O'REILLY®

Early Release
RAW & UNEDITED

Stream Processing with Apache Spark

Best Practices for Scaling and Optimizing Apache Spark

Gerard Maas & François Garillot

Stream Processing with Apache Spark

by Gerard Maas and Francois Garillot

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com/safari). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

- Acquisitions Editor: Rachel Roumeliotis
- Developmental Editor: Jeff Bleiel
- Production Editor: Nan Barber
- Cover Designer: Karen Montgomery
- Illustrator: Rebecca Demarest

Revision History for the Early Release

- 2019-03-04: First Early Release

See http://oreilly.com/catalog/errata.csp?isbn=9781491944240 for release details.

978-1-491-94417-2

Preface

Why we wrote this book

How's this book for?

Installing Spark

Spark is an Apache open-source project hosted officially by the Apache Foundation, but which mostly uses Github for its development (*https://github.com/apache/spark*). It can also be downloaded as a binary, pre-compiled package at the address:

https://spark.apache.org/downloads.html

From there, the user can start running Spark on one or more machines, as we will explain later. Packages exist for all the major Linux distributions, which should help installation.

For the purposes of this book, we will use examples and code compatible with Spark 2.1.0, and except for minor output and formatting details, those examples should stay compatible with future Spark versions.

Note however that Spark, is a program that runs on the Java Virtual Machine (JVM), which you should install and make accessible on every machine in which any Spark component will run.

To install a Java Development Kit (JDK), we recommend OpenJDK, which is packaged on many systems and architectures as well: *http://openjdk.java.net/install/*

You can also install the Sun JDK: *https://www.java.com/en/download/*

Spark, as any Scala program, runs on any system on which a JDK version 6 or later is present. The recommended Java runtime for Spark depends on the version:

- For Spark versions below 2.0, Java 7 is the recommended version.
- For Spark versions 2.0 and above, Java 8 is the recommended version.

Learning Scala

The examples of this book are in Scala. This is the implementation language of core Spark, but by far not the only language in which it can be used, since Spark offers APIs in Python, Java and R, at the time of this writing.

Scala is one of the most feature-complete programming langages today, in that it offers both functional and object-oriented aspects. Yet its concision and type inference makes the basic elements of its syntax easy to understand.

Scala as a beginner language has many advantages from a pedagogical viewpoint, its regular syntax and semantics being one of the most important.
Björn Regnell, Lund University

Hence, we hope the examples will stay clear enough for any reader to pick up what the example means. However, for the readers who might want a primer on the language, and that are more comfortable learning using a book, we can advise *Atomic Scala* [Eckel2013]. For users looking for a reference book to touch up on their knowledge, the reference book is *Programming in Scala*[Odersky2010].

The Way Ahead

This book is organized in five parts:

- Part I extends and deepens on the concepts we started to discuss in this chapter. We cover the fundamental concepts of stream processing, the general blueprints of the architectures that implement streaming and study Spark in detail.
- In Part II, we learn Structured Streaming, its programming model and how to implement streaming applications, from relatively simple stateless transformations to advanced stateful operations. We also discuss its interagration with monitoring tools supporting 24/7 operations and discover the experimental areas currently under development.
- In Part III, we study Spark Streaming. In a similar organization to Structured Streaming, we learn how to create streaming applications, operate Spark Streaming jobs and how to integrate it with other APIs in Spark. We close this part with a modest guide to performance tuning.
- Part IV brings us in contact with advanced streaming techniques. We discuss the use of probabilistic data structures and approximation techniques to address stream processing challenges and examine the limited space of online machine learning with Spark Streaming.
- To close, Part V brings us to streaming beyond Apache Spark. We survey other available stream processors and provide a glimpse into further steps to keep learning about Spark and stream processing.

We recommend you to go through part I to gain an understanding of the concepts supporting stream processing. That will facilitate the use of a common language and concepts across the rest of the book.

Part II, Structured Streaming and Part III, Spark Streaming, follow a consistent structure. You may chose to cover one or the other first, to match your interest and most immediate priorities. Maybe you are starting a new project and want to know Structured Streaming? Check! Start in Part II. Or you may be jumping in an existing code base that uses Spark Streaming and want to understant it better? Okey. Start in Part III.

Part IV goes initially deep into some mathematical background required to understand the probabilistic structures discussed. We like to think of it as “the road ahead is steep but the scenery is beatiful”

Part V will put stream processing using Spark in perspective with other available frameworks and libraries out there. It might help you decide to try one or more alternatives before settling for a technology.

