Chapter 22. Event-Time and Stateful Processing

Event Time

Event time is an important topic to cover discretely because Spark's DStream API does not support processing information with respect to event-time. At a higher level, in stream processing systems there are effectively two relevant times for each event:

- the time at which it actually occurred (*event time*)
- and the time that it was processed or reached the stream processing system (*processing time*)

Event time

- Event time is the time that is embedded in the data itself. It is most often, though not required to be, the time that an event actually occurs.
- This is important to use because it provides a more robust way of comparing events against one another.
- The challenge here is that event data can be late or out of order.
- This means that the stream processing system must be able to handle out-of-order or late data.

Processing time

- Processing time is the time at which the stream-processing system actually receives data.
- This is usually less important than event time because when it's processed is largely an implementation detail.
- This can't ever be out of order because it's a property of the streaming system at a certain time (not an external system like event time).

A more tangible example:

- Suppose that we have a datacenter located in San Francisco.
- An event occurs in two places at the same time: one in Ecuador, the other in Virginia

Due to the location of the datacenter, *the event in Virginia is likely to show up in our datacenter before the event in Ecuador*. If we were to analyze this data based on processing time, it would **appear** that the event in

Virginia occurred before the event in Ecuador: something that we know to be wrong. However, if we were to analyze the data based on **event time** (largely ignoring the time at which it's processed), we would see that these events occurred at the same time. As we mentioned, the fundamental idea is that the order of the series of events in the processing system does not guarantee an ordering in event time.

- Computer networks are unreliable. That means that events can be dropped, slowed down, repeated, or be sent without issue.
- Because individual events are not guaranteed to suffer one fate or the other, we must acknowledge that any number of things can happen to these events on the way from the source of the information to our stream processing system.
- For this reason, **we need to operate on event time** and look at the overall stream with reference to this information contained in the data rather than on when it arrives in the system.

Stateful Processing

- Stateful processing is only necessary when you need to use or update intermediate information (state) over longer periods of time (in either a microbatch or a record-at-a-time approach).
- This can happen when you are using event time or when you are performing an aggregation on a key, whether that involves event time or not.
- For the most part, when you're performing stateful operations, Spark handles all of this complexity for you.
    - For example, when you specify a grouping, Structured Streaming maintains and updates the information for you.
    - When performing a stateful operation, Spark stores the intermediate information in a state store.
    - Spark's current state store implementation is an **in-memory state store** that is made fault tolerant by storing intermediate state to the **checkpoint directory**.

Arbitrary Stateful Processing

- There are times when you need fine-grained control over what state should be stored, how it is updated, and when it should be removed, either explicitly or via a time-out.
- This is called arbitrary (or custom) stateful processing and Spark allows you to essentially store whatever information you like over the course of the processing of a stream - provides immense flexibility and power and allows for some complex business logic to be handled quite easily.
- Some examples:
    - You'd like to record information about user sessions on an ecommerce site.
        - For instance, you might want to track what pages users visit over the course of this session in order to provide recommendations in real time during their next session.
        - Naturally, these sessions have completely arbitrary start and stop times that are unique to that user.
    - Your company would like to report on errors in the web application but only if five events occur during a user's session.
        - You could do this with **count-based windows** that only emit a result if five events of some type occur.

- You'd like to deduplicate records over time.
  - To do so, you're going to need to keep track of every record that you see before deduplicating it.

When working with event time, it's just another column in our dataset, and that's really all we need to concern ourselves with

```
spark.conf.set("spark.sql.shuffle.partitions", 5)
static = spark.read.json("/FileStore/tables/activity_data")
streaming = spark\
  .readStream\
  .schema(static.schema)\
  .option("maxFilesPerTrigger", 10)\
  .json("/FileStore/tables/activity_data")
```

```
root
|-- Arrival_Time: long (nullable = true)
|-- Creation_Time: long (nullable = true)
|-- Device: string (nullable = true)
|-- Index: long (nullable = true)
|-- Model: string (nullable = true)
|-- User: string (nullable = true)
|-- gt: string (nullable = true)
|-- x: double (nullable = true)
|-- y: double (nullable = true)
|-- z: double (nullable = true)
```

In this dataset, there are two time-based columns.

