1. SPARK STREAMING

1.1 Overview

Spark's streaming library (Structured Streaming) is a scalable, fault-tolerant engine built on Spark SQL that lets you write streaming jobs with the same DataFrame/Dataset APIs used for batch, executing them incrementally via micro-batches or a low-latency continuous mode. It provides end-to-end reliability with checkpointing and replayed offsets, and can achieve exactly-once semantics with appropriate sources/sinks—while the older DStreams-based "Spark Streaming" API is now legacy/deprecated in favor of Structured Streaming. Common uses include ingesting from Kafka, performing stateful aggregations and windowing, and delivering near real-time analytics with configurable triggers.

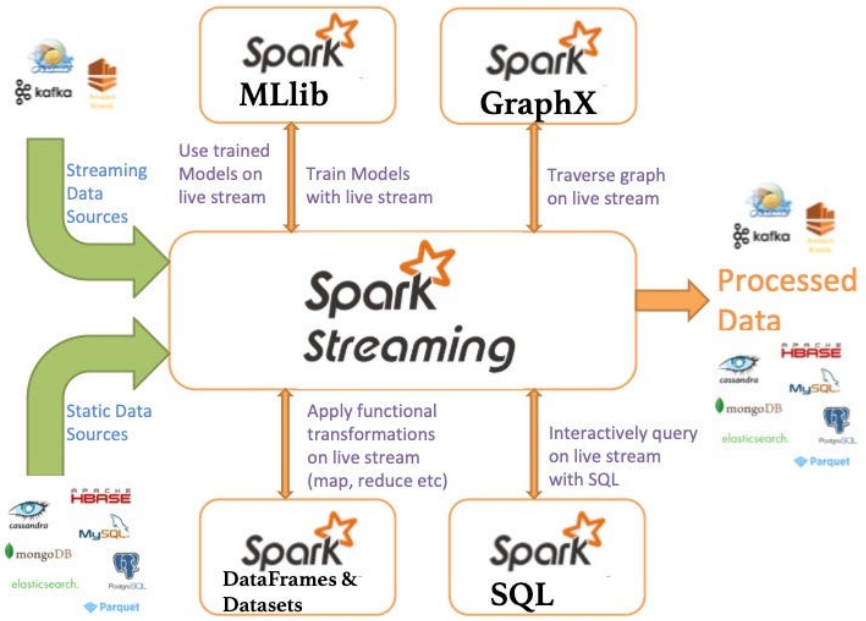

Figure 1: Spark Streaming Pipeline.

1.2 Watermark & Window

Watermark. A mark for handling (evict) late-arriving data, requires data to have a timestamp field (be it event time - time of event occuring IRL, or processing time - time of Spark receiving the data). For example, the latest data captured is at 10:10 AM, watermark = 4 mins ⟶ Reject all data sooner than 10:06 AM. If data at 10:15 AM arrives ⟶ Update latest data and reject all data sooner than 10:11 AM.

Window. There are 3 core window types in Azure Stream Analytics, *i.e.*, tumbling, hopping, and sliding.

Tumbling windows split the stream into fixed-length, non-overlapping, contiguous intervals (for example, every 10 seconds or every 5 minutes). Each event belongs to exactly one window, which makes tumbling ideal for periodic rollups (per-minute counts, average sensor values per hour, etc.) and easy-to-reason, "bucketed" aggregations. Results are produced once per window at its end, with the output timestamp set to the window's end time.

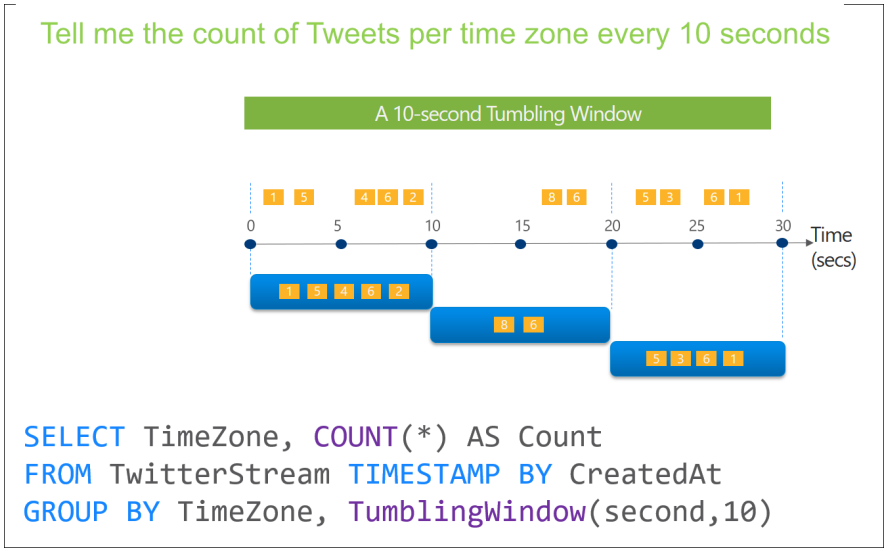

Figure 2: Tumbling windows.

