Open | DSE

Architecting Big Data
Solutions with Apache Spark
Lecture 5: Spark Streaming vs Structured

Ekhtiar Syed

Data Engineer/Scientist

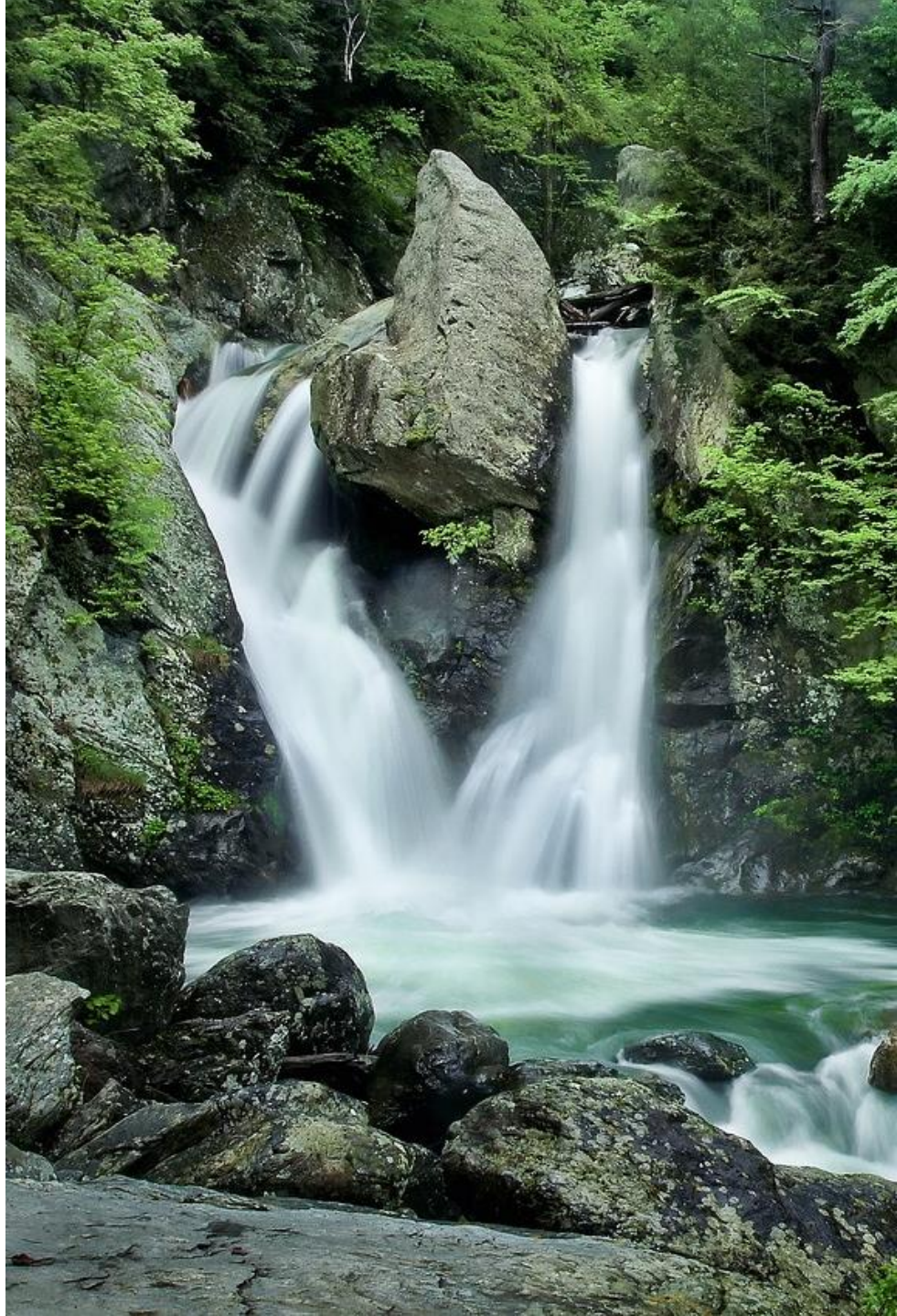
Signify Research (formerly known as Philips Lighting)

Streaming With Spark

> Spark Streaming operates in micro-batch mode.

> You have two options / module to choose from:
>> > Option 1: RDD based DStreams
>> > Option 2: DataFrame based Structured-Streaming

> In this lecture, we will dive into both of their under the hood implementation / architecture.



