Chapter 8. Structured Streaming

In earlier chapters, you learned how to use structured APIs to process very large but finite volumes of data. However, often data arrives continuously and needs to be processed in a real-time manner. In this chapter, we will discuss how the same Structured APIs can be used for processing data streams as well.

Evolution of the Apache Spark Stream Processing Engine

Stream processing is defined as the continuous processing of endless streams of data. With the advent of big data, stream processing systems transitioned from single-node processing engines to multiple-node, distributed processing engines. Traditionally, distributed stream processing has been implemented with a *record-at-a-time processing model*, as illustrated in Figure 8-1.

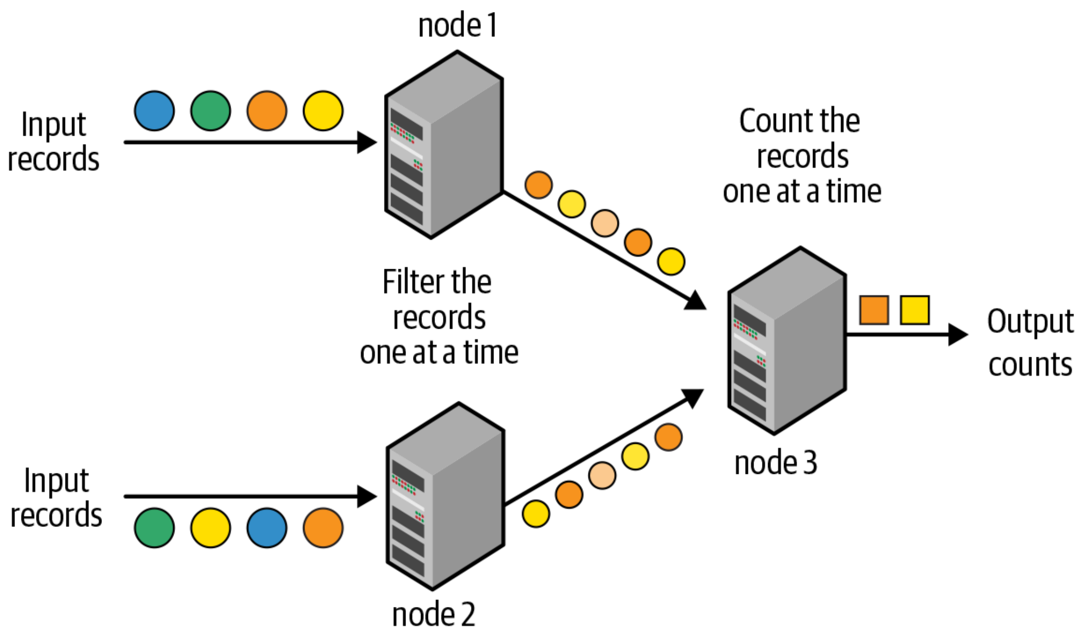

node 1
Input records
Count the records one at a time
Filter the records one at a time
Output counts
node 3
Input records
node 2

Figure 8-1. Traditional record-at-a-time processing model

The processing pipeline is composed of a directed graph of nodes, as shown in Figure 8-1; each node continuously receives one record at a time, processes it, and then forwards the generated record(s) to the next node in the graph. This processing model can achieve very low latencies —that is, an input record can be processed by the pipeline and the resulting output can be generated within milliseconds. However, this model is not very efficient at recovering from node failures and straggler nodes (i.e., nodes that are slower than others); it can either recover from a failure very fast with a lot of extra failover resources, or use minimal extra resources but recover slowly.[1]

The Advent of Micro-Batch Stream Processing

This traditional approach was challenged by Apache Spark when it introduced Spark Streaming (also called DStreams). It introduced the idea of *micro-batch stream processing*, where the streaming computation is modeled as a continuous series of small, map/reduce-style batch processing jobs (hence, "micro-batches") on small chunks of the stream data. This is illustrated in Figure 8-2.

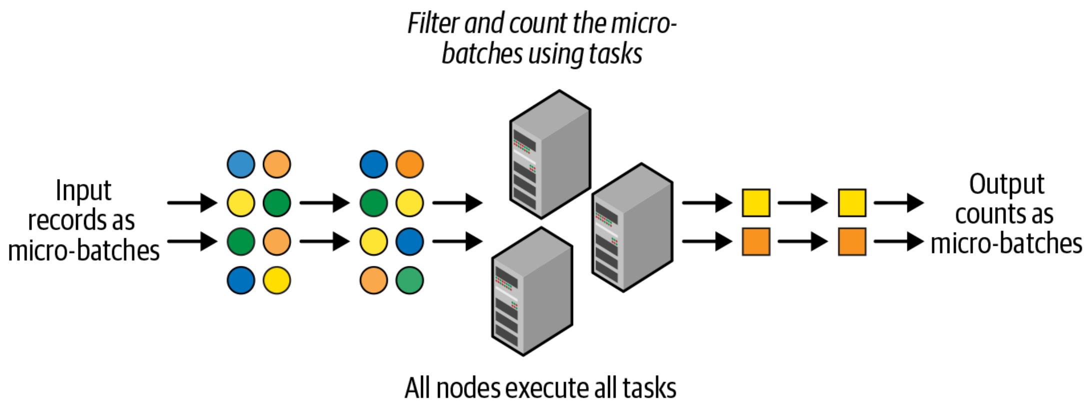

Filter and count the micro-batches using tasks
Input records as micro-batches
Output counts as micro-batches
All nodes execute all tasks

Figure 8-2. Structured Streaming uses a micro-batch processing model

As shown here, Spark Streaming divides the data from the input stream into, say, 1-second micro-batches. Each batch is processed in the Spark cluster in a distributed manner with small deterministic tasks that generate the output in micro-batches. Breaking down the streaming computation into these small tasks gives us two advantages over the traditional, continuous-operator model:

- Spark's agile task scheduling can very quickly and efficiently recover from failures and straggler executors by rescheduling one or more copies of the tasks on any of the other executors.

- The deterministic nature of the tasks ensures that the output data is the same no matter how many times the task is reexecuted. This crucial characteristic enables Spark Streaming to provide end-to-end exactly-once processing guarantees, that is, the generated output results will be such that every input record was processed exactly once.

This efficient fault tolerance does come at the cost of latency—the micro-batch model cannot achieve millisecond-level latencies; it usually achieves latencies of a few seconds (as low as half a second in some cases). However, we have observed that for an overwhelming majority of stream processing use cases, the benefits of micro-batch processing outweigh the drawback of second-scale latencies. This is because most streaming pipelines have at least one of the following characteristics:

- The pipeline does not need latencies lower than a few seconds. For example, when the streaming output is only going to be read by hourly jobs, it is not useful to generate output with subsecond latencies.
- There are larger delays in other parts of the pipeline. For example, if the writes by a sensor into Apache Kafka (a system for ingesting data streams) are batched to achieve higher throughput, then no amount of optimization in the downstream processing systems can make the end-to-end latency lower than the batching delays.

Furthermore, the DStream API was built upon Spark's batch RDD API. Therefore, DStreams had the same functional semantics and fault-tolerance model as RDDs. Spark Streaming thus proved that it is possible for a single, unified processing engine to provide consistent APIs and semantics for batch, interactive, and streaming workloads. This fundamental paradigm shift in stream processing propelled Spark Streaming to become one of the most widely used open source stream processing engines.

Lessons Learned from Spark Streaming (DStreams)

Despite all the advantages, the DStream API was not without its flaws. Here are a few key areas for improvement that were identified:

Lack of a single API for batch and stream processing

Even though DStreams and RDDs have consistent APIs (i.e., same operations and same semantics), developers still had to explicitly rewrite their code to use different classes when converting their batch jobs to streaming jobs.

Lack of separation between logical and physical plans

Spark Streaming executes the DStream operations in the same sequence in which they were specified by the developer. Since developers effectively specify the exact physical plan, there is no scope for automatic optimizations, and developers have to hand-optimize their code to get the best performance.

Lack of native support for event-time windows

DStreams define window operations based only on the time when each record is received by Spark Streaming (known as *processing time*). However, many use cases need to calculate windowed aggregates based on the time when the records were generated (known as *event time*) instead of when they were received or processed. The lack of native support of event-time windows made it hard for developers to build such pipelines with Spark Streaming.

These drawbacks shaped the design philosophy of Structured Streaming, which we will discuss next.

The Philosophy of Structured Streaming

Based on these lessons from DStreams, Structured Streaming was designed from scratch with one core philosophy—for developers, writing stream processing pipelines should be as easy as writing batch pipelines. In a nutshell, the guiding principles of Structured Streaming are:

A single, unified programming model and interface for batch and stream processing

This unified model offers a simple API interface for both batch and streaming workloads. You can use familiar SQL or batch-like DataFrame queries (like those you've learned about in the previous chapters) on your stream as you would on a batch, leaving dealing with the underlying complexities of fault tolerance, optimizations, and tardy data to the engine. In the coming sections, we will examine some of the queries you might write.

A broader definition of stream processing

Big data processing applications have grown complex enough that the line between real-time processing and batch processing has blurred significantly. The aim with Structured Streaming was to broaden its applicability from traditional stream processing to a larger class of applications; any application that periodically (e.g., every few hours) to continuously (like traditional streaming applications) processes data should be expressible using Structured Streaming.

Next, we'll discuss the programming model used by Structured Streaming.

The Programming Model of Structured Streaming

"Table" is a well-known concept that developers are familiar with when building batch applications. Structured Streaming extends this concept to streaming applications by treating a stream as an unbounded, continuously appended table, as illustrated in Figure 8-3.

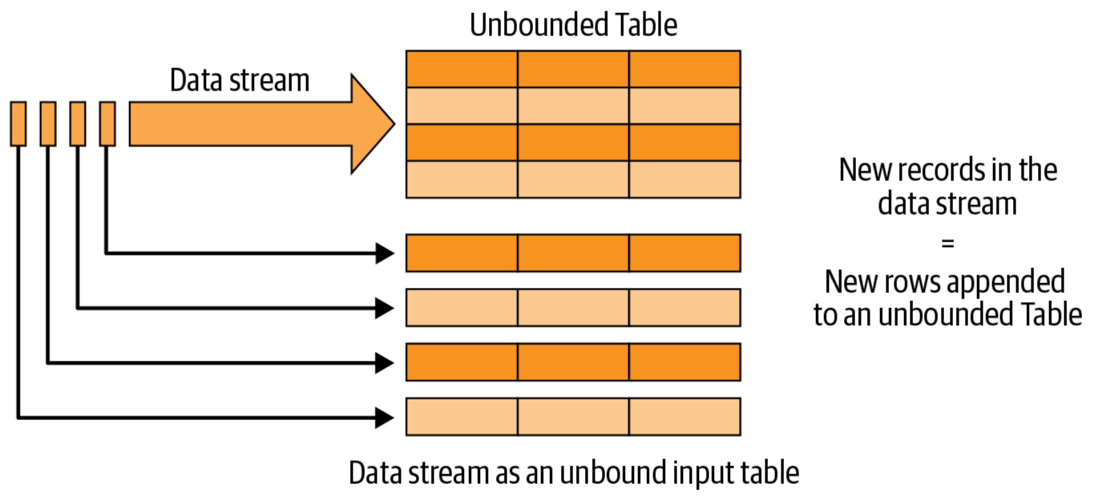

Unbounded Table
Data stream
New records in the data stream = New rows appended to an unbounded Table
Data stream as an unbound input table

Figure 8-3. The Structured Streaming programming model: data stream as an unbounded table

Every new record received in the data stream is like a new row being appended to the unbounded input table. Structured Streaming will not actually retain all the input, but the output produced by Structured Streaming until time T will be equivalent to having all of the input until T in a static, bounded table and running a batch job on the table.

As shown in Figure 8-4, the developer then defines a query on this conceptual input table, as if it were a static table, to compute the result table that will be written to an output sink. Structured Streaming will automatically convert this batch-like query to a streaming execution plan. This is called *incrementalization*: Structured Streaming figures out what state needs to be maintained to update the result each time a record arrives. Finally, developers specify triggering policies to control when to update the results. Each time a trigger fires, Structured Streaming checks for new data (i.e., a new row in the input table) and incrementally updates the result.

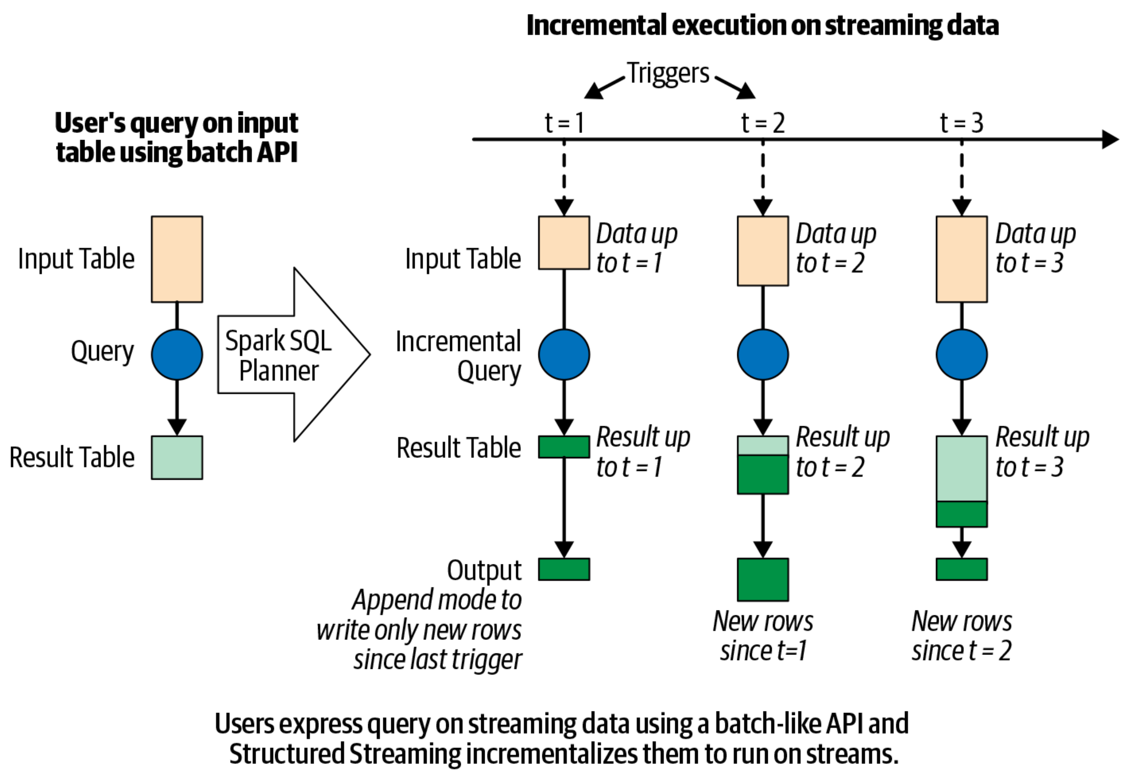

Incremental execution on streaming data
Triggers
t = 1
t = 2
t = 3
User's query on input table using batch API
Input Table
Query
Result Table
Spark SQL Planner
Input Table
Incremental Query
Result Table
Data up to t = 1
Data up to t = 2
Data up to t = 3
Result up to t = 1
Result up to t = 2
Result up to t = 3
Output
Append mode to write only new rows since last trigger
New rows since t=1
New rows since t = 2
Users express query on streaming data using a batch-like API and Structured Streaming incrementalizes them to run on streams.

