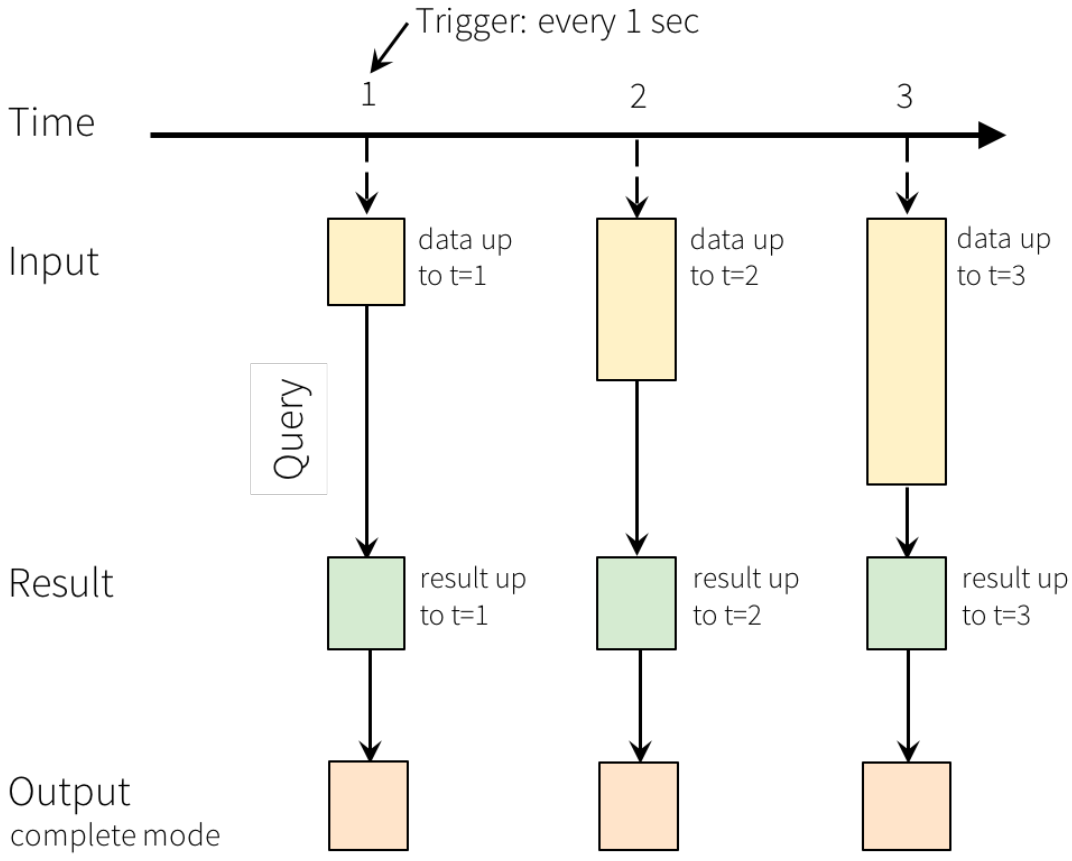
- Unbounded, continuously appended table.
  - The key idea in Structured Streaming is to treat a live data stream as a table that is being continuously appended.
  - Every new record received in the data stream is a new row being appended to the unbounded input table.
  - Does not retain all the input
  - The output produced during time T is equivalent to having all of the input (during T) in a static bounded table and running a batch job on the table.
  - Streaming computations are expressed as standard batch-like query as on a static table.
  - Spark runs it as an incremental query on unbounded input table.

Structured Streaming – *incrementalization*

- Incrementalization: Structured Streaming automatically converting batch-like query to a streaming execution plan.
  - Structured Streaming figures out what state needs to be maintained to update the result each time a new record arrives.
- Developers specify triggering policies to control when to update the results
  - When a trigger fires, Structured Streaming checks for new data and incrementally updates the result.

Structured Streaming – Programming Model

- A query on the input generates the "Result Table".
  - After every trigger interval (e.g. 1 second), new rows get appended to the Input Table, which eventually updates the Result Table.
  - When the result table is updated, changed result rows can be written out to an external sink.
  - "Output" is defined as what gets written out to the external storage.



Programming Model for Structured Streaming

Structured Streaming – Output Mode 1/2

- Complete Mode – The entire updated Result Table is written to the external storage.
  - Up to the storage connector to decide how to handle writing of the entire table.
  - Supported by queries where the result table is likely to be much smaller than the input data (can be maintained in memory)
- Append Mode (default) – Only the rows *appended* in the Result Table since the last trigger are written to the external storage.
  - On applicable to the queries where existing rows in the Result Table are not expected to change.
  - Supported by only stateless queries (to be covered soon) that never modify previously output data.

