Ch.21 Structured Streaming Basics

Structured Streaming

- stream processing framework built on the Spark SQL engine
- rather than introducing a separate API, Structured Streaming uses the existing structured APIs in Spark (DataFrames, Datasets, and SQL), meaning that all the operations you are familiar with there are supported
- Users express a streaming computation in the same way they'd write a batch computation on static data.
- Upon specifying this, and specifying a streaming destination, the Structured Streaming engine will take care of running your query incrementally and continuously as new data arrives into the system.
- These logical instructions for the computation are then executed using the same Catalyst engine discussed in Part II of this book, including query optimization, code generation, etc.
- For instance, Structured Streaming ensures end-to-end, exactly-once processing as well as fault-tolerance through checkpointing and write-ahead logs.
- **The main idea behind Structured Streaming is to treat a stream of data as a table to which data is continuously appended.**
- The job then periodically checks for new input data, process it, updates some internal state located in a state store if needed, and updates its result.
- A cornerstone of the API is that you should not have to change your query's code when doing batch or stream processing—you should have to specify only whether to run that query in a batch or streaming fashion.
- Internally, Structured Streaming will automatically figure out how to "**incrementalize**" your query, i.e., update its result efficiently whenever new data arrives, and will run it in a fault tolerant fashion.

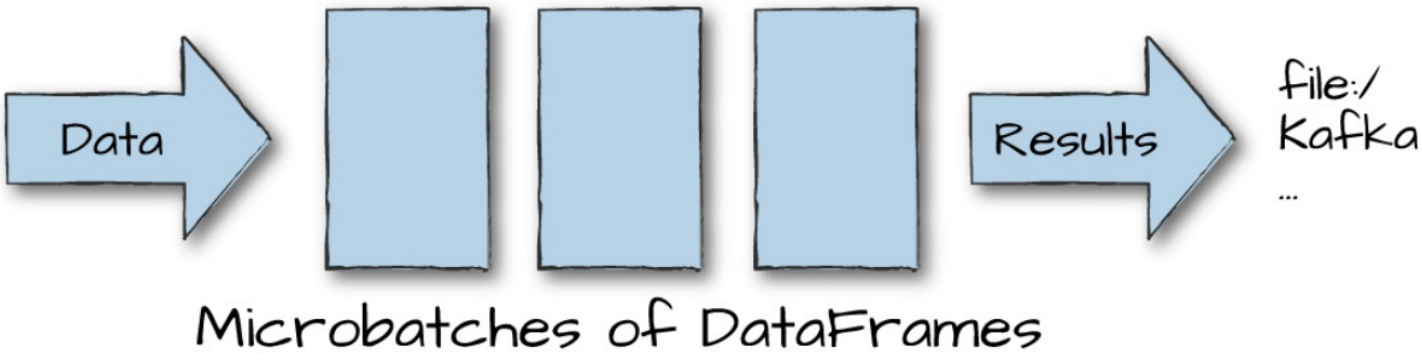


Figure 20-2. Micro-batch

- Structured Streaming is "your DataFrame, but streaming."
- easy to get started using streaming applications.
- There are some limits to the types of queries Structured Streaming will be able to run
- new concepts you have to think about that are specific to streaming, such as **event-time** and **out-of-order** data.
- Structured Streaming enables users to build what we call continuous applications.
- A continuous application is an end-to-end application that reacts to data in real time by combining a

variety of tools: streaming jobs, batch jobs, joins between streaming and offline data, and interactive ad-hoc queries.

- Because most streaming jobs today are deployed within the context of a larger continuous application, the Spark developers sought to make it easy to specify the whole application in one framework and get consistent results across these different portions of it.
- For example, you can use Structured Streaming to continuously update a table that users query interactively with Spark SQL, serve a machine learning model trained by MLlib, or join streams with offline data in any of Spark's data sources —applications that would be much more complex to build using a mix of different tools.

