Apache Streaming

Why Spark Streaming

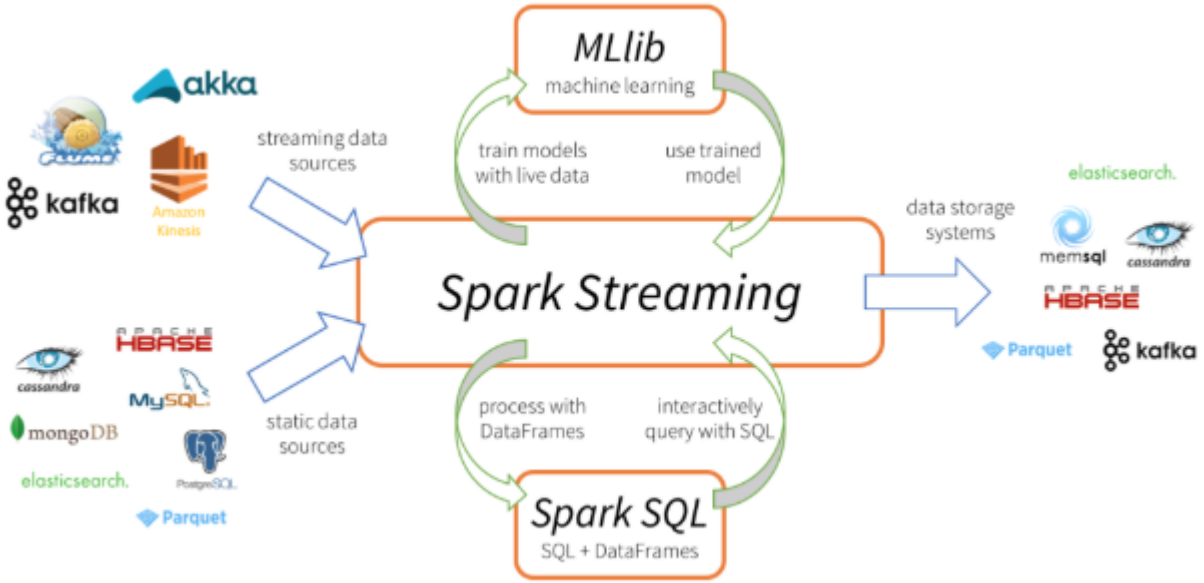
- Many big-data applications need to process large data streams in real time
- Streaming is a technique for transferring data so that it can be processed as a steady and continuous stream.
- Apache Spark has provided an unified engine that natively supports both batch and streaming workloads.

Spark Streaming

Spark Streaming is used to stream real-time data from various sources like Twitter, Stock Market and Geographical Systems and perform powerful analytics to help businesses.

Spark Streaming

- Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.



Basic sources:

- Sources directly available in the StreamingContext API. Examples: file systems, and socket connections.

Advanced sources:

- Sources like Kafka, Flume, Kinesis, etc. are available through extra utility classes.

DStream

- Discretized Stream or DStream is the basic abstraction provided by Spark Streaming
- Represents continuous stream of data
- Implemented as a sequence of RDDs

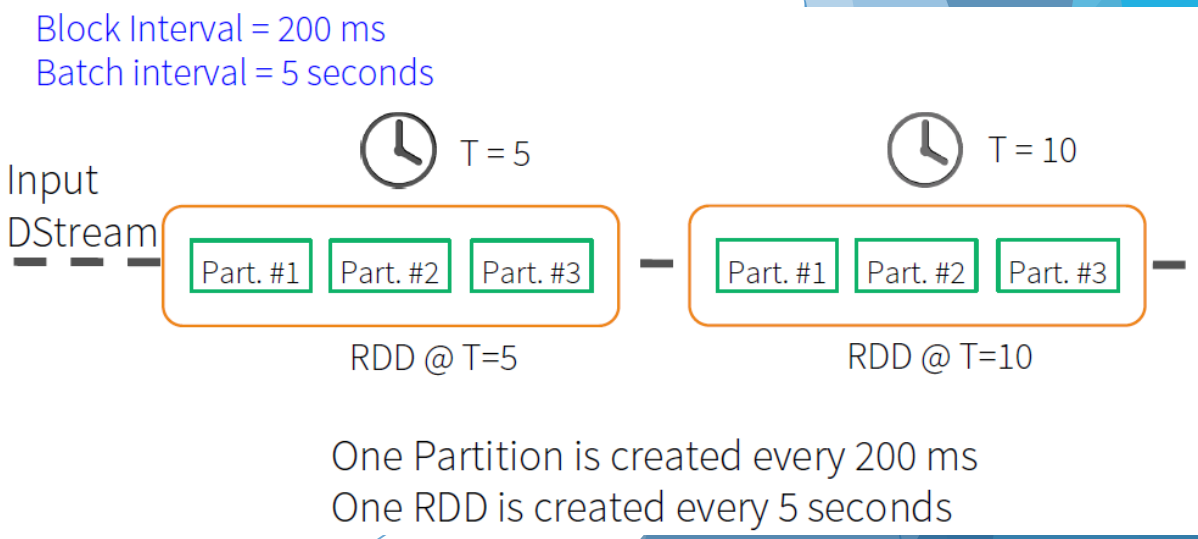
- Created from streaming input sources
- Created by applying transformations on existing DStreams

Block Interval:

- Controls how often we create partition
- 200ms by default

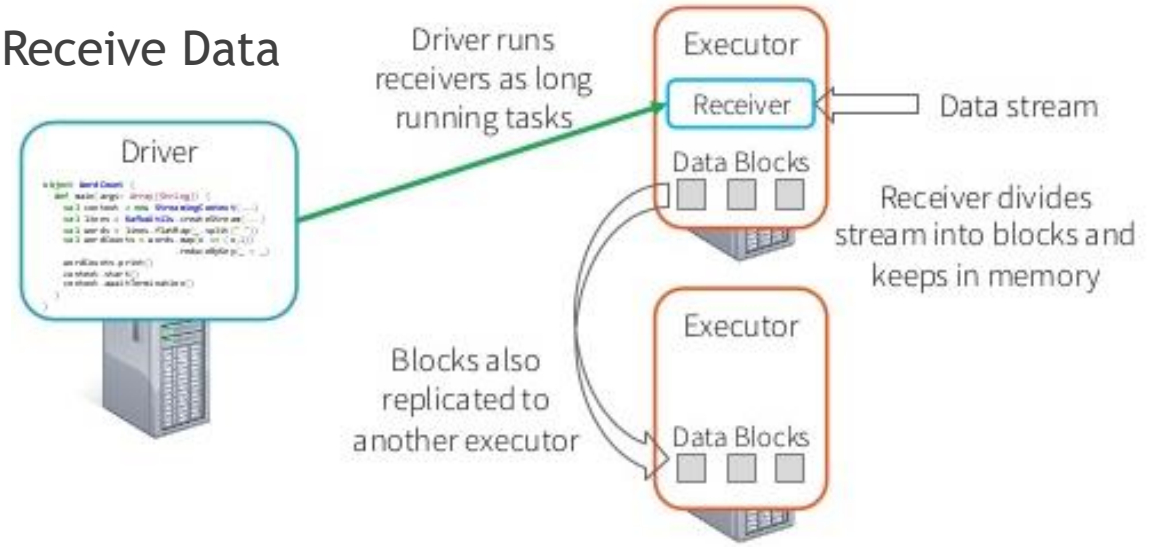
Batch Interval:

- Timer till it needs to be buffered for creating RDD
- It is programmatically controlled while creating Dstream



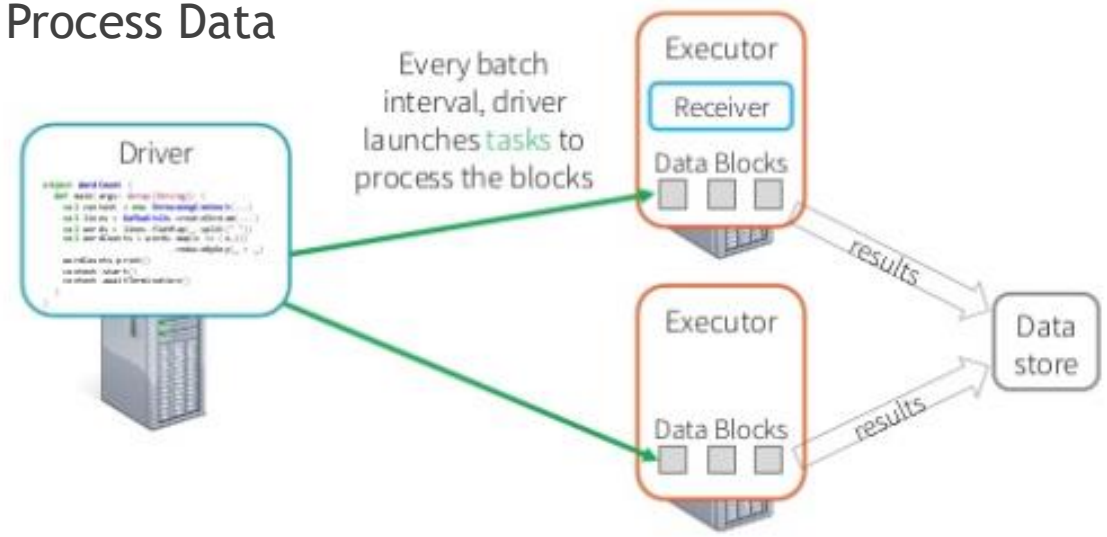
Spark Streaming - Workflow

Receive Data




```
Entry point                                    Batch Interval
val context = new StreamingContext(conf, Seconds(1))

val lines = context.socketTextStream(...)
DStream: represents a data stream

val words = lines.flatMap(_.split(" "))
Transformations: transform data to create new DStreams

val wordCounts = words.map(x => (x, 1)).reduceByKey(_+_)
wordCounts.print()   Print the DStream contents on screen
context.start()      Start the streaming job
```

Process Data



Receiver-based:

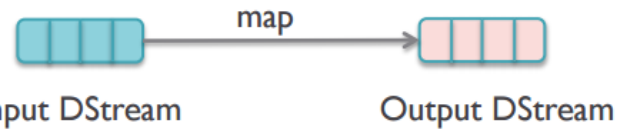
- For sources without a buffer, like TCP sockets
- A long running task called the Receiver is launched in one of the executors

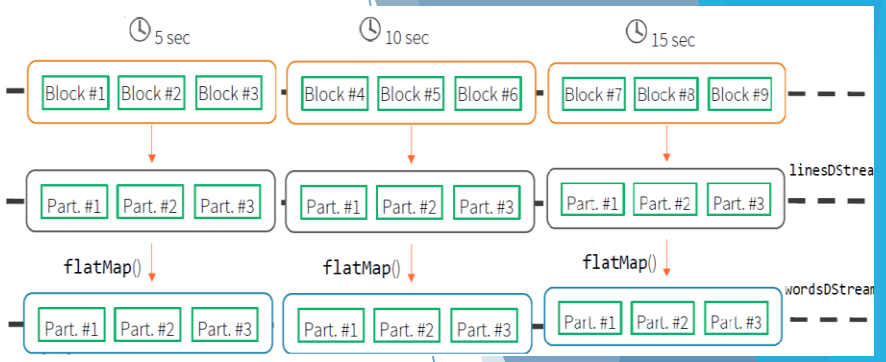
Receiver-less or Direct:

- For sources with a buffer, like Kafka
- Streaming context gets partition info form source and then creates RDD

Transformations

Transformations modify data from one DStream to another





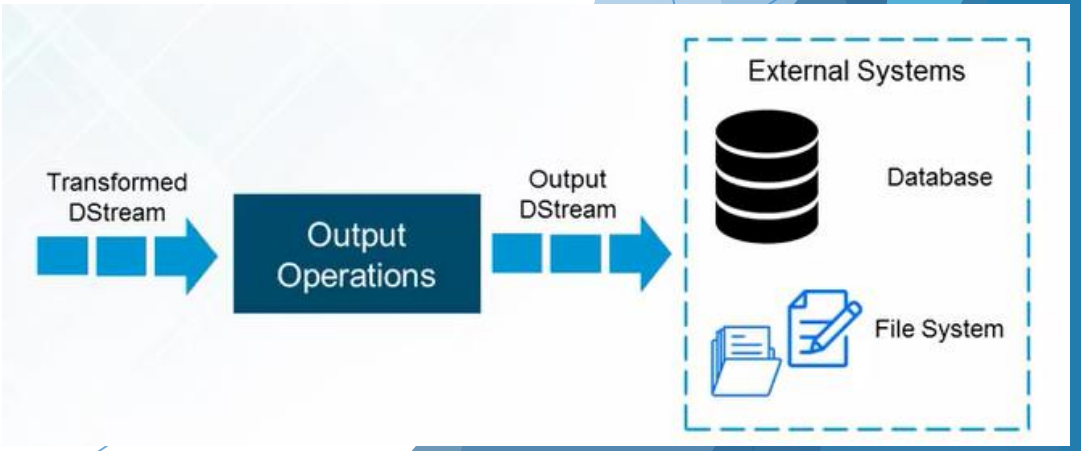
Transformations	Description
• map(func) • flatMap(func) • filter(func) • repartition(numPartitions) • union(otherStream) • count() • reduce(func) • reduceByKey(func, [numTasks]) • countByValue() • join(otherStream, [numTasks]) • cogroup(otherStream, [numTasks])	• All these return new DStream
transform(func)	• It can be used to apply any RDD operation that is not exposed in the DStream API

