
# Spark Streaming

## DStreams & Structured Streaming

# Stream Processing

- Stream processing is the act of continuously incorporating new data to compute a result.
- In stream processing, the input data is unbounded and has no predetermined beginning or end. It simply forms a series of events that arrive at the stream processing system
- Examples:
  - credit card transactions
  - clicks on a website
  - sensor readings from IoT devices etc.

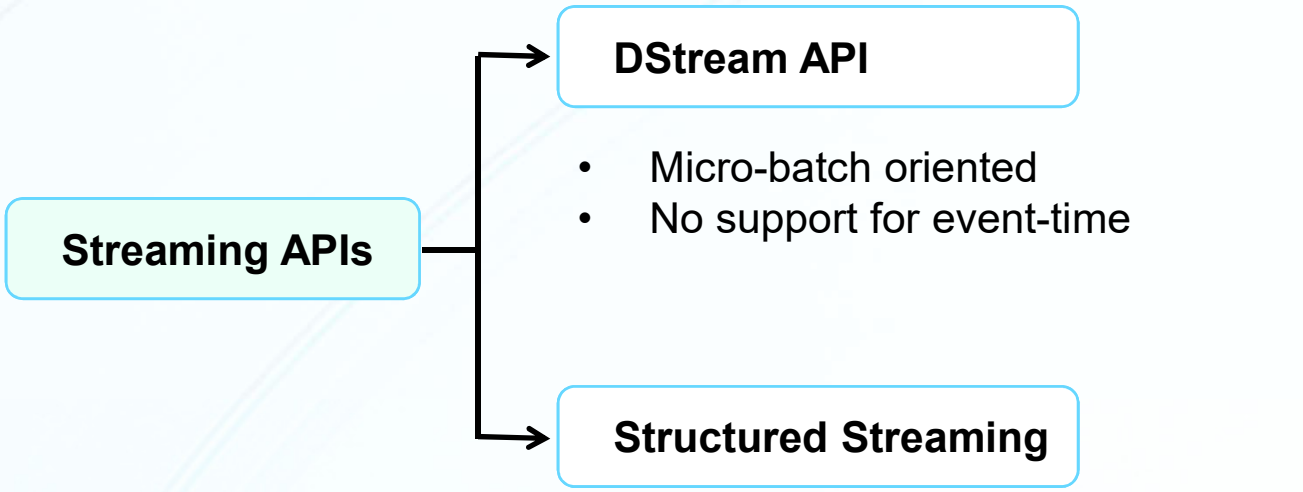
# Stream Processing Use Cases

- Notifications and alerting
  - Ex: Driving an alert to an employee at a fulfillment center
- Real-time reporting
  - Ex: Real-time Dashboards about a systems usage patterns
- Incremental ETL
  - Ex: Streaming processing of batch jobs
- Update data to serve in real time
  - Ex: Web analytics products such as Google Analytics
- Real-time decision making
  - Ex: Tracking fraudulent credit card transactions

# Spark Streaming

- Spark Streaming is an extension of the core API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.
- Data can be ingested from different streaming sources and can be processed using complex algorithms expressed with high-level functions like `map`, `reduce`, `join` and `window`. Finally, processed data can be pushed out to file systems, databases, and live dashboards.

# Spark's Stream APIs



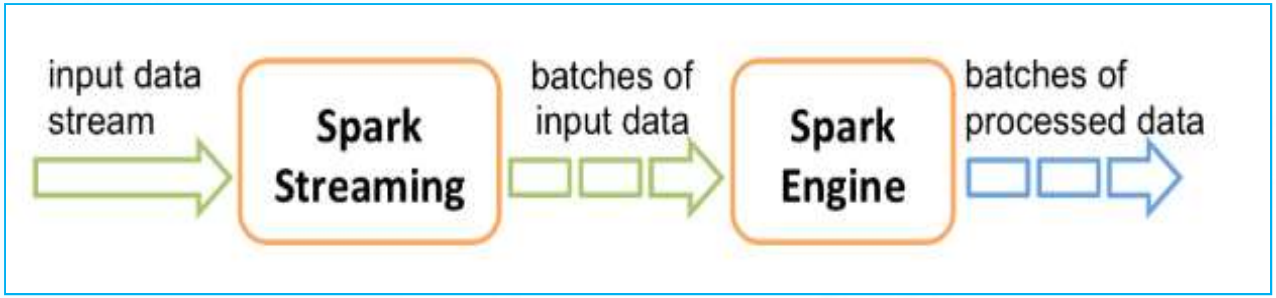

- Micro-batch oriented
- No support for event-time

- Higher level optimizations
- Supports event-time processing
- Support for continuous processing