Spark Streaming Programming Guide

- Extension of the core Spark API
- Enables scalable, high-throughput, fault-tolerant stream processing of live data streams
- Data can be ingested from many sources like Kafka, Flume, Kinesis, or TCP sockets
- You can use all of the complex processing capabilities of RDD (map, reduce, join and window)
- Finally, processed data can be pushed out to filesystems, databases, and live dashboards
- You can also apply Spark's machine learning and graph processing algorithms on data streams

Spark Streaming Architecture

Spark Streaming Programming Guide

- Internally, DStreams receive live input data streams and divides the data into batches, which are then processed by the Spark's engine to generate the final stream of results in batches
- Spark Streaming provides a high-level abstraction called discretized stream or DStream, which represents a continuous stream of data
- DStreams can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying high-level operations on other Dstreams
- Internally, a DStream is represented as a sequence of RDDs
- You can write DStreaming programs in Scala, Java or Python

Spark Streaming Data Flow

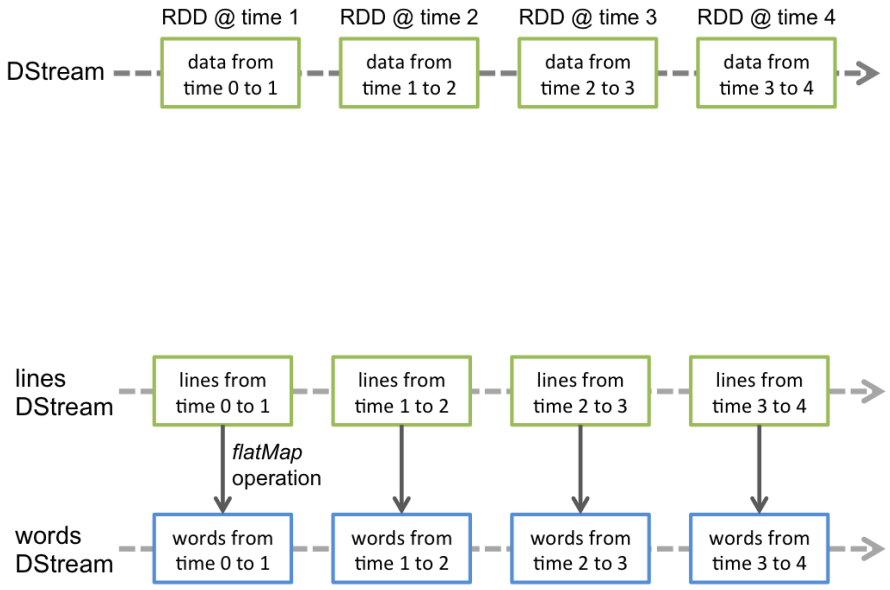
Quick Example

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

# Create a local StreamingContext with batch interval of 1s
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 1)

# Create a DStream that will connect to hostname:port
lines = ssc.socketTextStream("localhost", 9999)

# Split each line into words
words = lines.flatMap(lambda line: line.split(" "))

# Count each word in each batch
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)

# Print the first ten elements of each RDD generated
wordCounts.pprint()
```

Spark Streaming Data Flow

Discretized Streams (DStreams)

- Discretized Stream or DStream is the basic abstraction provided by Spark Streaming
- Each RDD in a DStream contains data from a certain interval, as shown in the following figure
- Any operation applied on a DStream translates to operations on the underlying RDDs
- Lets imagine we are listening to a stream of news titles and doing a word count using a flatMap operation
- The flatMap operation is applied on each RDD in the lines DStream to generate the RDDs of the words DStream



Spark Streaming Data Flow

Transform Operation

The transform operation (along with its variations like transformWith) allows arbitrary RDD-to-RDD functions to be applied on a Dstream:

```
# RDD containing spam information
spamInfoRDD = sc.pickleFile(...)