Output Operations

Actions	Description
• saveAsTextFiles(…) • saveAsObjectFiles(...) • saveAsHadoopFile(...)	• Save DStream's contents as text files/ SequenceFiles /Hadoop files
print()	• Prints the first ten elements of every batch of data in a DStream • This is useful for development and debugging.
foreachRDD(func)	• To call an action that is available in RDD API not in Dstream API

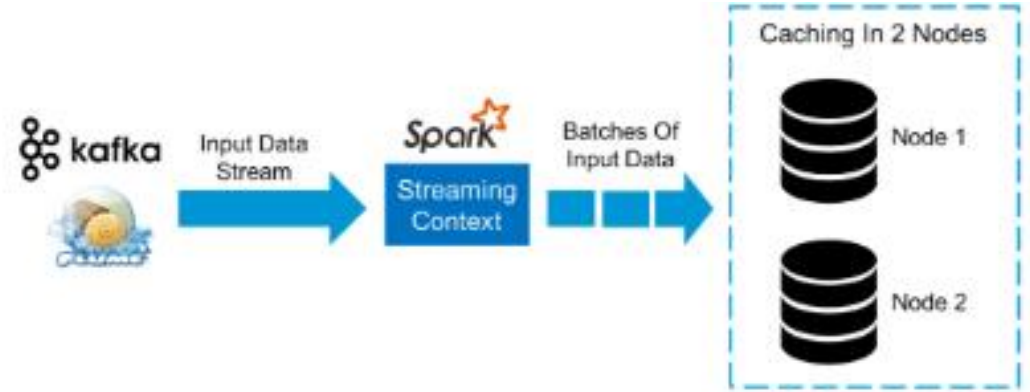
Power Combo = Dstream + RDD

- Combine live data streams with historical data
- Combine streaming with MLlib, GraphX algos
- Query streaming data using SQL



Caching

- DStreams allow developers to cache/ persist the stream's data in memory.
- This is useful if the data in the DStream will be computed multiple times.
- This can be done using the persist() method on a DStream.



Program Structure

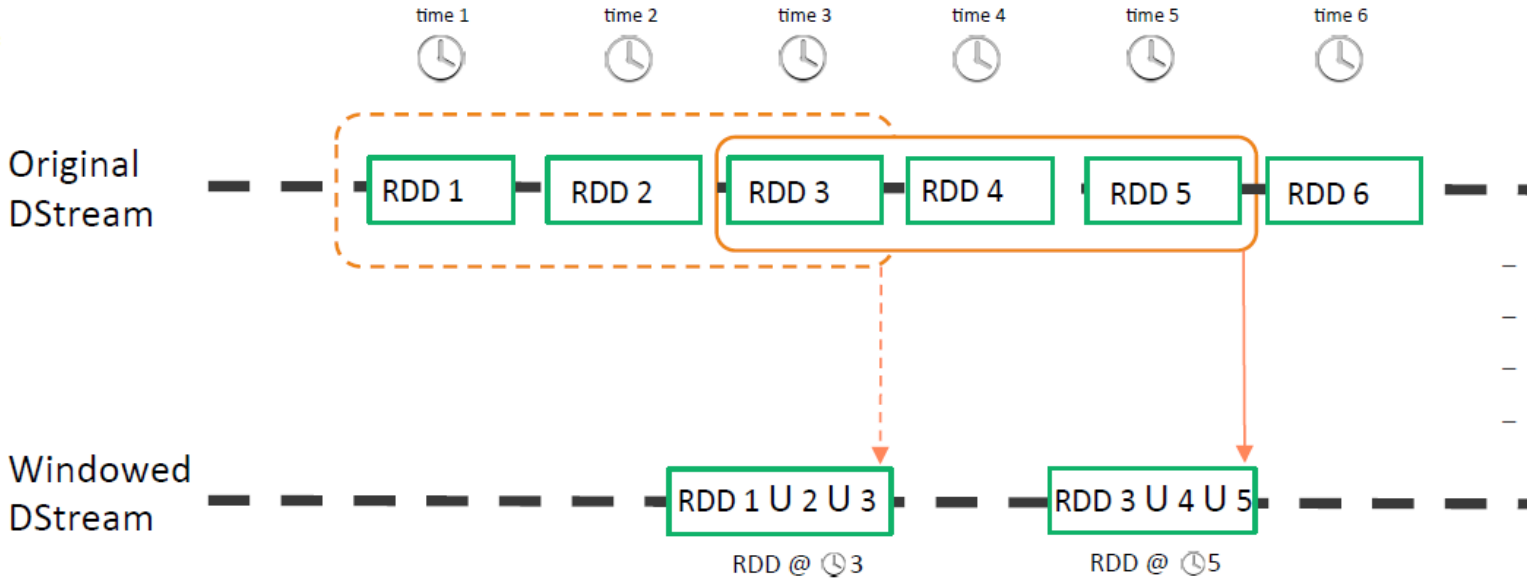
- StreamingContext is the main entry point for Spark Streaming functionality
  - *socketTextStream(host, port)*
  - *fileStream(directory), textFileStream(directory)*
  - *actorStream(actorProps, actorName)*
  - *queueStream(queueOfRDDs)*
- Specify the batchInterval in the program
- Create Dstream object by specifying source
- Construct DStream DAG

- *StreamingContext.start()* to kickoff the streaming process
- *StreamingContext.awaitTermination() to wait for termination*
- *StreamingContext.stop()* to stop the streaming process

- Once the streaming process is started the DAG cannot be changed
- Once streaming context is stopped, streaming context will be garbage collected
- Stopping StreamingContext stops SparkContext as well by default