# DStreams API

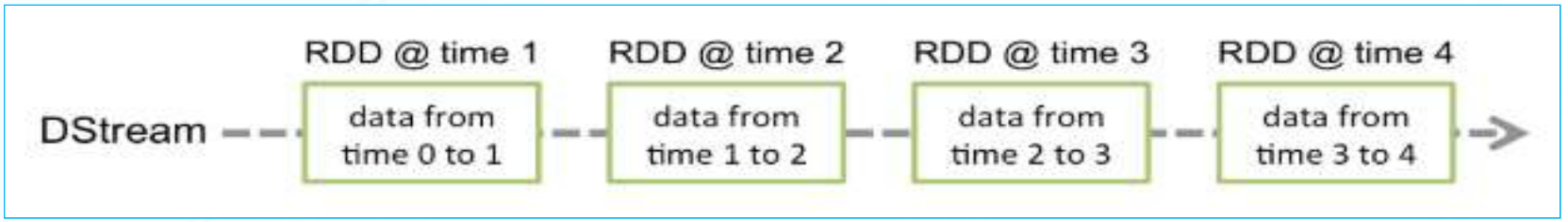
# Discretized Stream Processing

- Spark Streaming provides a high-level abstraction called ***discretized stream*** or ***DStream***, which represents a continuous stream of data, represented as a **sequence of RDDs**
- DStreams can be created either from input data stream sources such as Kafka, Flume or by applying high-level operations on other DStreams.
- Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.



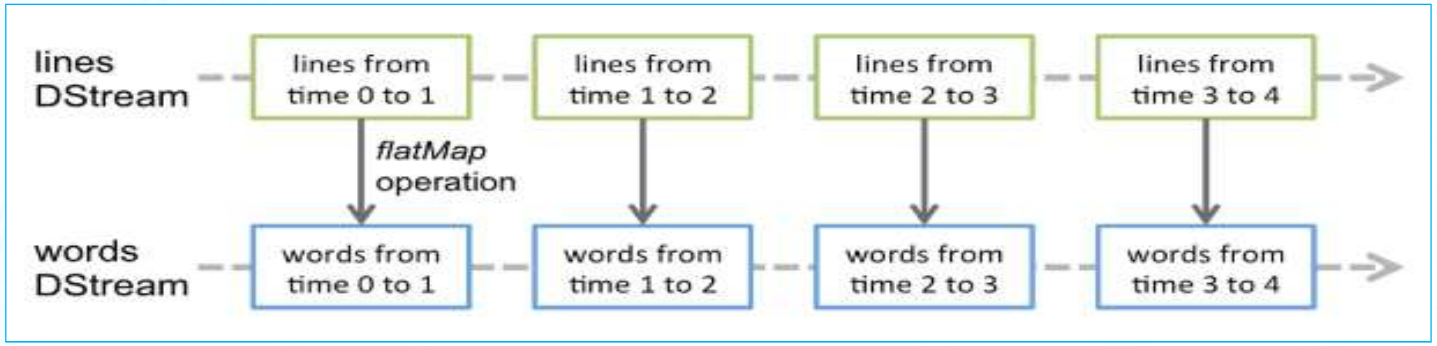
# DStreams

- A DStream is represented by a continuous series of RDDs.
- Each RDD in a DStream contains data from a certain interval.



# DStreams

- Any operation applied on a DStream translates to operations on the underlying RDDs.
  - For example, we can convert a stream of lines to words, apply **`flatMap`** operation on each RDD in the lines DStream to generate the RDDs of the words DStream.



# Streaming ‘Word Count’ Example

```
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 1)

lines = ssc.socketTextStream("localhost", 9999)
words = lines.flatMap(lambda line: line.split(" "))
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)
wordCounts.print()

ssc.start()
ssc.awaitTermination()
```

Here we are creating a stateless DStream that listens to the streaming data generated from a socket program that runs on localhost @ port 9999 with a batch interval of 1 second.

