Labs 3. Structured Streaming Basics

Structured Streaming in Action

We're going to be working with the Heterogeneity Human Activity Recognition Dataset. The data consists of smartphone and smartwatch sensor readings from a variety of devices—specifically, the accelerometer and gyroscope, sampled at the highest possible frequency supported by the devices. Readings from these sensors were recorded while users performed activities like biking, sitting, standing, walking, and so on. There are several different smartphones and smartwatches used, and nine total users. You can download the data from the activity data folder.

1. **Let's read in the static version of the dataset as a DataFrame:**

```
# in Python
static = spark.read.json("/data/activity-
data/")
dataSchema = static.schema
```

Here's the schema:

```
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)
|-- _corrupt_record: string (nullable =
true)
|-- gt: string (nullable = true)
|-- x: double (nullable = true)
|-- y: double (nullable = true)
|-- z: double (nullable = true)
```

Here's a sample of the DataFrame:

```
+-------------+-----------------+--------+-----+------+----+--------+-----+-----
| Arrival_Time|  Creation_Time|  Device|Index| Model|User|_c...ord|.  gt|   x
|1424696634224|142469663222623685|nexus4_1|   62|nexus4|  a|    null|stand|-0...
...
|1424696660715|142469665872381726|nexus4_1| 2342|nexus4|  a|    null|stand|-0...
+-------------+-----------------+--------+-----+------+----+--------+-----+-----
```

You can see in the preceding example, which includes a number of timestamp columns, models, user, and device information. The `gt` field specifies what activity the user was doing at that time.

2. Let’s create a streaming version of the same Dataset, which will read each input file in the dataset one by one as if it was a stream.

```
# in Python
streaming =
spark.readStream.schema(dataSchema).option("maxFilesPerTrigger",
1)\
.json("/data/activity-data")
```

> **NOTE**
>
> Essentially `maxFilesPerTrigger` allows to control how quickly Spark will read all of the files in the folder. By specifying this value lower, we’re artificially limiting the flow of the stream to one file per trigger. This helps us demonstrate how Structured Streaming runs incrementally in our example, but probably isn’t something you’d use in production.

3. We'll show one simple transformation—we will group and count data by the `gt` column, which is the activity being performed by the user at that point in time:

```
# in Python
activityCounts = streaming.groupBy("gt").count()
```

Because this code is being written in local mode on a small machine, we are going to set the shuffle partitions to a small value to avoid creating too many shuffle partitions:

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

4. As mentioned in the course, we should specify an output destination, or output sink for our result of this query. For this basic example, we are going to write to a *memory sink* which keeps an in-memory table of the results.

In the process of specifying this sink, we're going to need to define *how* Spark will output that data. In this example, we use the *complete* output mode. This mode rewrites all of the keys along with their counts after every trigger:

```
# in Python
activityQuery =
activityCounts.writeStream.queryName("activity_counts")\
.format("memory").outputMode("complete")\
.start()
```

We are now writing out our stream! You'll notice that we set a *unique* query name to represent this stream, in this case `activity_counts`. We specified our format as an in-memory table and we set the output mode.

5. Spark lists this stream, and other active ones, under the active streams in our `SparkSession`. We can see a list of those streams by running the following:

```
spark.streams.active
```

Spark also assigns each stream a UUID, so if need be you could iterate through the list of running streams and select the above one. In this case, we assigned it to a variable, so that's not necessary.

6. Now that this stream is running, we can experiment with the results by querying the in-memory table it is maintaining of the current output of our streaming aggregation. This table will be called `activity_counts`, the same as the stream. To see the current data in this output table, we simply need to query it! We'll do this in a simple loop that will print the results of the streaming query every second:

```
# in Python
from time import sleep
for x in range(5):
```

```
    spark.sql("SELECT * FROM activity_counts").show()
    sleep(1)
```

As the preceding queries run, you should see the counts for each activity change over time. For instance, the first `show` call displays the following result (because we queried it while the stream was reading the first file):

```
+---+     +
| gt|count|
+---+     +
+---+     +
```

The previous `show` call shows the following result—note that the result will probably vary when you're running this code personally because you will likely start it at a different time:

```
+----------+  +
|    gt|count|
+----------+  +
|    sit| 8207|
...
|    null| 6966|
|    bike| 7199|
+----------+  +
```

With this simple example, the power of Structured Streaming should become clear. You can take the same operations that you use in batch and run them on a stream of data with very few code changes (essentially just specifying that it's a stream). The rest of this labs touches on some of the details about the various manipulations, sources, and sinks that you can use with Structured Streaming.

Transformations on Streams

Selections and Filtering

All select and filter transformations are supported in Structured Streaming, as are all DataFrame functions and individual column manipulations.