- The **Creation_Time** column defines when an event was created, whereas the **Arrival_Time** defines when an event hit our servers somewhere upstream.
- We will use Creation_Time in this chapter.
- This example reads from a file but, as we saw in the previous chapter, it would be simple to change it to Kafka if you already have a cluster up and running.

Windows on Event Time

The first step in event-time analysis is to convert the timestamp column into the proper Spark SQL **timestamp** type.

- Our current column is unixtime nanoseconds (represented as a long), therefore we're going to have to do a little manipulation to get it into the proper format:

```
withEventTime = streaming.selectExpr("*", "cast(cast(Creation_Time as double)/1000000000 as
timestamp) as event_time")
```

- now prepared to do arbitrary operations on event time!
- Note how this experience is just like we'd do in batch operations—there's no special API or DSL.
- We simply use columns, just like we might in batch, the aggregation, and we're working with event time.

Tumbling Windows

The simplest operation is simply to count the number of occurrences of an event in a given window. Figure 22-2 depicts the process when performing a simple summation based on the input data and a key:

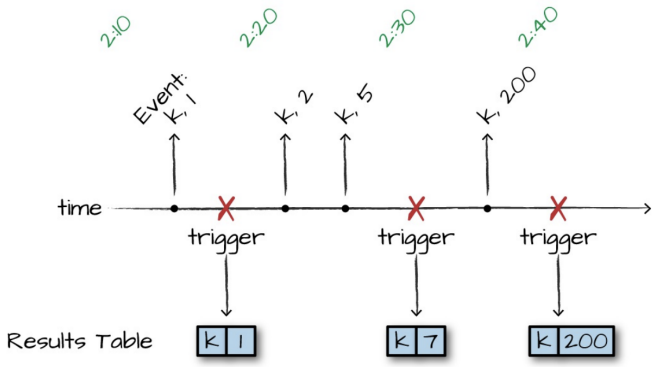
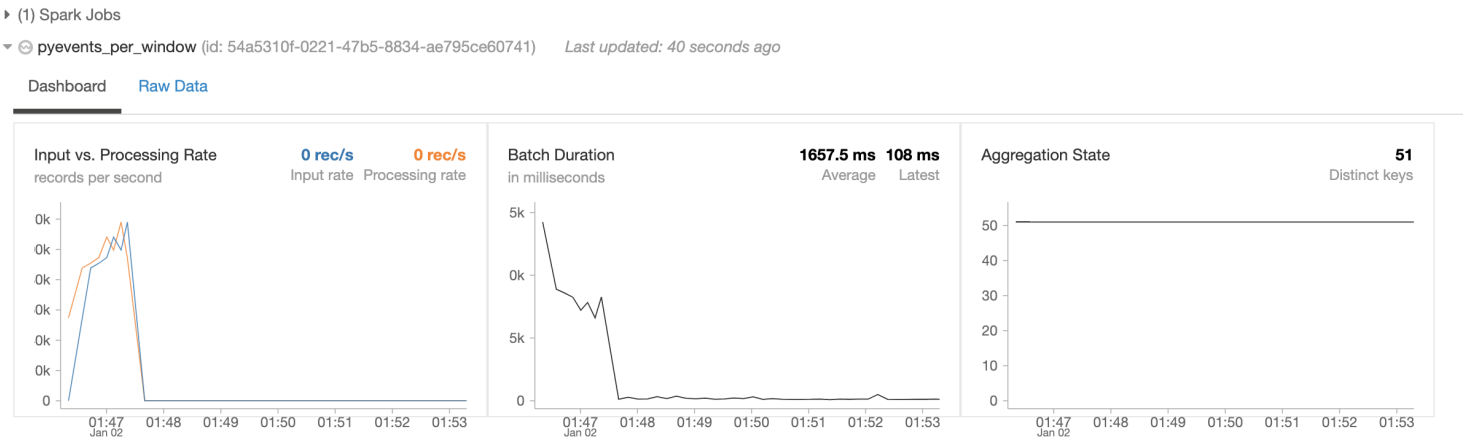


Figure 22-2. Tumbling Windows

We're performing **an aggregation of keys over a window of time**.