# Dependencies for Spark Streaming

- All the dependencies that are required to be included in your Spark Streaming projects can be obtained from Maven Central.
- Shown below is primary dependency that need to be added.

```
<dependency>
        <groupId>org.apache.spark</groupId>
        <artifactId>spark-streaming_2.11</artifactId>
        <version>2.3.1</version>
</dependency>
```

# Dependencies for Spark Streaming

- To use streaming data ingestion tools like Kafka, Flume, and Kinesis that are not present in the Spark Streaming core API, you will have to add the corresponding artifact to the dependencies.

| Source | Artifact |
|---|---|
| Kafka | spark-streaming-kafka-0-10_2.11 |
| Flume | spark-streaming-flume_2.11 |
| Amazon Kinesis | spark-streaming-kinesis-asl_2.11 |

# StreamingContext

- **`StreamingContext`** is the main entry point of all Spark Streaming functionality that can be created from a **`SparkConf`** object or from an existing **`SparkContext`** object.

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

sc = SparkContext(master, appName)
ssc = StreamingContext(sc, 1)
```

# StreamingContext

After a context is defined, you have to do the following:

- Define the input sources by creating input DStreams.
- Define the streaming computations to DStreams.
- Start receiving data and processing it using **`streamingContext.start()`**
- Wait for the processing to be stopped using **`streamingContext.awaitTermination()`**
- The processing can be manually stopped using **`streamingContext.stop()`**

# StreamingContext

- Once a context has been started, no new streaming computations can be set up or added to it.
- Once a context has been stopped, it cannot be restarted.
- Only one **`StreamingContext`** can be active in a JVM at a time.
- **`stop()`** on **`StreamingContext`** also stops the **`SparkContext`**. To stop only the **`StreamingContext`**, set the optional parameter of **`stop()`** called **`stopSparkContext`** to **`false`**.
- A **`SparkContext`** can be re-used to create multiple StreamingContexts, as long as the previous **`StreamingContext`** is stopped (without stopping the **`SparkContext`**) before the next **`StreamingContext`** is created.

# Streaming Sources

- The stream of input data received from streaming sources is represented by **Input DStreams**.
- Every Input DStream is associated with a **Receiver** object which receives the data from a source and stores it in Spark's memory for processing.
- Spark Streaming provides two categories of built-in streaming sources:
  - **Basic Sources:**
    - Directly available in the StreamingContext API.
    - Examples: file systems, and socket connections.
  - **Advanced Sources:**
    - Available through extra utility classes & dependencies.
    - Examples: Kafka, Flume, Kinesis, etc.

# File Streams

- A File Stream can read data from any HDFS API compatible File Systems such as HDFS, S3, NFS etc.