# join data stream with spam information to do data cleaning
cleanedDStream = wordCounts.transform(lambda rdd: rdd.join(spamInfoRDD).filter(...))
```

Window Operation

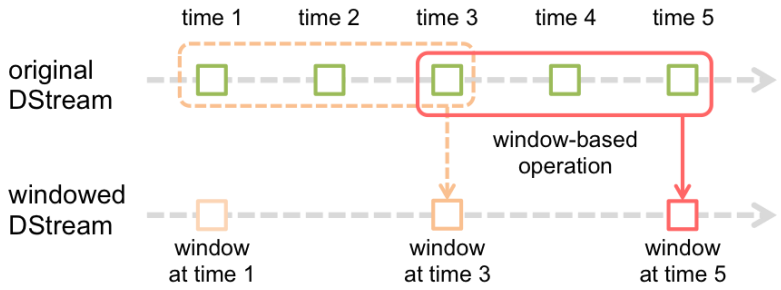
Spark Streaming also provides *windowed computations*, which allow you to apply transformations over a sliding window of data.

As shown in the figure, every time the window *slides* over a source DStream, the source RDDs that fall within the window are combined and operated upon to produce the RDDs of the windowed DStream.

In this specific case, the operation is applied over the last 3 time units of data, and slides by 2 time units. This shows that any window operation needs to specify two parameters.

- *window length* - The duration of the window (3 in the figure).
- *sliding interval* - The interval at which the window operation is performed (2 in the figure).

These two parameters must be multiples of the batch interval of the source DStream (1 in the figure).



Spark Streaming Window Operation

Window Operations

Transformation	Meaning
window(windowLength, slideInterval)	Perform windowed batches of the source DStream
countByWindow(windowLength, slideInterval)	Return a sliding window count of elements
reduceByWindow(func, windowLength, slideInterval)	Reduce by window (i.e. sum, count)
reduceByKeyAndWindow(func, windowLength, slideInterval, [numTasks])	Reduce by key, per window. (lambda v1, v2: v1 + v2)
reduceByKeyAndWindow(func, invFunc, windowLength, slideInterval, [numTasks])	A more efficient version of the above reduceByKeyAndWindow()
countByValueAndWindow(windowLength,slideInterval, [numTasks])	When called on a DStream of (K, V) pairs, returns a new DStream of (K, Long) pairs where the value of each key is its frequency within a sliding window.

Checkpointing

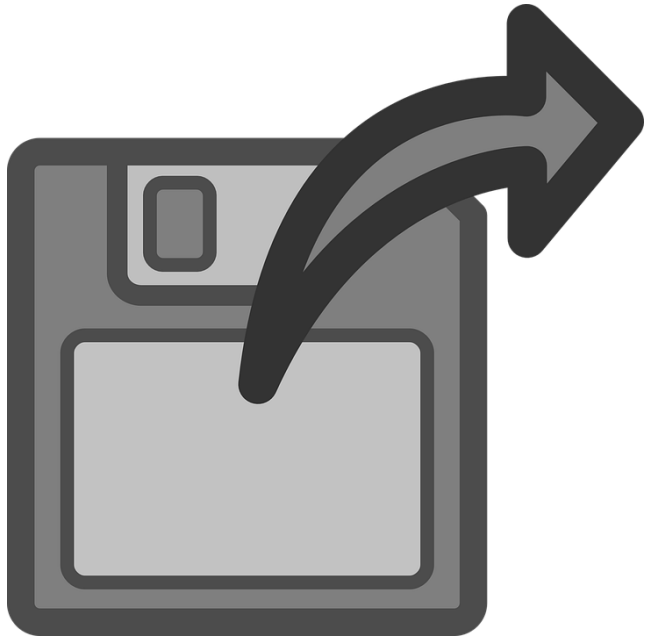
A streaming application must operate 24/7 and hence must be resilient to failures unrelated to the application logic i.e. system failures, JVM crash..

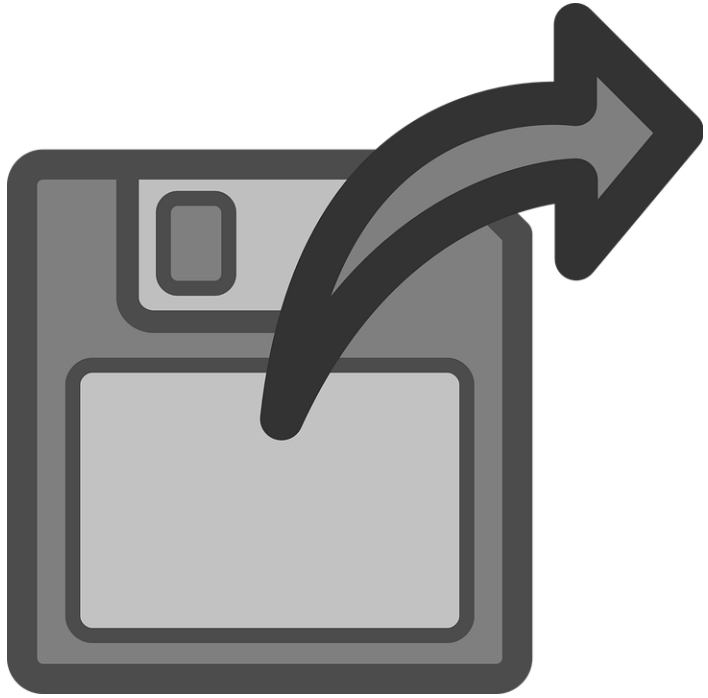
For this to be possible, Spark Streaming needs to *checkpoint* enough information to a fault- tolerant storage system such that it can recover from failures