Window Operations

- Apply transformations over a sliding window of data
- Every time the window slides over a source DStream, the source RDDs that fall within the window are combined and operated to produce the RDDs
- Window length - The duration of the window
- Sliding interval - The interval at which the window will slide or move forward
- Ex: Window Length = 30 Secs ; Sliding Interval = 20 Secs



- *window(windowLength, slideInterval)*
- *countByWindow(windowLength, slideInterval)*
- *reduceByWindow(func, windowLength, slideInterval)*
- *reduceByKeyAndWindow(…)*

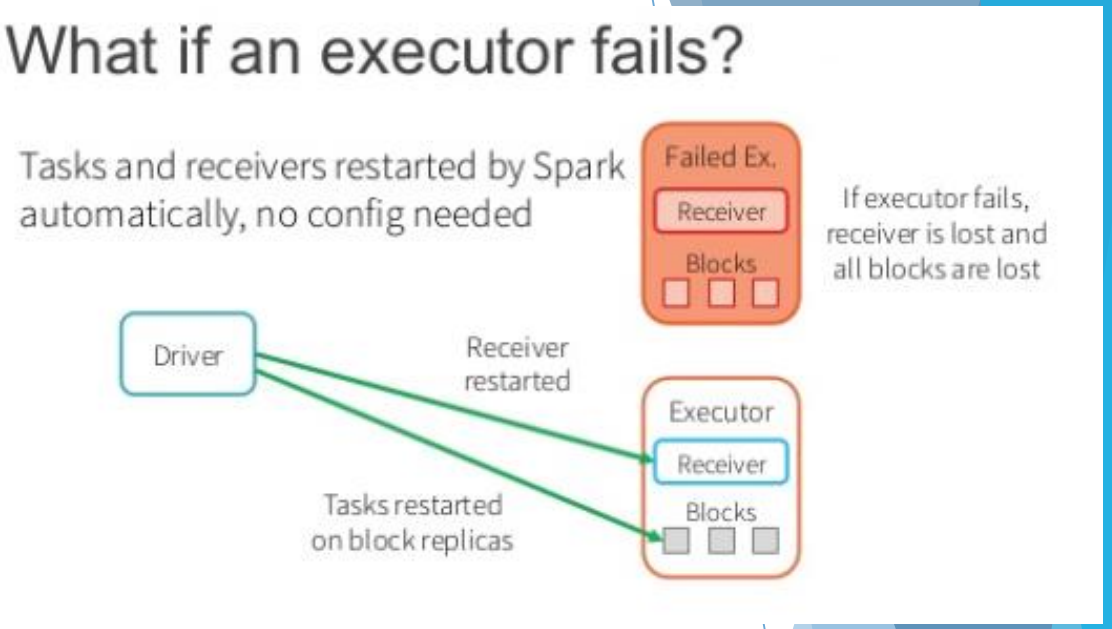
Fault Tolerance

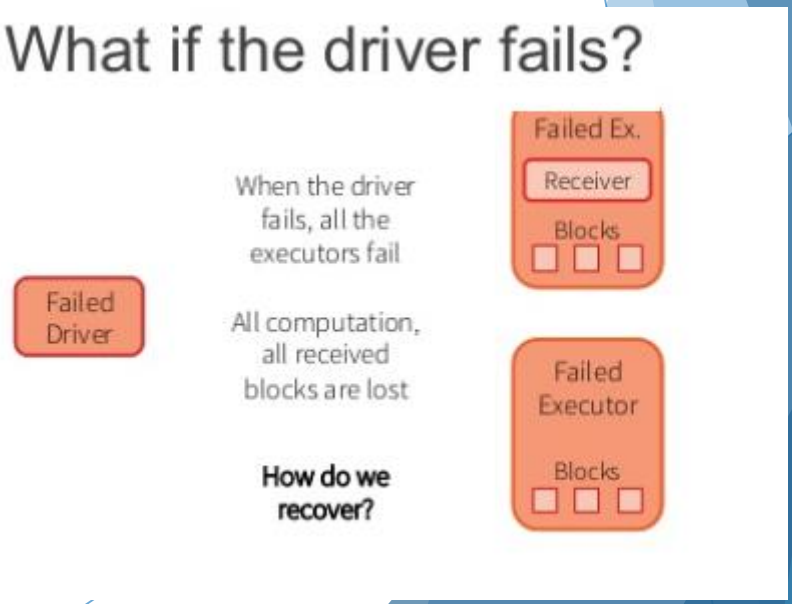
Executor/Receiver failure:

- Batches of input data are replicated in memory for fault-tolerance
- Data lost due to worker failure, can be recomputed from replicated input data
- Data buffered but not yet replicated will be lost. This can be handled by WAL
- WAL – Write Ahead Log
- Synchronously save the received data to a fault tolerant storage

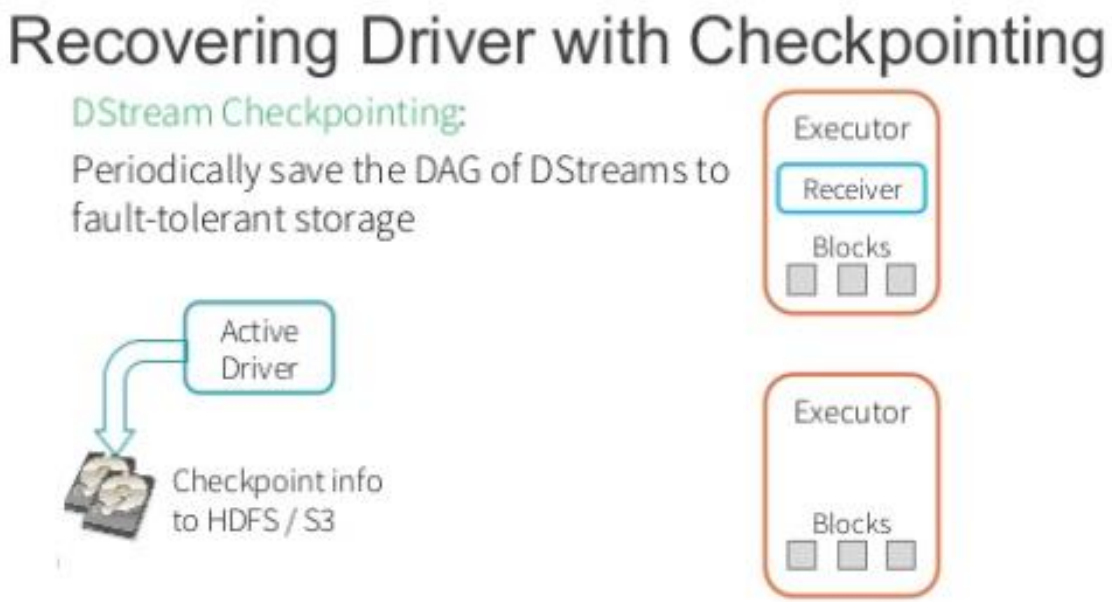
Driver failure:

- Executors are lost including any Receivers they were running

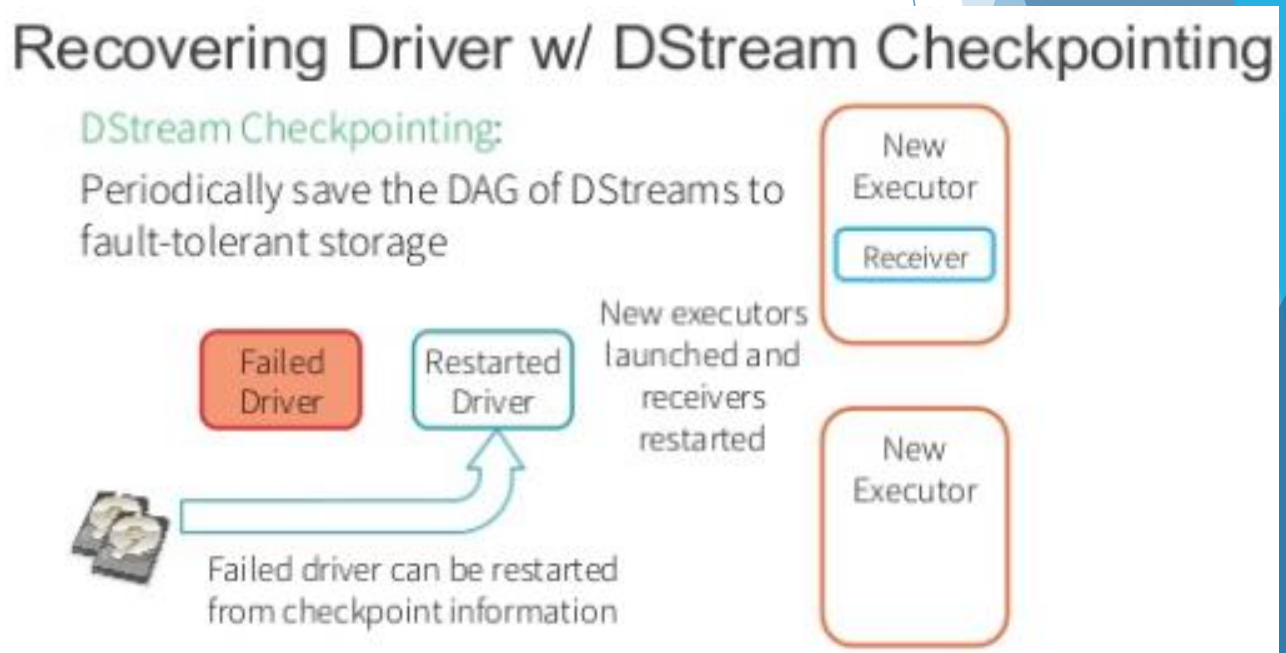




Checkpoint – Driver Failure



Metadata Checkpointing



Checkpointing

- A process of writing received records at checkpoint intervals to HDFS is checkpointing
- Checkpointing creates fault-tolerant stream processing pipelines.
- DStreams can checkpoint input data at specified time intervals.

Metadata Checkpointing:

- Saving of the information defining the streaming computation to fault-tolerant storage like HDFS.
- This is used to recover from failure of the node running the driver of the streaming application

Data Checkpointing:

- Saving of the generated RDDs to reliable storage
- This is necessary in some stateful transformations that combine data across multiple batches.

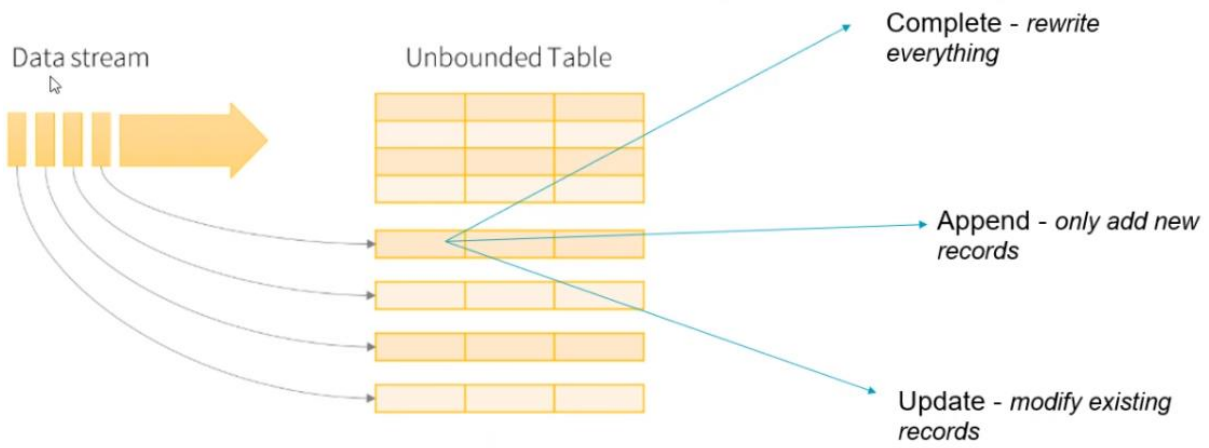
```
val checkpointDir = ...

def creatingFunc(): StreamingContext = {
   val newSsc = ...                          // create and setup a new StreamingContext

   newSsc.checkpoint(checkpointDir)      // enable checkpointing

   ...
}

// Recreate context from checkpoints info in checkpointDir, or create a new one by
val ssc = StreamingContext.getOrCreate(checkpointDir, creatingFunc _)
```

Structured Streaming

Structured Streaming

- Structured Streaming is a stream processing engine built on the Spark SQL engine
- Streaming computation can be done on the same way as that of a batch computation on static data.
- Rich, unified and high level APIs (Dataframes & Dataset)
- No more dealing with RDD directly!
- The biggest difference is latency and message delivery guarantees

