**Spark-Streaming – Word Count - Netcat example**

**Step-1:** Create a python file(I.e. Say SparkStreamingWordCount.py) to create a streaming DataFrame that represents text data received from a server listening on localhost:9999, and transform the DataFrame to calculate word counts.

**Step-2:** Run the Netcat (a small utility found in most Unix-like systems) as a data server by using below command in one terminal

nc -lk 9999

**Step-3:**

In another terminal run/ execute the above python code using below command

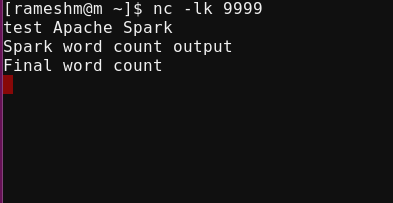
./bin/spark-submit ../../SparkStreamingWordCount.py localhost 9999

**Step4:**

Any lines typed in the terminal running the netcat server will be counted and printed on screen every second. It will look something like the following.

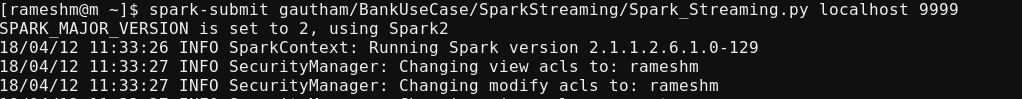
**Output:**

Terminal-1:

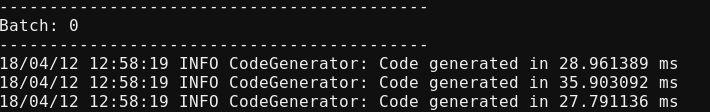


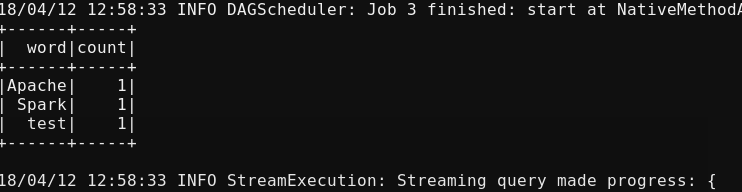
Terminal-2:

Executing the python code



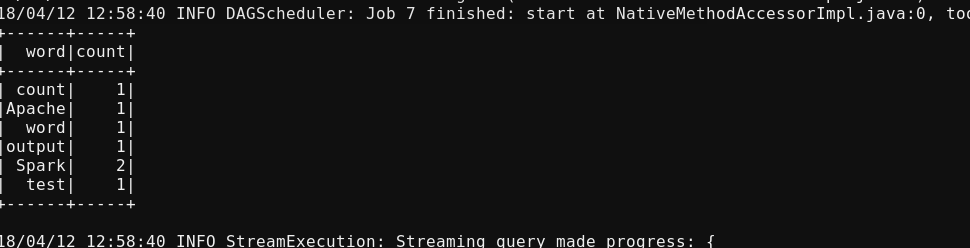
After typing the first line in terminal-1 (test Apache Spark), below is the output in terminal-2



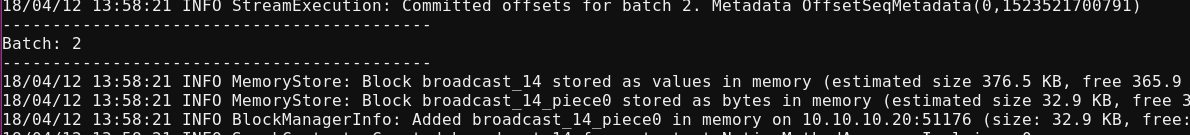


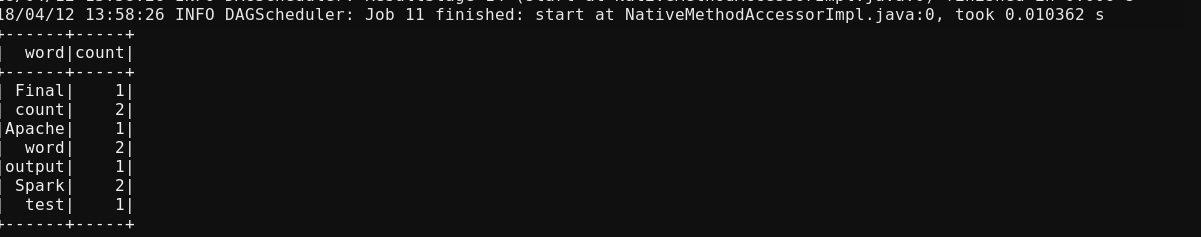
After typing the Second line in terminal-1, below is the output in terminal-2





After typing the Third line in terminal-1, below is the output in terminal-2





**Structured Streaming flow:**

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.

## Basic Concepts

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.

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.