There are two types of data that are checkpointed:

Metadata checkpointing: Saving of the information defining the streaming computation to fault-tolerant storage. It is primarily needed for recovery from driver failures.

Data checkpointing: Saving of the generated RDDs to reliable storage. It is necessary even for basic functioning if stateful transformations are used.





When to enable Checkpointing

Simple streaming applications without stateful transformations can be run without enabling checkpointing. However, checkpointing must be enabled for applications with any of the following requirements:

Usage of stateful transformations: If either updateStateByKey or reduceByKeyAndWindow (with inverse function) is used in the application, then the checkpoint directory must be provided to allow for periodic RDD checkpointing.

***Recovering from failures of the driver running the application*:** Metadata checkpoints are used to recover with progress information.

Need For A New Approach

Providing end-to-end reliability and correctness guarantees: Long running data processing systems must be resilient to failures by ensuring that outputs are consistent with results processed in batch. Additionally, unusual activities (e.g failures in upstream components, traffic spikes, etc.) must be continuously monitored and automatically mitigated to ensure highly available insights are delivered in real-time.

Performing complex transformations – Data arrives in a myriad formats (CSV, JSON, Avro, etc.) that often must be restructured, transformed and augmented before being consumed. Such restructuring requires that all the traditional tools from batch processing systems are available, but without the added latencies that they typically entail.

Handling late or out-of-order data – When dealing with the physical world, data arriving late or out-of-order is a fact of life. As a result, aggregations and other complex computations must be continuously (and accurately) revised as new information arrives.

