**Chapter 22. Event-Time and Stateful Processing**

[Chapter 21](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch21.html#s5c1---structured-streaming-basics) covered the core concepts and basic APIs; this chapter dives into event-time and stateful processing. Event-time processing is a hot topic because we analyze information with respect to the time that it was created, not processed. The key idea between this style of processing is that over the lifetime of the job, Spark will maintain relevant state that it can update over the course of the job before outputting it to the sink.

Let’s cover these concepts in greater detail before we begin working with code to show they work.

**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).

Those explanations are nice and abstract, so let’s use 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 (see Figure 22-1).



*Figure 22-1. Event Time Across the World*

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. This can be somewhat unintuitive, but is worth reinforcing. 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. This means that we hope to compare events based on the time at which those events occurred.

**Stateful Processing**

The other topic we need to cover in this chapter is stateful processing. Actually, we already demonstrated this many times in [Chapter 21](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch21.html#s5c1---structured-streaming-basics). 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. You simply specify the logic. 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**

The stateful processing capabilities described above are sufficient to solve many streaming problems. However, 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. This provides immense flexibility and power and allows for some complex business logic to be handled quite easily. Just as we did before, let’s ground this with 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.

Now that we’ve explained the core concepts that we’re going to need in this chapter, let’s cover all of this with some examples that you can follow along with and explain some of the important caveats that you need to consider when processing in this manner.

**Event-Time Basics**

Let’s begin with the same dataset from the previous chapter. When working with event time, it’s just another column in our dataset, and that’s really all we need to concern ourselves with; we simply use that column, as demonstrated here:

*// in Scala*

spark.conf.set("spark.sql.shuffle.partitions", 5)

**val** static **=** spark.read.json("/data/activity-data")

**val** streaming **=** spark

.readStream

.schema(static.schema)

.option("maxFilesPerTrigger", 10)

.json("/data/activity-data")

*# in Python*

spark.conf.set("spark.sql.shuffle.partitions", 5)

static = spark.read.json("/data/activity-data")

streaming = spark\

.readStream\

.schema(static.schema)\

.option("maxFilesPerTrigger", 10)\

.json("/data/activity-data")

streaming.printSchema()

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:

*// in Scala*

**val** withEventTime **=** streaming.selectExpr(

"\*",

"cast(cast(Creation\_Time as double)/1000000000 as timestamp) as event\_time")

*# in Python*

withEventTime = streaming\.selectExpr(

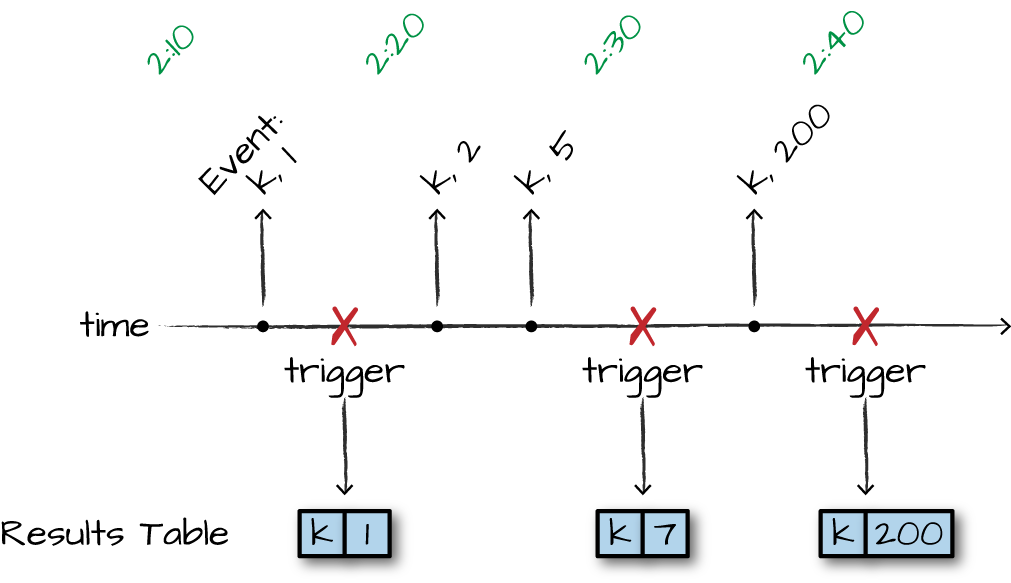
"\*",

"cast(cast(Creation\_Time as double)/1000000000 as timestamp) as event\_time")