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

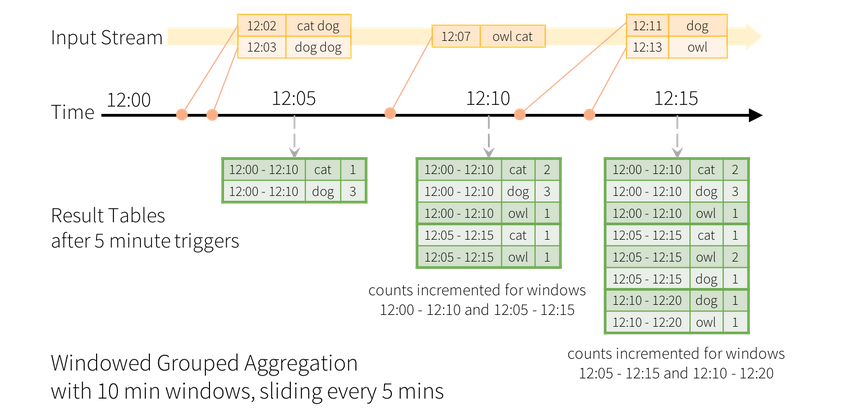
* Complete Mode - The entire updated Result Table will be written to the external storage. It is up to the storage connector to decide how to handle writing of the entire table.
* Append Mode - Only the new rows appended in the Result Table since the last trigger will be written to the external storage. 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 (available since Spark 2.1.1). Note that this is different from the Complete Mode in that this mode only outputs the rows that have changed since the last trigger. If the query doesn’t contain aggregations, it will be equivalent to Append mode.

**Spark Streaming - 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.

**Example:**

Imagine our [quick example](https://spark.apache.org/docs/2.2.0/structured-streaming-programming-guide.html#quick-example) is modified and the stream now contains lines along with the time when the line was generated. Instead of running word counts, we want to count words within 10 minute windows, updating every 5 minutes. That is, word counts in words received between 10 minute windows 12:00 - 12:10, 12:05 - 12:15, 12:10 - 12:20, etc. Note that 12:00 - 12:10 means data that arrived after 12:00 but before 12:10. Now, 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. So the counts will be indexed by both, the grouping key (i.e. the word) and the window (can be calculated from the event-time).



**Steps To execute word count with window operations:**

**Step-1:** Create a python file(I.e. Say Spark\_Streaming\_WindowOperations.py) to create a streaming DataFrame that represents text data received from a server listening on localhost:9999, and the DataFrame with aggregate values are maintained for each window the event-time of a row falls into.

**Step-2:** Run the Netcat (a small utility found in most Unix-like systems) as a data server by using below command in one terminal

nc -lk 9999

**Step-3:**

In another terminal run/ execute the above python code using below command

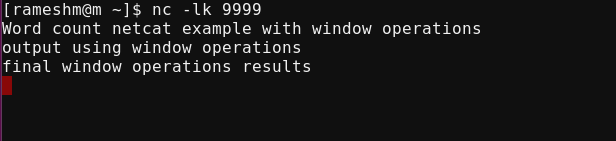
./bin/spark-submit ../../Spark\_Streaming\_WindowOperations.py localhost 9999

**Step4:**

Any lines typed in the terminal running the netcat server will be counted and printed on terminal-2 for every window.

**Output:**

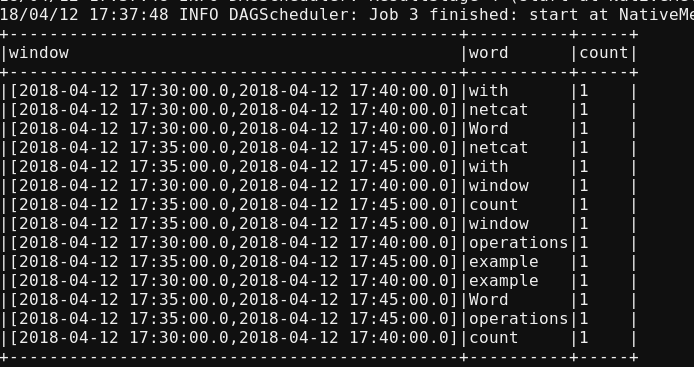
**Terminal-1:**

****

**Terminal-2:**

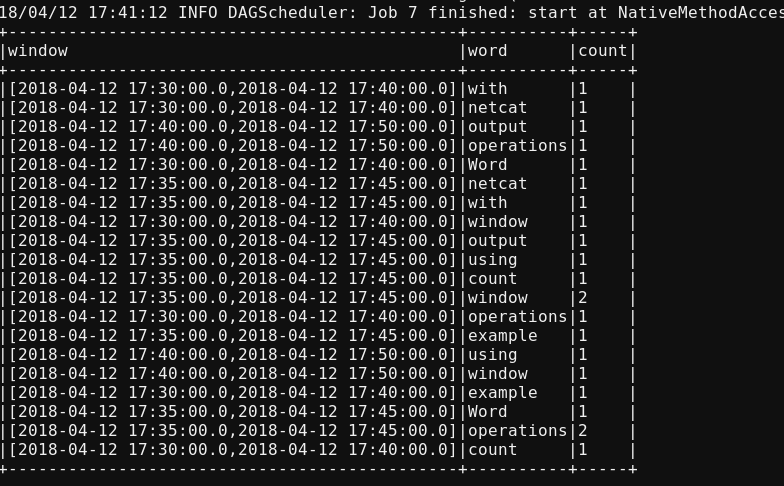
**Batch-0 output: Shows the word count between intervals 17:30:00 – 17:40:00 and 17:35:00 – 17:45:00**