Transformations and Actions

- Structured Streaming maintains the same concept of transformations and actions
- The transformations available in Structured Streaming are, with a few restrictions, the exact same transformations that we saw in Part II.
- The restrictions usually involve some types of queries that the engine cannot incrementalize yet, although some of the limitations are being lifted in new versions of Spark.
- There is generally only one action available in Structured Streaming: that of starting a stream, which will then run continuously and output results.

Input Sources

Spark 2.2, the supported input sources are as follows:

- Apache Kafka 0.10
- Files on a distributed file system like HDFS or S3 (Spark will continuously read new files in a directory)
- A socket source for testing
- Spark are working on a stable source API so that you can build your own streaming connectors.

Sinks

- sinks specify the destination for the result set of that stream
- Sinks and the execution engine are also responsible for reliably tracking the exact progress of data processing
- supported output sinks as of Spark 2.2:
    - Apache Kafka 0.10
    - Almost any file format
    - A foreach sink for running arbitary computation on the output records
    - A console sink for testing
    - A memory sink for debugging

Output Modes

- Defining a sink for our Structured Streaming job is only half of the story.
- We also need to define how we want Spark to write data to that sink.
- For instance, do we only want to append new information? Do we want to update rows as we receive

more information about them over time (e.g., updating the click count for a given web page)? Do we want to completely overwrite the result set every single time (i.e. always write a file with the complete click counts for all pages)?

- To do this, we define an output mode, similar to how we define output modes in the static Structured APIs.
- The supported output modes are as follows:
  - **Append** (only add new records to the output sink)
  - **Update** (update changed records in place)
  - **Complete** (rewrite the full output)
- One important detail is that certain queries, and certain sinks, only support certain output modes, as we will discuss later in the book.
- For example, suppose that your job is just performing a map on a stream. The output data will grow indefinitely as new records arrive, so it would not make sense to use Complete mode, which requires writing all the data to a new file at once.
- In contrast, if you are doing an aggregation into a limited number of keys, Complete and Update modes would make sense, but Append would not, because the values of some keys' need to be updated over time.

Triggers

- Whereas output modes define how data is output, **triggers define when data is output** — that is, when Structured Streaming should check for new input data and update its result.
- By default, Structured Streaming will look for new input records as soon as it has finished processing the last group of input data, giving the lowest latency possible for new results.
- However, this behavior can lead to writing many small output files when the sink is a set of files. Thus, Spark also supports triggers based on processing time (only look for new data at a fixed interval).
- In the future, other types of triggers may also be supported

Event-time processing

(i.e., processing data based on timestamps included in the record that may arrive out of order).

Event-time data

- Event-time means time fields that are embedded in your data.
- This means that rather than processing data according to the time it reaches your system, you process it according to the time that it was generated, even if records arrive out of order at the streaming application due to slow uploads or network delays.
- Expressing event-time processing is simple in Structured Streaming
- Because the system views the input data as a table, the event time is just another field in that table, and your application can do grouping, aggregation, and windowing using standard SQL operators.
- However, under the hood, Structured Streaming can take some special actions when it knows that one of your columns is an event-time field, including optimizing query execution or determining when it is safe to forget state about a time window

to forget state about a time window.

- Many of these actions can be controlled using watermarks.

Watermarks

- Watermarks are a feature of streaming systems that allow you to **specify how late they expect to see data** in event time.
- For example, in an application that processes logs from mobile devices, one might expect logs to be up to 30 minutes late due to upload delays.
- Systems that support event time, including Structured Streaming, usually allow setting watermarks to limit how long they need to remember old data.
- Watermarks can also be used to control when to output a result for a particular event time window (e.g., waiting until the watermark for it has passed).

Applied example

- we're going to be working with the Heterogeneity Human Activity Recognition Dataset
- 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.

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

- The gt field specifies what activity the user was doing at that time.
- Next will 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.
- Streaming DataFrames are largely the same as static DataFrames. We create them within Spark applications and then perform transformations on them to get our data into the correct format.
- Basically, all of the transformations that are available in the static Structured APIs apply to Streaming DataFrames.

DataFrames.