We’re 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:

*// in Scala*

**import** **org.apache.spark.sql.functions.**{window, col}

withEventTime.groupBy(window(col("event\_time"), "10 minutes")).count()

.writeStream

.queryName("events\_per\_window")

.format("memory")

.outputMode("complete")

.start()

*# in Python*

**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()

Now we’re writing out to the in-memory sink for debugging, so we can query it with SQL after we have the stream running:

spark.sql("SELECT \* FROM events\_per\_window").printSchema()

**SELECT** \* **FROM** events\_per\_window

This shows us something like the following result, depending on the amount of data processed when you had run the query:

+---------------------------------------------+-----+

|window |count|

+---------------------------------------------+-----+

|[2015-02-23 10:40:00.0,2015-02-23 10:50:00.0]|11035|

|[2015-02-24 11:50:00.0,2015-02-24 12:00:00.0]|18854|

...

|[2015-02-23 13:40:00.0,2015-02-23 13:50:00.0]|20870|

|[2015-02-23 11:20:00.0,2015-02-23 11:30:00.0]|9392 |

+---------------------------------------------+-----+

For reference, here’s the 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:

*// in Scala*

**import** **org.apache.spark.sql.functions.**{window, col}

withEventTime.groupBy(window(col("event\_time"), "10 minutes"), "User").count()

.writeStream

.queryName("events\_per\_window")

.format("memory")

.outputMode("complete")

.start()

*# in Python*

**from** **pyspark.sql.functions** **import** window, col

withEventTime.groupBy(window(col("event\_time"), "10 minutes"), "User").count()\

.writeStream\

.queryName("pyevents\_per\_window")\

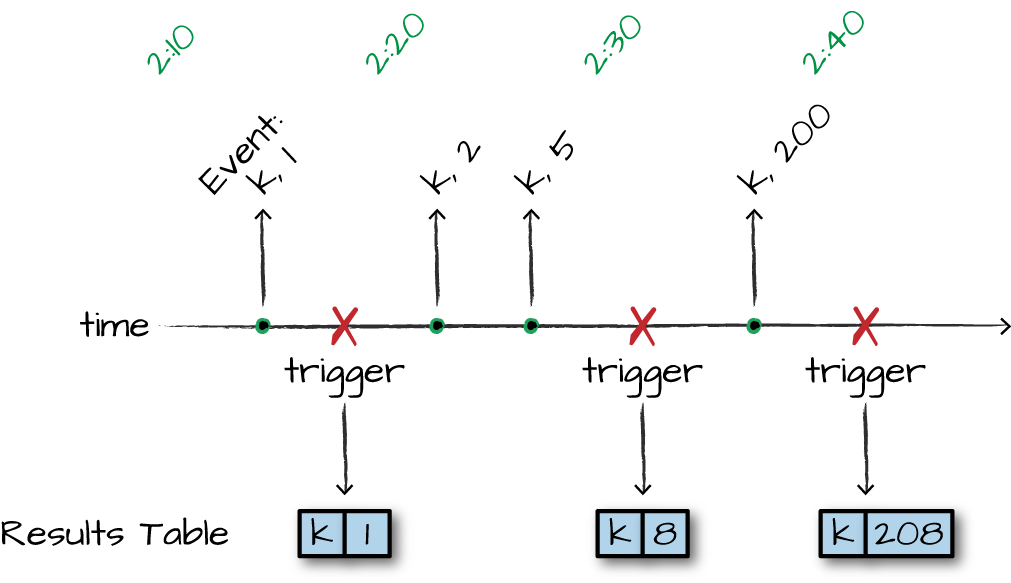
.format("memory")\

.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:

*// in Scala*

**import** **org.apache.spark.sql.functions.**{window, col}

withEventTime.groupBy(window(col("event\_time"), "10 minutes", "5 minutes"))

.count()

.writeStream

.queryName("events\_per\_window")

.format("memory")

.outputMode("complete")

.start()

*# in Python*

**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()

Naturally, we can query the in-memory table:

**SELECT** \* **FROM** events\_per\_window

This query gives us the following result. Note that the starting times for each window are now in 5-minute intervals instead of 10, like we saw in the previous query:

+---------------------------------------------+-----+

|window |count|

+---------------------------------------------+-----+

|[2015-02-23 14:15:00.0,2015-02-23 14:25:00.0]|40375|

|[2015-02-24 11:50:00.0,2015-02-24 12:00:00.0]|56549|

...

|[2015-02-24 11:45:00.0,2015-02-24 11:55:00.0]|51898|

|[2015-02-23 10:40:00.0,2015-02-23 10:50:00.0]|33200|

+---------------------------------------------+-----+

**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, shown now in Figure 22-4.



*Figure 22-4. Event Time Watermarking*

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:

*// in Scala*

**import** **org.apache.spark.sql.functions.**{window, col}

withEventTime

.withWatermark("event\_time", "5 hours")

.groupBy(window(col("event\_time"), "10 minutes", "5 minutes"))

.count()

.writeStream

.queryName("events\_per\_window")

.format("memory")

.outputMode("complete")

.start()

*# in Python*

**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()

It’s pretty amazing, but 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.

**SELECT** \* **FROM** events\_per\_window

+---------------------------------------------+-----+

|window |count|

+---------------------------------------------+-----+

|[2015-02-23 14:15:00.0,2015-02-23 14:25:00.0]|9505 |

|[2015-02-24 11:50:00.0,2015-02-24 12:00:00.0]|13159|

...

|[2015-02-24 11:45:00.0,2015-02-24 11:55:00.0]|12021|

|[2015-02-23 10:40:00.0,2015-02-23 10:50:00.0]|7685 |

+---------------------------------------------+-----+

At this point, you really know all that you need to know about handling late data. Spark does all of the heavy lifting for you. Just to reinforce the point, if you do not specify how late you think you will see data, then Spark will maintain that data in memory forever. Specifying a watermark allows it to free those objects from memory, allowing your stream to continue running for a long time.

**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.

Let’s begin the de-duplication process. 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:

*// in Scala*

**import** **org.apache.spark.sql.functions.expr**

withEventTime

.withWatermark("event\_time", "5 seconds")

.dropDuplicates("User", "event\_time")

.groupBy("User")

.count()

.writeStream

.queryName("deduplicated")

.format("memory")

.outputMode("complete")

.start()

*# in Python*

**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()

The result will be similar to the following and will continue to update over time as more data is read by your stream:

+----+-----+

|User|count|

+----+-----+

| a| 8085|

| b| 9123|

| c| 7715|

| g| 9167|

| h| 7733|

| e| 9891|

| f| 9206|

| d| 8124|

| i| 9255|

+----+-----+

**Arbitrary Stateful Processing**

The first section if this chapter demonstrates how Spark maintains information and updates windows based on our specifications. But things differ when you have more complex concepts of windows; this is, where arbitrary stateful processing comes in. This section includes several examples of different use cases along with examples that show you how you might go about setting up your business logic. Stateful processing is available only in Scala in Spark 2.2. This will likely change in the future.

When performing stateful processing, you might want to do the following:

* Create window based on counts of a given key
* Emit an alert if there is a number of events within a certain time frame
* Maintain user sessions of an undetermined amount of time and save those sessions to perform some analysis on later.

At the end of the day, there are two things you will want to do when performing this style of processing:

* Map over groups in your data, operate on each group of data, and generate at most a single row for each group. The relevant API for this use case is mapGroupsWithState.
* Map over groups in your data, operate on each group of data, and generate one or more rows for each group. The relevant API for this use case is flatMapGroupsWithState.

When we say “operate” on each group of data, that means that you can arbitrarily update each group independent of any other group of data. This means that you can define arbitrary window types that don’t conform to tumbling or sliding windows like we saw previously in the chapter. One important benefit that we get when we perform this style of processing is control over configuring time-outs on state. With windows and watermarks, it’s very simple: you simply time-out a window when the watermark passes the window start. This doesn’t apply to arbitrary stateful processing, because you manage the state based on user-defined concepts. Therefore, you need to properly time-out your state. Let’s discuss this a bit more.