```
sc = SparkContext(master, appName)
ssc = StreamingContext(sc, 1)
lines = ssc.textFileStream( <filePath> )
```

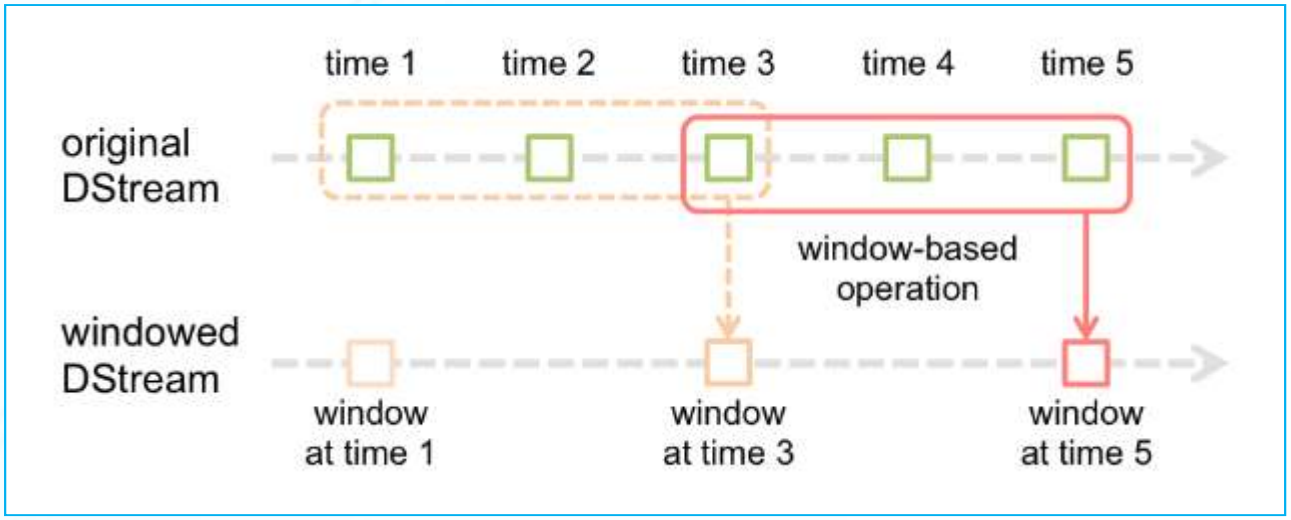
# DStream Transformations

- DStreams support many of the transformations available on normal Spark RDD's. Some of the common ones are as follows:

| Common DStream Transformations | | |
|---|---|---|
| **map**(*func*) | **flatMap**(*func*) | **filter**(*func*) |
| **repartition**(*numPart*) | **union**(*otherStream*) | **count**() |
| **reduce**(*func*) | **countByValue**() | **reduceByKey**(*func*, [*#Tasks*]) |
| **join**(*otherStream*, [*#Task*]) | **updateStateByKey**(*func*) | **transform**(*func*) |

# Windowed Transformations

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



# Windowed Transformations

- 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.
- 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

# Windowed Transformations

| Transformation | Meaning |
|---|---|
| **window**(*windowLength, slideInterval*) | Return a new DStream which is computed based on windowed batches of the source DStream |
| **countByWindow**(*windowLength,slideInterval*) | Return a sliding window count of elements in the stream. |
| **reduceByWindow**(*func, windowLength, slideInterval*) | Return a new single-element stream, created by aggregating elements in the stream over a sliding interval using *func*. The function should be associative and commutative so that it can be computed correctly in parallel. |
| **reduceByKeyAndWindow**(*func, windowLength, slideInterval,* [*numTasks*]) | When called on a DStream of (K, V) pairs, returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function *func* over batches in a sliding window. |
| **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. Like in reduceByKeyAndWindow, the number of reduce tasks is configurable through an optional argument. |

# Structured Streaming

- Structured Streaming provides fast, scalable, fault-tolerant, end-to-end exactly-once stream processing without the user having to reason about streaming.
- Built on the Spark SQL engine. You can express your streaming computation the same way you would express a batch computation on static data.
- Spark SQL engine will take care of running it incrementally and continuously and updating the final result as streaming data continues to arrive.

# Structured Streaming

- You can use the DataFrame API to express :
  - streaming aggregations
  - event-time windows
  - stream-to-batch joins, etc.
- The computation is executed on the same optimized Spark SQL engine.
- The system ensures end-to-end exactly-once fault-tolerance guarantees through check-pointing and Write-Ahead Logs.

# An Example - Streaming Wordcount

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

words = lines.select(
       explode( split(lines.value, " ")).alias("word"))

wordCounts = words.groupBy("word").count()

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

query.awaitTermination()
```