Figure 8-4. The Structured Streaming processing model

The last part of the model is the output mode. Each time the result table is updated, the developer will want to write the updates to an external system, such as a filesystem (e.g., HDFS, Amazon S3) or a database (e.g., MySQL, Cassandra). We usually want to write output incrementally. For this purpose, Structured Streaming provides three output modes:

Append mode

Only the new rows appended to the result table since the last trigger will be written to the external storage. This is applicable only in queries where existing rows in the result table cannot change (e.g., a map on an input stream).

Update mode

Only the rows that were updated in the result table since the last trigger will be changed in the external storage. This mode works for output sinks that can be updated in place, such as a MySQL table.

Complete mode

The entire updated result table will be written to external storage.

NOTE

Unless complete mode is specified, the result table will not be fully materialized by Structured Streaming. Just enough information (known as "state") will be maintained to ensure that the changes in the result table can be computed and the updates can be output.

Thinking of the data streams as tables not only makes it easier to conceptualize the logical computations on the data, but also makes it easier to express them in code. Since Spark's DataFrame is a programmatic representation of a table, you can use the DataFrame API to express your computations on streaming data. All you need to do is define an input DataFrame (i.e., the input table) from a streaming data source, and then you apply operations on the DataFrame in the same way as you would on a DataFrame defined on a batch source.

In the next section, you will see how easy it is to write Structured Streaming queries using DataFrames.

The Fundamentals of a Structured Streaming Query

In this section, we are going to cover some high-level concepts that you'll need to understand to develop Structured Streaming queries. We will first walk through the key steps to define and start a streaming query, then we will discuss how to monitor the active query and manage its life cycle.

Five Steps to Define a Streaming Query

As discussed in the previous section, Structured Streaming uses the same DataFrame API as batch queries to express the data processing logic. However, there are a few key differences you need to know about for defining a Structured Streaming query. In this section, we will explore the steps involved in defining a streaming query by building a simple query that reads streams of text data over a socket and counts the words.

Step 1: Define input sources

As with batch queries, the first step is to define a DataFrame from a streaming source. However, when reading batch data sources, we need `spark.read` to create a `DataFrameReader`, whereas with streaming sources we need `spark.readStream` to create a `DataStreamReader`. `DataStreamReader` has most of the same methods as `DataFrameReader`, so you can use it in a similar way. Here is an example of creating a DataFrame from a text data stream to be received over a socket connection:

```
# In Python
spark = SparkSession...
lines = (spark
  .readStream.format("socket")
  .option("host", "localhost")
  .option("port", 9999)
  .load())
```

```
// In Scala
val spark = SparkSession...
val lines = spark
  .readStream.format("socket")
  .option("host", "localhost")
  .option("port", 9999)
  .load()
```

This code generates the `lines` DataFrame as an unbounded table of newline-separated text data read from localhost:9999. Note that, similar to batch sources with `spark.read`, this does not immediately start reading the streaming data; it only sets up the configurations necessary for reading the data once the streaming query is explicitly started.

Besides sockets, Apache Spark natively supports reading data streams from Apache Kafka and all the various file-based formats that `DataFrameReader` supports (Parquet, ORC, JSON, etc.). The details of these sources and their supported options are discussed later in this chapter. Furthermore, a streaming query can define multiple input sources, both streaming and batch, which can be combined using DataFrame operations like unions and joins (also discussed later in this chapter).

Step 2: Transform data

Now we can apply the usual DataFrame operations, such as splitting the lines into individual words and then counting them, as shown in the following code:

```
# In Python
from pyspark.sql.functions import *
words = lines.select(explode(split(col("value"), "\\s")).alias("word"))
counts = words.groupBy("word").count()
```

```
// In Scala
import org.apache.spark.sql.functions._
val words = lines.select(explode(split(col("value"), "\\s")).as("word"))
val counts = words.groupBy("word").count()
```

`counts` is a *streaming DataFrame* (that is, a DataFrame on unbounded, streaming data) that represents the running word counts that will be computed once the streaming query is started and the streaming input data is being continuously processed.

Note that these operations to transform the `lines` streaming DataFrame would work in the exact same way if `lines` were a batch DataFrame. In general, most DataFrame operations that can be applied on a batch DataFrame can also be applied on a streaming DataFrame. To understand which operations are supported in Structured Streaming, you have to recognize the two broad classes of data transformations:

Stateless transformations

Operations like `select()`, `filter()`, `map()`, etc. do not require any information from previous rows to process the next row; each row can be processed by itself. The lack of previous "state" in these operations make them stateless. Stateless operations can be applied to both batch and streaming DataFrames.

Stateful transformations

> In contrast, an aggregation operation like `count()` requires maintaining state to combine data across multiple rows. More specifically, any DataFrame operations involving grouping, joining, or aggregating are stateful transformations. While many of these operations are supported in Structured Streaming, a few combinations of them are not supported because it is either computationally hard or infeasible to compute them in an incremental manner.

The stateful operations supported by Structured Streaming and how to manage their state at runtime are discussed later in the chapter.

Step 3: Define output sink and output mode

After transforming the data, we can define how to write the processed output data with `DataFrame.writeStream` (instead of `DataFrame.write`, used for batch data). This creates a `DataStreamWriter` which, similar to `DataFrameWriter`, has additional methods to specify the following:

- Output writing details (where and how to write the output)
- Processing details (how to process data and how to recover from failures)

Let's start with the output writing details (we will focus on the processing details in the next step). For example, the following snippet shows how to write the final `counts` to the console:

```
# In Python
writer = counts.writeStream.format("console").outputMode("complete")
```

```
// In Scala
val writer = counts.writeStream.format("console").outputMode("complete")
```

Here we have specified `"console"` as the output streaming sink and `"complete"` as the output mode. The output mode of a streaming query specifies what part of the updated output to write out after processing new input data. In this example, as a chunk of new input data is processed and the word counts are updated, we can choose to print to the console either the counts of all the words seen until now (that is, *complete mode*), or only those words that were updated in the last chunk of input data. This is decided by the specified output mode, which can be one of the following (as we already saw in "The Programming Model of Structured Streaming"):

Append mode

This is the default mode, where only the new rows added to the result table/DataFrame (for example, the `counts` table) since the last trigger will be output to the sink. Semantically, this mode guarantees that any row that is output is never going to be changed or updated by the query in the future. Hence, append mode is supported by only those queries (e.g., stateless queries) that will never modify previously output data. In contrast, our word count query can update previously generated counts; therefore, it does not support append mode.

Complete mode

In this mode, all the rows of the result table/DataFrame will be output at the end of every trigger. This is supported by queries where the result table is likely to be much smaller than the input data and therefore can feasibly be retained in memory. For example, our word count query supports complete mode because the counts data is likely to be far smaller than the input data.

Update mode

In this mode, only the rows of the result table/DataFrame that were updated since the last trigger will be output at the end of every trigger. This is in contrast to append mode, as the output rows may be modified by the query and output again in the future. Most queries support update mode.

NOTE

Complete details on the output modes supported by different queries can be found in the latest Structured Streaming Programming Guide.

Besides writing the output to the console, Structured Streaming natively supports streaming writes to files and Apache Kafka. In addition, you can write to arbitrary locations using the `foreachBatch()` and `foreach()` API methods. In fact, you can use `foreachBatch()` to write streaming outputs using existing batch data sources (but you will lose exactly-once guarantees). The details of these sinks and their supported options are discussed later in this chapter.

Step 4: Specify processing details

The final step before starting the query is to specify details of how to process the data. Continuing with our word count example, we are going to specify the processing details as follows:

```
# In Python
checkpointDir = "..."
writer2 = (writer
  .trigger(processingTime="1 second")
  .option("checkpointLocation", checkpointDir))
```

```
// In Scala
import org.apache.spark.sql.streaming._
val checkpointDir = "..."
val writer2 = writer
  .trigger(Trigger.ProcessingTime("1 second"))
  .option("checkpointLocation", checkpointDir)
```

Here we have specified two types of details using the `DataStreamWriter` that we created with `DataFrame.writeStream`:

Triggering details

This indicates when to trigger the discovery and processing of newly available streaming data. There are four options:

Default

When the trigger is not explicitly specified, then by default, the streaming query executes data in micro-batches where the next micro-batch is triggered as soon as the previous micro-batch has completed.

Processing time with trigger interval

You can explicitly specify the `ProcessingTime` trigger with an interval, and the query will trigger micro-batches at that fixed interval.

Once

In this mode, the streaming query will execute exactly one micro-batch—it processes all the new data available in a single batch and then stops itself. This is useful when you want to control the triggering and processing from an external scheduler that will restart the query using any custom schedule (e.g., to control cost by only executing a query once per day).

Continuous

This is an experimental mode (as of Spark 3.0) where the streaming query will process data continuously instead of in micro-batches. While only a small subset of DataFrame operations allow this mode to be used, it can provide much lower latency (as low as milliseconds) than the micro-batch trigger modes. Refer to the latest Structured Streaming Programming Guide for the most up-to-date information.

Checkpoint location

This is a directory in any HDFS-compatible filesystem where a streaming query saves its progress information—that is, what data has been successfully processed. Upon failure, this metadata is used to restart the failed query exactly where it left off. Therefore, setting this option is necessary for failure recovery with exactly-once guarantees.

Step 5: Start the query

Once everything has been specified, the final step is to start the query, which you can do with the following:

```
# In Python
streamingQuery = writer2.start()
```

```
// In Scala
val streamingQuery = writer2.start()
```

The returned object of type `streamingQuery` represents an active query and can be used to manage the query, which we will cover later in this chapter.

Note that `start()` is a nonblocking method, so it will return as soon as the query has started in the background. If you want the main thread to block until the streaming query has terminated, you can use `streamingQuery.awaitTermination()`. If the query fails in the background with an error, `awaitTermination()` will also fail with that same exception.

You can wait up to a timeout duration using `awaitTermination(timeoutMillis)`, and you can explicitly stop the query with `streamingQuery.stop()`.

Putting it all together

To summarize, here is the complete code for reading streams of text data over a socket, counting the words, and printing the counts to the console:

```
# In Python
from pyspark.sql.functions import *
spark = SparkSession...
lines = (spark
  .readStream.format("socket")
  .option("host", "localhost")
  .option("port", 9999)
  .load())

words = lines.select(explode(split(col("value"), "\\s")).alias("word"))
counts = words.groupBy("word").count()
checkpointDir = "..."
streamingQuery = (counts
  .writeStream
  .format("console")
  .outputMode("complete")
  .trigger(processingTime="1 second")
  .option("checkpointLocation", checkpointDir)
  .start())
streamingQuery.awaitTermination()
```

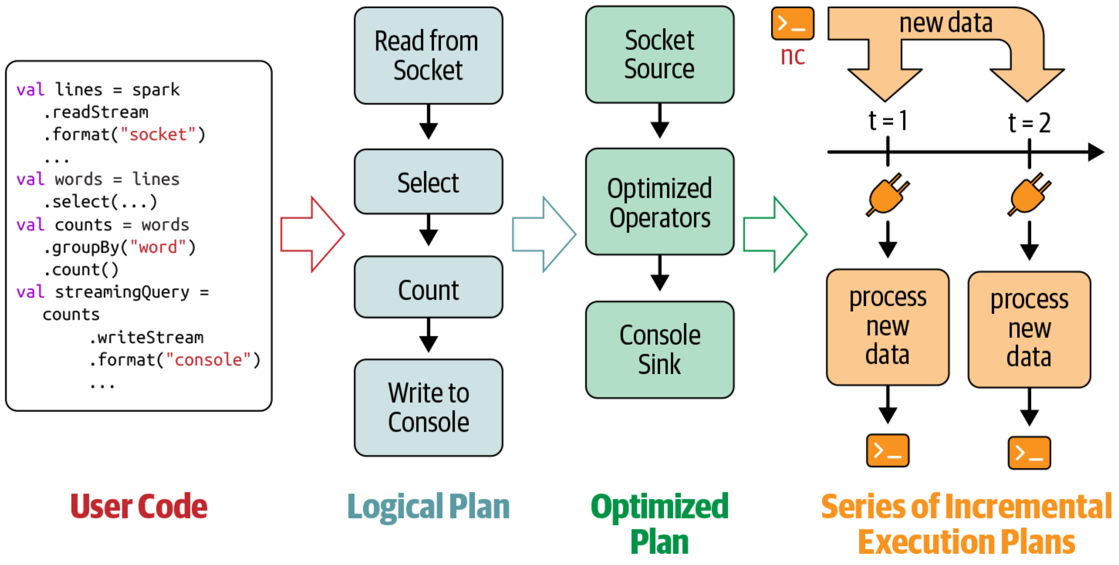
```
  // In Scala
  import org.apache.spark.sql.functions._
  import org.apache.spark.sql.streaming._
  val spark = SparkSession...
  val lines = spark
    .readStream.format("socket")
    .option("host", "localhost")
    .option("port", 9999)
    .load()

  val words = lines.select(explode(split(col("value"), "\\s")).as("word"))
  val counts = words.groupBy("word").count()

  val checkpointDir = "..."
  val streamingQuery = counts.writeStream
    .format("console")
    .outputMode("complete")
    .trigger(Trigger.ProcessingTime("1 second"))
    .option("checkpointLocation", checkpointDir)
    .start()
  streamingQuery.awaitTermination()
```

After the query has started, a background thread continuously reads new data from the streaming source, processes it, and writes it to the streaming sinks. Next, let's take a quick peek under the hood at how this is executed.