- However, one small difference is that Structured Streaming does not let you perform schema inference without explicitly enabling it.
- You can enable schema inference for this by setting the configuration **spark.sql.streaming.schemaInference** to true.
- Given that fact, we will read the schema from one file (that we know has a valid schema) and pass the dataSchema object from our static DataFrame to our streaming DataFrame.
- As mentioned, you should avoid doing this in a production scenario where your data may (accidentally) change out from under you

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

maxFilesPerTrigger

- allows you 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.

Streaming DataFrame creation and execution is lazy

- specify transformations on our streaming DataFrame before finally calling an action to start the stream. In this case, 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:

```
activityCounts = streaming.groupBy("gt").count()
```

specify our action to start the query.

- specify an output destination, or output sink for our result of this query.
- write to a memory sink which keeps an in-memory table of the results.
- define how Spark will output that data.
- use the complete output mode - rewrites all of the keys along with their counts after every trigger

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

- **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.
- When we run the preceding code, we also want to include the following line:

```
activityQuery.awaitTermination()
```

- After this code is executed, the streaming computation will have started in the background.

- The query object is a handle to that active streaming query, and we must specify that we would like to wait for the termination of the query using activityQuery.awaitTermination() to **prevent the driver process from exiting while the query is active.**
- must be included in your production applications; otherwise, your stream won't be able to run.
- 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.
- 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

```
from time import sleep
for x in range(5):
    spark.sql("SELECT * FROM activity_counts").show()
    sleep(1)
```

- see the counts for each activity change over time
- 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
- 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).

```
|        gt| count|
+----------+------+
|  stairsup|167255|
|       sit|196927|
|     stand|182165|
|      walk|212095|
|      bike|172762|
|stairsdown|149819|
|      null|167168|
+----------+------+

+----------+------+
|        gt| count|
+----------+------+
|  stairsup|177710|
|       sit|209235|
```

```
|       sit|209235|
|     stand|193552|
|      walk|225351|
|      bike|183559|
|stairsdown|159182|
|      null|177614|
+----------+------+

+----------+------+
|        gt| count|
+----------+------+
|  stairsup|177710|
|       sit|209235|
|     stand|193552|
|      walk|225351|
|      bike|183559|
|stairsdown|159182|
|      null|177614|
+----------+------+

+----------+------+
|        gt| count|
+----------+------+
|  stairsup|188165|
|       sit|221543|
|     stand|204938|
|      walk|238607|
|      bike|194355|
|stairsdown|168545|
|      null|188061|
+----------+------+

+----------+------+
|        gt| count|
+----------+------+
|  stairsup|188165|
|       sit|221543|
|     stand|204938|
|      walk|238607|
|      bike|194355|
|stairsdown|168545|
|      null|188061|
+----------+------+
```

-

Transformations on Streams

- Streaming transformations include almost all static DataFrame transformations that you already saw in Part II.
- All select, filter, and simple transformations are supported, as are all DataFrame functions and individual column manipulations.
- The limitations arise on transformations that do not make sense in context of streaming data

