**Chapter 23. Structured Streaming in Production**

The previous chapters of this part of the book have covered Structured Streaming from a user’s perspective. Naturally this is the core of your application. This chapter covers some of the operational tools needed to run Structured Streaming robustly in production after you’ve developed an application.

Structured Streaming was marked as production-ready in Apache Spark 2.2.0, meaning that this release has all the features required for production use and stabilizes the API. Many organizations are already using the system in production because, frankly, it’s not much different from running other production Spark applications. Indeed, through features such as transactional sources/sinks and exactly-once processing, the Structured Streaming designers sought to make it as easy to operate as possible. This chapter will walk you through some of the key operational tasks specific to Structured Streaming. This should supplement everything we saw and learned about Spark operations in [Part II](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/part02.html#part2).

**Fault Tolerance and Checkpointing**

The most important operational concern for a streaming application is failure recovery. Faults are inevitable: you’re going to lose a machine in the cluster, a schema will change by accident without a proper migration, or you may even intentionally restart the cluster or application. In any of these cases, Structured Streaming allows you to recover an application by just restarting it. To do this, you must configure the application to use checkpointing and write-ahead logs, both of which are handled automatically by the engine. Specifically, you must configure a query to write to a *checkpoint location* on a reliable file system (e.g., HDFS, S3, or any compatible filesystem). Structured Streaming will then periodically save all relevant progress information (for instance, the range of offsets processed in a given trigger) as well as the current intermediate state values to the checkpoint location. In a failure scenario, you simply need to restart your application, making sure to point to the same checkpoint location, and it will automatically recover its state and start processing data where it left off. You do not have to manually manage this state on behalf of the application—Structured Streaming does it for you.

To use checkpointing, specify your checkpoint location *before* starting your application through the checkpointLocation option on writeStream. You can do this as follows:

*// in Scala*

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

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

.readStream

.schema(static.schema)

.option("maxFilesPerTrigger", 10)

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

.groupBy("gt")

.count()

**val** query **=** streaming

.writeStream

.outputMode("complete")

.option("checkpointLocation", "/some/location/")

.queryName("test\_stream")

.format("memory")

.start()

*# in Python*

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

streaming = spark\

.readStream\

.schema(static.schema)\

.option("maxFilesPerTrigger", 10)\

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

.groupBy("gt")\

.count()

query = streaming\

.writeStream\

.outputMode("complete")\

.option("checkpointLocation", "/some/python/location/")\

.queryName("test\_python\_stream")\

.format("memory")\

.start()

If you lose your checkpoint directory or the information inside of it, your application will not be able to recover from failures and you will have to restart your stream from scratch.

**Updating Your Application**

Checkpointing is probably the most important thing to enable in order to run your applications in production. This is because the checkpoint will store all of the information about what your stream has processed thus far and what the intermediate state it may be storing is. However, checkpointing does come with a small catch—you’re going to have to reason about your old checkpoint data when you update your streaming application. When you update your application, you’re going to have to ensure that your update is not a breaking change. Let’s cover these in detail when we review the two types of updates: either an update to your application code or running a new Spark version.

**Updating Your Streaming Application Code**

Structured Streaming is designed to allow certain types of changes to the application code between application restarts. Most importantly, you are allowed to change user-defined functions (UDFs) as long as they have the same type signature. This feature can be very useful for bug fixes. For example, imagine that your application starts receiving a new type of data, and one of the data parsing functions in your current logic crashes. With Structured Streaming, you can recompile the application with a new version of that function and pick up at the same point in the stream where it crashed earlier.

While small adjustments like adding a new column or changing a UDF are not breaking changes and do not require a new checkpoint directory, there are larger changes that do require an entirely new checkpoint directory. For example, if you update your streaming application to add a new aggregation key or fundamentally change the query itself, Spark cannot construct the required state for the new query from an old checkpoint directory. In these cases, Structured Streaming will throw an exception saying it cannot begin from a checkpoint directory, and you must start from scratch with a new (empty) directory as your checkpoint location.

**Updating Your Spark Version**

Structured Streaming applications should be able to restart from an old checkpoint directory across patch version updates to Spark (e.g., moving from Spark 2.2.0 to 2.2.1 to 2.2.2). The checkpoint format is designed to be forward-compatible, so the only way it may be broken is due to critical bug fixes. If a Spark release cannot recover from old checkpoints, this will be clearly documented in its release notes. The Structured Streaming developers also aim to keep the format compatible across *minor* version updates (e.g., Spark 2.2.x to 2.3.x), but you should check the release notes to see whether this is supported for each upgrade. In either case, if you cannot start from a checkpoint, you will need to start your application again using a new checkpoint directory.