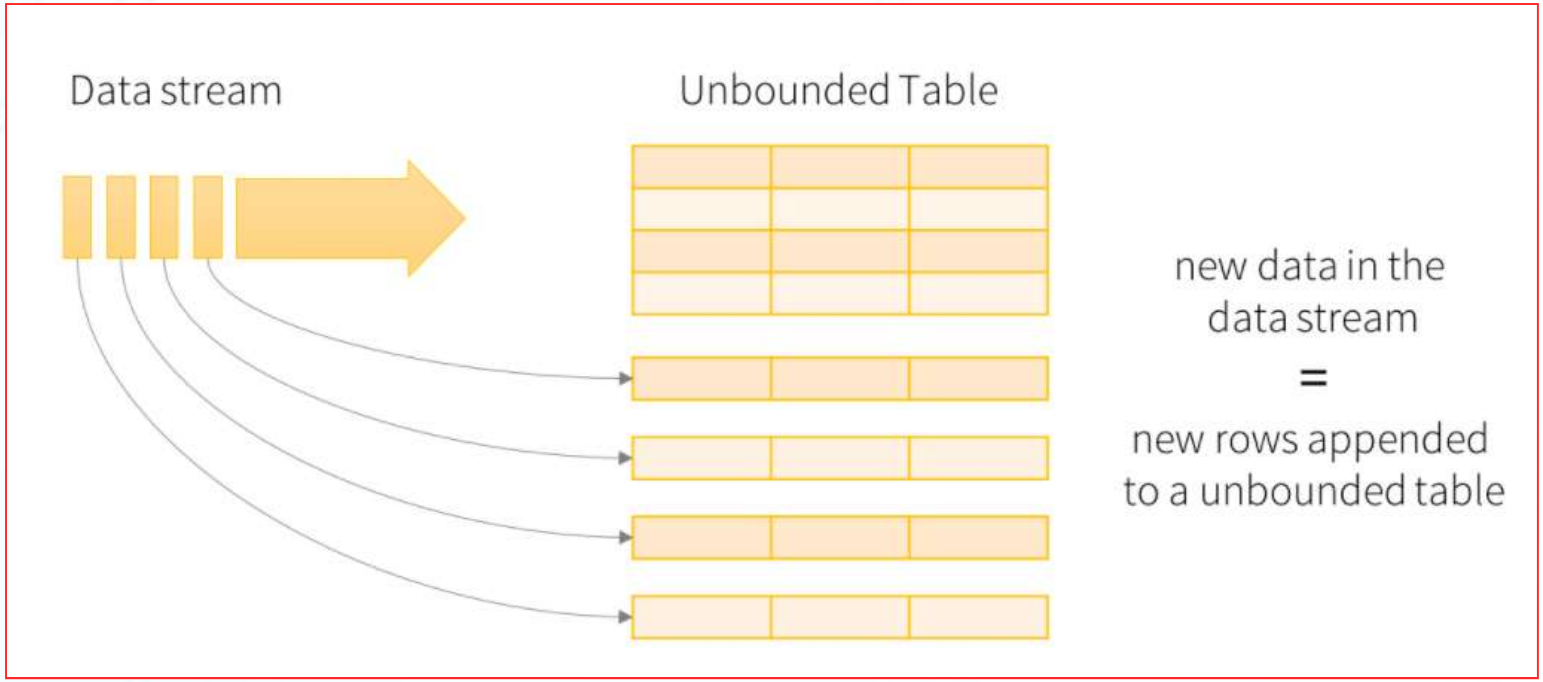
# Structured Streaming

- 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.
  - Since Spark 2.3, a low-latency processing mode called Continuous Processing, which can achieve end-to-end latencies as low as 1 millisecond with at-least-once guarantees had been introduced.
  - Without changing the DataFrame operations in your queries, you will be able to choose the mode based on your application requirements.