**Time-Outs**

As mentioned in [Chapter 21](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch21.html#s5c1---structured-streaming-basics), a time-out specifies how long you should wait before timing-out some intermediate state. A time-out is a global parameter across all groups that is configured on a per-group basis. Time-outs can be either based on processing time (GroupStateTimeout.ProcessingTimeTimeout) or event time (GroupStateTimeout.EventTimeTimeout). When using time-outs, check for time-out first before processing the values. You can get this information by checking the state.hasTimedOut flag or checking whether the values iterator is empty. You need to set some state (i.e., state must be defined, not removed) for time-outs to be set.

With a time-out based on processing time, you can set the time-out duration by calling GroupState.setTimeoutDuration (we’ll see code examples of this later in this section of the chapter). The time-out will occur when the clock has advanced by the set duration. Guarantees provided by this time-out with a duration of *D* ms are as follows:

* Time-out will never occur before the clock time has advanced by *D* ms
* Time-out will occur eventually when there is a trigger in the query (i.e., after *D* ms). So there is a no strict upper bound on when the time-out would occur. For example, the trigger interval of the query will affect when the time-out actually occurs. If there is no data in the stream (for any group) for a while, there won’t be any trigger and the time-out function call will not occur until there is data.

Because the processing time time-out is based on the clock time, it is affected by the variations in the system clock. This means that time zone changes and clock skew are important variables to consider.

With a time-out based on event time, the user also must specify the event-time watermark in the query using watermarks. When set, data older than the watermark is filtered out. As the developer, you can set the timestamp that the watermark should reference by setting a time-out timestamp using the GroupState.setTimeoutTimestamp(...) API. The time-out would occur when the watermark advances beyond the set timestamp. Naturally, you can control the time-out delay by either specifying longer watermarks or simply updating the time-out as you process your stream. Because you can do this in arbitrary code, you can do it on a per-group basis. The guarantee provided by this time-out is that it will never occur before the watermark has exceeded the set time-out.

Similar to processing-time time-outs, there is a no strict upper bound on the delay when the time-out actually occurs. The watermark can advance only when there is data in the stream, and the event time of the data has actually advanced.

**NOTE**

We mentioned this a few moments ago, but it’s worth reinforcing. Although time-outs are important, they might not always function as you expect. For instance, as of this writing, Structured Streaming does not have asynchronous job execution, which means that Spark will not output data (or time-out data) between the time that a epoch finishes and the next one starts, because it is not processing any data at that time. Also, if a processing batch of data has no records (keep in mind this is a batch, not a group), there are no updates and there cannot be an event-time time-out. This might change in future versions.

**Output Modes**

One last “gotcha” when working with this sort of arbitrary stateful processing is the fact that not all output modes discussed in [Chapter 21](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch21.html#s5c1---structured-streaming-basics) are supported. This is sure to change as Spark continues to change, but, as of this writing, mapGroupsWithState supports only the update output mode, whereas flatMapGroupsWithState supports append and update. append mode means that only after the time-out (meaning the watermark has passed) will data show up in the result set. This does not happen automatically, it is your responsibility to output the proper row or rows.

Please see [Table 21-1](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch21.html#using_output_modes) to see which output modes can be used when.

**mapGroupsWithState**

Our first example of stateful processing uses a feature called mapGroupsWithState. This is similar to a user-defined aggregation function that takes as input an update set of data and then resolves it down to a specific key with a set of values. There are several things you’re going to need to define along the way:

* Three class definitions: an input definition, a state definition, and optionally an output definition.
* A function to update the state based on a key, an iterator of events, and a previous state.
* A time-out parameter (as described in the time-outs section).

With these objects and definitions, you can control arbitrary state by creating it, updating it over time, and removing it. Let’s begin with a example of simply updating the key based on a certain amount of state, and then move onto more complex things like sessionization.

Because we’re working with sensor data, let’s find the first and last timestamp that a given user performed one of the activities in the dataset. This means that the key we will be grouping on (and mapping on) is a user and activity combination.

**NOTE**

When you use mapGroupsWithState, the output of the dream will contain only one row per key (or group) at all times. If you would like each group to have multiple outputs, you should use flatMapGroupsWithState (covered shortly).