Under the Hood of an Active Streaming Query

Once the query starts, the following sequence of steps transpires in the engine, as depicted in Figure 8-5. The DataFrame operations are converted into a logical plan, which is an abstract representation of the computation that Spark SQL uses to plan a query:

1. Spark SQL analyzes and optimizes this logical plan to ensure that it can be executed incrementally and efficiently on streaming data.
2. Spark SQL starts a background thread that continuously executes the following loop:[2]
    1. Based on the configured trigger interval, the thread checks the streaming sources for the availability of new data.
    2. If available, the new data is executed by running a micro-batch. From the optimized logical plan, an optimized Spark execution plan is generated that reads the new data from the source, incrementally computes the updated result, and writes the output to the sink according to the configured output mode.
    3. For every micro-batch, the exact range of data processed (e.g., the set of files or the range of Apache Kafka offsets) and any associated state are saved in the configured checkpoint location so that the query can deterministically reprocess the exact range if needed.
3. This loop continues until the query is terminated, which can occur for one of the following reasons:
    1. A failure has occurred in the query (either a processing error or a failure in the cluster).
    2. The query is explicitly stopped using `streamingQuery.stop()`.
    3. If the trigger is set to `Once`, then the query will stop on its own after executing a single micro-batch containing all the available data.



```
val lines = spark
   .readStream
   .format("socket")
   ...
val words = lines
   .select(...)
val counts = words
   .groupBy("word")
   .count()
val streamingQuery =
   counts
        .writeStream
        .format("console")
        ...
```

Figure 8-5. Incremental execution of streaming queries

NOTE

A key point you should remember about Structured Streaming is that underneath it is using Spark SQL to execute the data. As such, the full power of Spark SQL's hyperoptimized execution engine is utilized to maximize the stream processing throughput, providing key performance advantages.

Next, we will discuss how to restart a streaming query after termination and the life cycle of a streaming query.

Recovering from Failures with Exactly-Once Guarantees

To restart a terminated query in a completely new process, you have to create a new `SparkSession`, redefine all the DataFrames, and start the streaming query on the final result using the same checkpoint location as the one used when the query was started the first time. For our word count example, you can simply reexecute the entire code snippet shown earlier, from the definition of `spark` in the first line to the final `start()` in the last line.

The checkpoint location must be the same across restarts because this directory contains the unique identity of a streaming query and determines the life cycle of the query. If the checkpoint directory is deleted or the same query is started with a different checkpoint directory, it is like starting a new query from scratch. Specifically, checkpoints have record-level information (e.g., Apache Kafka offsets) to track the data range the last incomplete micro-batch was processing. The restarted query will use this information to start processing records precisely after the last successfully completed micro-batch. If the previous query had planned a micro-batch but had terminated before completion, then the restarted query will reprocess the same range of data before processing new data. Coupled with Spark's deterministic task execution, the regenerated output will be the same as it was expected to be before the restart.

Structured Streaming can ensure *end-to-end exactly-once guarantees* (that is, the output is as if each input record was processed exactly once) when the following conditions have been satisfied:

Replayable streaming sources

> The data range of the last incomplete micro-batch can be reread from the source.

Deterministic computations

> All data transformations deterministically produce the same result when given the same input data.

Idempotent streaming sink

> The sink can identify reexecuted micro-batches and ignore duplicate writes that may be caused by restarts.

Note that our word count example does not provide exactly-once guarantees because the socket source is not replayable and the console sink is not idempotent.

As a final note regarding restarting queries, it is possible to make minor modifications to a query between restarts. Here are a few ways you can modify the query:

DataFrame transformations

> You can make minor modifications to the transformations between restarts. For example, in our streaming word count example, if you

want to ignore lines that have corrupted byte sequences that can crash the query, you can add a filter in the transformation:

```
# In Python
# isCorruptedUdf = udf to detect corruption in string

filteredLines = lines.filter("isCorruptedUdf(value) = false")
words = lines.select(explode(split(col("value"), "\\s")).alias("word"))
```

```
// In Scala
// val isCorruptedUdf = udf to detect corruption in string

val filteredLines = lines.filter("isCorruptedUdf(value) = false")
val words = lines.select(explode(split(col("value"), "\\s")).as("word"))
```

Upon restarting with this modified `words` DataFrame, the restarted query will apply the filter on all data processed since the restart (including the last incomplete micro-batch), preventing the query from failing again.

Source and sink options

Whether a `readStream` or `writeStream` option can be changed between restarts depends on the semantics of the specific source or sink. For example, you should not change the `host` and `port` options for the socket source if data is going to be sent to that host and port. But you can add an option to the console sink to print up to one hundred changed counts after every trigger:

```
writeStream.format("console").option("numRows", "100")...
```

Processing details

As discussed earlier, the checkpoint location must not be changed between restarts. However, other details like trigger interval can be changed without breaking fault-tolerance guarantees.

For more information on the narrow set of changes that are allowed between restarts, see the latest Structured Streaming Programming Guide.

Monitoring an Active Query

An important part of running a streaming pipeline in production is tracking its health. Structured Streaming provides several ways to track the status and processing metrics of an active query.

Querying current status using StreamingQuery

You can query the current health of an active query using the `StreamingQuery` instance. Here are two methods:

Get current metrics using StreamingQuery

When a query processes some data in a micro-batch, we consider it to have made some progress. `lastProgress()` returns information on the last completed micro-batch. For example, printing the returned object (`StreamingQueryProgress` in Scala/Java or a dictionary in Python) will produce something like this:

```
// In Scala/Python
{
  "id" : "ce011fdc-8762-4dcb-84eb-a77333e28109",
  "runId" : "88e2ff94-ede0-45a8-b687-6316fbef529a",
  "name" : "MyQuery",
  "timestamp" : "2016-12-14T18:45:24.873Z",
  "numInputRows" : 10,
  "inputRowsPerSecond" : 120.0,
  "processedRowsPerSecond" : 200.0,
  "durationMs" : {
    "triggerExecution" : 3,
    "getOffset" : 2
  },
  "stateOperators" : [ ],
  "sources" : [ {
    "description" : "KafkaSource[Subscribe[topic-0]]",
    "startOffset" : {
      "topic-0" : {
        "2" : 0,
        "1" : 1,
        "0" : 1
      }
    },
    "endOffset" : {
      "topic-0" : {
        "2" : 0,
        "1" : 134,
        "0" : 534
      }
    },
    "numInputRows" : 10,
    "inputRowsPerSecond" : 120.0,
    "processedRowsPerSecond" : 200.0
  } ],
  "sink" : {
    "description" : "MemorySink"
  }
}
```

Some of the noteworthy columns are:

`id`

Unique identifier tied to a checkpoint location. This stays the same throughout the lifetime of a query (i.e., across restarts).

`runId`

Unique identifier for the current (re)started instance of the query. This changes with every restart.

`numInputRows`

Number of input rows that were processed in the last micro-batch.

`inputRowsPerSecond`

Current rate at which input rows are being generated at the source (average over the last micro-batch duration).

`processedRowsPerSecond`

Current rate at which rows are being processed and written out by the sink (average over the last micro-batch duration). If this rate is consistently lower than the input rate, then the query is unable to process data as fast as it is being generated by the source. This is a key indicator of the health of the query.

`sources and sink`

Provides source/sink-specific details of the data processed in the last batch.

Get current status using StreamingQuery.status()

This provides information on what the background query thread is doing at this moment. For example, printing the returned object will produce something like this:

```
// In Scala/Python
{
  "message" : "Waiting for data to arrive",
  "isDataAvailable" : false,
  "isTriggerActive" : false
}
```

Publishing metrics using Dropwizard Metrics

Spark supports reporting metrics via a popular library called Dropwizard Metrics. This library allows metrics to be published to many popular monitoring frameworks (Ganglia, Graphite, etc.). These metrics are by default not enabled for Structured Streaming queries due to their high volume of reported data. To enable them, apart from configuring Dropwizard Metrics for Spark, you have to explicitly set the `SparkSession` configuration `spark.sql.streaming.metricsEnabled` to `true` before starting your query.

Note that only a subset of the information available through `StreamingQuery.lastProgress()` is published through Dropwizard Metrics. If you want to continuously publish more progress information to arbitrary locations, you have to write custom listeners, as discussed next.

Publishing metrics using custom StreamingQueryListeners

`StreamingQueryListener` is an event listener interface with which you can inject arbitrary logic to continuously publish metrics. This developer API is available only in Scala/Java. There are two steps to using custom listeners:

1. Define your custom listener. The `StreamingQueryListener` interface provides three methods that can be defined by your implementation to get three types of events related to a streaming query: start, progress (i.e., a trigger was executed), and termination. Here is an example:

   ```
   // In Scala
   import org.apache.spark.sql.streaming._
   val myListener = new StreamingQueryListener() {
     override def onQueryStarted(event: QueryStartedEvent): Unit = {
       println("Query started: " + event.id)
     }
     override def onQueryTerminated(event: QueryTerminatedEvent): Unit = {
       println("Query terminated: " + event.id)
     }
     override def onQueryProgress(event: QueryProgressEvent): Unit = {
       println("Query made progress: " + event.progress)
     }
   }
   ```

2. Add your listener to the `SparkSession` before starting the query:

   ```
   // In Scala
   spark.streams.addListener(myListener)
   ```

   After adding the listener, all events of streaming queries running on this `SparkSession` will start calling the listener's methods.

Streaming Data Sources and Sinks

Now that we have covered the basic steps you need to express an end-to-end Structured Streaming query, let's examine how to use the built-in streaming data sources and sinks. As a reminder, you can create DataFrames from streaming sources using `SparkSession.readStream()` and write the output from a result DataFrame using `DataFrame.writeStream()`. In each case, you can specify the source type using the method `format()`. We will see a few concrete examples later.

Files

Structured Streaming supports reading and writing data streams to and from files in the same formats as the ones supported in batch processing: plain text, CSV, JSON, Parquet, ORC, etc. Here we will discuss how to operate Structured Streaming on files.

Reading from files

Structured Streaming can treat files written into a directory as a data stream. Here is an example:

```
# In Python
from pyspark.sql.types import *
inputDirectoryOfJsonFiles =  ...

fileSchema = (StructType()
  .add(StructField("key", IntegerType()))
  .add(StructField("value", IntegerType())))

inputDF = (spark
  .readStream
  .format("json")
  .schema(fileSchema)
  .load(inputDirectoryOfJsonFiles))


// In Scala
import org.apache.spark.sql.types._
val inputDirectoryOfJsonFiles =  ...

val fileSchema = new StructType()
  .add("key", IntegerType)
  .add("value", IntegerType)

val inputDF = spark.readStream
  .format("json")
  .schema(fileSchema)
  .load(inputDirectoryOfJsonFiles)
```

The returned streaming DataFrame will have the specified schema. Here are a few key points to remember when using files:

- All the files must be of the same format and are expected to have the same schema. For example, if the format is `"json"`, all the files must be in the JSON format with one JSON record per line. The schema of each JSON record must match the one specified with `readStream()`. Violation of these assumptions can lead to incorrect parsing (e.g., unexpected `null` values) or query failures.
- Each file must appear in the directory listing atomically—that is, the whole file must be available at once for reading, and once it is available, the file cannot be updated or modified. This is because Structured Streaming will process the file when the engine finds it (using directory listing) and internally mark it as processed. Any changes to that file will not be processed.
- When there are multiple new files to process but it can only pick some of them in the next micro-batch (e.g., because of rate limits), it will select the files with the earliest timestamps. Within the micro-batch, however, there is no predefined order of reading of the selected files; all of them will be read in parallel.

NOTE

This streaming file source supports a number of common options, including the file format–specific options supported by `spark.read()` (see "Data Sources for DataFrames and SQL Tables" in Chapter 4) and several streaming-specific options (e.g., `maxFilesPerTrigger` to limit the file processing rate). See the programming guide for full details.

Writing to files

Structured Streaming supports writing streaming query output to files in the same formats as reads. However, it only supports append mode, because while it is easy to write new files in the output directory (i.e., append data to a directory), it is hard to modify existing data files (as would be expected with update and complete modes). It also supports partitioning. Here is an example:

```
# In Python
outputDir = ...
checkpointDir = ...
resultDF = ...

streamingQuery = (resultDF.writeStream
  .format("parquet")
  .option("path", outputDir)
  .option("checkpointLocation", checkpointDir)
  .start())
```

```
// In Scala
val outputDir = ...
val checkpointDir = ...
val resultDF = ...

val streamingQuery = resultDF
  .writeStream
  .format("parquet")
  .option("path", outputDir)
  .option("checkpointLocation", checkpointDir)
  .start()
```

Instead of using the `"path"` option, you can specify the output directory directly as `start(outputDir)`.

A few key points to remember:

- Structured Streaming achieves end-to-end exactly-once guarantees when writing to files by maintaining a log of the data files that have been written to the directory. This log is maintained in the subdirectory *_spark_metadata*. Any Spark query on the directory (not its subdirectories) will automatically use the log to read the correct set of data files so that the exactly-once guarantee is maintained (i.e., no duplicate data or partial files are read). Note that other processing engines may not be aware of this log and hence may not provide the same guarantee.
- If you change the schema of the result DataFrame between restarts, then the output directory will have data in multiple schemas. These

schemas have to be reconciled when querying the directory.

Apache Kafka

Apache Kafka is a popular publish/subscribe system that is widely used for storage of data streams. Structured Streaming has built-in support for reading from and writing to Apache Kafka.

Reading from Kafka

To perform distributed reads from Kafka, you have to use options to specify how to connect to the source. Say you want to subscribe to data from the topic `"events"`. Here is how you can create a streaming DataFrame:

```
# In Python
inputDF = (spark
  .readStream
  .format("kafka")
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
  .option("subscribe", "events")
  .load())
```

```
// In Scala
val inputDF = spark
  .readStream
  .format("kafka")
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
  .option("subscribe", "events")
  .load()
```

The returned DataFrame will have the schema described in Table 8-1.

Table 8-1. Schema of the DataFrame generated by the Kafka source

Column name	Column type	Description
`key`	`binary`	Key data of the record as bytes.
`value`	`binary`	Value data of the record as bytes.
`topic`	`string`	Kafka topic the record was in. This is useful when subscribed to multiple topics.
`partition`	`int`	Partition of the Kafka topic the record was in.
`offset`	`long`	Offset value of the record.
`timestamp`	`long`	Timestamp associated with the record.
`timestampType`	`int`	Enumeration for the type of the timestamp associated with the record.

You can also choose to subscribe to multiple topics, a pattern of topics, or even a specific partition of a topic. Furthermore, you can choose whether to read only new data in the subscribed-to topics or process all the available data in those topics. You can even read Kafka data from batch queries—that is, treat Kafka topics like tables. See the Kafka Integration Guide for more details.

Writing to Kafka

For writing to Kafka, Structured Streaming expects the result DataFrame to have a few columns of specific names and types, as outlined in Table 8-2.

Table 8-2. Schema of DataFrame that can be written to the Kafka sink

Column name	Column type	Description
`key` (optional)	`string` or `binary`	If present, the bytes will be written as the Kafka record key; otherwise, the key will be empty.
`value` (required)	`string` or `binary`	The bytes will be written as the Kafka record value.
`topic` (required only if `"topic"` is not specified as option)	`string`	If `"topic"` is not specified as an option, this determines the topic to write the key/value to. This is useful for fanning out the writes to multiple topics. If the `"topic"` option has been specified, this value is ignored.