# 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.

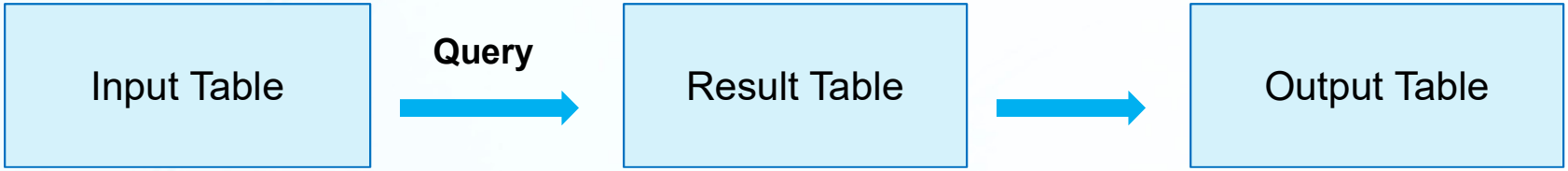
# Programming Model




Data stream as an unbounded table

# Programming Model

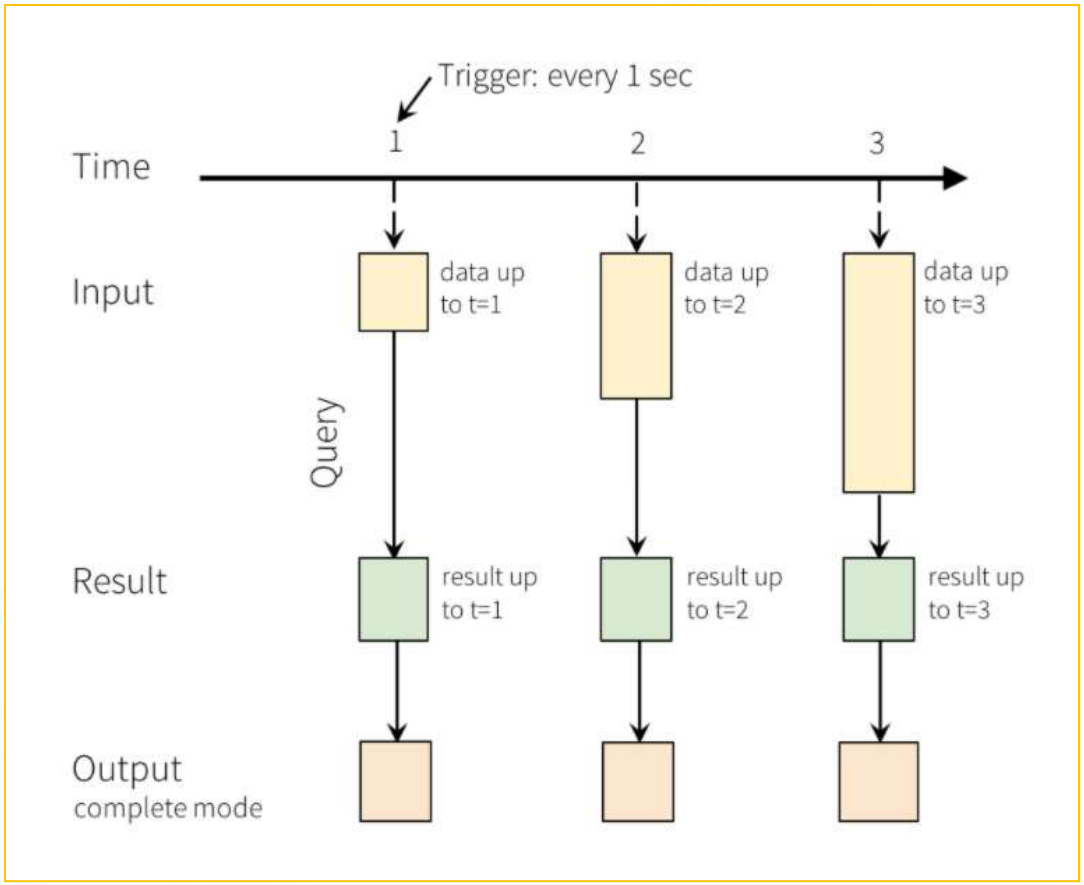
- A query on the input will generate the “Result Table”.
- Every trigger interval (say, every 1 second), new rows get appended to the Input Table, which eventually updates the Result Table.
- Whenever the result table gets updated, we would want to write the changed result rows to an external sink.



# Programming Model

- Structured Streaming does not materialize the entire table.
- It reads the latest available data, processes it incrementally to update the result, and then discards the source data.
- Spark is responsible for updating the Result Table when there is new data, thus relieving the users from reasoning about fault-tolerance and data consistency (at-least-once, or at-most-once, or exactly-once