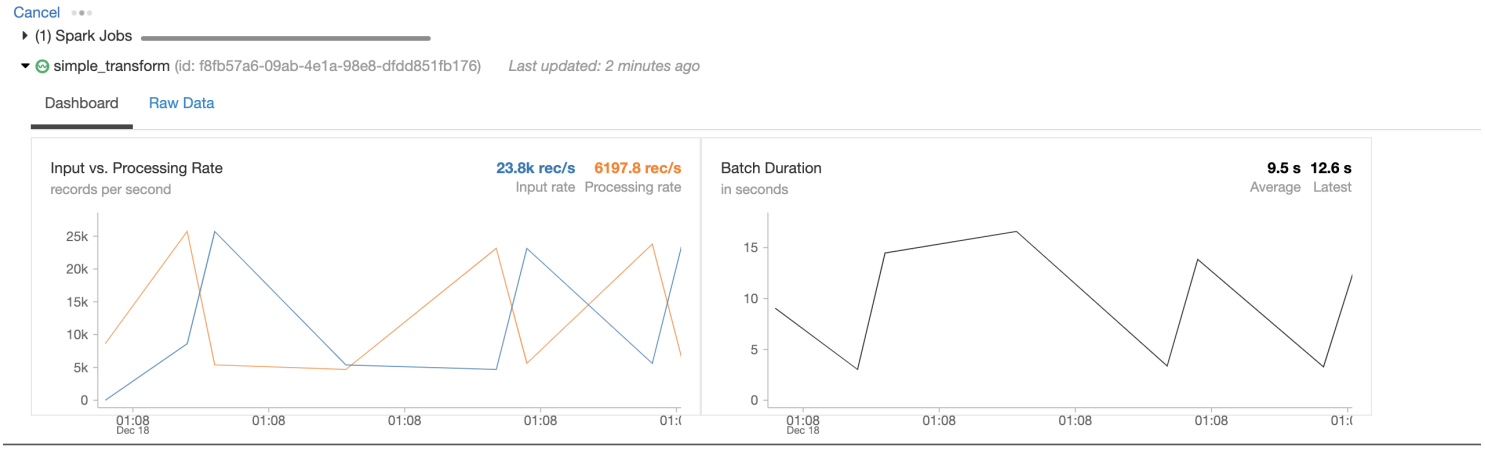
The limitations arise on transformations that do not make sense in context of streaming data.

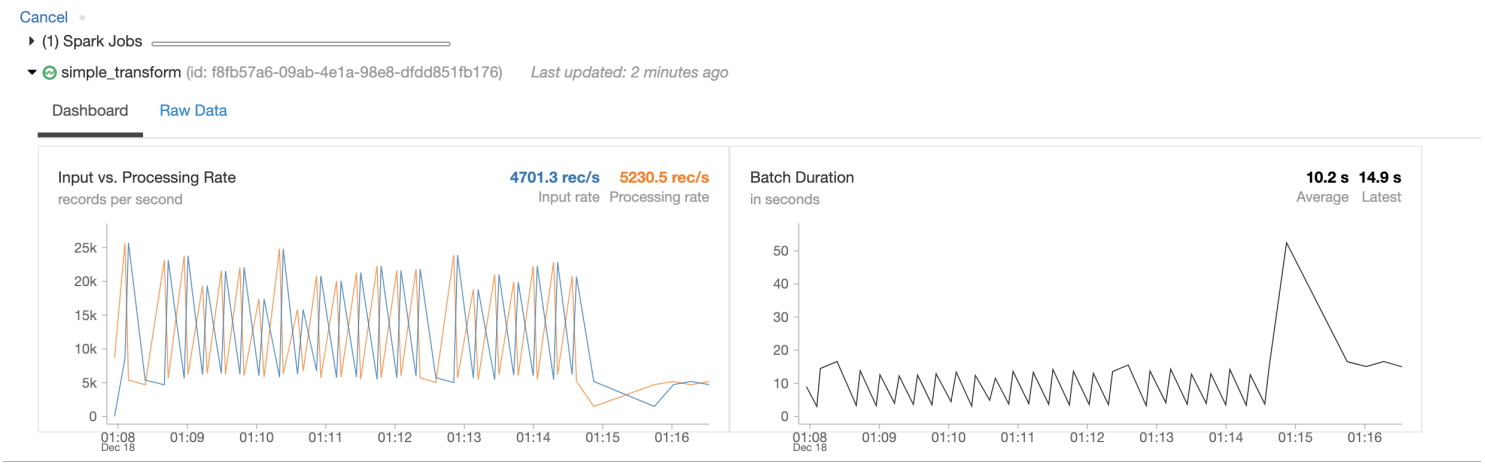
- For example, as of Apache Spark 2.2, users cannot sort streams that are not aggregated, and cannot perform multiple levels of aggregation without using Stateful Processing (covered in the next chapter).
- These limitations may be lifted as Structured Streaming continues to develop, so we encourage you to check the documentation of your version of Spark for updates.