You can write to Kafka in all three output modes, though complete mode is not recommended as it will repeatedly output the same records. Here is a concrete example of writing the output of our earlier word count query into Kafka in update mode:

```python
# In Python
counts = ... # DataFrame[word: string, count: long]
streamingQuery = (counts
  .selectExpr(
    "cast(word as string) as key",
    "cast(count as string) as value")
  .writeStream
  .format("kafka")
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
  .option("topic", "wordCounts")
  .outputMode("update")
  .option("checkpointLocation", checkpointDir)
  .start())
```

```
// In Scala
val counts = ... // DataFrame[word: string, count: long]
val streamingQuery = counts
  .selectExpr(
    "cast(word as string) as key",
    "cast(count as string) as value")
  .writeStream
  .format("kafka")
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2")
  .option("topic", "wordCounts")
  .outputMode("update")
  .option("checkpointLocation", checkpointDir)
  .start()
```

See the Kafka Integration Guide for more details.

Custom Streaming Sources and Sinks

In this section, we will discuss how to read and write to storage systems that do not have built-in support in Structured Streaming. In particular, you'll see how to use the `foreachBatch()` and `foreach()` methods to implement custom logic to write to your storage.

Writing to any storage system

There are two operations that allow you to write the output of a streaming query to arbitrary storage systems: `foreachBatch()` and `foreach()`. They have slightly different use cases: while `foreach()` allows custom write logic on every row, `foreachBatch()` allows arbitrary operations and custom logic on the output of each micro-batch. Let's explore their usage in more detail.

Using foreachBatch()

`foreachBatch()` allows you to specify a function that is executed on the output of every micro-batch of a streaming query. It takes two parameters: a DataFrame or Dataset that has the output of a micro-batch, and the unique identifier of the micro-batch. As an example, say we want to write the output of our earlier word count query to Apache Cassandra. As of

Spark Cassandra Connector 2.4.2, there is no support for writing streaming DataFames. But you can use the connector's batch DataFrame support to write the output of each batch (i.e., updated word counts) to Cassandra, as shown here:

```
# In Python
hostAddr = "<ip address>"
keyspaceName = "<keyspace>"
tableName = "<tableName>"

spark.conf.set("spark.cassandra.connection.host", hostAddr)

def writeCountsToCassandra(updatedCountsDF, batchId):
    # Use Cassandra batch data source to write the updated counts
    (updatedCountsDF
      .write
      .format("org.apache.spark.sql.cassandra")
      .mode("append")
      .options(table=tableName, keyspace=keyspaceName)
      .save())

streamingQuery = (counts
  .writeStream
  .foreachBatch(writeCountsToCassandra)
  .outputMode("update")
  .option("checkpointLocation", checkpointDir)
  .start())
```

```
// In Scala
import org.apache.spark.sql.DataFrame

val hostAddr = "<ip address>"
val keyspaceName = "<keyspace>"
val tableName = "<tableName>"

spark.conf.set("spark.cassandra.connection.host", hostAddr)

def writeCountsToCassandra(updatedCountsDF: DataFrame, batchId: Long) {
    // Use Cassandra batch data source to write the updated counts
    updatedCountsDF
      .write
      .format("org.apache.spark.sql.cassandra")
```

```
        .options(Map("table" -> tableName, "keyspace" -> keyspaceName))
        .mode("append")
        .save()
      }

val streamingQuery = counts
  .writeStream
  .foreachBatch(writeCountsToCassandra _)
  .outputMode("update")
  .option("checkpointLocation", checkpointDir)
  .start()
```

With `foreachBatch()`, you can do the following:

Reuse existing batch data sources

As shown in the previous example, with `foreachBatch()` you can use existing batch data sources (i.e., sources that support writing batch DataFrames) to write the output of streaming queries.

Write to multiple locations

If you want to write the output of a streaming query to multiple locations (e.g., an OLAP data warehouse and an OLTP database), then you can simply write the output DataFrame/Dataset multiple times. However, each attempt to write can cause the output data to be recomputed (including possible rereading of the input data). To avoid recomputations, you should cache the `batchOutputDataFrame`, write it to multiple locations, and then uncache it:

```
# In Python
def writeCountsToMultipleLocations(updatedCountsDF, batchId):
  updatedCountsDF.persist()
  updatedCountsDF.write.format(...).save()  # Location 1
  updatedCountsDF.write.format(...).save()  # Location 2
  updatedCountsDF.unpersist()
```

```
// In Scala
def writeCountsToMultipleLocations(
  updatedCountsDF: DataFrame,
  batchId: Long) {
    updatedCountsDF.persist()
```

```
      updatedCountsDF.write.format(...).save()  // Location 1
      updatedCountsDF.write.format(...).save()  // Location 2
      updatedCountsDF.unpersist()
    }
```

Apply additional DataFrame operations

Many DataFrame API operations are not supported[3] on streaming DataFrames because Structured Streaming does not support generating incremental plans in those cases. Using `foreachBatch()` , you can apply some of these operations on each micro-batch output. However, you will have to reason about the end-to-end semantics of doing the operation yourself.

NOTE

`foreachBatch()` only provides at-least-once write guarantees. You can get exactly-once guarantees by using the `batchId` to deduplicate multiple writes from reexecuted micro-batches.

Using foreach()

If `foreachBatch()` is not an option (for example, if a corresponding batch data writer does not exist), then you can express your custom writer logic using `foreach()` . Specifically, you can express the data-writing logic by dividing it into three methods: `open()` , `process()` , and `close()` . Structured Streaming will use these methods to write each partition of the output records. Here is an abstract example:

```
# In Python
# Variation 1: Using function
def process_row(row):
    # Write row to storage
    pass

query = streamingDF.writeStream.foreach(process_row).start()

# Variation 2: Using the ForeachWriter class
class ForeachWriter:
  def open(self, partitionId, epochId):
    # Open connection to data store
    # Return True if write should continue
    # This method is optional in Python
    # If not specified, the write will continue automatically
    return True

  def process(self, row):
    # Write string to data store using opened connection
    # This method is NOT optional in Python
    pass

  def close(self, error):
    # Close the connection. This method is optional in Python
    pass

resultDF.writeStream.foreach(ForeachWriter()).start()


// In Scala
import org.apache.spark.sql.ForeachWriter
val foreachWriter = new ForeachWriter[String] {  // typed with Strings

    def open(partitionId: Long, epochId: Long): Boolean = {
      // Open connection to data store
      // Return true if write should continue
    }

    def process(record: String): Unit = {
      // Write string to data store using opened connection
    }

    def close(errorOrNull: Throwable): Unit = {
      // Close the connection
    }
 }

resultDSofStrings.writeStream.foreach(foreachWriter).start()
```

The detailed semantics of these methods as executed are discussed in the Structured Streaming Programming Guide.

Reading from any storage system

Unfortunately, as of Spark 3.0, the APIs to build custom streaming sources and sinks are still experimental. The DataSourceV2 initiative in Spark 3.0

introduces the streaming APIs but they are yet to be declared as stable. Hence, there is no official way to read from arbitrary storage systems.

Data Transformations

In this section, we are going to dig deeper into the data transformations supported in Structured Streaming. As briefly discussed earlier, only the DataFrame operations that can be executed incrementally are supported in Structured Streaming. These operations are broadly classified into *stateless* and *stateful* operations. We will define each type of operation and explain how to identify which operations are stateful.

Incremental Execution and Streaming State

As we discussed in "Under the Hood of an Active Streaming Query", the Catalyst optimizer in Spark SQL converts all the DataFrame operations to an optimized logical plan. The Spark SQL planner, which decides how to execute a logical plan, recognizes that this is a streaming logical plan that needs to operate on continuous data streams. Accordingly, instead of converting the logical plan to a one-time physical execution plan, the planner generates a continuous sequence of execution plans. Each execution plan updates the final result DataFrame incrementally—that is, the plan processes only a chunk of new data from the input streams and possibly some intermediate, partial result computed by the previous execution plan.

Each execution is considered as a micro-batch, and the partial intermediate result that is communicated between the executions is called the streaming "state." DataFrame operations can be broadly classified into stateless and stateful operations based on whether executing the operation incrementally requires maintaining a state. In the rest of this section, we are going to explore the distinction between stateless and stateful operations and how their presence in a streaming query requires different runtime configuration and resource management.

NOTE

Some logical operations are fundamentally either impractical or very expensive to compute incrementally, and hence they are not supported in Structured Streaming. For example, any attempt to start a streaming query with an operation like `cube()` or `rollup()` will throw an `UnsupportedOperationException`.

Stateless Transformations

All projection operations (e.g., `select()`, `explode()`, `map()`, `flatMap()`) and selection operations (e.g., `filter()`, `where()`) process each input record individually without needing any information from previous rows. This lack of dependence on prior input data makes them stateless operations.

A streaming query having only stateless operations supports the append and update output modes, but not complete mode. This makes sense: since any processed output row of such a query cannot be modified by any future data, it can be written out to all streaming sinks in append mode (including append-only ones, like files of any format). On the other hand, such queries naturally do not combine information across input records, and therefore may not reduce the volume of the data in the result. Complete mode is not supported because storing the ever-growing result data is usually costly. This is in sharp contrast with stateful transformations, as we will discuss next.

Stateful Transformations

The simplest example of a stateful transformation is `DataFrame.groupBy().count()`, which generates a running count of the number of records received since the beginning of the query. In every micro-batch, the incremental plan adds the count of new records to the previous count generated by the previous micro-batch. This partial count communicated between plans is the state. This state is maintained in the memory of the Spark executors and is checkpointed to the configured location in order to tolerate failures. While Spark SQL automatically manages the life cycle of this state to ensure correct results, you typically have to tweak a few knobs to control the resource usage for maintaining state.

In this section, we are going to explore how different stateful operators manage their state under the hood.

Distributed and fault-tolerant state management

Recall from Chapters 1 and 2 that a Spark application running in a cluster has a driver and one or more executors. Spark's scheduler running in the driver breaks down your high-level operations into smaller tasks and puts them in task queues, and as resources become available, the executors pull the tasks from the queues to execute them. Each micro-batch in a streaming query essentially performs one such set of tasks that read new data from streaming sources and write updated output to streaming sinks. For stateful stream processing queries, besides writing to sinks, each micro-batch of tasks generates intermediate state data which will be consumed by the next micro-batch. This state data generation is completely partitioned and distributed (as all reading, writing, and processing is in Spark), and it is cached in the executor memory for efficient consumption. This is illustrated in Figure 8-6, which shows how the state is managed in our original streaming word count query.

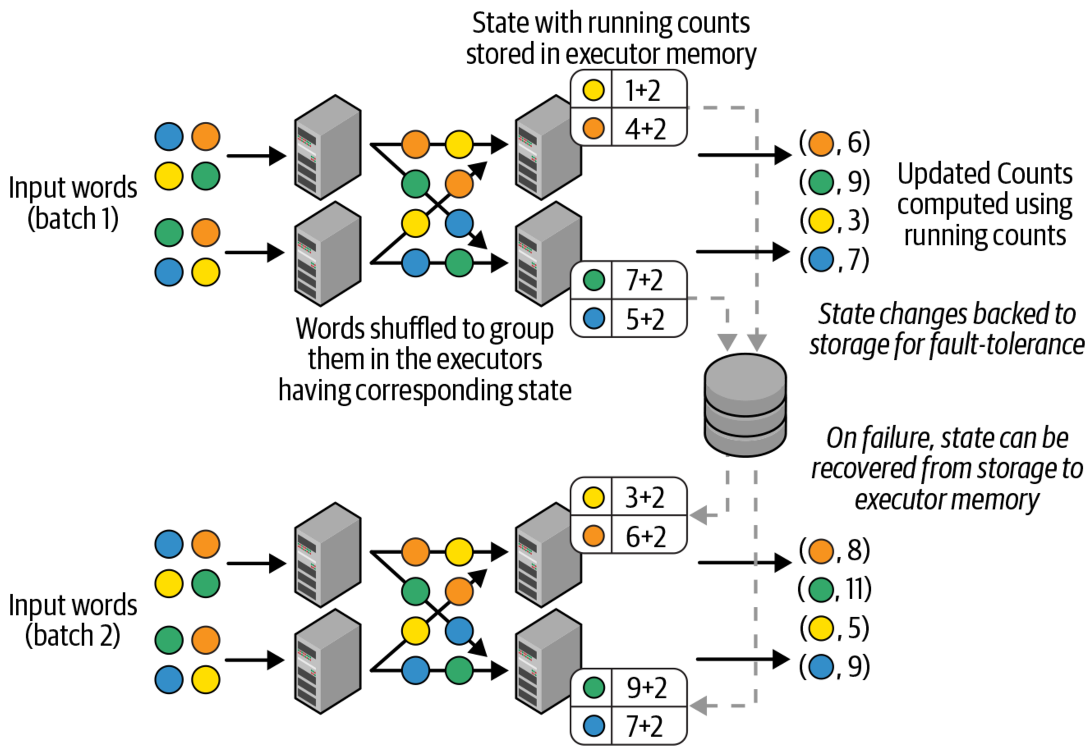

State with running counts stored in executor memory
1+2
4+2
Input words (batch 1)
(●, 6)
(●, 9)
(●, 3)
(●, 7)
Updated Counts computed using running counts
7+2
5+2
Words shuffled to group them in the executors having corresponding state
State changes backed to storage for fault-tolerance
On failure, state can be recovered from storage to executor memory
3+2
6+2
Input words (batch 2)
(●, 8)
(●, 11)
(●, 5)
(●, 9)
9+2
7+2

Figure 8-6. Distributed state management in Structured Streaming

Each micro-batch reads a new set of words, shuffles them within the executors to group them, computes the counts within the micro-batch, and finally adds them to the running counts to produce the new counts. These new counts are both the output and the state for the next micro-batch, and hence they are cached in the memory of the executors. The next micro-batch of data is grouped between executors in exactly the same way as before, so that each word is always processed by the same executor, and can therefore locally read and update its running count.

However, it is not sufficient to just keep this state in memory, as any failure (either of an executor or of the entire application) will cause the in-memory state to be lost. To avoid loss, we synchronously save the key/value state update as change logs in the checkpoint location provided by the user. These changes are co-versioned with the offset ranges processed in each batch, and the required version of the state can be automatically reconstructed by reading the checkpointed logs. In case of any failure, Structured Streaming is able to re-execute the failed micro-batch by reprocessing the same input data along with the same state that it had before that micro-batch, thus producing the same output data as it would have if there had been no failure. This is critical for ensuring end-to-end exactly-once guarantees.

To summarize, for all stateful operations, Structured Streaming ensures the correctness of the operation by automatically saving and restoring the state in a distributed manner. Depending on the stateful operation, all you may have to do is tune the state cleanup policy such that old keys and values can be automatically dropped from the cached state. This is what we will discuss next.