**Sizing and Rescaling Your Application**

In general, the size of your cluster should be able to comfortably handle bursts above your data rate. The key metrics you should be monitoring in your application and cluster are discussed as follows. In general, if you see that your input rate is much higher than your processing rate (elaborated upon momentarily), it’s time to scale up your cluster or application. Depending on your resource manager and deployment, you may just be able to dynamically add executors to your application. When it comes time, you can scale-down your application in the same way—remove executors (potentially through your cloud provider) or restart your application with lower resource counts. These changes will likely incur some processing delay (as data is recomputed or partitions are shuffled around when executors are removed). In the end, it’s a business decision as to whether it’s worthwhile to create a system with more sophisticated resource management capabilities.

While making underlying infrastructure changes to the cluster or application are sometimes necessary, other times a change may only require a restart of the application or stream with a new configuration. For instance, changing spark.sql.shuffle.partitions is not supported while a stream is currently running (it won’t actually change the number of shuffle partitions). This requires restarting the actual stream, not necessarily the entire application. Heavier weight changes, like changing arbitrary Spark application configurations, will likely require an application restart.

**Metrics and Monitoring**

Metrics and monitoring in streaming applications is largely the same as for general Spark applications using the tools described in [Chapter 18](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch18.html#s4c3---monitoring-and-debugging). However, Structured Streaming does add several more specifics in order to help you better understand the state of your application. There are two key APIs you can leverage to query the status of a streaming query and see its recent execution progress. With these two APIs, you can get a sense of whether or not your stream is behaving as expected.

**Query Status**

The query status is the most basic monitoring API, so it’s a good starting point. It aims to answer the question, “What processing is my stream performing right now?” This information is reported in the status field of the query object returned by startStream. For example, you might have a simple counts stream that provides counts of IOT devices defined by the following query (here we’re just using the same query from the previous chapter without the initialization code):

query.status

To get the status of a given query, simply running the command query.status will return the current status of the stream. This gives us details about what is happening at that point in time in the stream. Here’s a sample of what you’ll get back when querying this status:

{

"message" : "Getting offsets from ...",

"isDataAvailable" : true,

"isTriggerActive" : true

}

The above snippet describes getting the offsets from a Structured Streaming data source (hence the message describing getting offsets). There are a variety of messages to describe the stream’s status.

**NOTE**

We have shown the status command inline here the way you would call it in a Spark shell. However, for a standalone application, you may not have a shell attached to run arbitrary code inside your process. In that case, you can expose its status by implementing a monitoring server, such as a small HTTP server that listens on a port and returns query.status when it gets a request. Alternatively, you can use the richer StreamingQueryListener API described later to listen to more events.

**Recent Progress**

While the query’s current status is useful to see, equally important is an ability to view the query’s progress. The progress API allows us to answer questions like “At what rate am I processing tuples?” or “How fast are tuples arriving from the source?” By running query.recentProgress, you’ll get access to more time-based information like the processing rate and batch durations. The streaming query progress also includes information about the input sources and output sinks behind your stream.

query.recentProgress

Here’s the result of the Scala version after we ran the code from before; the Python one will be similar:

Array({

"id" : "d9b5eac5-2b27-4655-8dd3-4be626b1b59b",

"runId" : "f8da8bc7-5d0a-4554-880d-d21fe43b983d",

"name" : "test\_stream",

"timestamp" : "2017-08-06T21:11:21.141Z",

"numInputRows" : 780119,

"processedRowsPerSecond" : 19779.89350912779,

"durationMs" : {

"addBatch" : 38179,

"getBatch" : 235,

"getOffset" : 518,

"queryPlanning" : 138,

"triggerExecution" : 39440,

"walCommit" : 312

},

"stateOperators" : [ {

"numRowsTotal" : 7,

"numRowsUpdated" : 7

} ],

"sources" : [ {

"description" : "FileStreamSource[/some/stream/source/]",

"startOffset" : null,

"endOffset" : {

"logOffset" : 0

},

"numInputRows" : 780119,

"processedRowsPerSecond" : 19779.89350912779

} ],

"sink" : {

"description" : "MemorySink"

}

})

As you can see from the output just shown, this includes a number of details about the state of the stream. It is important to note that this is a snapshot in time (according to when we asked for the query progress). In order to consistently get output about the state of the stream, you’ll need to query this API for the updated state repeatedly. The majority of the fields in the previous output should be self-explanatory. However, let’s review some of the more consequential fields in detail.

**INPUT RATE AND PROCESSING RATE**

The input rate specifies how much data is flowing into Structured Streaming from our input source. The processing rate is how quickly the application is able to analyze that data. In the ideal case, the input and processing rates should vary together. Another case might be when the input rate is much greater than the processing rate. When this happens, the stream is falling behind and you will need to scale the cluster up to handle the larger load.

**BATCH DURATION**

Nearly all streaming systems utilize batching to operate at any reasonable throughput (some have an option of high latency in exchange for lower throughput). Structured Streaming achieves both. As it operates on the data, you will likely see batch duration oscillate as Structured Streaming processes varying numbers of events over time. Naturally, this metric will have little to no relevance when the continuous processing engine is made an execution option.