Integrating with other systems – Information originates from a variety of sources (Kafka, HDFS, S3, etc), which must be integrated to see the complete picture.

Structured Streaming

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

You can express your streaming computation the same way you would express 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.

Finally, 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.*

Structured Streaming

Internally, by default, 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 as low as 100 milliseconds and exactly-once fault-tolerance guarantees.

However, since Spark 2.3, we have 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.

Next, we are going to walk you through the programming model and the APIs. We are going to explain the concepts mostly using the default micro-batch processing model, and then finally discuss Continuous Processing model.

First, let's start with a simple example of a Structured Streaming query - a streaming word count.

Structured Streaming

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

spark = SparkSession.builder.appName(“WordCount").getOrCreate()

# Create DataFrame representing the stream of input lines from localhost
# lines DataFrame represents an unbounded table
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()
```

```
# TERMINAL 2: RUNNING structured_network_wordcount.py

$ spark-submit wordcount.py localhost 9999

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

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

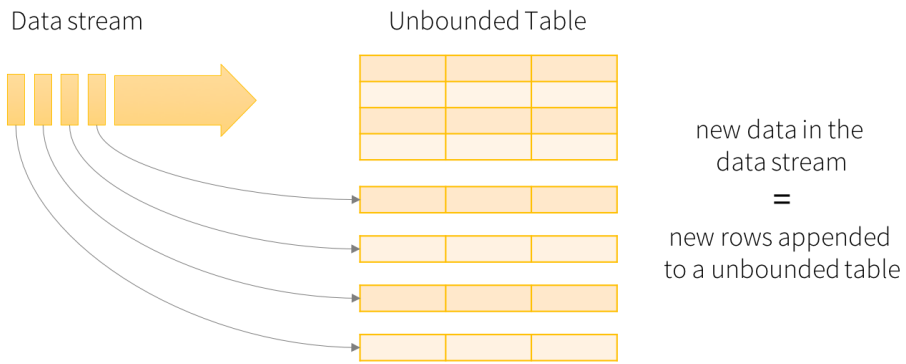
Programming Model

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.

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

Programming Model

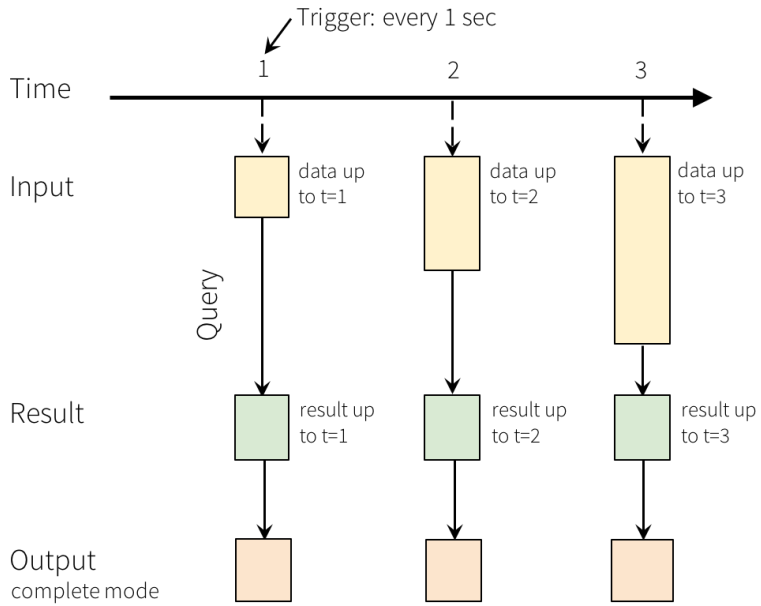
A query on the input will generate the “Result Table”. Every trigger interval (say, every 1 second), new rows get appended to the Input Table, which eventually updates the Result Table. Whenever the result table gets updated, we would want to write the changed result rows to an external sink.

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 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. If the query doesn’t contain aggregations, it will be equivalent to Append mode.