Hopping windows also have a fixed size, but they overlap and "hop" forward by a configurable hop size (for example, 10-minute windows starting every 5 minutes). Events can appear in multiple windows, which is useful for moving averages, rolling KPIs, or smoothing volatile metrics. Conceptually, a tumbling window is a special case of a hopping window where hop size = window size. Like tumbling, hopping windows emit at the end of each scheduled window.

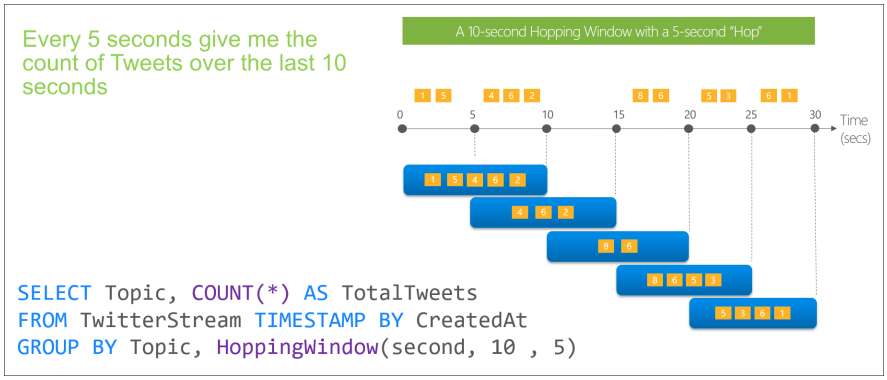

Figure 3: Hopping windows.

Sliding windows consider all possible windows of a given length, but to avoid infinite outputs, the engine emits a result only when the window's contents change—that is, when an event enters or exits the window. Sliding windows have no fixed schedule, so they are excellent for reactive, "tell me when the last N minutes changed" scenarios, bursty traffic, and threshold crossings without producing idle duplicates when the

stream is quiet. Unlike tumbling/hopping, outputs are event-driven rather than time-scheduled.

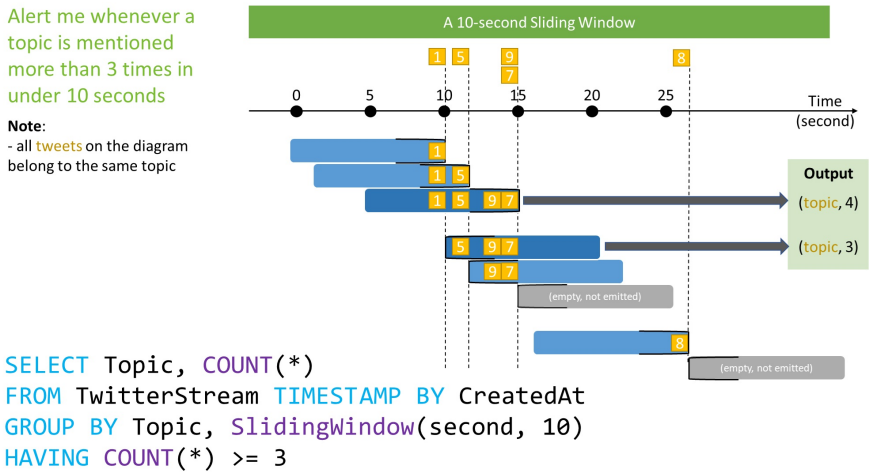

Figure 4: Sliding windows.

Suggestion: Choose tumbling for clean, non-overlapping buckets; hopping for overlapping, time-aligned rollups (moving stats); and sliding for change-driven analytics that react immediately to arrivals/departures within a lookback window.
*For more information, please read **this document**.*

1.3 Execution modes

There are 3 execution modes:

1. **Update:** Outputs changed rows since last trigger requires watermarks for most aggregations.
   - Why? Without them, how can Spark knows when an aggregation is final (ex: Data row count increases every 1 second, when do you want it?)
   - When to use? Aggregations with watermarks, stateful operations.
2. **Append:** Outputs new rows that will NEVER change $\longrightarrow$ Cannot be used with aggregations unless with watermarks.
   - Why? Aggregated results are not absolute (*e.g.*, Data row count always increases, but is static for some interval).
   - When to use? Simple transformations, filters and map operations.
3. **Complete:** Outputs entire result table every time, independent of watermarks, but can only be used with aggregations (not transformation).

The near-future of Spark: **real-time mode** coming in **4.1.0**.

2. TUTORIAL

This section is leveraged to help students get familiar with Spark Streaming library. Let's create a streaming job, reads data from socket **9999**, and process grocery sale count in the form of json output to console.