**TIP**

Generally it’s a best practice to visualize the changes in batch duration and input and processing rates. It’s much more helpful than simply reporting changes over time.

**Spark UI**

The Spark web UI, covered in detail in [Chapter 18](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch18.html#s4c3---monitoring-and-debugging), also shows tasks, jobs, and data processing metrics for Structured Streaming applications. On the Spark UI, each streaming application will appear as a sequence of short jobs, one for each trigger. However, you can use the same UI to see metrics, query plans, task durations, and logs from your application. One departure of note from the DStream API is that the Streaming Tab is not used by Structured Streaming.

**Alerting**

Understanding and looking at the metrics for your Structured Streaming queries is an important first step. However, this involves constantly watching a dashboard or the metrics in order to discover potential issues. You’re going to need robust *automatic* alerting to notify you when your jobs are failing or not keeping up with the input data rate without monitoring them manually. There are several ways to integrate existing alerting tools with Spark, generally building on the recent progress API we covered before. For example, you may directly feed the metrics to a monitoring system such as the open source Coda Hale Metrics library or Prometheus, or you may simply log them and use a log aggregation system like Splunk. In addition to monitoring and alerting on queries, you’re also going to want to monitor and alert on the state of the cluster and the overall application (if you’re running multiple queries together).

**Advanced Monitoring with the Streaming Listener**

We already touched on some of the high-level monitoring tools in Structured Streaming. With a bit of glue logic, you can use the status and queryProgress APIs to output monitoring events into your organization’s monitoring platform of choice (e.g., a log aggregation system or Prometheus dashboard). Beyond these approaches, there is also a lower-level but more powerful way to observe an application’s execution: the StreamingQueryListener class.

The StreamingQueryListener class will allow you to receive asynchronous updates from the streaming query in order to automatically output this information to other systems and implement robust monitoring and alerting mechanisms. You start by developing your own object to extend StreamingQueryListener, then attach it to a running SparkSession. Once you attach your custom listener with sparkSession.streams.addListener(), your class will receive notifications when a query is started or stopped, or progress is made on an active query. Here’s a simple example of a listener from the Structured Streaming documentation:

**val** spark**:** **SparkSession** = ...

spark.streams.addListener(**new** **StreamingQueryListener**() {

**override** **def** onQueryStarted(queryStarted**:** **QueryStartedEvent**)**:** **Unit** = {

println("Query started: " + queryStarted.id)

}

**override** **def** onQueryTerminated(

queryTerminated**:** **QueryTerminatedEvent**)**:** **Unit** = {

println("Query terminated: " + queryTerminated.id)

}

**override** **def** onQueryProgress(queryProgress**:** **QueryProgressEvent**)**:** **Unit** = {

println("Query made progress: " + queryProgress.progress)

}

})

Streaming listeners allow you to process each progress update or status change using custom code and pass it to external systems. For example, the following code for a StreamingQueryListener that will forward all query progress information to Kafka. You’ll have to parse this JSON string once you read data from Kafka in order to access the actual metrics:

**class** **KafkaMetrics**(servers**:** **String**) **extends** **StreamingQueryListener** {

**val** kafkaProperties **=** **new** **Properties**()

kafkaProperties.put(

"bootstrap.servers",

servers)

kafkaProperties.put(

"key.serializer",

"kafkashaded.org.apache.kafka.common.serialization.StringSerializer")

kafkaProperties.put(

"value.serializer",

"kafkashaded.org.apache.kafka.common.serialization.StringSerializer")

**val** producer **=** **new** **KafkaProducer**[**String**, **String**](kafkaProperties)

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

**import** **org.apache.kafka.clients.producer.KafkaProducer**

**override** **def** onQueryProgress(event**:**

**StreamingQueryListener.QueryProgressEvent**)**:** **Unit** = {

producer.send(**new** **ProducerRecord**("streaming-metrics",

event.progress.json))

}

**override** **def** onQueryStarted(event**:**

**StreamingQueryListener.QueryStartedEvent**)**:** **Unit** = {}

**override** **def** onQueryTerminated(event**:**

**StreamingQueryListener.QueryTerminatedEvent**)**:** **Unit** = {}

}

Using the StreamingQueryListener interface, you can even monitor Structured Streaming applications on one cluster by running a Structured Streaming application on that same (or another) cluster. You could also manage multiple streams in this way.

**Conclusion**

In this chapter, we covered the main tools needed to run Structured Streaming in production: checkpoints for fault tolerance and various monitoring APIs that let you observe how your application is running. Lucky for you, if you’re running Spark in production already, many of the concepts and tools are similar, so you should be able to reuse a lot of your existing knowledge. Be sure to check [Part IV](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/part04.html#part4) to see some other helpful tools for monitoring Spark Applications.