Selections and Filtering

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

```
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()
```



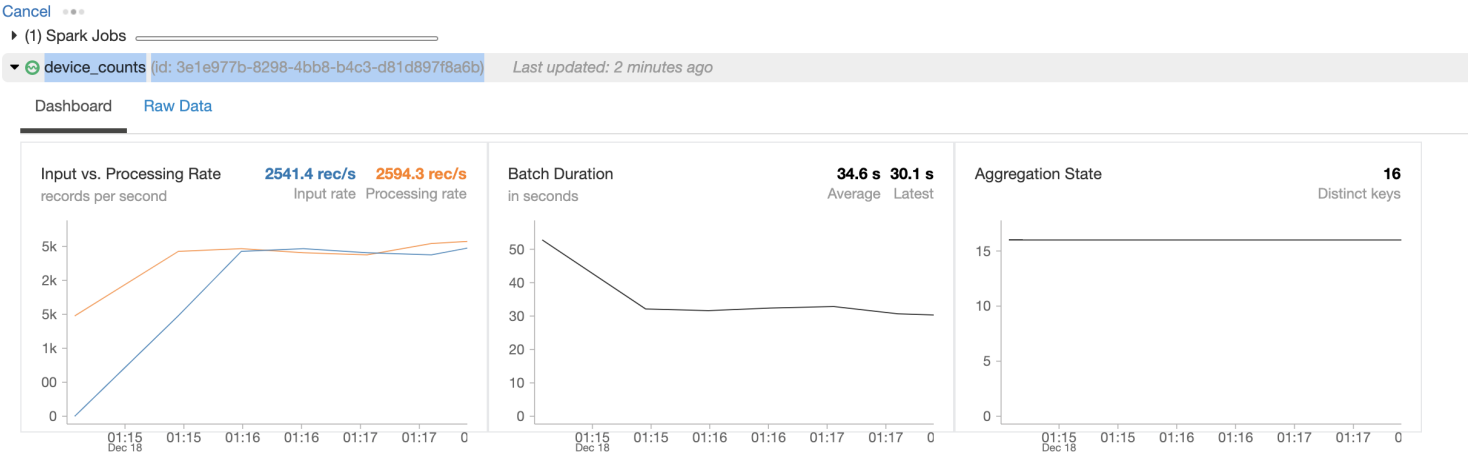


Aggregations

- can specify arbitrary aggregations, as you saw in the Structured APIs
- 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
- (jump back to Chapter 7 in order to see potential aggregations that you can run on your stream)

```
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()
```

```
1  deviceModelStats = streaming.cube("gt", "model").avg()\
2    .drop("avg(Arrival_time)")\
3    .drop("avg(Creation_Time)")\
4    .drop("avg(Index)")\
5    .writeStream.queryName("device_counts").format("memory")\
6    .outputMode("complete")\
7    .start()
8
```



http://mariuszrafalo.pl/sgh/projekty/db_clean.html#:~:text=There%20are%20two%20main%20reasons,up%20files%20to%20restore%20service%20.

NOTE As of Spark 2.2, the one limitation of aggregations is that multiple "chained" aggregations (aggregations on streaming aggregations) are not supported at this time. However, you can achieve this by writing out to an intermediate sink of data, like Kafka or a file sink. This will change in the future as the Structured Streaming community adds this functionality.

Joins

- 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

```
historicalAgg = static.groupBy("gt", "model").avg()
```

```
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()
```

Structured Streaming supports several sources and sinks, including Apache Kafka, files, and several sources and sinks for testing and debugging.

- in reality you can mix and match them (e.g., use a Kafka input source with a file sink).

Where Data Is Read and Written (Sources and Sinks)

- Structured Streaming supports several production sources and sinks (files and Apache Kafka),
- as well as some debugging tools like the memory table sink.
- We mentioned these at the beginning of the chapter, but now let's cover the details of each one.

