databricks

Introduction to Streaming

LECTURE

Event Time + Aggregations over Time Windows

In this lecture, you will learn how to reason about time in streaming data, apply time-based windowing for event-time aggregations, and address challenges like memory pressure using solutions such as external state stores, watermarking for handling late-arriving events, and techniques to control state size for efficient real-time analytics.

Reasoning About Time

Event time vs. Processing time

Event time vs. processing time

- **Event Time:** time at which the event (record in the data) actually occurred.
- **Processing time:** time at which a record is actually processed.
- Important in every use case processing unbounded data in whatever order (otherwise no guarantee on correctness)

Event time refers to the time at which the event actually occurred. Examples of this are when an IoT device records a BPM or what the sales for a movie was on a particular date.

Processing time refers to the time at which the record is actually processed.

Event Time is critical to Streaming

- Necessary for accurate and consistent processing of data based on the time the event actually occurred, rather than when it was received or processed.
- Especially important in scenarios where data may be delayed or out of order, as it ensures that the data is processed in the correct sequence and that any time-based aggregations or calculations are accurate

Time Based Windows

Tumbling window vs. Sliding window

Tumbling Window

- **No window overlap**
- Any given event gets aggregated into **only one** window group (e.g. 1:00–2:00 am, 2:00–3:00 am, 3:00-4:00 am, ...)

Sliding Window

- **Windows overlap**
- Any given event gets aggregated into **multiple window** groups (e.g. 1:00-2:00 am, 1:30–2:30 am, 2:00–3:00 am, ...)

There are two different types of windows for aggregating data: tumbling windows and sliding windows.
Tumbling windows have no window overlap.
In the one case if there's a tumbling window, then that box is discrete from every other box, and there's no overlap between them.

Given events get aggregated into only one window group.
If we have a sliding window, that means that the windows are overlapping.
So as we're moving that box over, there is overlap between two boxes.
And then in this one, in any given event, gets aggregated into multiple window groups.

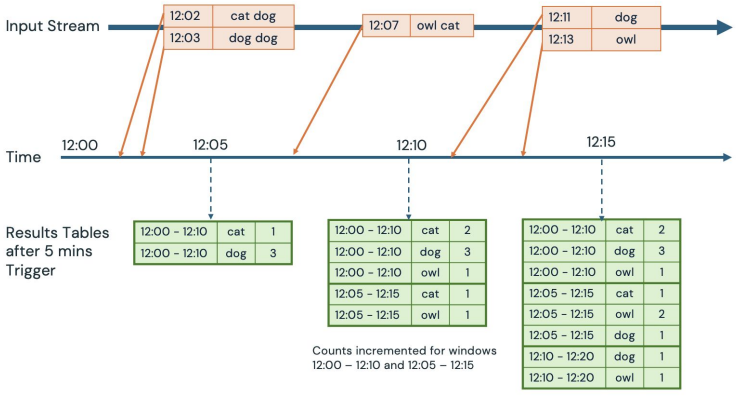
Note: In all cases, the leading time is included in the window. However the trailing time is excluded from that time window.

For example, suppose your Time window is one hour. You receive a row at 2;00 PM.
Does the row go in the 1:00 - 2:00 Time window or the 2:00 - 3:00 Time window?
The answer is the row would be included in the 2:00 - 3:00 Time window.

Time Based Windows

Sliding window example

```
windowedDF =
 (
    eventsDF
    .groupBy(window("eventTime",
       "10 minutes",
       "5 minutes"))
    .count()
    .writeStream
    .trigger(processingTime="5 minutes")
 )
```



Let's use the following results table as an example. After five-minute triggers, we want to count incremental windows from 12:00 to 12:10, and then from 12:05 to 12:15.

