Data 603 – Big Data Platforms

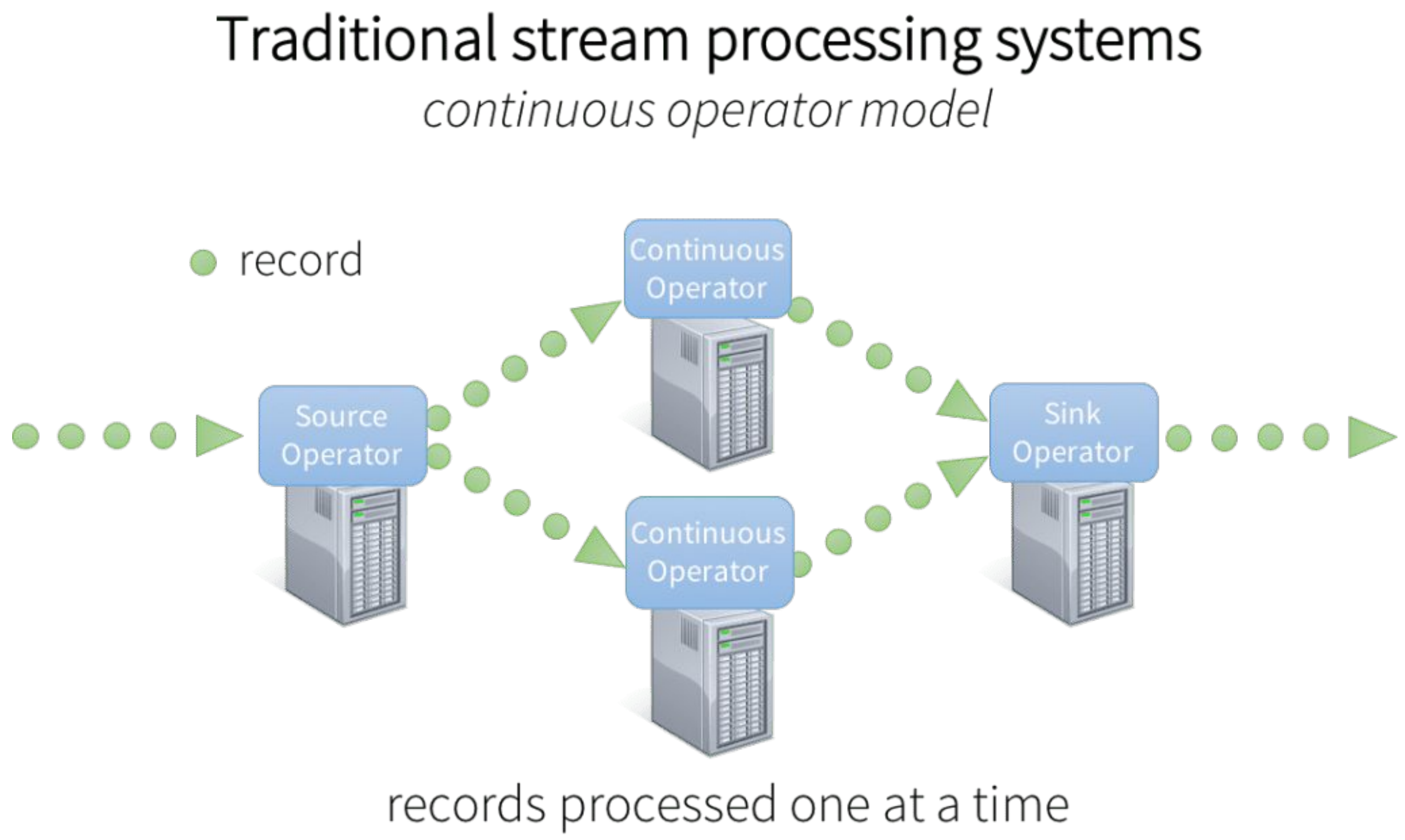
Lecture 7
Structured Streaming (Part 1)

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.
  - Pro: Can achieve low latencies
  - Cons: No good failure recovery strategy.