Types of stateful operations

The essence of streaming state is to retain summaries of past data. Sometimes old summaries need to be cleaned up from the state to make room for new summaries. Based on how this is done, we can distinguish two types of stateful operations:

Managed stateful operations

These automatically identify and clean up old state, based on an operation-specific definition of "old." You can tune what is defined as old in order to control the resource usage (e.g., executor memory used to store state). The operations that fall into this category are those for:

- Streaming aggregations
- Stream–stream joins
- Streaming deduplication

Unmanaged stateful operations

These operations let you define your own custom state cleanup logic. The operations in this category are:

- `MapGroupsWithState`
- `FlatMapGroupsWithState`

These operations allow you to define arbitrary stateful operations (sessionization, etc.).

Each of these operations are discussed in detail in the following sections.

Stateful Streaming Aggregations

Structured Streaming can incrementally execute most DataFrame aggregation operations. You can aggregate data by keys (e.g., streaming word count) and/or by time (e.g., count records received every hour). In this section, we are going to discuss the semantics and operational details of tuning these different types of streaming aggregations. We'll also briefly discuss the few types of aggregations that are not supported in streaming. Let's begin with aggregations not involving time.

Aggregations Not Based on Time

Aggregations not involving time can be broadly classified into two categories:

Global aggregations

Aggregations across all the data in the stream. For example, say you have a stream of sensor readings as a streaming DataFrame named `sensorReadings`. You can calculate the running count of the total number of readings received with the following query:

```
# In Python
runningCount = sensorReadings.groupBy().count()
```

```
// In Scala
val runningCount = sensorReadings.groupBy().count()
```

NOTE

You cannot use direct aggregation operations like `DataFrame.count()` and `Dataset.reduce()` on streaming DataFrames. This is because, for static DataFrames, these operations immediately return the final computed aggregates, whereas for streaming DataFrames the aggregates have to be continuously updated. Therefore, you have to always use `DataFrame.groupBy()` or `Dataset.groupByKey()` for aggregations on streaming DataFrames.

Grouped aggregations

Aggregations within each group or key present in the data stream. For example, if `sensorReadings` contains data from multiple sensors, you can calculate the running average reading of each sensor (say, for setting up a baseline value for each sensor) with the following:

```
# In Python
baselineValues = sensorReadings.groupBy("sensorId").mean("value")
```

```
// In Scala
val baselineValues = sensorReadings.groupBy("sensorId").mean("value")
```

Besides counts and averages, streaming DataFrames support the following types of aggregations (similar to batch DataFrames):

All built-in aggregation functions

`sum()`, `mean()`, `stddev()`, `countDistinct()`, `collect_set()`, `approx_count_distinct()`, etc. Refer to the API documentation (Python and Scala) for more details.

Multiple aggregations computed together

You can apply multiple aggregation functions to be computed together in the following manner:

```
# In Python
from pyspark.sql.functions import *
multipleAggs = (sensorReadings
  .groupBy("sensorId")
  .agg(count("*"), mean("value").alias("baselineValue"),
    collect_set("errorCode").alias("allErrorCodes")))
```

```
// In Scala
import org.apache.spark.sql.functions.*
val multipleAggs = sensorReadings
  .groupBy("sensorId")
  .agg(count("*"), mean("value").alias("baselineValue"),
    collect_set("errorCode").alias("allErrorCodes"))
```

User-defined aggregation functions

All user-defined aggregation functions are supported. See the Spark SQL programming guide for more details on untyped and typed user-defined aggregation functions.

Regarding the execution of such streaming aggregations, we have already illustrated in previous sections how the running aggregates are maintained as a distributed state. In addition to this, there are two very important points to remember for aggregations not based on time: the output mode to use for such queries and planning the resource usage by state. These are discussed toward the end of this section. Next, we are going to discuss aggregations that combine data within time windows.

Aggregations with Event-Time Windows

In many cases, rather than running aggregations over the whole stream, you want aggregations over data bucketed by time windows. Continuing with our sensor example, say each sensor is expected to send at most one reading per minute and we want to detect if any sensor is reporting an unusually high number of times. To find such anomalies, we can count the number of readings received from each sensor in five-minute intervals. In addition, for robustness, we should be computing the time interval based on when the data was generated at the sensor and not based on when the data was received, as any transit delay would skew the results. In other words, we want to use the *event time*—that is, the timestamp in the record representing when the reading was generated. Say the `sensorReadings` DataFrame has the generation timestamp as a column named `eventTime`. We can express this five-minute count as follows:

```
# In Python
from pyspark.sql.functions import *
(sensorReadings
  .groupBy("sensorId", window("eventTime", "5 minute"))
  .count())
```

```
// In Scala
import org.apache.spark.sql.functions.*
sensorReadings
  .groupBy("sensorId", window("eventTime", "5 minute"))
  .count()
```

The key thing to note here is the `window()` function, which allows us to express the five-minute windows as a dynamically computed grouping column. When started, this query will effectively do the following for each sensor reading:

- Use the `eventTime` value to compute the five-minute time window the sensor reading falls into.
- Group the reading based on the composite group `(<computed window>, SensorId)`.
- Update the count of the composite group.

Let's understand this with an illustrative example. Figure 8-7 shows how a few sensor readings are mapped to groups of five-minute tumbling (i.e., nonoverlapping) windows based on their event time. The two timelines show when each received event will be processed by Structured Streaming, and the timestamp in the event data (usually, the time when the event was generated at the sensor).

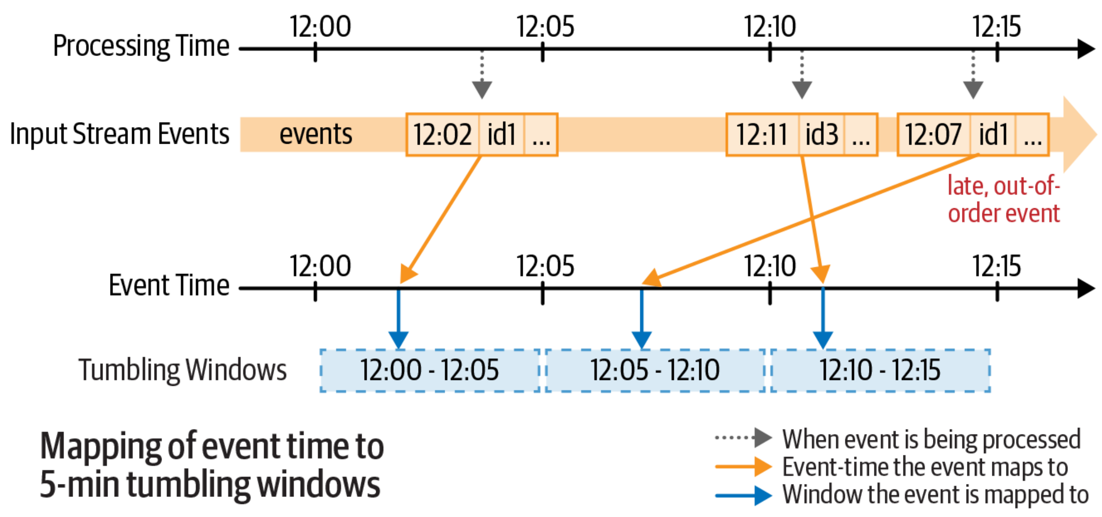


Figure 8-7. Mapping of event time to tumbling windows

Each five-minute window over event time is considered for the grouping based on which the counts will be calculated. Note that events may come late and out of order in terms of event time. As shown in the figure, the event with event time 12:07 was received and processed after the event with time 12:11. However, irrespective of when they arrive, each event is assigned to the appropriate group based on its event time. In fact, depending on the window specification, each event can be assigned to multiple groups. For example, if you want to compute counts corresponding to 10-minute windows sliding every 5 minutes, then you can do the following:

```
# In Python
(sensorReadings
  .groupBy("sensorId", window("eventTime", "10 minute", "5 minute"))
  .count())
```

```
// In Scala
sensorReadings
  .groupBy("sensorId", window("eventTime", "10 minute", "5 minute"))
  .count()
```

In this query, every event will be assigned to two overlapping windows as illustrated in Figure 8-8.

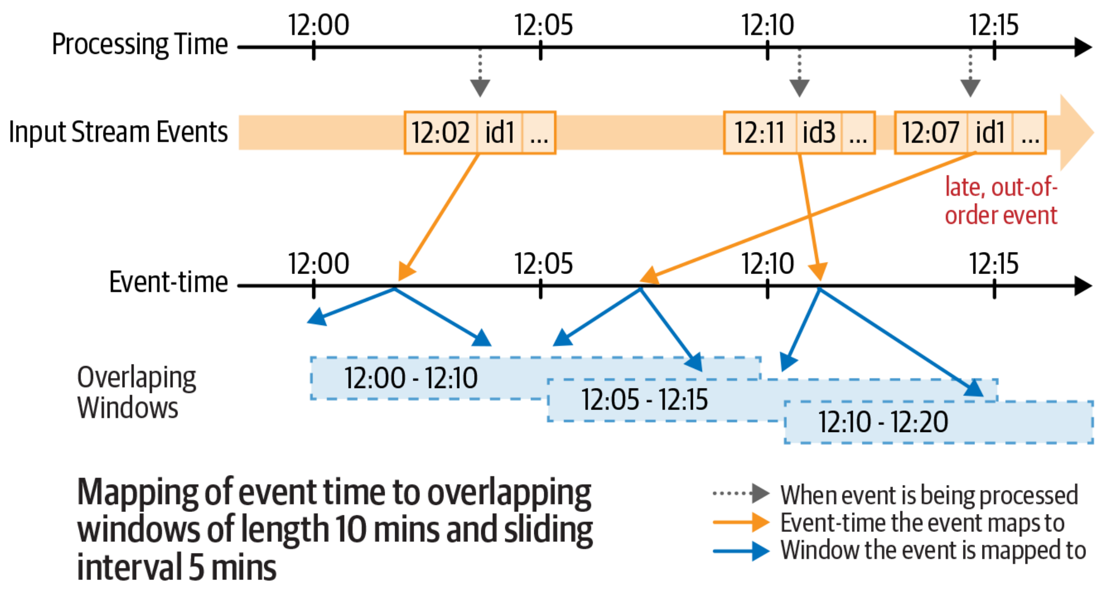


Figure 8-8. Mapping of event time to multiple overlapping windows

Each unique tuple of `(<assigned time window>, sensorId)` is considered a dynamically generated group for which counts will be computed. For example, the event `[eventTime = 12:07, sensorId = id1]` gets mapped to two time windows and therefore two groups, `(12:00-12:10, id1)` and `(12:05-12:15, id1)`. The counts for these two windows are each incremented by 1. Figure 8-9 illustrates this for the previously shown events.

Assuming that the input records were processed with a trigger interval of five minutes, the tables at the bottom of Figure 8-9 show the state of the result table (i.e., the counts) at each of the micro-batches. As the event time moves forward, new groups are automatically created and their aggregates are automatically updated. Late and out-of-order events get handled automatically, as they simply update older groups.

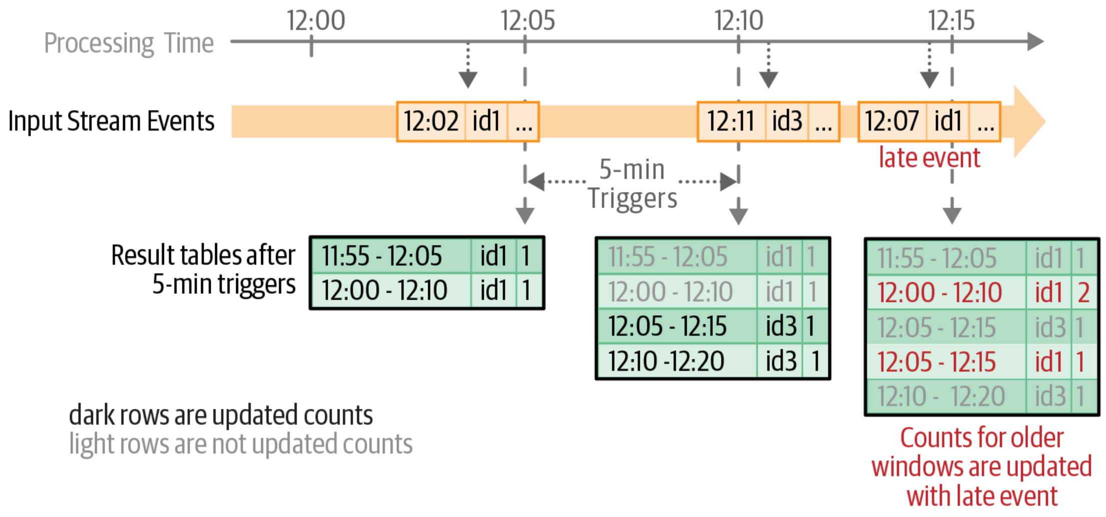


Figure 8-9. Updated counts in the result table after each five-minute trigger

However, from the point of view of resource usage, this poses a different problem—indefinitely growing state size. As new groups are created corresponding to the latest time windows, the older groups continue to occupy the state memory, waiting for any late data to update them. Even if in practice there is a bound on how late the input data can be (e.g., data cannot be more than seven days late), the query does not know that information. Hence, it does not know when to consider a window as "too old to receive updates" and drop it from the state. To provide a lateness bound to a query (and prevent unbounded state), you can specify *watermarks*, as we discuss next.

Handling late data with watermarks

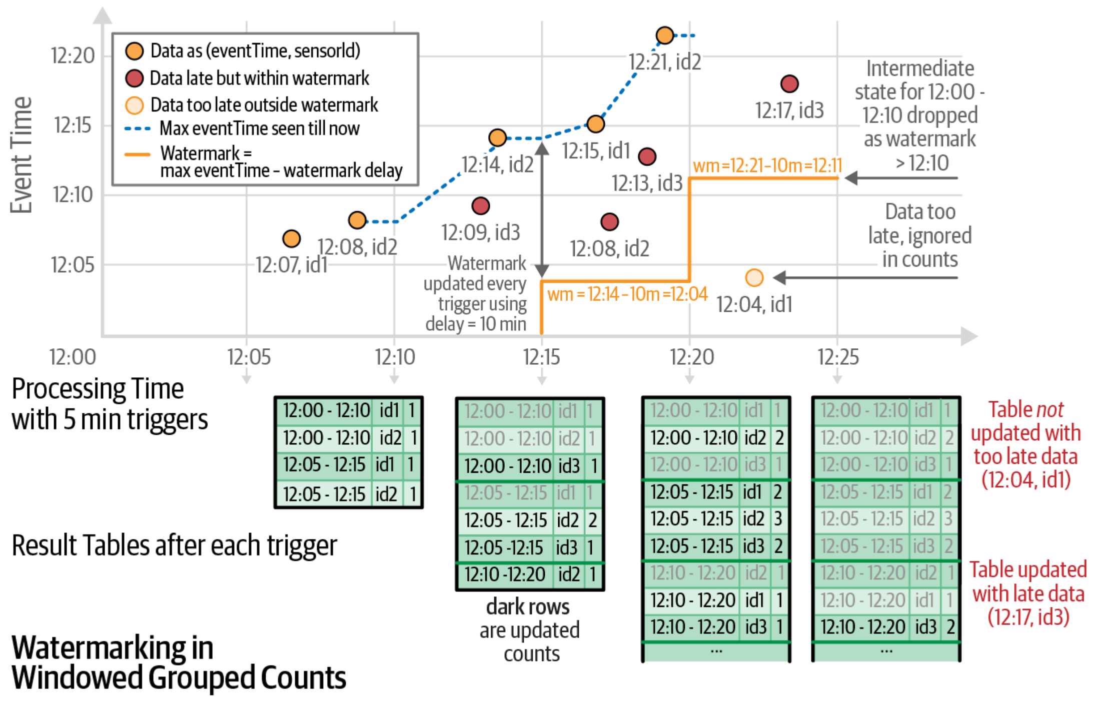
A *watermark* is defined as a moving threshold in event time that trails behind the maximum event time seen by the query in the processed data. The trailing gap, known as the *watermark delay*, defines how long the engine will wait for late data to arrive. By knowing the point at which no more data will arrive for a given group, the engine can automatically finalize the aggregates of certain groups and drop them from the state. This limits the total amount of state that the engine has to maintain to compute the results of the query.