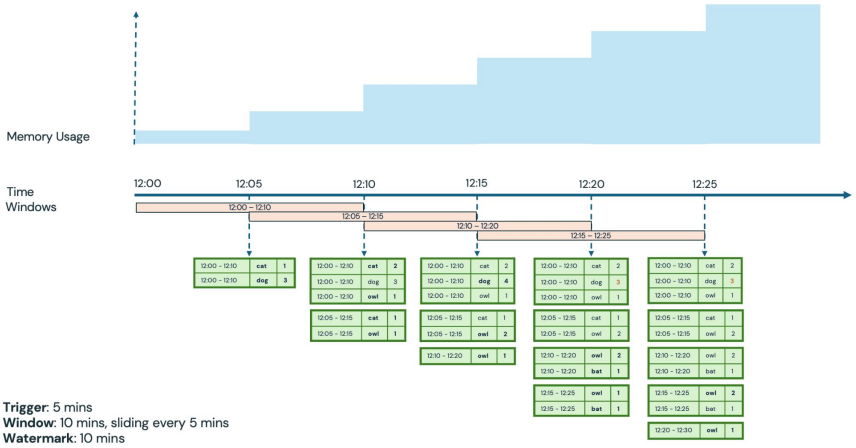
It starts at 12:00 and goes until 12:05, which is when the next trigger is happening. After that, the next trigger happens at 12:10.
But if we have sliding windows of 10 minutes, that means that these are going to overlap.
So our 12:00 to 12:10 has one window, but then our trigger happens at 12:05, which means that the next 10 minutes, starting at 12:05, are going to overlap with our first window.

This means it is possible for the same row to be in multiple Time windows.
For example, the 12:07 ‘cat’ row, is counted in both the 12:00 - 12:10 Time window as well as the 12:05 - 12:15 Time window.

Challenges: Memory Pressure

Long running query example

- Size of intermediate state will keep increase over time.
- Default behaviour is to maintain state in executor memory and DBFS.
- GC pauses will take longer and longer. Failure is inevitable.



Another concern arises when we start architecting long running processes.

When a Stateful Streaming query needs to run for days, that means it must hold the ‘State’ in RAM.
This can be problematic since there is not an infinite amount of RAM on a Compute.

Thus it becomes necessary to instruct the system on how to manage its memory and sustain performance over time.

There are two approaches to do this.

Solution 1: RocksDB as external state store

One technique to reduce memory usage on the cluster is to save intermediate state off-heap.

```
spark
spark.conf.set(
  "spark.sql.streaming.stateStore.providerClass",
  "com.databricks.sql.streaming.state.RocksDBStateStoreProvider")
```

References:
Databricks Blog on RocksDB
Structured Streaming Programming Guide

https://www.databricks.com/blog/feature-deep-dive-watermarking-apache-spark-structured-streaming

When managing a streaming query where Spark may need to manage millions of keys and keep state for each of them, the default state store that comes with Databricks clusters may not be effective.
You might start to see higher memory utilization, and then longer garbage collection pauses. These will both impede the performance and scalability of your Structured Streaming application.

This is where Apache RocksDB comes in. This open source project hold RAM off-heap, making it more performant.
You can leverage RocksDB natively in Databricks by enabling it like so in the Spark configuration:

```
spark conf set(
  'spark.sql.streaming.stateStore.providerClass',
  'com.databricks.sql.streaming.state.RocksDBStateStoreProvider')
```

Solution 2: Watermarking / Late Threshold

Capture “Event Time” on incoming data and set Watermarks.

Purpose:

1. Reclaim memory by purging old state
2. Set expectations on “how late is too late?”

   Event Time < (Max event time seen by the engine - late threshold)

```
Spark
.readStream
.withWatermark(“event_timestamp
”, “10 minutes”)
.<source info>
```


We use the watermarks to define a threshold. In the example above, events with event_timestamp within the 1 hour watermark are OK.
Events outside watermark are “too late”. The watermark is always relative to the trigger time and works like a sliding window.

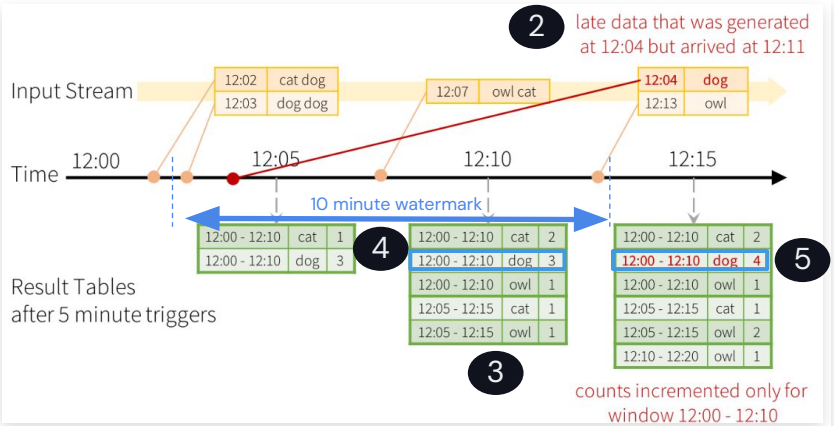
Watermarking controls how long the system should wait for late-arriving data when performing stateful operations.
It defines a time-based threshold, allowing the system to determine when it is safe to close a time window or discard old state, thus preventing unbounded memory usage

Handling Late Arrival Events with Watermark

Delayed data will be processed if within Watermark threshold - Guaranteed!