Let’s establish the input, state, and output definitions:

**case** **class** **InputRow**(user**:String**, timestamp**:java.sql.Timestamp**, activity**:String**)

**case** **class** **UserState**(user**:String**,

**var** activity**:String**,

**var** start**:java.sql.Timestamp**,

**var** end**:java.sql.Timestamp**)

For readability, set up the function that defines how you will update your state based on a given row:

**def** updateUserStateWithEvent(state**:UserState**, input**:InputRow**)**:UserState** = {

**if** (**Option**(input.timestamp).isEmpty) {

**return** state

}

**if** (state.activity == input.activity) {

**if** (input.timestamp.after(state.end)) {

state.end **=** input.timestamp

}

**if** (input.timestamp.before(state.start)) {

state.start **=** input.timestamp

}

} **else** {

**if** (input.timestamp.after(state.end)) {

state.start **=** input.timestamp

state.end **=** input.timestamp

state.activity **=** input.activity

}

}

state

}

Now, write the function that defines the way state is updated based on an epoch of rows:

**import** **org.apache.spark.sql.streaming.**{**GroupStateTimeout**, **OutputMode**, **GroupState**}

**def** updateAcrossEvents(user**:String**,

inputs**:** **Iterator**[**InputRow**],

oldState**:** **GroupState**[**UserState**])**:UserState** = {

**var** state**:UserState** = **if** (oldState.exists) oldState.get **else** **UserState**(user,

"",

**new** java.sql.**Timestamp**(6284160000000L),

**new** java.sql.**Timestamp**(6284160L)

)

*// we simply specify an old date that we can compare against and*

*// immediately update based on the values in our data*

**for** (input **<-** inputs) {

state **=** updateUserStateWithEvent(state, input)

oldState.update(state)

}

state

}

When we have that, it’s time to start your query by passing in the relevant information. The one thing that you’re going to have to add when you specify mapGroupsWithState is whether you need to time-out a given group’s state. This just gives you a mechanism to control what should be done with state that receives no update after a certain amount of time. In this case, you want to maintain state indefinitely, so specify that Spark should not time-out.

Use the update output mode so that you get updates on the user activity:

**import** **org.apache.spark.sql.streaming.GroupStateTimeout**

withEventTime

.selectExpr("User as user",

"cast(Creation\_Time/1000000000 as timestamp) as timestamp", "gt as activity")

.as[**InputRow**]

.groupByKey(**\_**.user)

.mapGroupsWithState(**GroupStateTimeout**.**NoTimeout**)(updateAcrossEvents)

.writeStream

.queryName("events\_per\_window")

.format("memory")

.outputMode("update")

.start()

**SELECT** \* **FROM** events\_per\_window **order** **by** **user**, **start**

Here’s a sample of our result set:

+----+--------+--------------------+--------------------+

|user|activity| start| end|

+----+--------+--------------------+--------------------+

| a| bike|2015-02-23 13:30:...|2015-02-23 14:06:...|

| a| bike|2015-02-23 13:30:...|2015-02-23 14:06:...|

...

| d| bike|2015-02-24 13:07:...|2015-02-24 13:42:...|

+----+--------+--------------------+--------------------+

An interesting aspect of our data is that the last activity performed at any given time is “bike.” This is related to how the experiment was likely run, in which they had each participant perform the same activities in order.

**EXAMPLE: COUNT-BASED WINDOWS**

Typical window operations are built from start and end times for which all events that fall in between those two points contribute to the counting or summation that you’re performing. However, there are times when instead of creating windows based on time, you’d rather create them based on a number of events regardless of state and event times, and perform some aggregation on that window of data. For example, we may want to compute a value for every 500 events received, regardless of when they are received.

The next example analyzes the activity dataset from this chapter and outputs the average reading of each device periodically, creating a window based on the *count* of events and outputting it each time it has accumulated 500 events for that device. You define two case classes for this task: the input row format (which is simply a device and a timestamp); and the state and output rows (which contain the current count of records collected, device ID, and an array of readings for the events in the window).