For example, suppose you know that your sensor data will not be late by more than 10 minutes. Then you can set the watermark as follows:

```
# In Python
(sensorReadings
  .withWatermark("eventTime", "10 minutes")
  .groupBy("sensorId", window("eventTime", "10 minutes", "5 minutes"))
  .mean("value"))
```

```
// In Scala
sensorReadings
  .withWatermark("eventTime", "10 minutes")
  .groupBy("sensorId", window("eventTime", "10 minutes", "5 minute"))
  .mean("value")
```

Note that you must call `withWatermark()` before the `groupBy()` and on the same timestamp column as that used to define windows. When this query is executed, Structured Streaming will continuously track the maximum observed value of the `eventTime` column and accordingly update the watermark, filter the "too late" data, and clear old state. That is, any data late by more than 10 minutes will be ignored, and all time windows that are more than 10 minutes older than the latest (by event time) input data will be cleaned up from the state. To clarify how this query will be executed, consider the timeline in Figure 8-10 showing how a selection of input records were processed.



Event Time
12:20
12:15
12:10
12:05
12:00
12:05
12:10
12:15
12:20
12:25
● Data as (eventTime, sensorId)
● Data late but within watermark
○ Data too late outside watermark
Max eventTime seen till now
Watermark = max eventTime - watermark delay
12:07, id1
12:08, id2
12:14, id2
12:09, id3
12:15, id1
12:13, id3
12:08, id2
12:21, id2
12:17, id3
12:04, id1
Watermark updated every trigger using delay = 10 min
wm = 12:14 - 10m = 12:04
wm=12:21-10m=12:11
Intermediate state for 12:00 - 12:10 dropped as watermark > 12:10
Data too late, ignored in counts
Processing Time with 5 min triggers
Result Tables after each trigger
dark rows are updated counts
Table *not* updated with too late data (12:04, id1)
Table updated with late data (12:17, id3)
Watermarking in Windowed Grouped Counts

12:00 - 12:10	id1	1
12:00 - 12:10	id2	1
12:05 - 12:15	id1	1
12:05 - 12:15	id2	1

12:00 - 12:10	id1	1
12:00 - 12:10	id2	1
12:00 - 12:10	id3	1
12:05 - 12:15	id1	1
12:05 - 12:15	id2	2
12:05 -12:15	id3	1
12:10 -12:20	id2	1

12:00 - 12:10	id1	1
12:00 - 12:10	id2	2
12:00 - 12:10	id3	1
12:05 - 12:15	id1	2
12:05 - 12:15	id2	3
12:05 - 12:15	id3	2
12:10 - 12:20	id2	1
12:10 - 12:20	id1	1
12:10 - 12:20	id3	1
...		

12:00 - 12:10	id1	1
12:00 - 12:10	id2	2
12:00 - 12:10	id3	1
12:05 - 12:15	id1	2
12:05 - 12:15	id2	3
12:05 - 12:15	id3	2
12:10 - 12:20	id2	1
12:10 - 12:20	id1	1
12:10 - 12:20	id3	2
...		

Figure 8-10. Illustration of how the engine tracks the maximum event time across events, updates the watermark, and accordingly handles late data

This figure shows a two-dimensional plot of records processed in terms of their processing times (x-axis) and their event times (y-axis). The records are processed in micro-batches of five minutes and marked with circles. The tables at the bottom show the state of the result table after each micro-batch completes.

Each record was received and processed after all the records to its left. Consider the two records `[12:15, id1]` (processed around 12:17) and `[12:13, id3]` (processed around 12:18). The record for `id3` was considered late (and therefore marked in solid red) because it was generated by the sensor before the record for `id1` but it was processed after the latter. However, in the micro-batch for processing-time range 12:15–12:20, the watermark used was 12:04 which was calculated based on the maximum event time seen till the previous micro-batch (that is, 12:14 minus the 10-minute watermark delay). Therefore, the late record `[12:13, id3]` was not considered to be too late and was successfully counted. In contrast, in the next micro-batch, the record `[12:04, id1]` was considered to be too late compared to the new watermark of 12:11 and was discarded.

You can set the watermark delay based on the requirements of your application—larger values for this parameter allow data to arrive later, but at the cost of increased state size (i.e., memory usage), and vice versa.

Semantic guarantees with watermarks

Before we conclude this section about watermarks, let's consider the precise semantic guarantee that watermarking provides. A watermark of 10 minutes guarantees that the engine will *never drop any data* that is delayed by less than 10 minutes compared to the latest event time seen in the input data. However, the guarantee is strict only in one direction. Data delayed by more than 10 minutes is not guaranteed to be dropped—that is, it may get aggregated. Whether an input record more than 10 minutes late will actually be aggregated or not depends on the exact timing of when the record was received and when the micro-batch processing it was triggered.

Supported output modes

Unlike streaming aggregations not involving time, aggregations with time windows can use all three output modes. However, there are other implications regarding state cleanup that you need to be aware of, depending on the mode:

Update mode

In this mode, every micro-batch will output only the rows where the aggregate got updated. This mode can be used with all types of aggregations. Specifically for time window aggregations, watermarking will ensure that the state will get cleaned up regularly. This is the most useful and efficient mode to run queries with streaming aggregations. However, you cannot use this mode to write aggregates to append-only streaming sinks, such as any file-based formats like Parquet and ORC (unless you use Delta Lake, which we will discuss in the next chapter).

Complete mode

In this mode, every micro-batch will output all the updated aggregates, irrespective of their age or whether they contain changes. While this mode can be used on all types of aggregations, for time window aggregations, using complete mode means state will not be cleaned up even if a watermark is specified. Outputting all aggregates requires all past state, and hence aggregation data must be preserved even if a watermark has been defined. Use this mode on time window aggregations with caution, as this can lead to an indefinite increase in state size and memory usage.

Append mode

This mode can be used only with aggregations on event-time windows and with watermarking enabled. Recall that append mode does not allow previously output results to change. For any aggregation without watermarks, every aggregate may be updated with any future data, and hence these cannot be output in append mode. Only when watermarking is enabled on aggregations on event-time windows does the query know when an aggregate is not going to update any further. Hence, instead of outputting the updated rows, append mode outputs each key and its final aggregate value only when the watermark ensures that the aggregate is not going to be updated again. The advantage of this mode is that it allows you to write aggregates to append-only streaming sinks (e.g., files). The disadvantage is that the output will be delayed by the watermark duration—the query has to wait for the trailing watermark to exceed the time window of a key before its aggregate can be finalized.

Streaming Joins

Structured Streaming supports joining a streaming Dataset with another static or streaming Dataset. In this section we will explore what types of joins (inner, outer, etc.) are supported, and how to use watermarks to limit the state stored for stateful joins. We will start with the simple case of joining a data stream and a static Dataset.

Stream–Static Joins

Many use cases require joining a data stream with a static Dataset. For example, let's consider the case of ad monetization. Suppose you are an advertisement company that shows ads on websites and you make money when users click on them. Let's assume that you have a static Dataset of all the ads to be shown (known as impressions), and another stream of events for each time users click on the displayed ads. To calculate the click revenue, you have to match each click in the event stream to the corresponding ad impression in the table. Let's first represent the data as two DataFrames, a static one and a streaming one, as shown here:

```
# In Python
# Static DataFrame [adId: String, impressionTime: Timestamp, ...]
# reading from your static data source
impressionsStatic = spark.read. ...

# Streaming DataFrame [adId: String, clickTime: Timestamp, ...]
# reading from your streaming source
clicksStream = spark.readStream. ...
```

```
// In Scala
// Static DataFrame [adId: String, impressionTime: Timestamp, ...]
// reading from your static data source
val impressionsStatic = spark.read. ...

// Streaming DataFrame [adId: String, clickTime: Timestamp, ...]
// reading from your streaming source
val clicksStream = spark.readStream. ...
```

To match the clicks with the impressions, you can simply apply an inner equi-join between them using the common `adId` column:

```
# In Python
matched = clicksStream.join(impressionsStatic, "adId")
```

```
// In Scala
val matched = clicksStream.join(impressionsStatic, "adId")
```

This is the same code as you would have written if both impressions and clicks were static DataFrames—the only difference is that you use `spark.read()` for batch processing and `spark.readStream()` for a stream. When this code is executed, every micro-batch of clicks is inner-joined against the static impression table to generate the output stream of matched events.

Besides inner joins, Structured Streaming also supports two types of stream–static outer joins:

- Left outer join when the left side is a streaming DataFrame
- Right outer join when the right side is a streaming DataFrame

The other kinds of outer joins (e.g., full outer and left outer with a streaming DataFrame on the right) are not supported because they are not easy to run incrementally. In both supported cases, the code is exactly as it would be for a left/right outer join between two static DataFrames:

```
# In Python
matched = clicksStream.join(impressionsStatic, "adId", "leftOuter")
```

```
// In Scala
val matched = clicksStream.join(impressionsStatic, Seq("adId"), "leftOuter")
```

There are a few key points to note about stream–static joins:

- Stream–static joins are stateless operations, and therefore do not require any kind of watermarking.

- The static DataFrame is read repeatedly while joining with the streaming data of every micro-batch, so you can cache the static DataFrame to speed up the reads.
- If the underlying data in the data source on which the static DataFrame was defined changes, whether those changes are seen by the streaming query depends on the specific behavior of the data source. For example, if the static DataFrame was defined on files, then changes to those files (e.g., appends) will not be picked up until the streaming query is restarted.

In this stream–static example, we made a significant assumption: that the impression table is a static table. In reality, there will be a stream of new impressions generated as new ads are displayed. While stream–static joins are good for enriching data in one stream with additional static (or slowly changing) information, this approach is insufficient when both sources of data are changing rapidly. For that you need stream–stream joins, which we will discuss next.

Stream–Stream Joins

The challenge of generating joins between two data streams is that, at any point in time, the view of either Dataset is incomplete, making it much harder to find matches between inputs. The matching events from the two streams may arrive in any order and may be arbitrarily delayed. For example, in our advertising use case an impression event and its corresponding click event may arrive out of order, with arbitrary delays between them. Structured Streaming accounts for such delays by buffering the input data from both sides as the streaming state, and continuously checking for matches as new data is received. The conceptual idea is sketched out in Figure 8-11.

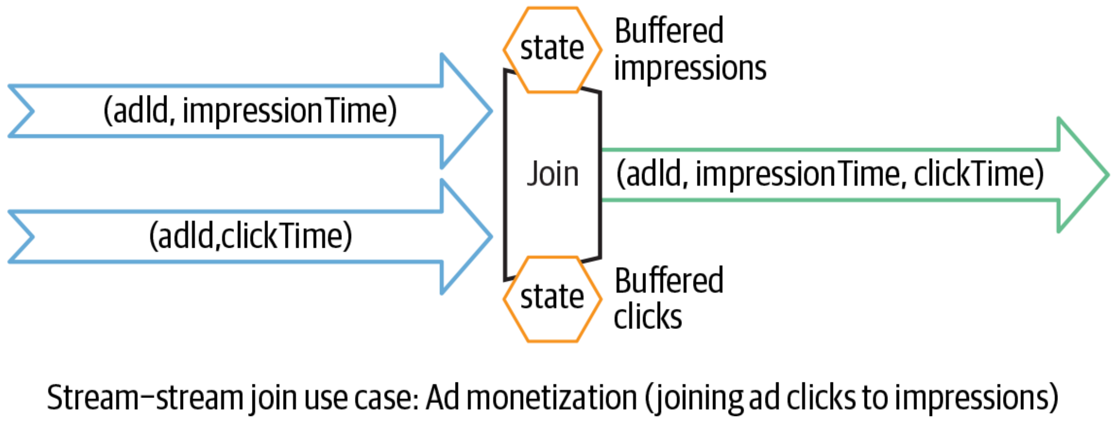

state
Buffered impressions
(adId, impressionTime)
Join
(adId, impressionTime, clickTime)
(adId,clickTime)
state
Buffered clicks
Stream–stream join use case: Ad monetization (joining ad clicks to impressions)

Figure 8-11. Ad monetization using a stream–stream join

Let's consider this in more detail, first with inner joins and then with outer joins.

Inner joins with optional watermarking

Say we have redefined our `impressions` DataFrame to be a streaming DataFrame. To get the stream of matching impressions and their corresponding clicks, we can use the same code we used earlier for static joins and stream–static joins:

```
# In Python
# Streaming DataFrame [adId: String, impressionTime: Timestamp, ...]
impressions = spark.readStream. ...

# Streaming DataFrame[adId: String, clickTime: Timestamp, ...]
clicks = spark.readStream. ...
matched = impressions.join(clicks, "adId")
```

```
// In Scala
// Streaming DataFrame [adId: String, impressionTime: Timestamp, ...]
val impressions = spark.readStream. ...

// Streaming DataFrame[adId: String, clickTime: Timestamp, ...]
val clicks = spark.readStream. ...
val matched = impressions.join(clicks, "adId")
```

Even though the code is the same, the execution is completely different. When this query is executed, the processing engine will recognize it to be a stream–stream join instead of a stream–static join. The engine will buffer all clicks and impressions as state, and will generate a matching impression-and-click as soon as a received click matches a buffered impression (or vice versa, depending on which was received first). Let's visualize how this inner join works using the example timeline of events in Figure 8-12.