File source and sink

- Probably the simplest source you can think of is the simple file source.
- While essentially any file source should work, the ones that we see in practice are Parquet, text, JSON, and CSV.
- The only difference between using the file source/sink and Spark's static file source is that with streaming, we can control the number of files that we read in during each trigger via the maxFilesPerTrigger option that we saw earlier.
- Keep in mind that any files you add into an input directory for a streaming job need to appear in it atomically.
- Otherwise, Spark will process partially written files before you have finished.
- On file systems that show partial writes, such as local files or HDFS, this is best done by writing the file in an external directory and moving it into the input directory when finished.
- On Amazon S3, objects normally only appear once fully written.

Kafka source and sink

- **Apache Kafka** is a distributed publish-and-subscribe system for streams of data.
- Kafka lets you publish and subscribe to streams of records like you might do with a message queue
  - these are stored as streams of records in a fault-tolerant way.
- Think of Kafka like a distributed buffer.
- Kafka lets you store streams of records in categories that are referred to as topics.
- Each record in Kafka consists of a **key**, a **value**, and a **timestamp**.
- Topics consist of immutable sequences of records for which the position of a record in a sequence is called an offset.
- Reading data is called **subscribing to a topic** and writing data is as simple as **publishing to a topic**
- Spark allows you to read from Kafka with both batch and streaming DataFrames

- Spark allows you to read from Kafka with both batch and streaming DataFrames.
- As of Spark 2.2, Structured Streaming supports Kafka version 0.10.
- There are only a few options that you need to specify when you read from Kafka.

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

```
# 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()
```

```
  .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

- 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:

```
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()
```

Foreach sink

- The foreach sink is akin to foreachPartitions in the Dataset API.
- This operation allows arbitrary operations to be computed on a per-partition basis, in parallel.
- This is available in Scala and Java initially, but it will likely be ported to other languages in the future.
- To use the foreach sink, you must implement the **ForeachWriter** interface, which is available in the Scala/Java documents, which contains three methods: open, process, and close.

- The relevant methods will be called whenever there is a sequence of rows generated as output after a trigger.
- Here are some important details:
  - The writer must be **Serializable**, as it were a UDF or a Dataset map function.
  - The three methods (**open, process, close**) will be called on each executor.
  - The writer must do all its initialization, like opening connections or starting transactions only in the open method.
  - A common source of errors is that if initialization occurs outside of the open method (say in the class that you're using), that happens on the driver instead of the executor.
- Because the Foreach sink runs arbitrary user code, one key issue you must consider when using it is fault tolerance.
- If Structured Streaming asked your sink to write some data, but then crashed, it cannot know whether your original write succeeded.
- Therefore, the API provides some additional parameters to help you achieve exactly-once processing.
- First, the open call on your ForeachWriter receives two parameters that uniquely identify the set of rows that need to be acted on.
- The version parameter is a monotonically increasing ID that increases on a per-trigger basis, and partitionId is the ID of the partition of the output in your task.
- Your open method should return whether to process this set of rows.
- If you track your sink's output externally and see that this set of rows was already output (e.g., you wrote the last version and partitionId written in your storage system), you can return false from open to skip processing this set of rows.
- Otherwise, return true. Your ForeachWriter will be opened again for each trigger's worth of data to write.
- Next, the process method will be called for each record in the data, assuming your open method returned true.
- This is fairly straightforward—just process or write your data.
- Finally, whenever open is called, the close method is also called (unless the node crashed before that), regardless of whether open returned true.
- If Spark witnessed an error during processing, the close method receives that error.
- It is your responsibility to clean up any open resources during close.
- Together, the ForeachWriter interface effectively lets you implement your own sink, including your own logic for tracking which triggers' data has been written or safely overwriting it on failures.
- We show an example of passing a ForeachWriter below:

```
//in Scala
datasetOfString.write.foreach(new ForeachWriter[String] {
def open(partitionId: Long, version: Long): Boolean = {
// open a database connection
}
def process(record: String) = {
// write string to connection
}
def close(errorOrNull: Throwable): Unit = {
// close the connection
```

```
}
})
```

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

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

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:

```
activityCounts.writeStream.format("memory").queryName("my_device_table")
```

```
activityCounts.writeStream.format("memory").queryName("my_device_table")
```

Append mode

- Append mode is the default behavior and the simplest to understand.
- When new rows are added to the result table, they will be output to the sink based on the trigger (explained next) that you specify.
- This mode ensures that each row is output once (and only once), assuming that you have a fault-tolerant sink.
- When you use append mode with event-time and watermarks (covered in Chapter 22), only the final result will output to the sink.

Complete mode

- Complete mode will output the entire state of the result table to your output sink.
- This is useful when you're working with some stateful data for which all rows are expected to change over time or the sink you are writing does not support row-level updates.
- Think of it like the state of a stream at the time the previous batch had run.

Update mode

- Update mode is similar to complete mode except that only the rows that are different from the previous write are written out to the sink.
- Naturally, your sink must support row-level updates to support this mode.
- If the query doesn't contain aggregations, this is equivalent to append mode.

When can you use each mode?

- Structured Streaming limits your use of each mode to queries where it makes sense.
- For example, if your query just does a map operation, Structured Streaming will not allow complete mode, because this would require it to remember all input records since the start of the job and rewrite the whole output table.
- **This requirement is bound to get prohibitively expensive as the job runs.**
- We will discuss when each mode is supported in more detail in the next chapter, once we also cover event-time processing and watermarks.
- If your chosen mode is not available, Spark Streaming will throw an exception when you start your stream:

Table 21-1. Structured streaming output modes as of Spark 2.2

Query Type	Query type (continued)	Supported Output Modes	Notes
	Aggregation		Append mode uses watermark to drop old aggregation state. This means that as new rows are brought into the table, Spark will only keep around

Queries with aggregation	on event-time with watermark	Append, Update, Complete	rows that are below the “watermark”. Update mode also uses the watermark to remove old aggregation state. By definition, complete mode does not drop old aggregation state since this mode preserves all data in the Result Table.
	Other aggregations	Complete, Update	Since no watermark is defined (only defined in other category), old aggregation state is not dropped. Append mode is not supported as aggregates can update thus violating the semantics of this mode.
Queries with mapGroupsWithState		Update	
Queries with flatMapGroupsWithState	Append operation mode	Append	Aggregations are allowed after flatMapGroupsWithState.
	Update operation mode	Update	Aggregations not allowed after flatMapGroupsWithState.
Other queries		Append, Update	Complete mode not supported as it is infeasible to keep all unaggregated data in the Result Table.

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.
- Currently, there is one periodic trigger type, based on processing time, as well as a “once” trigger to manually run a processing step once.

Processing time trigger

- 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).
- String Format:

```
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 immediately after 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:

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

Streaming Dataset API

- One final thing to note about Structured Streaming is that you are not limited to just the DataFrame API for streaming.
- You can also use Datasets to perform the same computation but in type-safe manner.
- You can turn a streaming DataFrame into a Dataset the same way you did with a static one.
- As before, the Dataset's elements need to be Scala case classes or Java bean classes.
- Other than that, the DataFrame and Dataset operators work as they did in a static setting, and will also turn into a streaming execution plan when run on a stream.

```
// in Scala
case class Flight(DEST_COUNTRY_NAME: String, ORIGIN_COUNTRY_NAME: String,
count: BigInt)
val dataSchema = spark.read
.parquet("/data/flight-data/parquet/2010-summary.parquet/")
.schema
val flightsDF = spark.readStream.schema(dataSchema)
.parquet("/data/flight-data/parquet/2010-summary.parquet/")
val flights = flightsDF.as[Flight]
def originIsDestination(flight_row: Flight): Boolean = {
return flight_row.ORIGIN_COUNTRY_NAME == flight_row.DEST_COUNTRY_NAME
}
flights.filter(flight_row => originIsDestination(flight_row))
.groupByKey(x => x.DEST_COUNTRY_NAME).count()
.writeStream.queryName("device_counts").format("memory").outputMode("complete")
```