Traditional stream processing systems
continuous operator model
record
Source Operator
Continuous Operator
Continuous Operator
Sink Operator
records processed one at a time

https://databricks.com/blog/2015/07/30/diving-into-apache-spark-streamings-execution-model.html

Spark Over Traditional Stream Processing

- Fast recovery from failures and stragglers
  - *The static allocation of continuous operators to worker nodes makes it challenging for traditional systems to recover quickly from faults and stragglers.*
- Better load balancing and resource usage
  - *Uneven allocation of the processing load between the workers can cause bottlenecks in a continuous operator system*

https://databricks.com/blog/2015/07/30/diving-into-apache-spark-streamings-execution-model.html

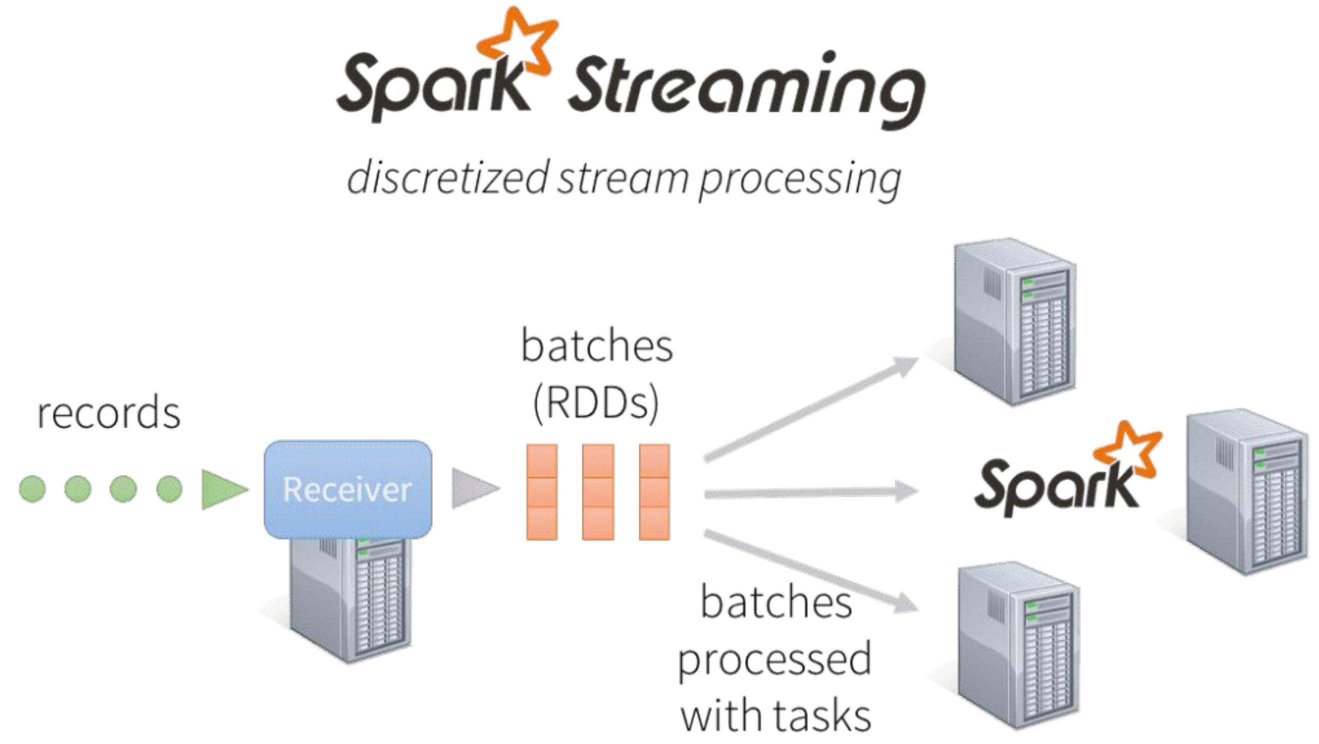
Spark Over Traditional Stream Processing

- Combining of streaming data with static datasets and interactive queries
  - *It provides support for many more use cases without introducing complexity to be able to query the streaming data interactively and to combine it with static datasets (e.g. pre-computed models).*
- Native integration with advanced processing libraries (SQL, machine learning, graph processing)
  - *More complex workloads require continuously learning and updating data models, or even querying the "latest" view of streaming data with SQL queries. Having a common abstraction across these analytic tasks makes the developer's job much easier.*

Reference:
https://databricks.com/blog/2015/07/30/diving-into-apache-spark-streamings-execution-model.html

Spark's Micro-Batch Stream Processing

- Streaming paradigm change brought by Apache Spark Streaming (Dstream)
  - From continuous record at a time to micro-batch 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



records processed in batches with short tasks
each batch is a RDD (partitioned dataset)

From: https://databricks.com/blog/2015/07/30/diving-into-apache-spark-streamings-execution-model.html

Micro-Batch Costs and Benefits

Benefits

- Recovery from failures
  - One or more copies of the tasks can be rescheduled on executors upon failures
  - 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.