- We update the result table (depending on the output mode) when every trigger runs, which will operate on the data received since the last trigger.
- In the case of our actual dataset (and Figure 22-2), we'll do so in 10-minute windows without any overlap between them (each, and only one event can fall into one window).
- This will update in real time, as well, meaning that if new events were being added upstream to our system, Structured Streaming would update those counts accordingly.
- This is the complete output mode, Spark will output the entire result table regardless of whether we've seen the entire dataset:

```
from pyspark.sql.functions import window, col
withEventTime.groupBy(window(col("event_time"), "10 minutes")).count()\
  .writeStream\
  .queryName("pyevents_per_window")\
  .format("memory")\
  .outputMode("complete")\
  .start()
```




```
Out[5]: <pyspark.sql.streaming.StreamingQuery at 0x7fcbc2487c10>
```

md 8

```
%sql
SELECT * FROM pyevents_per_window
```

	window	count
1	{"start": "2015-02-24T11:50:00.000+0000", "end": "2015-02-24T12:00:00.000+0000"}	150773
2	{"start": "2015-02-24T13:00:00.000+0000", "end": "2015-02-24T13:10:00.000+0000"}	133323
3	{"start": "2015-02-23T12:30:00.000+0000", "end": "2015-02-23T12:40:00.000+0000"}	100853

schema we get from the previous query:

```
root
|-- window: struct (nullable = false)
| |-- start: timestamp (nullable = true)
| |-- end: timestamp (nullable = true)
|-- count: long (nullable = false)
```

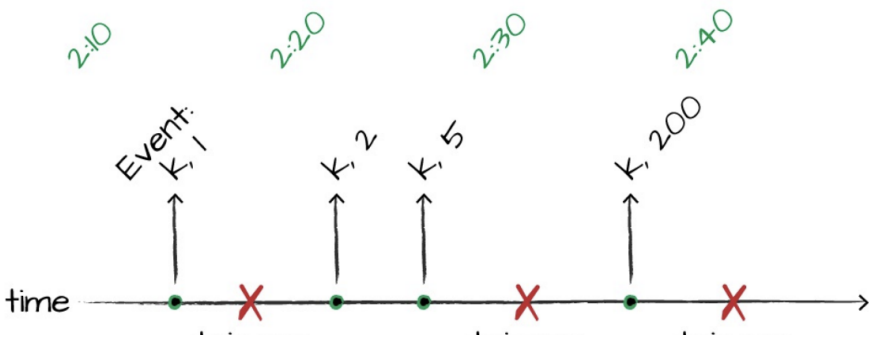
Notice how window is actually a **struct** (a complex type).

- Using this we can query this struct for the start and end times of a particular window.
- Of importance is the fact that we can also perform an aggregation on multiple columns, including the event time column.
- Just like we saw in the previous chapter, we can even perform these aggregations using methods like cube.
- While we won't repeat the fact that we can perform the multi-key aggregation below, this does apply to any window-style aggregation (or stateful computation) we would like:

```
from pyspark.sql.functions import window, col
withEventTime.groupBy(window(col("event_time"), "10 minutes"), "User").count()\
  .writeStream\
  .queryName("pyevents_per_window")\
  .format("memory")\
  .outputMode("complete")\
  .start()
```

Sliding windows

The previous example was simple counts in a given window. Another approach is that we can decouple the window from the starting time of the window. Figure 22-3 illustrates what we mean:



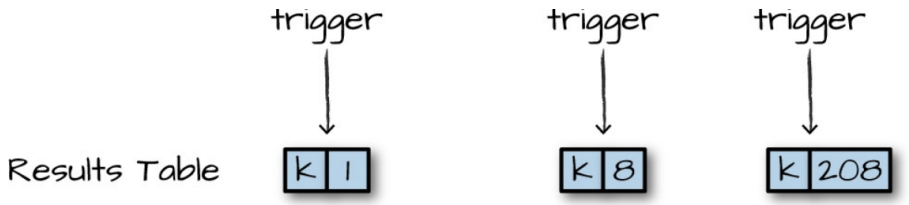


Figure 22-3. Sliding Windows

In the figure, we are running a sliding window through which we look at an hour increment, but we'd like to *get the state every 10 minutes*. This means that we will update the values over time and will include the last hours of data. In this example, we have **10-minute windows, starting every five minutes**. Therefore each event will fall into two different windows. You can tweak this further according to your needs:

```
from pyspark.sql.functions import window, col
withEventTime.groupBy(window(col("event_time"), "10 minutes", "5 minutes"))\
  .count()\
  .writeStream\
  .queryName("pyevents_per_window")\
  .format("memory")\
  .outputMode("complete")\
  .start()
```

Handling Late Data with Watermarks

The preceding examples are great, but they have a flaw. We never specified **how late** we expect to see data. This means that Spark is going to need to *store that intermediate data forever because we never specified a* **watermark**, or a time at which we don't expect to see any more data.

- This applies to all stateful processing that operates on event time.
- We must specify this watermark in order to age-out data in the stream (and, therefore, state) so that we don't overwhelm the system over a long period of time.
- Concretely, a watermark is an amount of time following a given event or set of events after which we do not expect to see any more data from that time.
- We know this can happen due to delays on the network, devices that lose a connection, or any number of other issues.
- In the DStreams API, there was no robust way to handle late data in this way—if an event occurred at a certain time but did not make it to the processing system by the time the batch for a given window started, it would show up in other processing batches.
- Structured Streaming remedies this: In event time and stateful processing, a given window's state or set of data is decoupled from a processing window.
- That means that as more events come in, Structured Streaming will continue to update a window with more information.

Let's return back to our event time example from the beginning of the chapter(the globe)

In this example, let's imagine that we frequently see some amount of delay from our customers in Latin America. Therefore, *we specify a watermark of 10 minutes.* When doing this, we instruct Spark that **any event that occurs more than 10 "event-time" minutes past a previous event should be ignored**.

- Conversely, this also states that we expect to see every event within 10 minutes.

After that, Spark should remove intermediate state and, depending on the output mode, do something with the result. As mentioned at the beginning of the chapter, *we need to specify watermarks because if we did not, we'd need to keep all of our windows around forever, expecting them to be updated forever.*

This brings us to the core question when working with event-time: **"how late do I expect to see data?"** The answer to this question will be the **watermark** that you'll configure for your data.

Returning to our dataset, if we know that we typically see data as produced downstream in minutes but we have seen delays in events up to five hours after they occur (perhaps the user lost cell phone connectivity), we'd specify the watermark in the following way:

```
from pyspark.sql.functions import window, col
withEventTime\
  .withWatermark("event_time", "30 minutes")\
  .groupBy(window(col("event_time"), "10 minutes", "5 minutes"))\
  .count()\
  .writeStream\
  .queryName("pyevents_per_window")\
  .format("memory")\
  .outputMode("complete")\
  .start()
```

Almost nothing changed about our query - we essentially just added another configuration. Now, Structured Streaming will wait until 30 minutes after the final timestamp of this 10-minute rolling window before it finalizes the result of that window. We can query our table and see the intermediate results because we're using **complete** mode—they'll be updated over time. In **append** mode, this information won't be output until the window closes.

Dropping Duplicates in a Stream

One of the more difficult operations in record-at-a-time systems is *removing duplicates from the stream.*

- Almost by definition, you must operate on a batch of records at a time in order to find duplicates—there's a **high coordination overhead** in the processing system.

Deduplication is an important tool in many applications, especially when messages might be delivered multiple times by upstream systems.