7. We show a simple example using selections and filtering below. In this case, because we are not updating any keys over time, we will use the Append output mode, so that new results are appended to the output table:

```
# in Python
from pyspark.sql.functions import expr
```

```
simpleTransform = streaming.withColumn("stairs", expr("gt like
'%stairs%'"))\
.where("stairs")\
.where("gt is not null")\
.select("gt", "model", "arrival_time", "creation_time")\
.writeStream\
.queryName("simple_transform")\
.format("memory")\
.outputMode("append")\
.start()
```

Aggregations

8. Structured Streaming has excellent support for aggregations. You can specify arbitrary aggregations, as you saw in the Structured APIs. For example, you can use a more exotic aggregation, like a cube, on the phone model and activity and the average x, y, z accelerations of our sensor:

```
# in Python
deviceModelStats = streaming.cube("gt", "model").avg()\
.drop("avg(Arrival_time)")\
.drop("avg(Creation_Time)")\
.drop("avg(Index)")\
.writeStream.queryName("device_counts").format("memory")\
.outputMode("complete")\
.start()
```

9. Querying that table allows us to see the results:

```
SELECT * FROM device_counts
```

```
+----------+------+------------------+--------------------+   +
|  gt| model|  avg(x)|     avg(y)|     avg(z)|
+----------+------+------------------+--------------------+   +
|  sit|  null|-3.682775300344...|1.242033094787975...|-4.22021191297611   |
|  stand|  null|-4.415368069618...|-5.30657295890281...|2.264837548081631 |
...
|  walk|nexus4|-0.007342235359...|0.004341030525168...|-6.01620400184307  |
|stairsdown|nexus4|0.0309175199508...|-0.02869185568293...| 0.11661923308518365|
...
+----------+------+------------------+--------------------+   +
```

Joins

10. As of Apache Spark 2.2, Structured Streaming supports joining streaming DataFrames to static DataFrames. Spark 2.3 will add the ability to join multiple streams together. You can do multiple column joins and supplement streaming data with that from static data sources:

```
# in Python
historicalAgg = static.groupBy("gt", "model").avg()
deviceModelStats = streaming.drop("Arrival_Time",
"Creation_Time", "Index")\
.cube("gt", "model").avg()\
.join(historicalAgg, ["gt", "model"])\
.writeStream.queryName("device_counts").format("memory")\
.outputMode("complete")\
.start()
```

In Spark 2.2, full outer joins, left joins with the stream on the right side, and right joins with the stream on the left are not supported. Structured Streaming also does not yet support stream-to- stream joins, but this is also a feature under active development.

Input and Output

Reading from the Kafka Source

To read, you first need to choose one of the following options: `assign, subscribe`, or `subscribePattern`. Only one of these can be present as an option when you go to read from Kafka.

- `Assign` is a fine-grained way of specifying not just the topic but also the topic partitions from which you would like to read. This is specified as a JSON string `{"topicA": [0,1],"topicB":[2,4]}`.
- `subscribe` and `subscribePattern` are ways of subscribing to one or more topics either by specifying a list of topics (in the former) or via a pattern (via the latter). Second, you will need to specify the `kafka.bootstrap.servers` that Kafka provides to connect to the service.

After you have specified your options, you have several other options to specify:

- `startingOffsets` and `endingOffsets`

The start point when a query is started, either `earliest`, which is from the earliest offsets; `latest`, which is just from the latest offsets; or a JSON string specifying a starting offset for each `TopicPartition`. In the JSON, `-2` as an offset can be used to refer to earliest, `-1` to latest. For example, the JSON specification could be `{"topicA": {"0":23,"1":-1},"topicB":{"0":-2}}`. This applies only when a new Streaming query is started, and that resuming will always pick up from where the

query left off. Newly discovered partitions during a query will start at earliest. The ending offsets for a given query.

- `failOnDataLoss`

Whether to fail the query when it's possible that data is lost (e.g., topics are deleted, or offsets are out of range). This might be a false alarm. You can disable it when it doesn't work as you expected. The default is `true`.

- `maxOffsetsPerTrigger`

The total number of offsets to read in a given trigger.

11. There are also options for setting Kafka consumer timeouts, fetch retries, and intervals. To read from Kafka, do the following in Structured Streaming:

```
# in Python
# Subscribe to 1 topic
df1 = spark.readStream.format("kafka")\
.option("kafka.bootstrap.servers",
"host1:port1,host2:port2")\
.option("subscribe", "topic1")\
.load()
# Subscribe to multiple topics
df2 = spark.readStream.format("kafka")\
.option("kafka.bootstrap.servers",
"host1:port1,host2:port2")\
.option("subscribe", "topic1,topic2")\
.load()
# Subscribe to a pattern
df3 = spark.readStream.format("kafka")\
.option("kafka.bootstrap.servers",
"host1:port1,host2:port2")\
.option("subscribePattern", "topic.*")\
.load()
```

Each row in the source will have the following schema:

- key: binary
- value: binary
- topic: string
- partition: int
- offset: long
- timestamp: long

Each message in Kafka is likely to be serialized in some way. Using native Spark functions in the Structured APIs, or a User-Defined Function (UDF), you can parse

the message into a more structured format analysis. A common pattern is to use JSON or Avro to read and write to Kafka.