Costs

- Higher latency – seconds vs milliseconds
  - For majority of the stream processing use cases, these delays are acceptable.

Streaming pipeline characteristics

The benefits of micro-batch processing outweigh its latency draw back

- Pipelines do not need latencies below few seconds
  - Downstream process may not read output of the stream process for a period of time (e.g. hourly jobs)
- Often, there are larger delays in other parts of the pipeline.
  - Batching at data ingestion layer (e.g. Apache Kafka) to achieve higher throughput.

Spark RDD Streaming (DStreams)

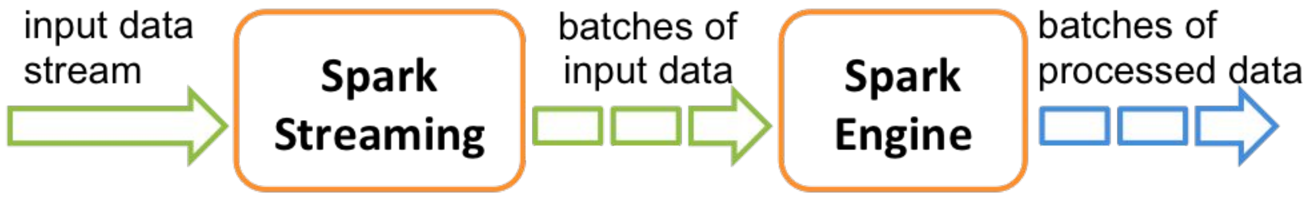
Shortcomings of DStream

- Lack of a single API for batch and stream processing
  - DStreams and RDD have consistent APIs .. .however
  - 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 processing time – when each record is received by Spark Streaming.
  - Many cases need to work with event time - when the records were generated.

Spark Structured Streaming

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

Spark Structured Streaming



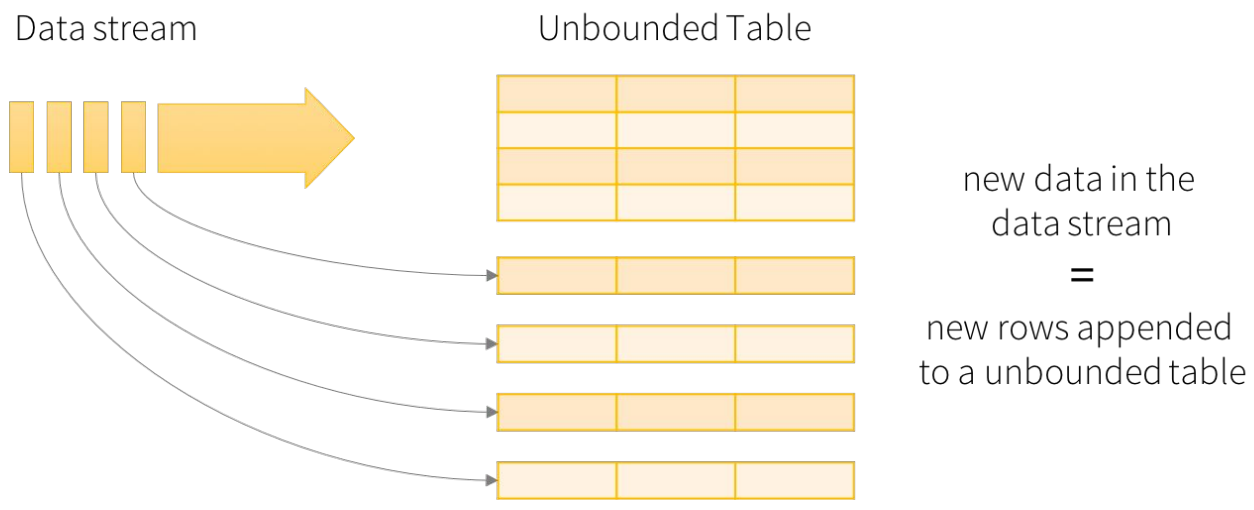
Spark Streaming is an extension of the core Spark API. It provides following benefits over traditional record-at-a-time processing model and DStream:

- Enables scalable, high-throughput, fault-tolerant stream processing of live data streams
- Same computational expressions used for batch processing can be used for Spark Streaming
- Dataset/DataFrame API can be used in Scala, Java, Python or R.
- Leverages the same Spark SQL engine.
- Guaranteed end-to-end fault-tolerance
- Exactly-once streaming processing

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
Unbounded Table
new data in the data stream
=
new rows appended to a unbounded table

Data stream as an unbounded table

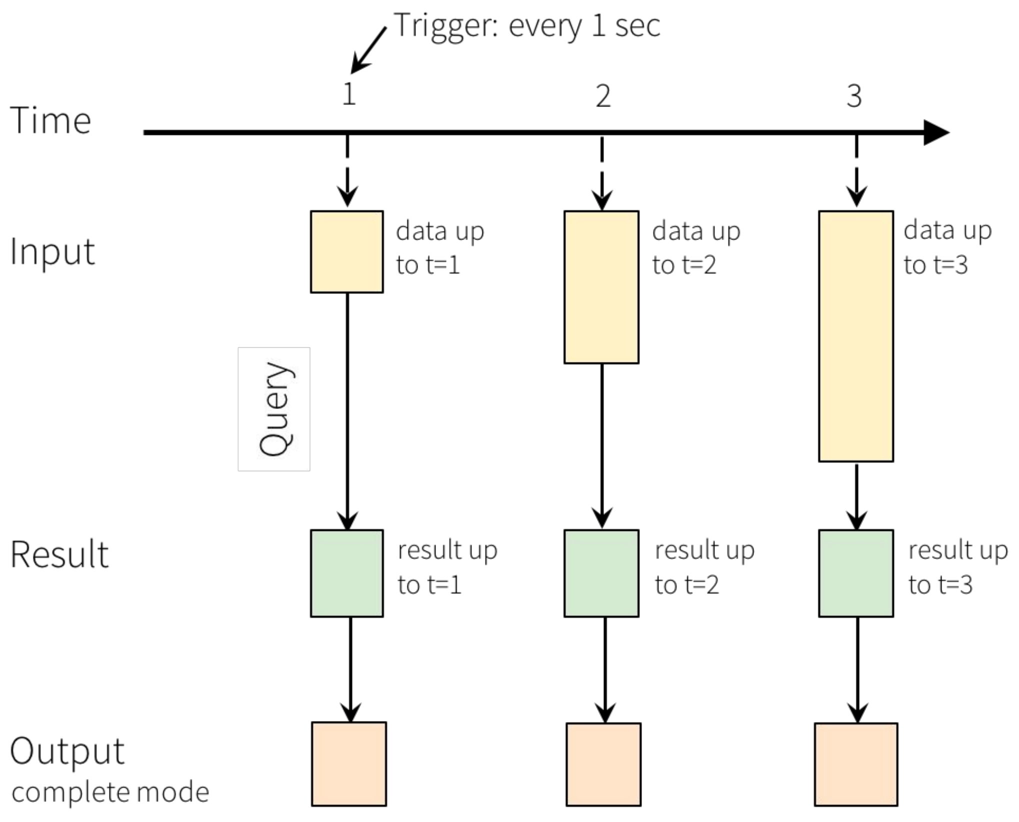
- Structured Streaming extends the concept of streaming applications by treating a stream as an unbounded, continuously appended table.
- Internally, Structured Streaming queries are processed using a *micro-batch processing engine*.
- Streaming computations are expressed as standard batch-like query as on a static table

Structured Streaming – Programming Model

- Every new record received in the data stream is a new row being appended to the unbounded input table.
- Structured Streaming 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.
- A query is defined on the conceptual input table, as if it were a static table, to compute the result table that will be written to an output sink.
- Structured Streaming queries are processed using a *micro-batch processing* engine
  - Data streams are processed as series of small batch jobs 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** is introduced to achieve end-to-end latencies as low as 1 millisecond with at-least-once guarantees.

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 (refer to Structured Streaming Output mode slides).



Programming Model for Structured Streaming

Structured Streaming – Programming Model

Structured Streaming does not materialize the entire table

- Latest available data are read from the streaming data sources and are processed immediately to update the result
- The source data are discarded after being processed.
- Only minimally required intermediate state data are retained to update the result.