Executed text (Word count netcat example with window operations) around 15:37:00 PM



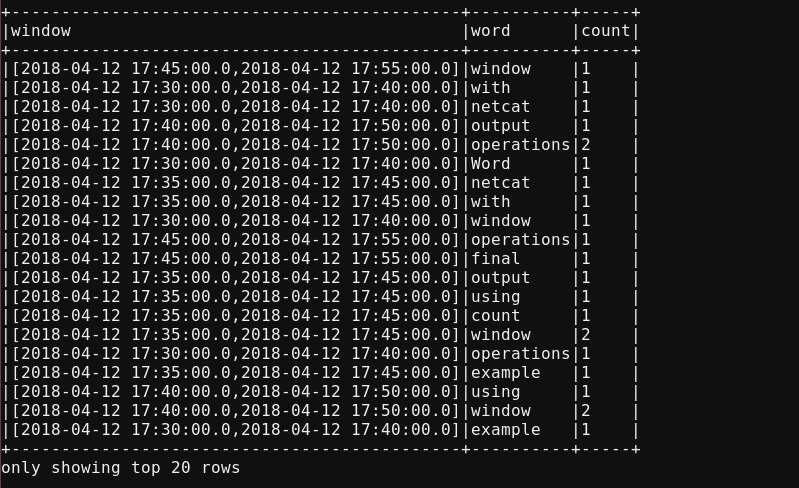
**Batch-1 output: Shows the word count between intervals 17:30:00 – 17:40:00, 17:35:00 – 17:45:00 and 17:40:00 – 17:50:00**

Executed text (output using window operations) around 15:42:00 PM



**Batch-2 output: Shows the word count between intervals 17:30:00 – 17:40:00, 17:35:00 – 17:45:00,17:40:00 – 17:50:00 and 17:45:00 – 17:55:00**

Executed text (final window operations results) around 15:47:00 PM



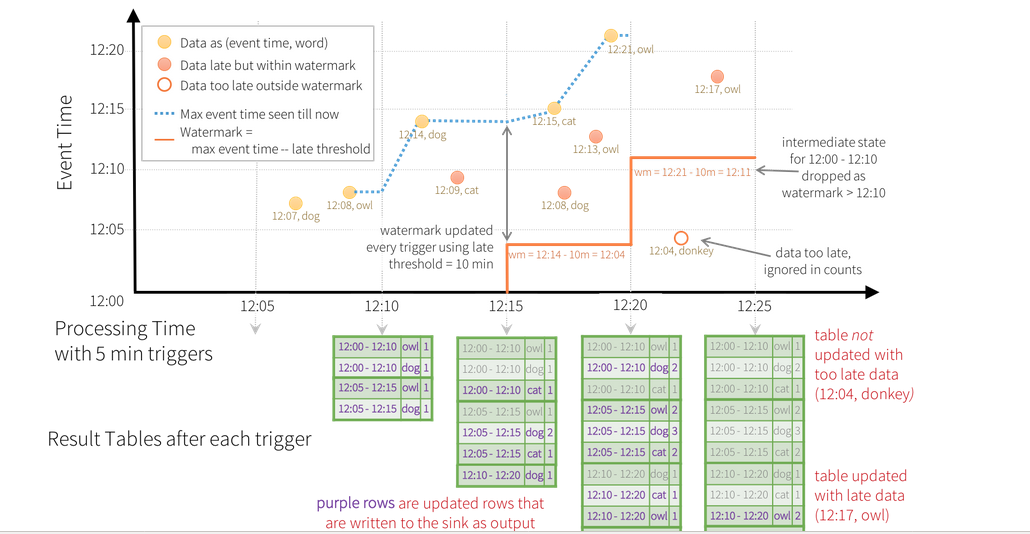
### Handling Late Data and Watermarking: Used to handle evnets that arrives late to the application.

**Watermarking in Windowed Grouped Aggregated with Update Mode:**

* Watermarking lets the engine automatically track the current event time in the data and attempt to clean up old state accordingly.
* User 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 be dropped.

**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](https://spark.apache.org/docs/2.2.0/structured-streaming-programming-guide.html#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.

****

**Execution process is same as the above**

**Spark Streaming – Kafka:**