1. Starting netcat in listen mode on TCP port 9999 on the local machine.

```
nc -l -p 9999
```

2. Copying data to the terminal.

```
{"timestamp": "2025-01-01 10:00:06", "item": "orange", "amount": 4}
{"timestamp": "2025-01-01 10:00:07", "item": "banana", "amount": 1}
{"timestamp": "2025-01-01 10:00:10", "item": "apple", "amount": 3}
{"timestamp": "2025-01-01 10:00:12", "item": "orange", "amount": 2}
{"timestamp": "2025-01-01 10:00:00", "item": "apple", "amount": 5}
{"timestamp": "2025-01-01 10:00:01", "item": "banana", "amount": 3}
{"timestamp": "2025-01-01 10:00:02", "item": "apple", "amount": 2}
```

3. Creating a static DataFrame to join with the streaming DataFrame to get the revenue of each item.

Code 1: DataFrame Generation.

```
df_static = spark.createDataFrame([("apple", 2.0), ("banana", 1.0), ("orange", 1.5)], ["item", "price"])
```

4. Getting the timestamp and the rest of the columns from the json string.

Code 2: Information extraction *e.g.* timestamp, value, etc.

```
df = (spark.readStream # this is a streaming DataFrame, not a static DataFrame
      .format("socket")         # read from socket, great for debugging
      .option("host", "bd-1")   # hostname
      .option("port", 9987)      # port
      .load()
      .select(from_json(col("value"), "timestamp STRING, item STRING, amount INT").alias("data")) # parse the json string, stored at "
  data" column
      .select("data.*")
      .withColumn("timestamp", to_timestamp(col("timestamp"))) # convert string to timestamp
     )
```

5. Aggregating the grocery count by item and print to console to validate the input data.

Code 3: Data aggregation.

```
MODE = "update" # "complete", "append", "update"
if MODE == "append":
    df_stream = (df
                 .withWatermark("timestamp", "1 seconds")                                  # set watermark to 1 seconds
                 .groupBy(window(col("timestamp"), "10 seconds", "5 seconds"), col("item")) # window of 10 seconds, slide of 5 seconds
                 .agg(sum("amount").alias("total_amount"))
                 .select("item", "total_amount")
                )
else:
    df_stream = (df
                 .groupBy(col("item")) # group by item
                 .agg(sum("amount").alias("total_amount"))
                )
```

6. Joining with the static DataFrame to get the revenue of each item over time, not the total revenue.

Code 4: Joining with the static DataFrame.

```
df_joined = (df_stream
             .join(df_static, on="item", how="left")
             .withColumn("revenue", col("total_amount") * col("price"))
             .drop("price")
            )
```

7. Writing the output to console.

Code 5: Visualization.

```
query = (df_final
         .writeStream
         .outputMode(MODE) # complete, append, update
         .format("console")
         .option("truncate", "false") # do not truncate the output
         .start()
        )
query.awaitTermination()
```

3. EXERCISES

This exercises is leveraged to help students practice Spark and Kafka operations professionally. The requirements of exercise are shown as the followings.

Exercise 1: Prepare movie data. This is mandatory to do any of the exercises.

- Reading data from 3 topics, including:
  - Movies (with “Lab1_movies" as topic name (*i.e.*, subscribe))
  - Ratings with the rate of 100 rows per trigger (with “Lab1_ratings" as topic name (*i.e.*, subscribe))
  - Tags similar to ratings (with “Lab1_tags" as topic name (*i.e.*, subscribe))

  Kafka addresses are 10.1.1.10:30090, 10.1.1.27:30091, 10.1.1.203:30092.
- Defining schema for movies, ratings and tags. Subsequently, converting data type of the DataFrame to its corresponding schema.

Exercise 2: *Which genres are hot right now?* ***Hint:*** Static-stream join and per-genre aggregation.

- Joining ratings with movies to get genres.
- Writing to console every 5 seconds.

Exercise 3: *What goes on the Trending Now rail?* ***Hint:*** Windowed counts + per-window Top-K using window functions.

- Secondary sorting by movieId for tie-break.
- Writing to console every 5 seconds.

Suggestion:

- Inputs: ratings (stream), movies (static).
- Must use: 5-minute tumbling window on eventTime; count by (window, movieId); join title; dense_rank over partitionBy(window).orderBy(cnt desc, movieId asc).
- Output: (window, movieId, title, cnt, rank) with rank $\leqslant$ 3, complete mode.
- Passing if:
  - Columns match: max 3 rows per window.
  - Watermark on ratings before the window.
  - Secondary sorting by movieId for tie-break.

Exercise 4: Stream-Stream Join: Tags and Ratings. ***Hint:*** Real-time monitoring system to identify which tags are being applied to movies as they receive ratings.

- Stream-stream join in APPEND mode (no aggregation)
- Writing to console every 5 seconds.

Suggestion:

- Inputs: ratings (stream), tags (stream).
- Must use: watermarks on both streams; join on movieId.
- Output: (movieId, tag, rating, ratingTime, tagTime) in append mode.
- Goal: Stream–stream join to correlate live rating events with tagging events.
- Passing if:
  - Watermarks on both sides exist in the plan.
  - Stream-stream join on movieId.
  - Output shows matched rating-tag pairs.

Submission: The works must be done in a group of at most 3.

- Compress all necessary files (sources and a report) as **CO3137_BigDataLab_Lab02_<Student ID-1>_<Student ID-2>_<Student ID-3>.zip**.
- Students must submit and demonstrate their results before the end of the class.
- Early submitters (within the lab class) get bonus points. The rest can still get a 10 if they deserve it comprehensively.