Structured Streaming – Output Mode 2/2

- Update Mode – Only the rows that were *updated* in the Result Table since the last trigger are written to the external storage.
  - If the query doesn't contain aggregations, it is equivalent to Append mode.
  - Most queries support update mode

Structured Streaming – DataFrame API

- DataFrame API can be used to express the computations on streaming data
  - Need to define an input DataFrame (i.e. the input table) from a streaming data source
  - Apply operations on the DataFrame in the same as as on a batch source

Five Steps to Define a Streaming Query

- Step 1: Define input sources
  - Define a DataFrame from a streaming source
  - Use spark.readStream to create a DatastreamReader (vs. using spark.read to create a DataFrameReader)
  - A streaming query can define multiple input sources, both streaming and batch, which can be combined using DataFrame operations.

```
spark = SparkSession...
lines = (
    spark.readStream.format("socket")
         .option("host", "localhost")
         .option("port", 9999)
         .load()
)
```

- *lines* is an unbounded table

Five Steps to Define a Streaming Query

- Step 2: Transform Data

```
from pyspark.sql.functions import *
words = lines.select(split(col("value"), "\\s")
                          .alias("word"))
counts = words.groupBy("word").count()
```

  - Counts is a streaming DataFrame (a DataFrame on unbounded, streaming data)

Five Steps to Define a Streaming Query

Two broad classes of data transformation

- Stateless transformations
  - Do not require information from previous rows to process the next row
  - Each row can be processed by itself.
  - The lack of previous "state" in these operations make them stateless.
  - Can be applied to both batch and streaming DataFrames.
  - select(), filter(), map()
- Stateful transformations
  - Requires maintaining state to combine data across multiple rows.
  - Any DataFrame operations involving grouping, joining , or aggregating
  - For Structured Streaming, few combinations are not supported.

Five Steps to Define a Streaming Query

Step 3: Define output sink and output mode

- Define how to write the processed output data with `DataFrame.writeStream` (vs. `DataFrame.write` for batch)
- Options
  - Output writing details (where and how to write the output)
  - Processing details (how to process data and recover from failures)

```
writer = (counts.writeStream.format("console")
                .outputMode("complete"))
```

- "console" is the output streaming sink
- "complete" is the output mode
  - Specifies what part of the updated output to write out after processing new data.

Five Steps to Define a Streaming Query

Step 4: Specify processing details

```
checkpointDir = "..."
writer2 = (
    writer.trigger(processingTime="1 second")
          .option("checkpointLocation", checkpointDir))
```

Triggering Details

- When to trigger the discovery and processing of newly available streaming data.
  - Four Options:
    - Default: processing of next micro-batch is triggered as soon as the previous micro-batch has completed
    - Processing time with trigger interval: Triggering on fixed interval
    - Once: Processes all the new data available in a single batch and stops itself
    - Continuous: Experimental mode (Spark 3.0), new data is processed continuously instead of in micro-batches. Provides lower latency.

Five Steps to Define a Streaming Query

Step 5: Start the query

```
streamingQuery = writer2.start()
```

- streamingQuery:
  - Returned object of type streamingQuery
  - Represents an active query. It can be used to manage the query.
- start() is a nonblocking method- returns as soon as the query has started in the background.
- If main thread is to be blocked until the query has terminiated, use streamingQuery.awaitTermination().
  - Explicitly stop the query with streamingQuery.stop()

Five Steps to Define a Streaming Query

```
lines = (spark.readStream.format("socket")
           .option("host","localhost")
           .option("port", 9999)
           .load())
words = lines.select(split(col("value"), "\\s").alias("word"))
counts = words.groupBy("word").count()
checkpointDir = "..."
streamingQuery = (
   counts.writeStream.format("console")
         .outputMode("complete")
         .trigger(processingTime="1 second")
         .option("checkpointLocation", checkpointDir)
         .start())

streamingQuery.awaitTermination()
```

Checkpoints

- Checkpoints contain the unique identify of a streaming query and determines the life cycle of the query
- Checkpoints have record-level information
  - It tracks the data range the last incomplete micro-batch was processing. This information is used by restarted query to start processing records after the last successfully completed micro-batch
  - If the check point directory is deleted, it is like starting new query from scratch
- Works with Spark's deterministic task executions to generate output to be the same as it was expected before the restart.

Checkpoints

Checkpoint Location

- Directory in any HDFS-compatible filesystem where streaming query saves its progress information – what data has been successfully processed.
- Metadata is used during failure to query exactly where it left off
- This option is necessary for failure recovery with exactly-one guarantee.

End-to-end Exactly-Once Guarantees

- Exactly-once guarantees: Output is as if each input record was processed exactly once.
- Following conditions have to be satisfied:
  - Replayable streaming sources
    - The data range of the last incomplete micro-batch can be reread from the source.
  - Deterministic computations
    - All data transformations deterministically produce the same result when given the same input data
  - Idempotent streaming sink
    - The sink can identify reexecuted micro-batches and ignore duplicate writes that may be caused by restarts.

Monitoring an Active Query

There are several ways to track the status and processing metrics of active query:

- Querying current status using StreamingQuery
  - lastProgress() returns information on the last completed micro-batch.
    - processedRowsPerSecond – Rate at which rows are being processed and written out by the sink. Key indicator of the health of the query.
  - StreamingQuery.status() provides information on what the background query thread is doing at this moment.
- Publishing metrics using Dropwizard Metrics
  - spark.sql.streaming.metricsEnabled to true
- Public metrics using custom StreamingQueryListeners
  - StreamingQueryListener event listener interface. Only available in Scala/Java.
  - spark.streams.addListener(myListener)

Demos

- `nc –lk <port number>` and
  `nc localhost <port number>`
- https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#quick-example

Questions

Homework

- Stream the location of the ISS into Spark over the course of an hour
- Visualize the path of the ISS in that time, ideally over a world map
- Submission details to follow in Blackboard
- Not due until **4/7**