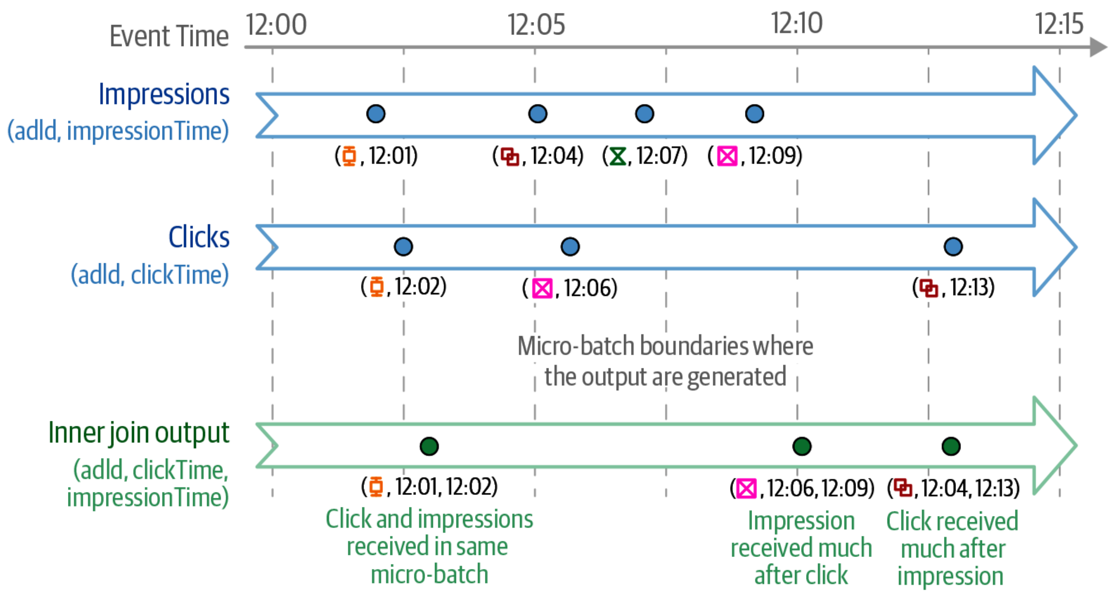


Figure 8-12. Illustrative timeline of clicks, impressions, and their joined output

In Figure 8-12, the blue dots represent the event times of impression and click events that were received across different micro-batches (separated by the dashed grey lines). For the purposes of this illustration, assume that each event was actually received at the same wall clock time as the event time. Note the different scenarios under which the related events are being joined. Both events with `adId = ⌶` were received in the same micro-batch, so their joined output was generated by that micro-batch. However, for `adId = ⧉` the impression was received at 12:04, much earlier than its corresponding click at 12:13. Structured Streaming will first receive the impression at 12:04 and buffer it in the state. For each received click, the engine will try to join it with all buffered impressions (and vice versa). Eventually, in a later micro-batch running around 12:13, the engine receives the click for `adId = ⧉` and generates the joined output.

However, in this query, we have not given any indication of how long the engine should buffer an event to find a match. Therefore, the engine may buffer an event forever and accumulate an unbounded amount of streaming state. To limit the streaming state maintained by stream–stream joins, you need to know the following information about your use case:

- *What is the maximum time range between the generation of the two events at their respective sources?* In the context of our use case, let's assume that a click can occur within zero seconds to one hour after the corresponding impression.
- *What is the maximum duration an event can be delayed in transit between the source and the processing engine?* For example, ad clicks from a browser may get delayed due to intermittent connectivity and arrive much later than expected, and out of order. Let's say that impressions and clicks can be delayed by at most two and three hours, respectively.

These delay limits and event-time constraints can be encoded in the DataFrame operations using watermarks and time range conditions. In other words, you will have to do the following additional steps in the join to ensure state cleanup:

1. Define watermark delays on both inputs, such that the engine knows how delayed the input can be (similar to with streaming aggregations).
2. Define a constraint on event time across the two inputs, such that the engine can figure out when old rows of one input are not going to be required (i.e., will not satisfy the time constraint) for matches with the other input. This constraint can be defined in one of the following ways:
    1. Time range join conditions (e.g., join condition = `"leftTime BETWEEN rightTime AND rightTime + INTERVAL 1 HOUR"`)
    2. Join on event-time windows (e.g., join condition = `"leftTimeWindow = rightTimeWindow"`)

In our advertisement use case, our inner join code will get a little bit more complicated:

```
# In Python
# Define watermarks
impressionsWithWatermark = (impressions
  .selectExpr("adId AS impressionAdId", "impressionTime")
  .withWatermark("impressionTime", "2 hours"))

clicksWithWatermark = (clicks
  .selectExpr("adId AS clickAdId", "clickTime")
  .withWatermark("clickTime", "3 hours"))

# Inner join with time range conditions
(impressionsWithWatermark.join(clicksWithWatermark,
  expr("""
    clickAdId = impressionAdId AND
    clickTime BETWEEN impressionTime AND impressionTime + interval 1 hour""")))
```

```
// In Scala
// Define watermarks
val impressionsWithWatermark = impressions
  .selectExpr("adId AS impressionAdId", "impressionTime")
  .withWatermark("impressionTime", "2 hours ")

val clicksWithWatermark = clicks
  .selectExpr("adId AS clickAdId", "clickTime")
  .withWatermark("clickTime", "3 hours")

// Inner join with time range conditions
impressionsWithWatermark.join(clicksWithWatermark,
  expr("""
    clickAdId = impressionAdId AND
    clickTime BETWEEN impressionTime AND impressionTime + interval 1 hour"""))
```

With these time constraints for each event, the processing engine can automatically calculate how long events need to be buffered to generate correct results, and when the events can be dropped from the state. For example, it will evaluate the following (illustrated in Figure 8-13):

- Impressions need to be buffered for at most four hours (in event time), as a three-hour-late click may match with an impression made

four hours ago (i.e., three hours late + up to one-hour delay between the impression and click).
- Conversely, clicks need to be buffered for at most two hours (in event time), as a two-hour-late impression may match with a click received two hours ago.

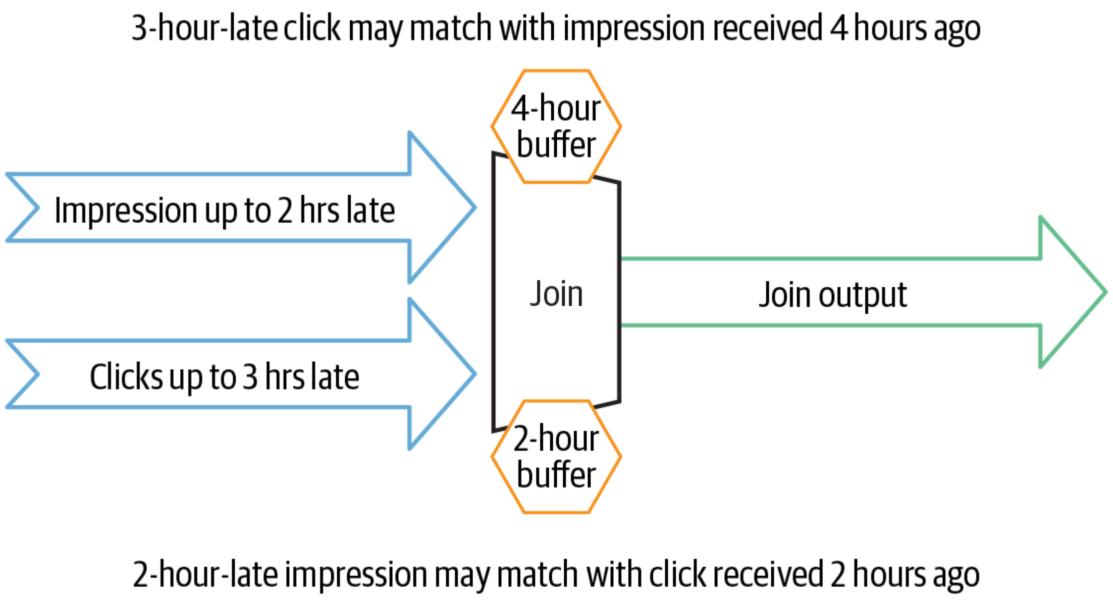

3-hour-late click may match with impression received 4 hours ago
4-hour buffer
Impression up to 2 hrs late
Join
Join output
Clicks up to 3 hrs late
2-hour buffer
2-hour-late impression may match with click received 2 hours ago

Figure 8-13. Structured Streaming automatically calculates thresholds for state cleanup using watermark delays and time range conditions

There are a few key points to remember about inner joins:

- For inner joins, specifying watermarking and event-time constraints are both optional. In other words, at the risk of potentially unbounded state, you may choose not to specify them. Only when both are specified will you get state cleanup.
- Similar to the guarantees provided by watermarking on aggregations, a watermark delay of two hours guarantees that the engine will never drop or not match any data that is less than two hours delayed, but data delayed by more than two hours may or may not get processed.

Outer joins with watermarking

The previous inner join will output only those ads for which both events have been received. In other words, ads that received no clicks will not be reported at all. Instead, you may want all ad impressions to be reported, with or without the associated click data, to enable additional analysis later (e.g., click-through rates). This brings us to *stream–stream outer joins*. All you need to do to implement this is specify the outer join type:

```
# In Python
# Left outer join with time range conditions
(impressionsWithWatermark.join(clicksWithWatermark,
  expr("""
    clickAdId = impressionAdId AND
    clickTime BETWEEN impressionTime AND impressionTime + interval 1 hour"""),
  "leftOuter"))  # only change: set the outer join type
```

```
// In Scala
// Left outer join with time range conditions
impressionsWithWatermark.join(clicksWithWatermark,
  expr("""
    clickAdId = impressionAdId AND
    clickTime BETWEEN impressionTime AND impressionTime + interval 1 hour"""),
  "leftOuter")  // Only change: set the outer join type
```

As expected of outer joins, this query will start generating output for every impression, with or without (i.e., using `NULL` s) the click data. However, there are a few additional points to note about outer joins:

- Unlike with inner joins, the watermark delay and event-time constraints are not optional for outer joins. This is because for generating the `NULL` results, the engine must know when an event is not going to match with anything else in the future. For correct outer join results and state cleanup, the watermarking and event-time constraints must be specified.
- Consequently, the outer `NULL` results will be generated with a delay as the engine has to wait for a while to ensure that there neither were nor would be any matches. This delay is the maximum buffering time (with respect to event time) calculated by the engine for each event as discussed in the previous section (i.e., four hours for impressions and two hours for clicks).

Arbitrary Stateful Computations

Many use cases require more complicated logic than the SQL operations we have discussed up to now. For example, say you want to track the statuses (e.g., signed in, busy, idle) of users by tracking their activities (e.g., clicks) in real time. To build this stream processing pipeline, you will have to track each user's activity history as a state with arbitrary data structure, and continuously apply arbitrarily complex changes on the data structure based on the user's actions. The operation `mapGroupsWithState()` and its more flexible counterpart `flatMapGroupsWithState()` are designed for such complex analytical use cases.

NOTE

As of Spark 3.0, these two operations are only available in Scala and Java.

In this section, we will start with a simple example with `mapGroupsWithState()` to illustrate the four key steps to modeling custom state data and defining custom operations on it. Then we will discuss the concept of timeouts and how you can use them to expire state that has not been updated for a while. We will end with `flatMapGroupsWithState()`, which gives you even more flexibility.

Modeling Arbitrary Stateful Operations with mapGroupsWithState()

State with an arbitrary schema and arbitrary transformations on the state is modeled as a user-defined function that takes the previous version of the state value and new data as inputs, and generates the updated state and computed result as outputs. Programmatically in Scala, you will have to define a function with the following signature (`K`, `V`, `S`, and `U` are data types, as explained shortly):

```
// In Scala
def arbitraryStateUpdateFunction(
    key: K,
    newDataForKey: Iterator[V],
    previousStateForKey: GroupState[S]
): U
```

This function is provided to a streaming query using the operations `groupByKey()` and `mapGroupsWithState()`, as follows:

```
// In Scala
val inputDataset: Dataset[V] =  // input streaming Dataset

inputDataset
  .groupByKey(keyFunction)   // keyFunction() generates key from input
  .mapGroupsWithState(arbitraryStateUpdateFunction)
```

When this streaming query is started, in each micro-batch Spark will call this `arbitraryStateUpdateFunction()` for each unique key in the micro-batch's data. Let's take a closer look at what the parameters are and what parameter values Spark will call the function with:

`key: K`

`K` is the data type of the common keys defined in the state and the input. Spark will call this function for each unique key in the data.

`newDataForKey: Iterator[V]`

`V` is the data type of the input Dataset. When Spark calls this function for a key, this parameter will have all the new input data corresponding to that key. Note that the order in which the input data objects will be present in the iterator is not defined.

`previousStateForKey: GroupState[S]`

`S` is the data type of the arbitrary state you are going to maintain, and `GroupState[S]` is a typed wrapper object that provides methods to access and manage the state value. When Spark calls this function for a key, this object will provide the state value set the previous time Spark called this function for that key (i.e., for one of the previous micro-batches).

`U`

`U` is the data type of the output of the function.

NOTE

There are a couple of additional parameters that you have to provide. All the types (`K` , `V` , `S` , `U`) must be encodable by Spark SQL's encoders. Accordingly, in `mapGroupsWithState()` , you have to provide the typed encoders for `S` and `U` either implicitly in Scala or explicitly in Java. See "Dataset Encoders" in Chapter 6 for more details.

Let's examine how to express the desired state update function in this format with an example. Say we want to understand user behavior based on their actions. Conceptually, it's quite simple: in every micro-batch, for each active user, we will use the new actions taken by the user and update the user's "status." Programmatically, we can define the state update function with the following steps:

1. Define the data types. We need to define the exact types of `K` , `V` , `S` , and `U` . In this case, we'll use the following:
    1. Input data (`V`) = `case class UserAction(userId: String, action: String)`
    2. Keys (`K`) = `String` (that is, the `userId`)
    3. State (`S`) = `case class UserStatus(userId: String, active: Boolean)`
    4. Output (`U`) = `UserStatus` , as we want to output the latest user status

    Note that all these data types are supported in encoders.
2. Define the function. Based on the chosen types, let's translate the conceptual idea into code. When this function is called with new user actions, there are two main situations we need to handle: whether a previous state (i.e., previous user status) exists for that key (i.e., `userId`) or not. Accordingly, we will initialize the user's status, or update the existing status with the new actions. We will explicitly update the state with the new running count, and finally return the updated `userId` - `userStatus` pair:

```
// In Scala
import org.apache.spark.sql.streaming._

 def updateUserStatus(
    userId: String,
    newActions: Iterator[UserAction],
    state: GroupState[UserStatus]): UserStatus = {

  val userStatus = state.getOption.getOrElse {
    new UserStatus(userId, false)
  }
  newActions.foreach { action =>
    userStatus.updateWith(action)
  }
  state.update(userStatus)
  return userStatus
}
```

3. Apply the function on the actions. We will group the input actions Dataset using `groupByKey()` and then apply the `updateUserStatus` function using `mapGroupsWithState()` :

```
// In Scala
val userActions: Dataset[UserAction] = ...
val latestStatuses = userActions
  .groupByKey(userAction => userAction.userId)
  .mapGroupsWithState(updateUserStatus _)
```

Once we start this streaming query with console output, we will see the updated user statuses being printed.