Programming Model for Structured Streaming

Programming Model

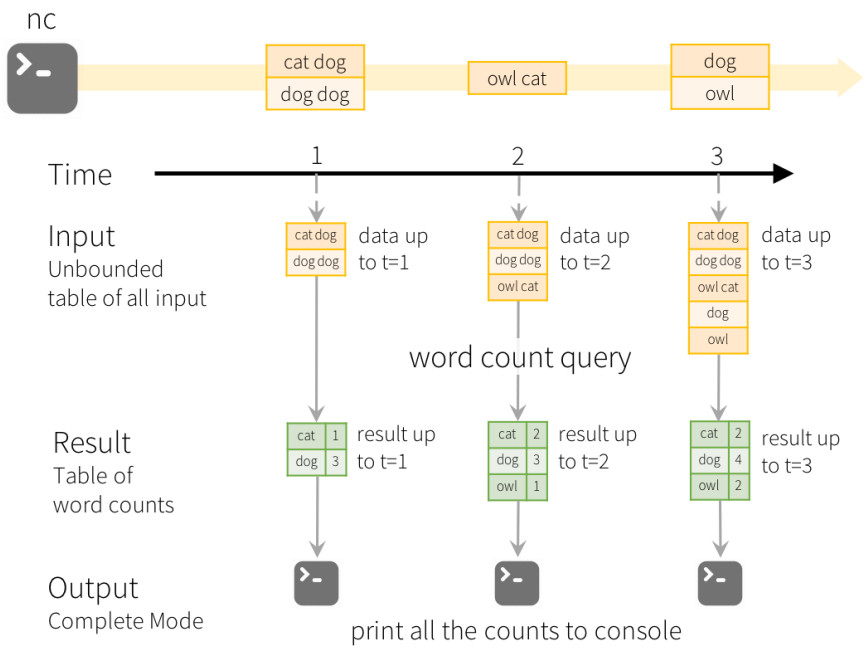
To illustrate the use of this model, let's understand the model in context of the Quick Example above.

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 in the figure.



Model of the Quick Example

A Final Note

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.

As an example, let’s see how this model handles event-time based processing and late arriving data.

Handling Event-time and Late Data

Event-time is the time embedded in the data itself.

For many applications, you may want to operate on this event-time.

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.

Handling Event-time and Late Data

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.

Furthermore, Spark supports **watermarking**, which allows the user to specify the threshold of late data, and allows the engine to accordingly clean up old state.

Fault Tolerance Semantics

Delivering end-to-end exactly-once semantics was one of key goals behind the design of Structured Streaming.

To achieve that, we have designed the Structured Streaming sources, the sinks and the execution engine to reliably track the exact progress of the processing so that it can handle any kind of failure by restarting and/or reprocessing.

Every streaming source is assumed to have offsets (similar to Kafka offsets, or Kinesis sequence numbers) to track the read position in the stream.

The engine uses checkpointing and write ahead logs to record the offset range of the data being processed in each trigger.

Write Ahead Logs are used in database and file systems to ensure the durability of any data operations. Before an operation is applied to the data, first it is written down into a durable log. If the system fails in the middle of applying the operation, it can recover by reading the log and reapplying the operations it had intended to do.

The streaming sinks are designed to be idempotent for handling reprocessing.

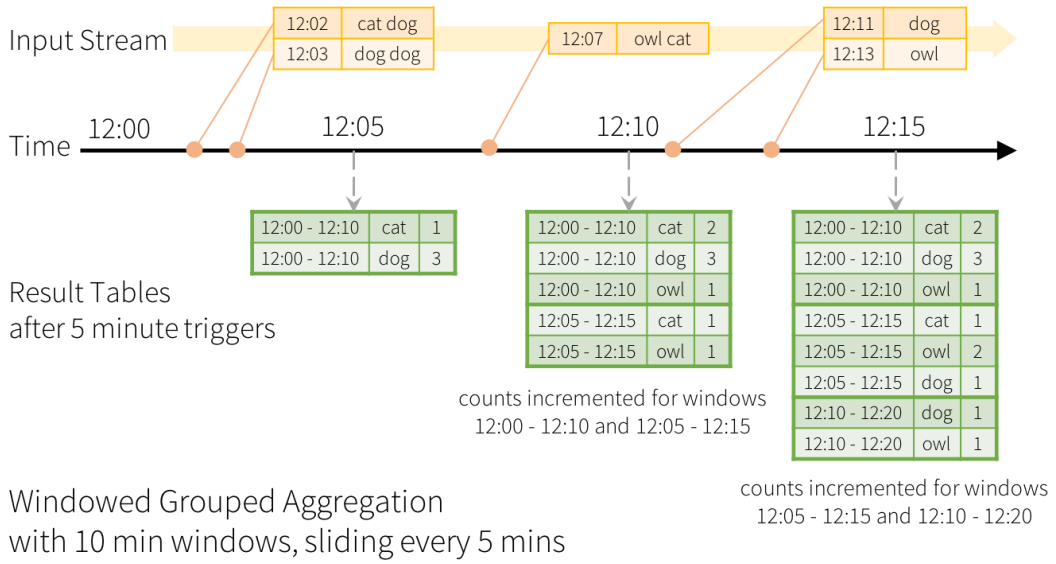