1 Configuration:

1. **Trigger**: 5 min
2. **Window**: 10 mins
3. **Sliding**: 5 min
4. **Watermark:** 10 mins

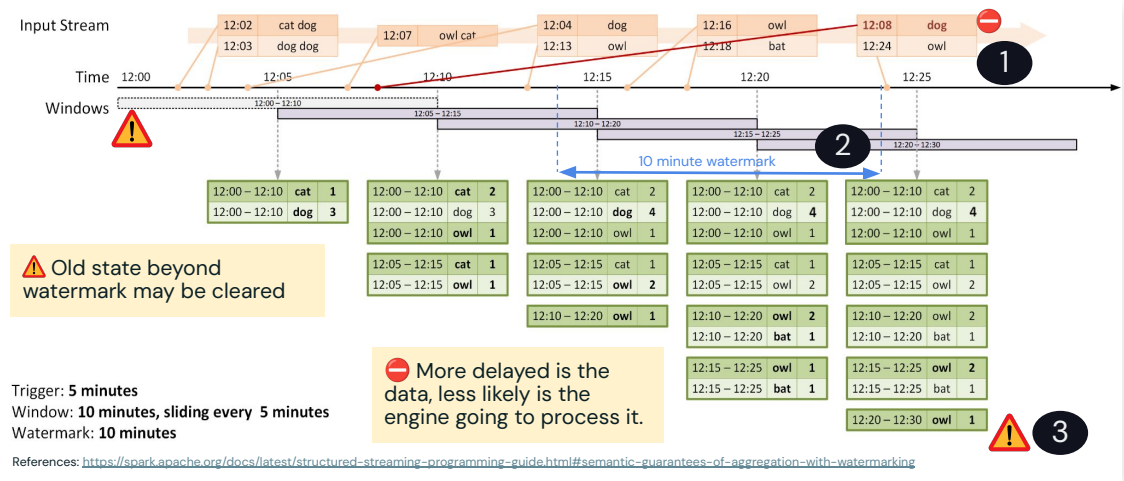


References: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#semantic-guarantees-of-aggregation-with-watermarking

1. Extending the previous examples, let's consider a case where we have a stream to an application with a:
   a. Trigger of every 5 minutes, starting at 12:05, 12:10, and 12:15
   b. Each trigger with a Windows of 10 minutes and a sliding value of 5 minutes.
   c. It is configured with a watermark of 10 minutes.
2. Then, we handled and watermarked late data of a "Dog" generated at 12:04 but arrived at 12:11, along with an "owl" at 12:13 and expected to be handled by the next trigger at 12:15.
3. Here, the dog count is only incremented for the window from 12:00 to 12:10 with a value of 3, missing that late dog data from 12:11.
4. The application should use the event time that was initially generated to update the older counts. Here, the intermediate state is held in memory for partial aggregates for an extended period so that late-arriving data can update the aggregates of old windows correctly. In this case, we have a watermark of 10 minutes configured, and considering the event at 12:13, we will create a sliding window from 12:03 through 12:13.
5. This will allow the next trigger at 12:15 to include the late "dog" data in the count aggregation with a total of 4.
6. Watermarking allows us to define a cutoff point, after which saved windows can be thrown away.

Control size of state

Late data MAY be dropped



Another key benefits of **watermarks** in Spark Structured Streaming is their ability to **control the size of state** during **stateful operations** such as windowed aggregations.

For example, consider a streaming query running at **12:25 PM** with a watermark delay threshold of **10 minutes**. This means the watermark is set to **12:14 PM**, based on the **maximum event time** seen so far (i.e., 12:24 PM) minus the configured delay.

Now, take a **window aggregation** covering the time range **12:00–12:10**. Since the **end of this window (12:10)** is **earlier than the watermark (12:14)**, Spark considers this window to be **expired**. Any events arriving after this point and belonging to this window would be deemed **too late** and thus **discarded**.

As a result, Spark can **safely remove the associated state** for the 12:00–12:10 window from memory. This **reclamation of state** prevents unbounded growth and ensures the system remains **resource-efficient**, avoiding excessive **RAM consumption**.

Thank you for completing this lesson and continuing your journey to develop your skills with us.