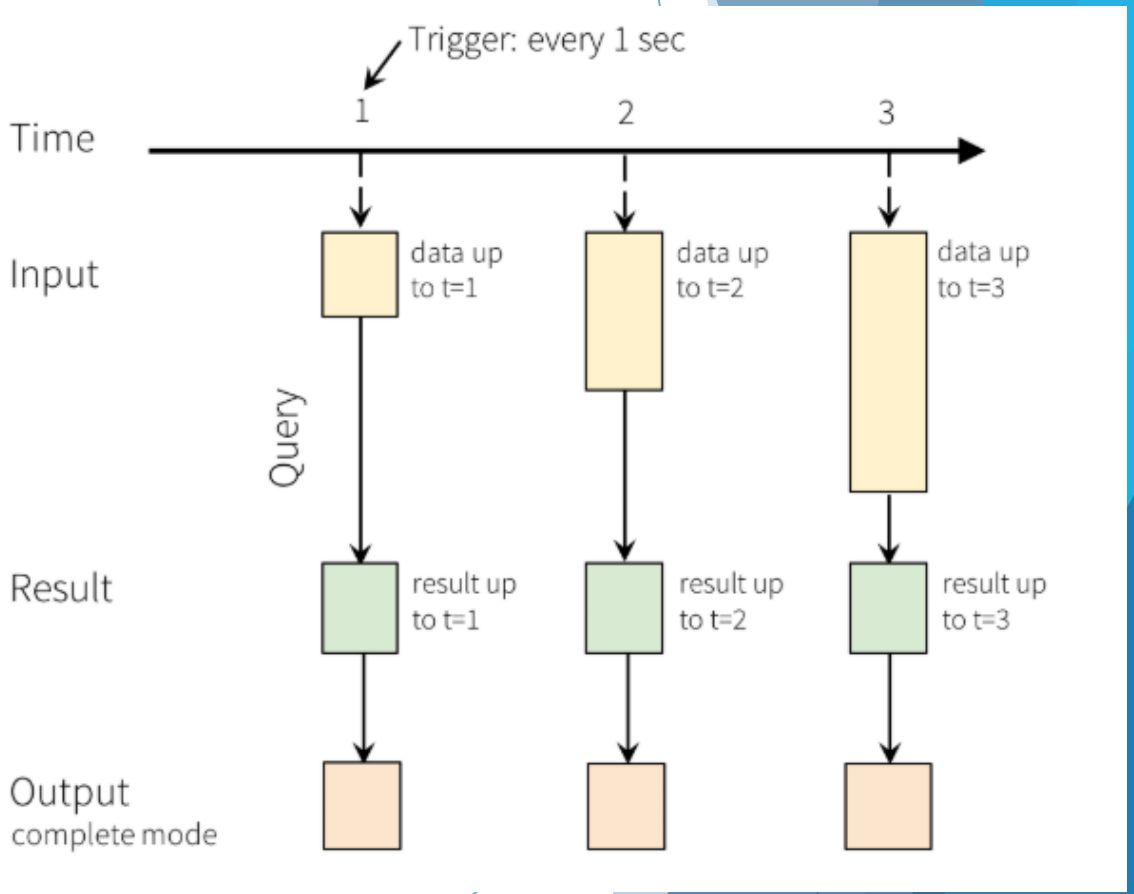
- Treat stream of data as unbounded tables
- Both Structured Streaming and Streaming with DStreams use micro-batching



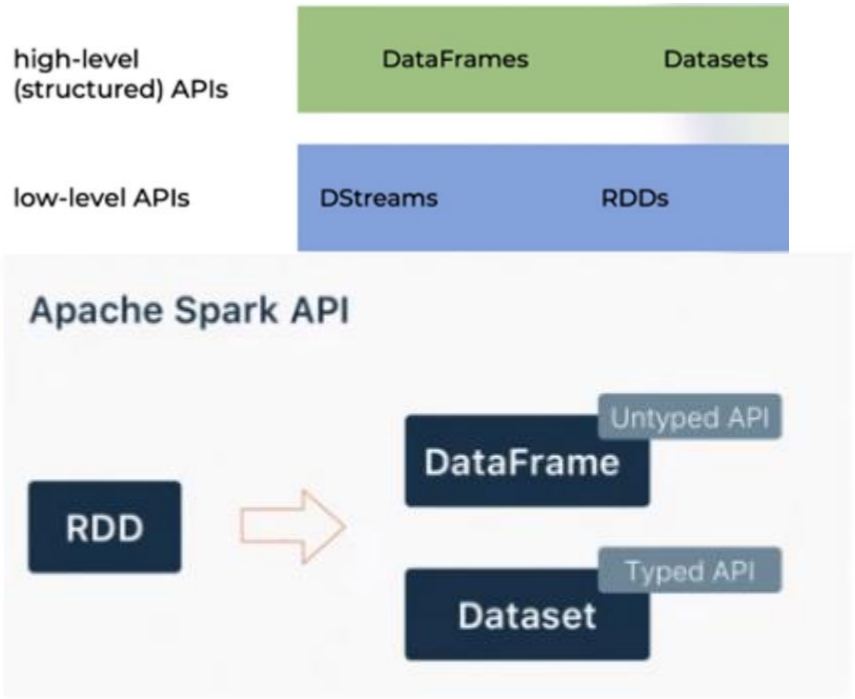
Processing Model

- Structured Streaming treats all the data arriving as an unbounded input table.
- Each new item in the stream is like a row appended to the input table.
- The developer then defines a query on this input table, as if it were a static table
- Spark automatically converts this batch-like query to a streaming execution plan (*incrementalization)*
- A query on the input will generate the “Result Table” which will be written to an output sink

- Developers specify triggers to control when to update the results.
- Each time a trigger fires, Spark checks for new data (new row in the input table), and incrementally updates the result.