Together, using replayable sources and idempotent sinks, Structured Streaming can ensure **end-to-end exactly-once semantics** under any failure.

Window Operations on Event Time

Aggregations over a sliding event-time window are straightforward with Structured Streaming and are 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.

Similar to our bitcoin window example, if we modify our wordcount example to be windowed over 10 mins window, updating every 5 minutes, it would aggregate the result for every window (as shown in the figure).



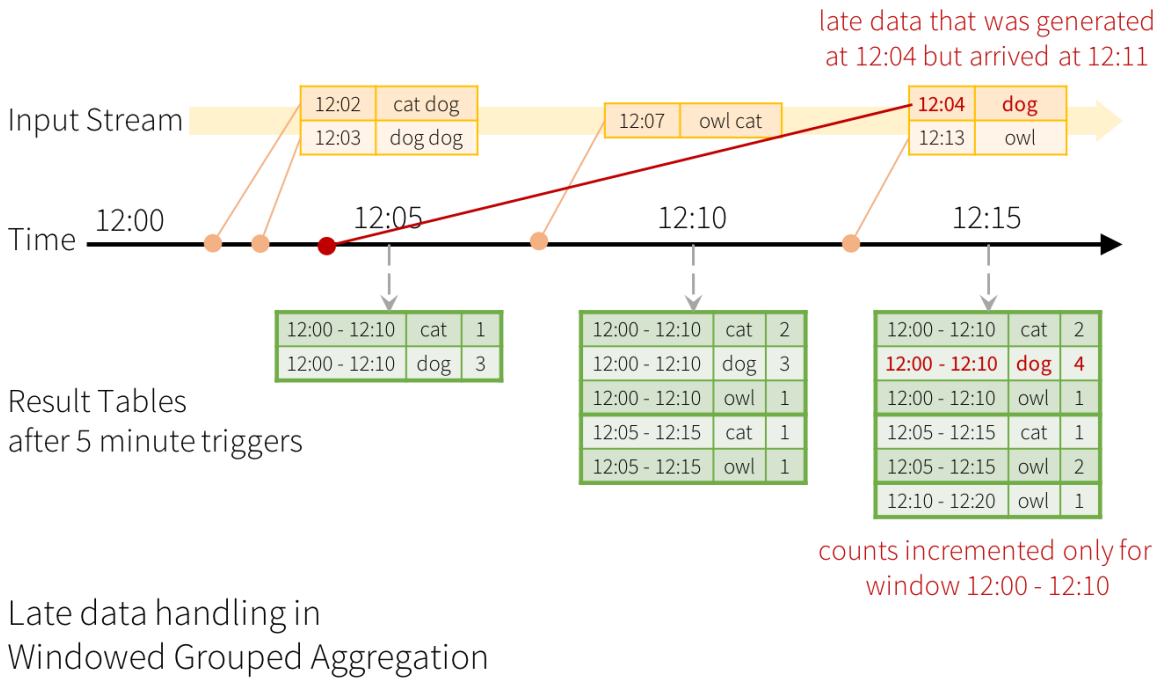
Handling Late Data and Watermarking

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.

This occurs naturally in our window-based grouping – 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, as illustrated.



Handling Late Data and Watermarking

However, to run this query for days, it's necessary for the system to bound the amount of intermediate in-memory state it accumulates.

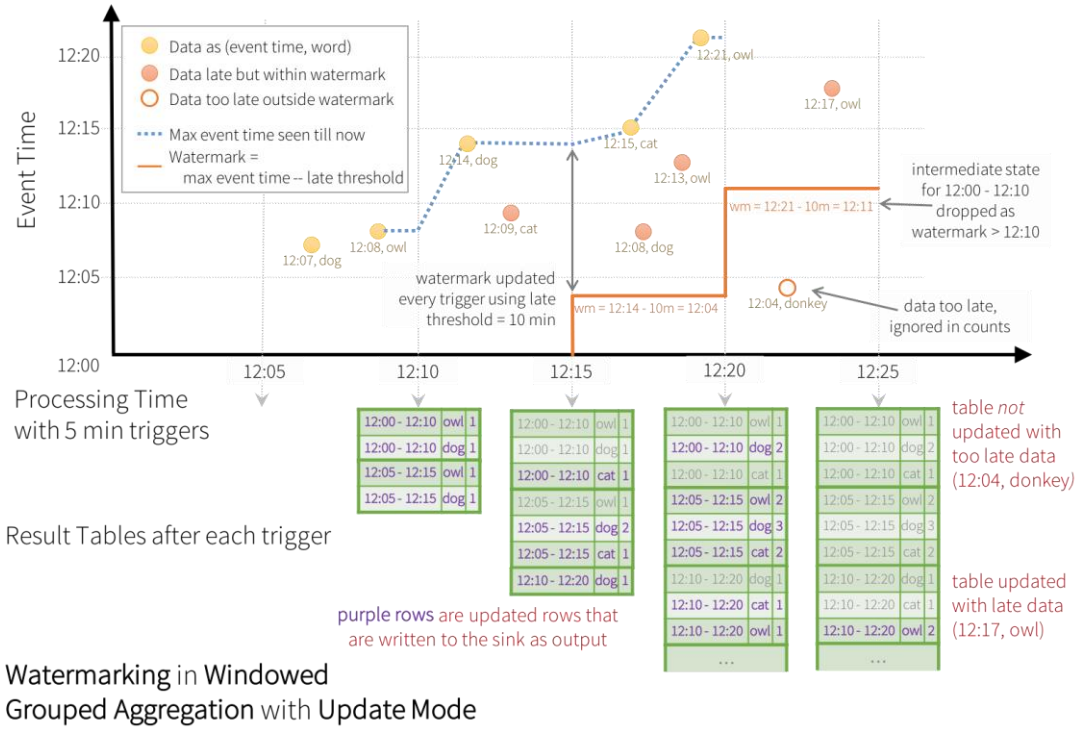
This means the system needs to know when an old aggregate can be dropped from the in-memory state because the application is not going to receive late data for that aggregate any more.

To enable this, in Spark 2.1, we have introduced watermarking, which lets the engine automatically track the current event time in the data and attempt to clean up old state accordingly.

You can define the watermark of a query by specifying the event time column and the threshold on how late the data is expected to be in terms of event time.

For a specific window starting at time T, the engine will maintain state and allow late data to update the state until (max event time seen by the engine - late threshold > T).

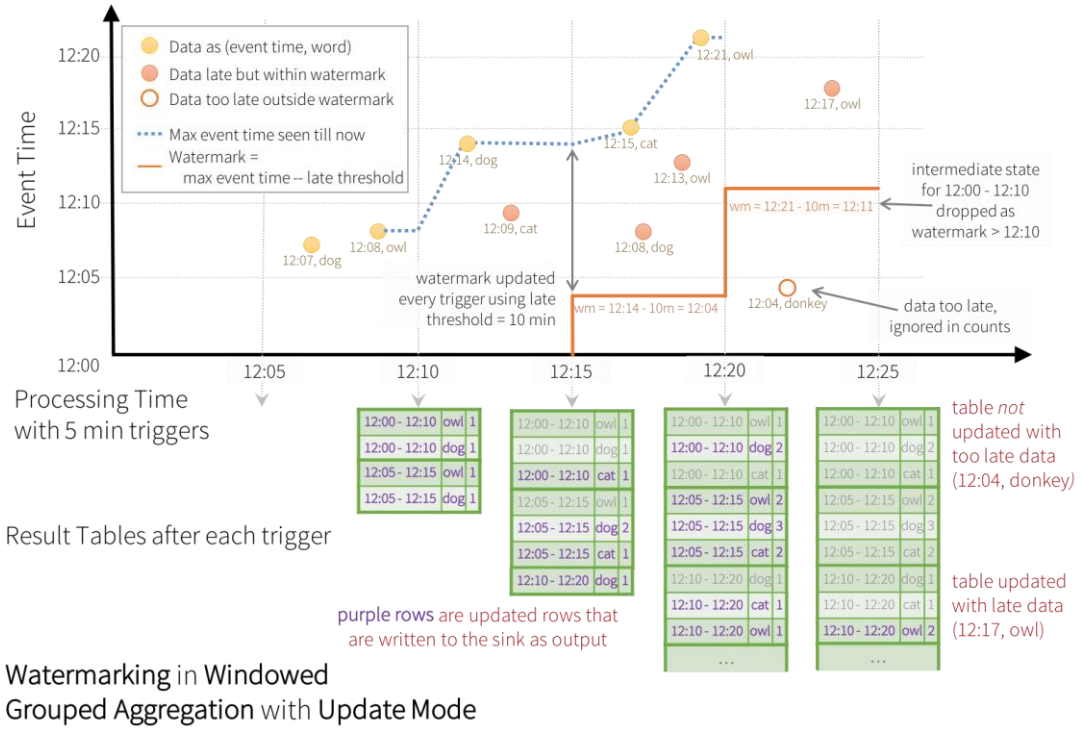
In other words, late data within the threshold will be aggregated, but data later than the threshold will start getting dropped (see later in the section for the exact guarantees).



Handling Late Data and Watermarking

In this example, we are defining the watermark of the query on the value of the column “timestamp”, and also defining “10 minutes” as the threshold of how late is the data allowed to be.