Here are our various, self-describing case class definitions:

**case** **class** **InputRow**(device**:** **String**, timestamp**:** **java.sql.Timestamp**, x**:** **Double**)

**case** **class** **DeviceState**(device**:** **String**, **var** values**:** **Array**[**Double**],

**var** count**:** **Int**)

**case** **class** **OutputRow**(device**:** **String**, previousAverage**:** **Double**)

Now, you can define the function to update the individual state based on a single input row. You could write this inline or in a number of other ways, but this example makes it easy to see exactly how you update based on a given row:

**def** updateWithEvent(state**:DeviceState**, input**:InputRow**)**:DeviceState** = {

state.count += 1

*// maintain an array of the x-axis values*

state.values **=** state.values ++ **Array**(input.x)

state

}

Now it’s time to define the function that updates across a series of input rows. Notice in the example that follows that we have a specific key, the iterator of inputs, and the old state, and we update that old state over time as we receive new events. This, in turn, will return our output rows with the updates on a per-device level based on the number of counts it sees. This case is quite straightforward, after a given number of events, you update the state and reset it. You then create an output row. You can see this row in the output table:

**import** **org.apache.spark.sql.streaming.**{**GroupStateTimeout**, **OutputMode**,

**GroupState**}

**def** updateAcrossEvents(device**:String**, inputs**:** **Iterator**[**InputRow**],

oldState**:** **GroupState**[**DeviceState**])**:Iterator**[**OutputRow**] **=** {

inputs.toSeq.sortBy(**\_**.timestamp.getTime).toIterator.flatMap { input **=>**

**val** state **=** **if** (oldState.exists) oldState.get

**else** **DeviceState**(device, **Array**(), 0)

**val** newState **=** updateWithEvent(state, input)

**if** (newState.count >= 500) {

*// One of our windows is complete; replace our state with an empty*

*// DeviceState and output the average for the past 500 items from*

*// the old state*

oldState.update(**DeviceState**(device, **Array**(), 0))

**Iterator**(**OutputRow**(device,

newState.values.sum / newState.values.length.toDouble))

}

**else** {

*// Update the current DeviceState object in place and output no*

*// records*

oldState.update(newState)

**Iterator**()

}

}

}

Now you can run your stream. You will notice that you need to explicitly state the output mode, which is append. You also need to set a GroupStateTimeout. This time-out specifies the amount of time you want to wait before a window should be output as complete (even if it did not reach the required count). In that case, set an infinite time-out, meaning if a device never gets to that required 500 count threshold, it will maintain that state forever as “incomplete” and not output it to the result table.

By specifying both of those parameters you can pass in the updateAcrossEvents function and start the stream:

**import** **org.apache.spark.sql.streaming.GroupStateTimeout**

withEventTime

.selectExpr("Device as device",

"cast(Creation\_Time/1000000000 as timestamp) as timestamp", "x")

.as[**InputRow**]

.groupByKey(**\_**.device)

.flatMapGroupsWithState(**OutputMode**.**Append**,

**GroupStateTimeout**.**NoTimeout**)(updateAcrossEvents)

.writeStream

.queryName("count\_based\_device")

.format("memory")

.outputMode("append")

.start()

After you start the stream, it’s time to query it. Here are the results:

**SELECT** \* **FROM** count\_based\_device

+--------+--------------------+

| device| previousAverage|

+--------+--------------------+

|nexus4\_1| 4.660034012E-4|

|nexus4\_1|0.001436279298199...|

...

|nexus4\_1|1.049804683999999...|

|nexus4\_1|-0.01837188737960...|

+--------+--------------------+

You can see the values change over each of those windows as you append new data to the result set.

**flatMapGroupsWithState**

Our second example of stateful processing will use a feature called flatMapGroupsWithState. This is quite similar to mapGroupsWithState except that rather than just having a single key with at most one output, a single key can have many outputs. This can provide us a bit more flexibility and the same fundamental structure as mapGroupsWithState applies. Here’s what we’ll need to define.

* Three class definitions: an input definition, a state definition, and optionally an output definition.
* A function to update the state based on a key, an iterator of events, and a previous state.
* A time-out parameter (as described in the time-outs section).

With these objects and definitions, we can control arbitrary state by creating it, updating it over time, and removing it. Let’s start with an example of sessionization.

**EXAMPLE: SESSIONIZATION**

Sessions are simply unspecified time windows with a series of events that occur. Typically, you want to record these different events in an array in order to compare these sessions to other sessions in the future. In a session, you will likely have arbitrary logic to maintain and update your state over time as well as certain actions to define when state ends (like a count) or a simple time-out. Let’s build on the previous example and define it a bit more strictly as a session.