RDD API Streaming vs Structured Streaming



Spark Streaming (D Streams) – RDD API

- RDD based Streaming
- Batch time has to be specified inside the application
- Higher latency is batch size is not properly assigned
- Triggers are bound often not customizable and are bound by batch and window time

```
val ssc= new StreamingContext(conf, Seconds(1))
val lines=ssc.socketTextStream("localhost",9999)
val words=lines.flatMap(_.split(" "))
val pairs=words.map(word=>(word,1))
val wordCount=pairs.reduceByKey(_+_)
wordCounts.print()
ssc.start()
```

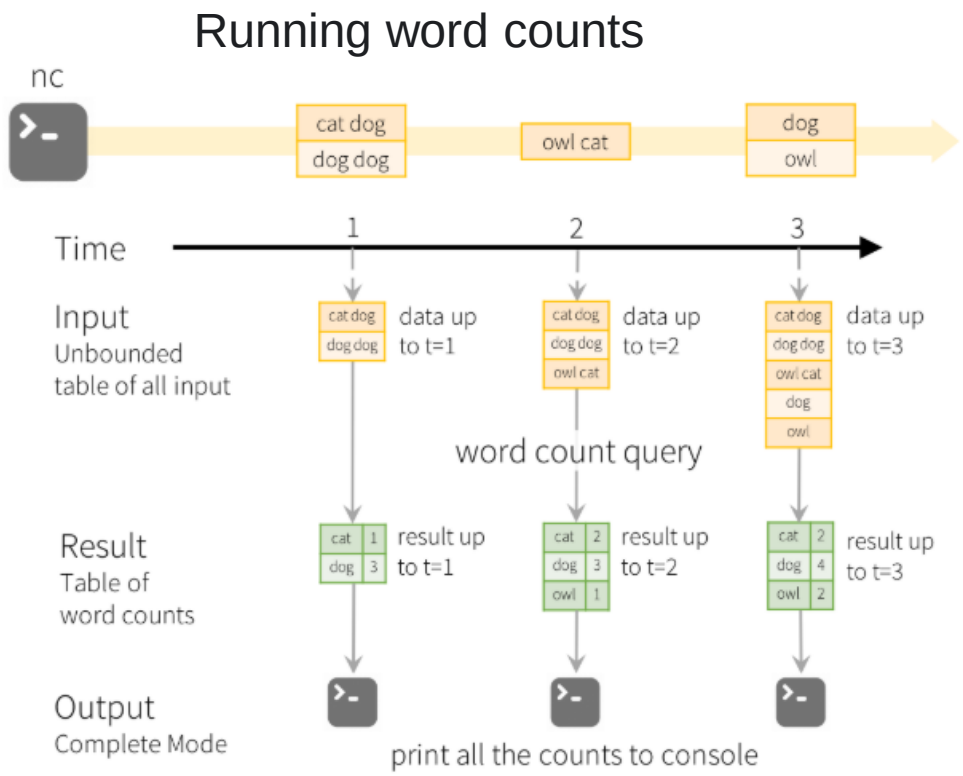
Spark Structured Streaming (DataFrame/Dataset API)

- Dataframe/Dataset based Streaming
- Uses micro-batch
- Low latency achieved through micro-batches
- Highly customizable triggers that can manipulate processing and event times separately

```
val lines=spark.readStream.format("socket").
          option("host","localhost").
          option("port","9999").
          load()
val words=lines.as[String]. flatMap(_.split(" "))
val wordCounts=words.groupby("value").count()
val query=wordCounts.writeStream.
          outputMode("complete").
          format("console").
          start()
```

Output Modes

- **Append:** Only the new rows appended to the result table since the last trigger will be written to the external storage.
- **Complete**: The entire updated result table will be written to external storage
- **Update**: Only the rows that were updated in the result table since the last trigger will be changed in the external storage.

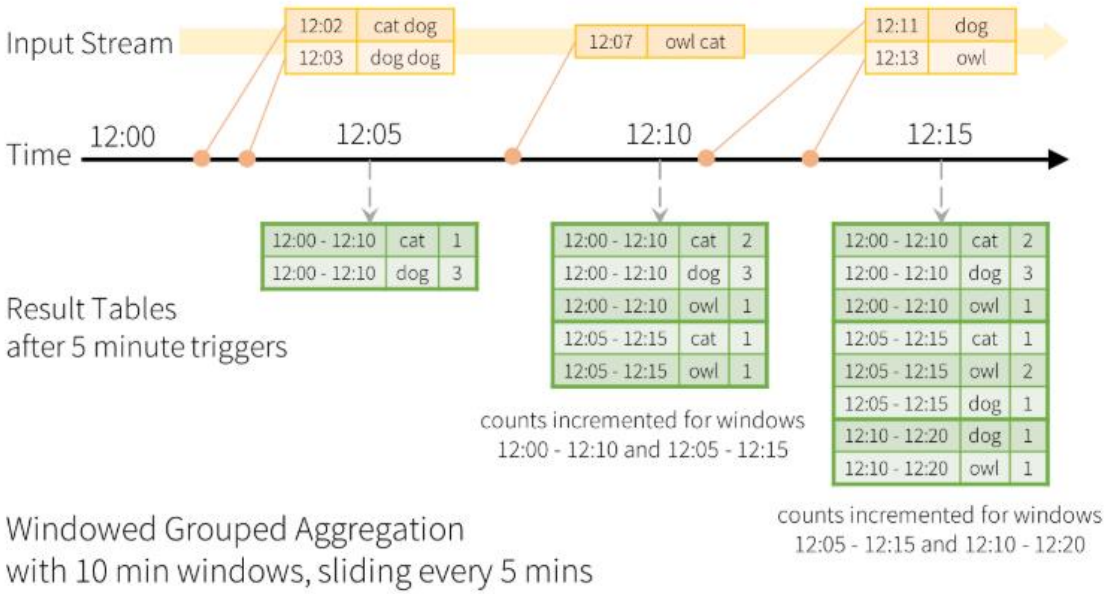



```scala
val lines = spark.readStream
  .format("socket")
  .option("host", "localhost")
  .option("port", 9999)
  .load()

// Split the lines into words
val words = lines.as[String].flatMap(_.split(" "))

// Generate running word count
val wordCounts = words.groupBy("value").count()
```