This model is different from many stream processing engines

- Many streaming systems require the user to maintain running aggregations.
  - Users need to be mindful of fault-tolerance and data consistency (at-least-once, at-most-once, or exactly-once)
- With Structured Streaming, Spark takes care of updating the Result Table when there is new data, relieving the users of the manual chores.

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 – 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)

- Step 1: Define input sources
  - Define a DataFrame from a streaming source
  - Use *spark.readStream* to create a *DataStreamReader* (vs. for batch processing, we use *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 a DataFrame that represents an unbounded table
  - This table contains one column of strings named “value”, and each line in the streaming text data becomes a row in the table.

Five Steps to Define a Streaming Query (Step 1)

- 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"
- Streaming data is not read immediately: only sets up necessary configurations until the query is explicitly started.

Five Steps to Define a Streaming Query (Step 2)

- 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)
    - Computed once the streaming query is started and the streaming input data is continuously processed.
  - Operations to transform *lines* streaming DataFrame work the exactly the same way as a batch DataFrame.

Two Broad Classes of Data Transformations

- Stateless transformations
  - These type of transformations (e.g. select(), filter(), map()) 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.
- 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.
    - Computationally too expensive

Five Steps to Define a Streaming Query (Step 3)

Step 3: Define output sink and output mode

- Define how to write the processed output data with *DataFrame.writeStream* (vs. *DataFrame.write* for batch data) which creates *DataStreamWriter*
- 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
- Output modes specify what part of the updated output to write out after processing new data.

Five Steps to Define a Streaming Query (Step 3)

- Structured streaming natively supports streaming writes to files and Apache Kafka.
- foreachBatch() and foreach() API methods can be used to write to arbitrary locations.

Structured Streaming – Output Mode 1/2

- 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 (or any row that is output) are not expected to change or updated by future queries.
  - Supported by only stateless queries that never modify previously output data.
- 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.
  - The output rows many be modified by the query and output again in the future.
  - Most queries support update mode
  - Works for output sinks that can be updated (e.g. MySQL table)

Structured Streaming – Output Mode 2/2

- 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)

NOTE: Each mode is applicable on certain types of queries.

For more details, refer here:
https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#output-modes

Five Steps to Define a Streaming Query (Step 4)

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.

Checkpoint Location

- HDFS-compatible filesystem location where streaming query saves its progress information.

Triggering 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
  - Explicitly specify the *ProcessingTrigger* option with an interval
- **Once**: Processes all the new data available in a single batch and stops itself
  - Useful in a scenario when it is necessary to control the triggering and processing from an external scheduler.
- **Continuous**: Experimental mode (Spark 3.0), new data is processed continuously instead of in micro-batches.
  - Provides much lower latency (milliseconds).

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-once guarantee.