At times, you might have an explicit session ID that you can use in your function. This obviously makes it much easier because you can just perform a simple aggregation and might not even need your own stateful logic. In this case, you’re creating sessions on the fly from a user ID and some time information and if you see no new event from that user in five seconds, the session terminates. You’ll also notice that this code uses time-outs differently than we have in other examples.

You can follow the same process of creating your classes, defining our single event update function and then the multievent update function:

**case** **class** **InputRow**(uid**:String**, timestamp**:java.sql.Timestamp**, x**:Double**,

activity**:String**)

**case** **class** **UserSession**(**val** uid**:String**, **var** timestamp**:java.sql.Timestamp**,

**var** activities**:** **Array**[**String**], **var** values**:** **Array**[**Double**])

**case** **class** **UserSessionOutput**(**val** uid**:String**, **var** activities**:** **Array**[**String**],

**var** xAvg**:Double**)

**def** updateWithEvent(state**:UserSession**, input**:InputRow**)**:UserSession** = {

*// handle malformed dates*

**if** (**Option**(input.timestamp).isEmpty) {

**return** state

}

state.timestamp **=** input.timestamp

state.values **=** state.values ++ **Array**(input.x)

**if** (!state.activities.contains(input.activity)) {

state.activities **=** state.activities ++ **Array**(input.activity)

}

state

}

**import** **org.apache.spark.sql.streaming.**{**GroupStateTimeout**, **OutputMode**,

**GroupState**}

**def** updateAcrossEvents(uid**:String**,

inputs**:** **Iterator**[**InputRow**],

oldState**:** **GroupState**[**UserSession**])**:Iterator**[**UserSessionOutput**] **=** {

inputs.toSeq.sortBy(**\_**.timestamp.getTime).toIterator.flatMap { input **=>**

**val** state **=** **if** (oldState.exists) oldState.get **else** **UserSession**(

uid,

**new** java.sql.**Timestamp**(6284160000000L),

**Array**(),

**Array**())

**val** newState **=** updateWithEvent(state, input)

**if** (oldState.hasTimedOut) {

**val** state **=** oldState.get

oldState.remove()

**Iterator**(**UserSessionOutput**(uid,

state.activities,

newState.values.sum / newState.values.length.toDouble))

} **else** **if** (state.values.length > 1000) {

**val** state **=** oldState.get

oldState.remove()

**Iterator**(**UserSessionOutput**(uid,

state.activities,

newState.values.sum / newState.values.length.toDouble))

} **else** {

oldState.update(newState)

oldState.setTimeoutTimestamp(newState.timestamp.getTime(), "5 seconds")

**Iterator**()

}

}

}

You’ll see in this one that we only expect to see an event at most five seconds late. Anything other than that and we will ignore it. We will use an EventTimeTimeout to set that we want to time-out based on the event time in this stateful operation:

**import** **org.apache.spark.sql.streaming.GroupStateTimeout**

withEventTime.where("x is not null")

.selectExpr("user as uid",

"cast(Creation\_Time/1000000000 as timestamp) as timestamp",

"x", "gt as activity")

.as[**InputRow**]

.withWatermark("timestamp", "5 seconds")

.groupByKey(**\_**.uid)

.flatMapGroupsWithState(**OutputMode**.**Append**,

**GroupStateTimeout**.**EventTimeTimeout**)(updateAcrossEvents)

.writeStream

.queryName("count\_based\_device")

.format("memory")

.start()

Querying this table will show you the output rows for each user over this time period:

**SELECT** \* **FROM** count\_based\_device

+---+--------------------+--------------------+

|uid| activities| xAvg|

+---+--------------------+--------------------+

| a| [stand, null, sit]|-9.10908533566433...|

| a| [sit, null, walk]|-0.00654280428601...|

...

| c|[null, stairsdown...|-0.03286657789999995|

+---+--------------------+--------------------+

As you might expect, sessions that have a number of activities in them have a higher x-axis gyroscope value than ones that have fewer activities. It should be trivial to extend this example to problem sets more relevant to your own domain, as well.

**Conclusion**

This chapter covered some of the more advanced topics in Structured Streaming, including event time and stateful processing. This is effectively the user guide to help you actually build out your application logic and turn it into something that provides value. Next, we will discuss what we’ll need to do in order to take this application to production and maintain and update it over time.