If this query is run in Update output mode (discussed later in Output Modes section), the engine will keep updating counts of a window in the Result Table until the window is older than the watermark, which lags behind the current event time in column “timestamp” by 10 minutes.



Streaming Deduplication

You can deduplicate records in data streams using a unique identifier in the events.

This is exactly same as deduplication on static using a unique identifier column.

The query will store the necessary amount of data from previous records such that it can filter duplicate records.

Similar to aggregations, you can use deduplication with or without watermarking.

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.

Continuous Processing [Experimental]

Continuous processing is a new, experimental streaming execution mode introduced in Spark 2.3 that enables low (~1 ms) end-to-end latency with **at-least-once** fault-tolerance guarantees.

Compare this with the default *micro-batch processing* engine which can achieve exactly-once guarantees but achieve latencies of ~100ms at best.

For some types of queries (discussed below), you can choose which mode to execute them in without modifying the application logic (i.e. without changing the DataFrame/Dataset operations).

Caveats

Continuous processing engine launches multiple long-running tasks that continuously read data from sources, process it and continuously write to sinks.

The number of tasks required by the query depends on how many partitions the query can read from the sources in parallel.

Therefore, before starting a continuous processing query, you must ensure there are enough cores in the cluster to all the tasks in parallel.

For example, if you are reading from a Kafka topic that has 10 partitions, then the cluster must have at least 10 cores for the query to make progress.

Stopping a continuous processing stream may produce spurious task termination warnings. These can be safely ignored.

There are currently no automatic retries of failed tasks. Any failure will lead to the query being stopped and it needs to be manually restarted from the checkpoint.