- A perfect example of this are Internet of Things (IoT) applications that have upstream producers generating messages in *nonstable* network environments, and the same message might end up being sent multiple times.

Your downstream applications and aggregations should be able to assume that there is only one of each message. Essentially, Structured Streaming makes it easy to take message systems that provide **at-least-once**

semantics, and convert them into **exactly-once** by *dropping duplicate messages as they come in*, based on **arbitrary keys.**

- To de-duplicate data, Spark will maintain a number of user specified keys and ensure that duplicates are ignored.

WARNING

- Just like other stateful processing applications, you need to specify a watermark to ensure that the maintained state does not grow infinitely over the course of your stream.

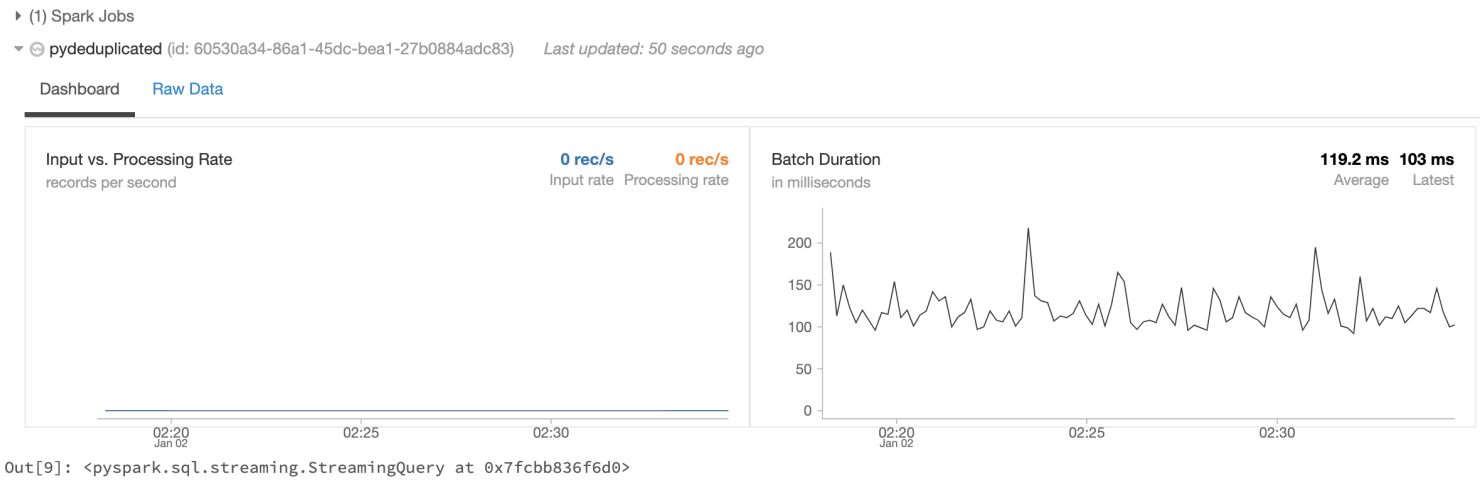
The goal here will be to de-duplicate the number of events per user by removing duplicate events.

- Notice how you need to specify the event time column as a duplicate column along with the column you should de-duplicate.

The core assumption is that duplicate events will have the same timestamp as well as identifier.
In this model, rows with two different timestamps are two different records:

```
from pyspark.sql.functions import expr

withEventTime\
  .withWatermark("event_time", "5 seconds")\
  .dropDuplicates(["User", "event_time"])\
  .groupBy("User")\
  .count()\
  .writeStream\
  .queryName("pydeduplicated")\
  .format("memory")\
  .outputMode("complete")\
  .start()
```




```
Out[9]: <pyspark.sql.streaming.StreamingQuery at 0x7fcbb836f6d0>
```

```
1 %sql
2 SELECT * FROM pydeduplicated
```

	User	count
1	a	80850

2	b	91230
3	c	77150
4	g	91679
5	h	77330