Writing to the Kafka Sink

12. Writing to Kafka queries is largely the same as reading from them except for fewer parameters. You'll still need to specify the Kafka bootstrap servers, but the only other option you will need to supply is either a column with the topic specification or supply that as an option. For example, the following writes are equivalent:

```
# in Python
df1.selectExpr("topic", "CAST(key AS STRING)", "CAST(value
AS STRING)")\
.writeStream\
.format("kafka")\
.option("kafka.bootstrap.servers",
"host1:port1,host2:port2")\
.option("checkpointLocation", "/to/HDFS-compatible/dir")\
.start()
df1.selectExpr("CAST(key AS STRING)", "CAST(value AS
STRING)")\
.writeStream\
.format("kafka")\
.option("kafka.bootstrap.servers",
"host1:port1,host2:port2")\
.option("checkpointLocation", "/to/HDFS-compatible/dir")\
.option("topic", "topic1")\
.start()
```

Sources and sinks for testing

Spark also includes several test sources and sinks that you can use for prototyping or debugging your streaming queries (these should be used only during development and not in production scenarios, because they do not provide end-to-end fault tolerance for your application):

Socket source

> The socket source allows you to send data to your Streams via TCP sockets. To start one, specify a host and port to read data from. Spark will open a new TCP connection to read from that address. The socket source should *not* be used in production because the socket sits on the driver and does not provide end-to-end fault-tolerance guarantees.

13. Here is a short example of setting up this source to read from `localhost:9999`:

```
# in Python
socketDF = spark.readStream.format("socket")\
.option("host", "localhost").option("port", 9999).load()
```

If you'd like to actually write data to this application, you will need to run a server that listens on port 9999. On Unix-like systems, you can do this using the NetCat utility, which will let you type text into the first connection that is opened to port 9999. Run the command below before starting your Spark application, then write into it:

```
nc -lk 9999
```

The socket source will return a table of text strings, one per line in the input data.

Console sink

The console sink allows you to write out some of your streaming query to the console. This is useful for debugging but is not fault-tolerant. Writing out to the console is simple and only prints some rows of your streaming query to the console. This supports both append and complete output modes:

```
activityCounts.format("console").write()
```

Memory sink

The memory sink is a simple source for testing your streaming system. It's similar to the console sink except that rather than printing to the console, it collects the data to the driver and then makes the data available as an in-memory table that is available for interactive querying. This sink is not fault tolerant, and you shouldn't use it in production, but is great for testing and querying your stream during development. This supports both append and complete output modes:

```
// in Scala
activityCounts.writeStream.format("memory").queryName("my_
device_table")
```

If you do want to output data to a table for interactive SQL queries in production, the authors recommend using the Parquet file sink on a distributed file system (e.g., S3). You can then query the data from any Spark application.

When Data Is Output (Triggers)

To control when data is output to our sink, we set a *trigger*. By default, Structured Streaming will start data as soon as the previous trigger completes processing. You can use triggers to ensure that you do not overwhelm your output sink with too many updates or to try and control file sizes in the output.

Processing time trigger

14. For the processing time trigger, we simply specify a duration as a string (you may also use a

`Duration` in Scala or `TimeUnit` in Java). We'll show the string format below.

```
# in Python
activityCounts.writeStream.trigger(processingTime='5
seconds')\
.format("console").outputMode("complete").start()
```

The `ProcessingTime` trigger will wait for multiples of the given duration in order to output data. For example, with a trigger duration of one minute, the trigger will fire at 12:00, 12:01, 12:02, and so on. If a trigger time is missed because the previous processing has not yet completed, then Spark will wait until the next trigger point (i.e., the next minute), rather than firing immediatelyafter the previous processing completes.

Once trigger

You can also just run a streaming job once by setting that as the trigger. This might seem like a weird case, but it's actually extremely useful in both development and production. During development, you can test your application on just one trigger's worth of data at a time. During production, the Once trigger can be used to run your job manually at a low rate (e.g., import new data into a summary table just occasionally). Because Structured Streaming still fully tracks all the input files processed and the state of the computation, this is easier than writing your own custom logic to track this in a batch job, and saves a lot of resources over running a continuous job 24/7:

```
# in Python
activityCounts.writeStream.trigger(once=True)\
.format("console").outputMode("complete").start()
```