Before we move on to more advanced topics, there are a few notable points to remember:

- When the function is called, there is no well-defined order for the input records in the new data iterator (e.g., `newActions`). If you need to update the state with the input records in a specific order (e.g., in the order the actions were performed), then you have to explicitly reorder them (e.g., based on the event timestamp or some other ordering ID). In fact, if there is a possibility that actions may be read out of order from the source, then you have to consider the possibility that a future micro-batch may receive data that should be processed before the data in the current batch. In that case, you have to buffer the records as part of the state.

- In a micro-batch, the function is called on a key once only if the micro-batch has data for that key. For example, if a user becomes inactive and provides no new actions for a long time, then by default, the function will not be called for a long time. If you want to update or remove state based on a user's inactivity over an extended period you have to use timeouts, which we will discuss in the next section.
- The output of `mapGroupsWithState()` is assumed by the incremental processing engine to be continuously updated key/value records, similar to the output of aggregations. This limits what operations are supported in the query after `mapGroupsWithState()`, and what sinks are supported. For example, appending the output into files is not supported. If you want to apply arbitrary stateful operations with greater flexibility, then you have to use `flatMapGroupsWithState()`. We will discuss that after timeouts.

Using Timeouts to Manage Inactive Groups

In the preceding example of tracking active user sessions, as more users become active, the number of keys in the state will keep increasing, and so will the memory used by the state. Now, in a real-world scenario, users are likely not going to stay active all the time. It may not be very useful to keep the status of inactive users in the state, as it is not going to change again until those users become active again. Hence, we may want to explicitly drop all information for inactive users. However, a user may not explicitly take any action to become inactive (e.g., explicitly logging off), and we may have to define inactivity as lack of any action for a threshold duration. This becomes tricky to encode in the function, as the function is not called for a user until there are new actions from that user.

To encode time-based inactivity, `mapGroupsWithState()` supports timeouts that are defined as follows:

- Each time the function is called on a key, a timeout can be set on the key based on a duration or a threshold timestamp.
- If that key does not receive any data, such that the timeout condition is met, the key is marked as "timed out." The next micro-batch will call the function on this timed-out key even if there is no data for that key in that micro-batch. In this special function call, the new in-

put data iterator will be empty (since there is no new data) and `GroupState.hasTimedOut()` will return `true`. This is the best way to identify inside the function whether the call was due to new data or a timeout.

There are two types of timeouts, based on our two notions of time: processing time and event time. The processing-time timeout is the simpler of the two to use, so we'll start with that.

Processing-time timeouts

Processing-time timeouts are based on the system time (also known as the wall clock time) of the machine running the streaming query and are defined as follows: if a key last received data at system timestamp `T`, and the current timestamp is more than `(T + <timeout duration>)`, then the function will be called again with a new empty data iterator.

Let's investigate how to use timeouts by updating our user example to remove a user's state based on one hour of inactivity. We will make three changes:

- In `mapGroupsWithState()`, we will specify the timeout as `GroupStateTimeout.ProcessingTimeTimeout`.
- In the state update function, before updating the state with new data, we have to check whether the state has timed out or not. Accordingly, we will update or remove the state.
- In addition, every time we update the state with new data, we will set the timeout duration.

Here's the updated code:

```
// In Scala
def updateUserStatus(
    userId: String,
    newActions: Iterator[UserAction],
    state: GroupState[UserStatus]): UserStatus = {

  if (!state.hasTimedOut) {       // Was not called due to timeout
    val userStatus = state.getOption.getOrElse {
      new UserStatus(userId, false)
    }
    newActions.foreach { action => userStatus.updateWith(action) }
    state.update(userStatus)
    state.setTimeoutDuration("1 hour") // Set timeout duration
    return userStatus

  } else {
    val userStatus = state.get()
    state.remove()                    // Remove state when timed out
    return userStatus.asInactive()  // Return inactive user's status
  }
}

val latestStatuses = userActions
  .groupByKey(userAction => userAction.userId)
  .mapGroupsWithState(
    GroupStateTimeout.ProcessingTimeTimeout)(
    updateUserStatus _)
```

This query will automatically clean up the state of users for whom the query has not processed any data for more than an hour. However, there are a few points to note about timeouts:

- The timeout set by the last call to the function is automatically cancelled when the function is called again, either for the new received data or for the timeout. Hence, whenever the function is called, the timeout duration or timestamp needs to be explicitly set to enable the timeout.
- Since the timeouts are processed during the micro-batches, the timing of their execution is imprecise and depends heavily on the trigger interval and micro-batch processing times. Therefore, it is not advised to use timeouts for precise timing control.
- While processing-time timeouts are simple to reason about, they are not robust to slowdowns and downtimes. If the streaming query suffers a downtime of more than one hour, then after restart, all the keys in the state will be timed out because more than one hour has passed since each key received data. Similar wide-scale timeouts can occur if the query processes data slower than it is arriving at the source (e.g., if data is arriving and getting buffered in Kafka). For ex-

ample, if the timeout is five minutes, then a sudden drop in processing rate (or spike in data arrival rate) that causes a five-minute lag could produce spurious timeouts. To avoid such issues we can use an event-time timeout, which we will discuss next.

Event-time timeouts

Instead of the system clock time, an event-time timeout is based on the event time in the data (similar to time-based aggregations) and a watermark defined on the event time. If a key is configured with a specific timeout timestamp of `T` (i.e., not a duration), then that key will time out when the watermark exceeds `T` if no new data was received for that key since the last time the function was called. Recall that the watermark is a moving threshold that lags behind the maximum event time seen while processing the data. Hence, unlike system time, the watermark moves forward in time at the same rate as the data is processed. This means (unlike with processing-time timeouts) any slowdown or downtime in query processing will not cause spurious timeouts.

Let's modify our example to use an event-time timeout. In addition to the changes we already made for using the processing-time timeout, we will make the following changes:

- Define watermarks on the input Dataset (assume that the class `UserAction` has an `eventTimestamp` field). Recall that the watermark threshold represents the acceptable amount of time by which input data can be late and out of order.
- Update `mapGroupsWithState()` to use `EventTimeTimeout`.
- Update the function to set the threshold timestamp at which the timeout will occur. Note that event-time timeouts do not allow setting a timeout duration, like processing-time timeouts. We will discuss the reason for this later. In this example, we will calculate this timeout as the current watermark plus one hour.

Here is the updated example:

```
// In Scala
def updateUserStatus(
    userId: String,
    newActions: Iterator[UserAction],
    state: GroupState[UserStatus]):UserStatus = {

  if (!state.hasTimedOut) {  // Was not called due to timeout
    val userStatus = if (state.getOption.getOrElse {
      new UserStatus()
    }
    newActions.foreach { action => userStatus.updateWith(action) }
    state.update(userStatus)

    // Set the timeout timestamp to the current watermark + 1 hour
    state.setTimeoutTimestamp(state.getCurrentWatermarkMs, "1 hour")
    return userStatus
  } else {
    val userStatus = state.get()
    state.remove()
    return userStatus.asInactive() }
}

val latestStatuses = userActions
  .withWatermark("eventTimestamp", "10 minutes")
  .groupByKey(userAction => userAction.userId)
  .mapGroupsWithState(
    GroupStateTimeout.EventTimeTimeout)(
    updateUserStatus _)
```

This query will be much more robust to spurious timeouts caused by restarts and processing delays.

Here are a few points to note about event-time timeouts:

- Unlike in the previous example with processing-time timeouts, we have used `GroupState.setTimeoutTimestamp()` instead of `GroupState.setTimeoutDuration()`. This is because with processing-time timeouts the duration is sufficient to calculate the exact future timestamp (i.e., current system time + specified duration) when the timeout would occur, but this is not the case for event-time timeouts. Different applications may want to use different strategies to calculate the threshold timestamp. In this example we simply calculate it based on the current watermark, but a different application

may instead choose to calculate a key's timeout timestamp based on the maximum event-time timestamp seen for that key (tracked and saved as part of the state).
- The timeout timestamp must be set to a value larger than the current watermark. This is because the timeout is expected to happen when the timestamp crosses the watermark, so it's illogical to set the timestamp to a value already larger than the current watermark.

Before we move on from timeouts, one last thing to remember is that you can use these timeout mechanisms for more creative processing than fixed-duration timeouts. For example, you can implement an approximately periodic task (say, every hour) on the state by saving the last task execution timestamp in the state and using that to set the processing-time timeout duration, as shown in this code snippet:

```
// In Scala
timeoutDurationMs = lastTaskTimstampMs + periodIntervalMs -
groupState.getCurrentProcessingTimeMs()
```

Generalization with flatMapGroupsWithState()

There are two key limitations with `mapGroupsWithState()` that may limit the flexibility that we want to implement more complex use cases (e.g., chained sessionizations):

- Every time `mapGroupsWithState()` is called, you have to return one and only one record. For some applications, in some triggers, you may not want to output anything at all.
- With `mapGroupsWithState()`, due to the lack of more information about the opaque state update function, the engine assumes that generated records are updated key/value data pairs. Accordingly, it reasons about downstream operations and allows or disallows some of them. For example, the DataFrame generated using `mapGroupsWithState()` cannot be written out in append mode to files. However, some applications may want to generate records that can be considered as appends.

`flatMapGroupsWithState()` overcomes these limitations, at the cost of slightly more complex syntax. It has two differences from `mapGroupsWithState()`:

- The return type is an iterator, instead of a single object. This allows the function to return any number of records, or, if needed, no records at all.
- It takes another parameter, called the *operator output mode* (not to be confused with the query output modes we discussed earlier in the chapter), that defines whether the output records are new records that can be appended (`OutputMode.Append`) or updated key/value records (`OutputMode.Update`).

To illustrate the use of this function, let's extend our user tracking example (we have removed timeouts to keep the code simple). For example, if we want to generate alerts only for certain user changes and we want to write the output alerts to files, we can do the following:

```
// In Scala
def getUserAlerts(
    userId: String,
    newActions: Iterator[UserAction],
    state: GroupState[UserStatus]): Iterator[UserAlert] = {

  val userStatus = state.getOption.getOrElse {
    new UserStatus(userId, false)
  }
  newActions.foreach { action =>
    userStatus.updateWith(action)
  }
  state.update(userStatus)

  // Generate any number of alerts
  return userStatus.generateAlerts().toIterator
}

val userAlerts = userActions
  .groupByKey(userAction => userAction.userId)
  .flatMapGroupsWithState(
    OutputMode.Append,
    GroupStateTimeout.NoTimeout)(
    getUserAlerts)
```

Performance Tuning

Structured Streaming uses the Spark SQL engine and therefore can be tuned with the same parameters as those discussed for Spark SQL in Chapters 5 and 7. However, unlike batch jobs that may process gigabytes to terabytes of data, micro-batch jobs usually process much smaller volumes of data. Hence, a Spark cluster running streaming queries usually needs to be tuned slightly differently. Here are a few considerations to keep in mind:

Cluster resource provisioning

Since Spark clusters running streaming queries are going to run 24/7, it is important to provision resources appropriately. Underprovisoning the resources can cause the streaming queries to fall behind (with micro-batches taking longer and longer), while overprovisioning (e.g., allocated but unused cores) can cause unnecessary costs. Furthermore, allocation should be done based on the nature of the streaming queries: stateless queries usually need more cores, and stateful queries usually need more memory.

Number of partitions for shuffles

For Structured Streaming queries, the number of shuffle partitions usually needs to be set much lower than for most batch queries—dividing the computation too much increases overheads and reduces throughput. Furthermore, shuffles due to stateful operations have significantly higher task overheads due to checkpointing. Hence, for streaming queries with stateful operations and trigger intervals of a few seconds to minutes, it is recommended to tune the number of shuffle partitions from the default value of 200 to at most two to three times the number of allocated cores.

Setting source rate limits for stability

After the allocated resources and configurations have been optimized for a query's expected input data rates, it's possible that sudden surges in data rates can generate unexpectedly large jobs and subsequent instability. Besides the costly approach of overprovisioning, you can safeguard against instability using source rate limits. Setting limits in supported sources (e.g., Kafka and files) prevents a query from consuming too much data in a single micro-batch. The surge data will stay buffered in the source, and the query will eventually catch up. However, note the following:

- Setting the limit too low can cause the query to underutilize allocated resources and fall behind the input rate.
- Limits do not effectively guard against sustained increases in input rate. While stability is maintained, the volume of buffered, unprocessed data will grow indefinitely at the source and so will the end-to-end latencies.

Multiple streaming queries in the same Spark application

Running multiple streaming queries in the same `SparkContext` or `SparkSession` can lead to fine-grained resource sharing. However:

- Executing each query continuously uses resources in the Spark driver (i.e., the JVM where it is running). This limits the number of queries that the driver can execute simultaneously. Hitting those limits can either bottleneck the task scheduling (i.e., underutilizing the executors) or exceed memory limits.
- You can ensure fairer resource allocation between queries in the same context by setting them to run in separate scheduler pools. Set the `SparkContext`'s thread-local property `spark.scheduler.pool` to a different string value for each stream:

```
// In Scala
// Run streaming query1 in scheduler pool1
spark.sparkContext.setLocalProperty("spark.scheduler.pool", "pool1")
df.writeStream.queryName("query1").format("parquet").start(path1)

// Run streaming query2 in scheduler pool2
spark.sparkContext.setLocalProperty("spark.scheduler.pool", "pool2")
df.writeStream.queryName("query2").format("parquet").start(path2)
```

```
# In Python
# Run streaming query1 in scheduler pool1
spark.sparkContext.setLocalProperty("spark.scheduler.pool", "pool1")
df.writeStream.queryName("query1").format("parquet").start(path1)
```

```
# Run streaming query2 in scheduler pool2
spark.sparkContext.setLocalProperty("spark.scheduler.pool", "pool2")
df.writeStream.queryName("query2").format("parquet").start(path2)
```

Summary

This chapter explored writing Structured Streaming queries using the DataFrame API. Specifically, we discussed:

- The central philosophy of Structured Streaming and the processing model of treating input data streams as unbounded tables
- The key steps to define, start, restart, and monitor streaming queries
- How to use various built-in streaming sources and sinks and write custom streaming sinks
- How to use and tune managed stateful operations like streaming aggregations and stream–stream joins
- Techniques for expressing custom stateful computations

By working through the code snippets in the chapter and the notebooks in the book's GitHub repo, you will get a feel for how to use Structured Streaming effectively. In the next chapter, we explore how you can manage structured data read and written simultaneously from batch and streaming workloads.

[1] For a more detailed explanation, see the original research paper "Discretized Streams: Fault-Tolerant Streaming Computation at Scale" by Matei Zaharia et al. (2013).

[2] This execution loop runs for micro-batch-based trigger modes (i.e., `ProcessingTime` and `Once`), but not for the `Continuous` trigger mode.

[3] For the full list of unsupported operations, see the Structured Streaming Programming Guide.