# Programming Model



# Output Modes

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

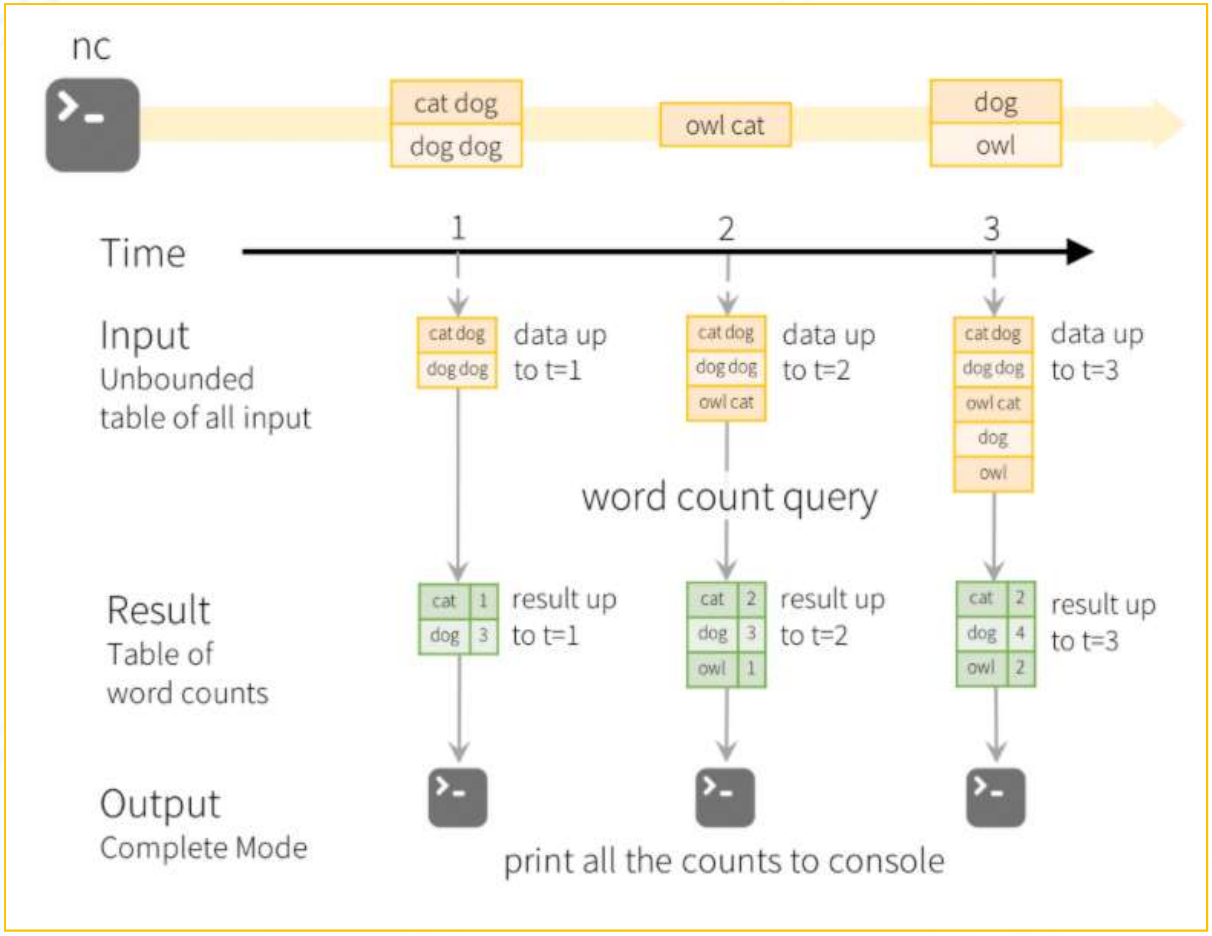
- ***Complete Mode*** - The entire updated Result Table will be written to the external storage. It is up to the storage connector to decide how to handle writing of the entire table.

- ***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 (since Spark 2.1.1).

Note that each mode is applicable on certain types of queries.

# Output Modes



# Built-in Input Sources

- **File source**
  - Reads files written in a directory as a stream of data. Files will be processed in the order of file modification time. If latestFirst is set, order will be reversed. Supported file formats are text, CSV, JSON, ORC, Parquet.
- **Kafka source**
  - Reads data from 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. (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. It is intended for testing and benchmarking.

# Window Operations on Event Time

- Event-time is the time embedded in the data itself.
  - This allows window-based aggregations (e.g. number of events every minute) where each time window is a group and each row can belong to multiple windows/groups.
- In case of window-based aggregations, aggregate values are maintained for each window the event-time of a row falls into.

```
words = ...  # streaming DataFrame of schema { timestamp: Timestamp, word: String }

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

# Window Operations on Event Time