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

Window Operations

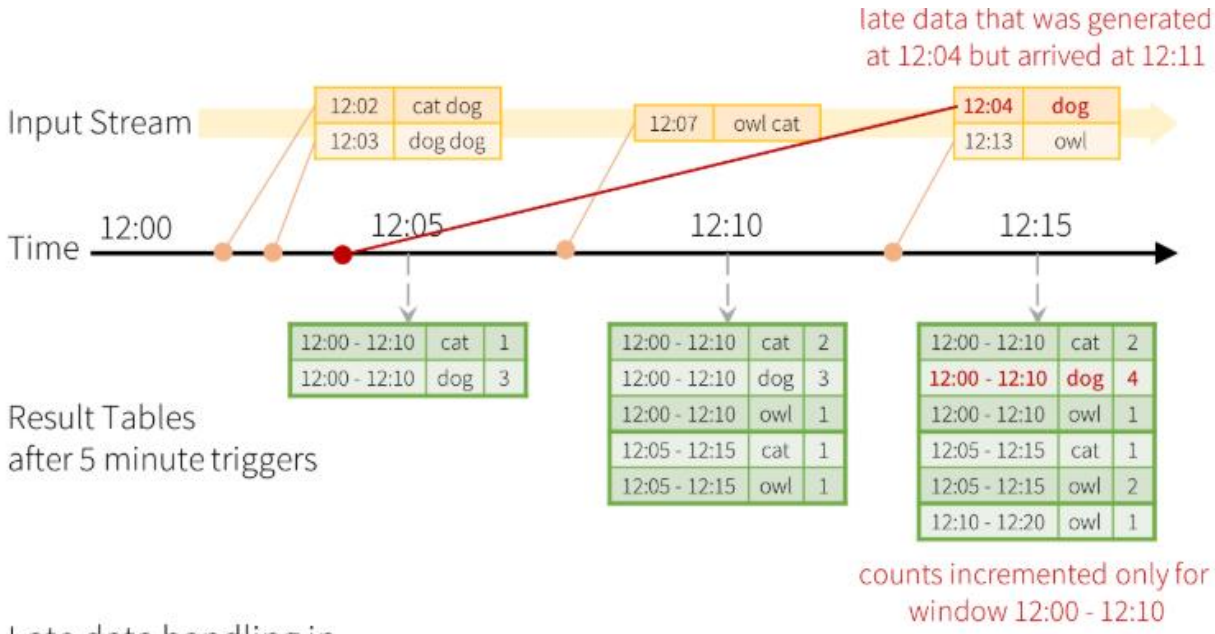
- Aggregations over a sliding event-time window are straightforward with Structured Streaming
- We want to count words within 10 minute windows *(similar to window length)*, updating every 5 minutes *(similar to sliding interval)*
- That is, word counts in words received between 10 minute windows 12:00 - 12:10, 12:05 - 12:15, 12:10 - 12:20, etc
- consider a word that was received at 12:07. This word should increment the counts corresponding to two windows 12:00 - 12:10 and 12:05 - 12:15




```scala
import spark.implicits._

val words = ... // streaming DataFrame of schema { timestamp: Timestamp, word: String }

// Group the data by window and word and compute the count of each group
val windowedCounts = words.groupBy(
  window($"timestamp", "10 minutes", "5 minutes"),
  $"word"
).count()
```

Late Data and Watermarking

- Now consider what happens if one of the events arrives late to the application. For example, say, a word generated at 12:04 (i.e. event time) could be received by the application at 12:11
- The application should use the time 12:04 instead of 12:11 to update the older counts for the window 12:00 - 12:10
- Structured Streaming can maintain the intermediate state for partial aggregates for a long period of time such that late data can update aggregates of old windows correctly



Late data handling in Windowed Grouped Aggregation

```scala
import spark.implicits._

val words = ... // streaming DataFrame of schema { timestamp: Timestamp, word: String }

// Group the data by window and word and compute the count of each group
val windowedCounts = words
    .withWatermark("timestamp", "10 minutes")
    .groupBy(
        window($"timestamp", "10 minutes", "5 minutes"),
        $"word")
    .count()
```

Output Sinks

Sink	Supported Output Modes	Options
File Sink	Append	`path`: path to the output directory, must be specified. For file-format-specific options, see the related methods in DataFrameWriter (Scala/Java/Python/R). E.g. for "parquet" format options see `DataFrameWriter.parquet()`
Kafka Sink	Append, Update, Complete	See the Kafka Integration Guide
Foreach Sink	Append, Update, Complete	None
ForeachBatch Sink	Append, Update, Complete	None
Console Sink	Append, Update, Complete	`numRows`: Number of rows to print every trigger (default: 20) `truncate`: Whether to truncate the output if too long (default: true)
Memory Sink	Append, Complete	None

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

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

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

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

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

Once

- When you set the trigger option to “Once” it processes only once and then terminates the stream.

Processing Time

- This is the most widely used and recommended practice in Spark Structured Streaming.
- The trigger option of processing time gives you better control over how often micro batch jobs should get triggered
- For example, if you set a trigger interval of “20 seconds” then for every 20 seconds a micro batch will process a batch of records

Continuous

- As of Spark 2.4.0, it’s still experimental. It enables you to process records in milliseconds which would otherwise take seconds in micro batch.
- Long-running task is created per write stream and processed as quickly as possible

```scala
import org.apache.spark.sql.streaming.Trigger

// Default trigger (runs micro-batch as soon as it can)
df.writeStream
  .format("console")
  .start()

// ProcessingTime trigger with two-seconds micro-batch interval
df.writeStream
  .format("console")
  .trigger(Trigger.ProcessingTime("2 seconds"))
  .start()

// One-time trigger
df.writeStream
  .format("console")
  .trigger(Trigger.Once())
  .start()
```

```scala
spark.readStream.format("kafka")
 .option("kafka.boostrap.servers",...)
.option("subscribe", "topic")
.load()
. selectExpr("cast (value as string) as json")
.select(from_json("json", schema).as("data"))
.writeStream .format("parquet")
.option("path", "/parquetTable/")
.trigger("1 second")
.option("checkpointLocation", "...")
.start()
```

Trigger: when to process data - Fixed interval micro-batches - As fast as possible micro-batches - Continuously
Checkpoint location: for tracking

Managing Streaming Queries

- The StreamingQuery object created when a query is started can be used to monitor and manage the query.

```
val query = df.writeStream.format("console").start()   // get the query object

query.id          // get the unique identifier of the running query that persists across restarts from checkpoin
t data

query.runId       // get the unique id of this run of the query, which will be generated at every start/restart

query.name        // get the name of the auto-generated or user-specified name

query.explain()   // print detailed explanations of the query

query.stop()      // stop the query

query.awaitTermination()   // block until query is terminated, with stop() or with error

query.exception       // the exception if the query has been terminated with error

query.recentProgress  // an array of the most recent progress updates for this query

query.lastProgress    // the most recent progress update of this streaming query
```

Reference

- Spark Streaming Guide

https://spark.apache.org/docs/0.9.2/streaming-programming-guide.html

- Streaming Example – Kafka

http://spark.apache.org/docs/latest/streaming-kafka-integration.html

- Structured Streaming

https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

Thank You

Keerthiga Barathan