Five Steps to Define a Streaming Query

Step 5: Start the query

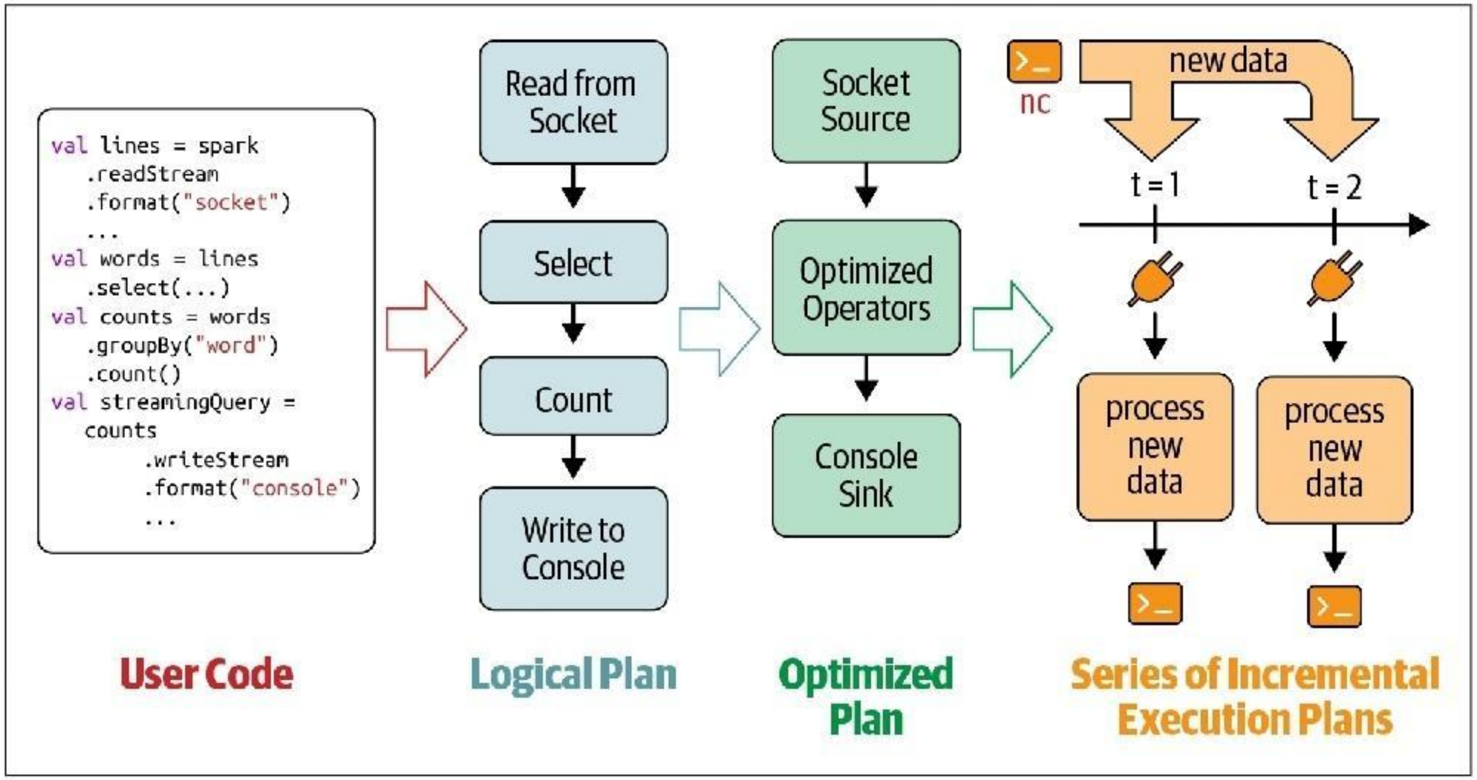
```
streamingQuery = writer2.start()
```

- *streamingQuery*:
  - Represents an active query.
  - Can be used to manage the query.
  - Explicitly stop the query with *streamingQuery.stop()*
- start() is a non-blocking 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()*.
- Wait up to a timeout duration using *awaitTermination(timeoutMillis)*.

Five Steps to Define a Streaming Query

```
from pyspark.sql.functions import *
spark = SparkSession …

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()
```

Streaming Query Execution



```
val lines = spark
   .readStream
   .format("socket")
   ...
val words = lines
   .select(...)
val counts = words
   .groupBy("word")
   .count()
val streamingQuery =
   counts
        .writeStream
        .format("console")
        ...
```

Figure 8-5. Incremental execution of streaming queries

Streaming Query Execution

1. The DataFrame operations are converted into a logical plan
2. Spark SQL analyzes and optimizes the logical plan to ensure that the query can be executed incrementally and efficiently on the streaming data
3. Spark SQL starts a background thread to continuously execute the following:
   a. Based on the trigger interval, the thread checks the streaming sources for the availability of new data
   b. The new data is executed by running a micro-batch. An optimized Spark execution plan in generated to read new data from the source, incrementally computes the update result, and writes the output to the sink based on the output mode configured.
   c. Progress and states are saved in the checkpoint location for each micro-batch
4. The loop continues until the query is terminated
   a. A failure in the query
   b. Explicitly stopped by streamingQuery.stop()
   c. If the trigger is set to Once, the query stops on its own after executing a single micro-batch with all available data.

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 re-executed 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)

Homework

Watch Tathagata Das' (the creator of Spark Structured Streaming) Spark Summit presentations:

- https://databricks.com/session/easy-scalable-fault-tolerant-stream-processing-with-structured-streaming-in-apache-spark
- https://databricks.com/session/easy-scalable-fault-tolerant-stream-processing-with-structured-streaming-in-apache-spark-continues (Unfortunately no audio … ☹.)
- https://databricks.com/session/deep-dive-into-stateful-stream-processing-in-structured-streaming
- Download the slide deck and follow along as you what the videos.

Complete this week's quiz.

- Questions will be based on the contents of the videos.
- There will be more questions than the usual.

UMBC

AN HONORS UNIVERSITY IN MARYLAND

Questions