The online resources of the book complement your learning experience with notebooks and code that you can use and experiment on your own. … Or even take a piece of code to bootstrap your own project. The online resources are located at: *https://github.com/StreamProcessingWithSpark*

We truly hope you enjoy reading this book as much as we enjoyed compiling all the information and bundling the experience it contains.

Bibliography

- [Eckel2013] Bruce Eckel and Dianne Marsh, *Atomic Scala*, Mindview, Inc. 2013 ISBN-13: 9780981872513
- [Odersky2010] Martin Odersky, Lex Spoon and Bill Venners, *Programming in Scala, 2d Ed*, Artima, 2011 ISBN: 0981531644

Part I. Fundamentals of Stream Processing with Apache Spark

Chapter 1. Introducing Stream Processing

In 2011, Marc Andreessen famously said that “software is eating the world”, refering to the booming digital economy, where many enterprises were facing the challenges of a digital transformation. Successful online businesses, using “online” and “mobile” operation modes, were taking over their traditional “brick and mortar” counterparts.

For example, imagine the traditional experience of buying a new camera in a photography shop: We would visit the shop, browse around, maybe ask a few questions to the clerk, make our mind and finally buy a model that fulfilled our desires and expectations. After fulfilling our purchase, the shop would have registered a credit card transaction — or only a casher balance change in case of a cash payment-- and they will know they have one less inventory item of that particular camera model.

Now, let us take that experience online: First, we start searching the web. We visit a couple of online stores, leaving digital traces as we pass by one to the other. Advertisements on websites suddently start showing us promotions for the camera we were looking at, as well as for competing alternatives. We finally find an online shop offering us the best deal and purchase the camera. We create an account. Our personal data gets registered and linked to the purchase. While we complete our purchase, we are offered additional options that are allegedly popular with other people that bought the same camera. Each of our digital interactions, like searching for keywords on the web, clicking on some link or spending time reading a particular page generate a series of events that are collected and transformed into business value, like personalized advertisement or upsale recommendations.

Commenting on Andreessen quote, in 2015, Dries Buytaert said “no, actually, *data* is eating the world”. What he meant is that the disruptive companies of today are no longer disruptive because of their software but because of the unique data they collect and their ability to transform that data into value.

The adoption of stream processing technologies is driven by the increasing need of businesses to improve the time required to react and adapt to changes in their operational environment. This way of processing data as it comes, provides a technical and strategical advantage. Examples on this on-going adoption includes sectors such as Internet commerce — continuously running data pipelines created by businesses that operate with customers on a 24/7 basis — or credit card companies — analysing transactions as they happen to detect and stop fraudulent activities as they happen.

Another driver of stream processing is that our ability to generate data far surpasses our ability to make sense of it. We are constantly increasing the number of computing-capable devices in our personal and professional environments. Televisions, connected cars, smartphones, bike computers, smart watches, survelliance cameras, thermostats, etc., we are surrounding ourselves with devices meant to produce event logs: streams of messages representing the actions and incidents that form part of the history of the device in its context. As we interconnect those devices more and more, we create an ability for us to access and therefore analyze those event logs. This phenomenon opens the door to an incredible burst of creativity and innovation in the domain of near real-time data analytics, on the condition that we find a way to make this analysis tractable. In this world of aggregated event logs, stream processing offers the most resource friendly way to operate the analysis of streams of data.

It is not a surprise that not only is data eating the world but *streaming data* is.

In this chapter, we are going to start our journey in stream processing using Apache Spark. To prepare us to discuss the capabilities of Spark in the stream processing area, we need to establish a common understanding of what is stream processing, its applications and challenges. Once we have built a common language, we introduce Apache Spark as a generic data processing framework able to handle the requirements of batch and streaming workloads using a unified model. Finally, we zoom in the streaming capabilities of Spark, where we present the two available APIs: Spark Streaming and Structured Streaming. We briefly discuss their salient characteristics to provide a sneak peek into what you will discover in the rest of this book.

What is Stream Processing?

Stream processing is the discipline and related set of techniques used to extract information from *unbounded data*.

In his book *Streaming Systems*, Tyler Akidau defines *unbounded* data as:

```
A type of dataset that is infinite in size (at least theoretically).
```

Akidau, Streaming Systems

Given that our information systems are built on hardware with finite resources, such as memory and storage capacity, they cannot possibly hold unbounded datasets. Instead, we observe the data as it is received at the processing system in the form of a flow of events over time. We call this a *stream* of data.

In contrast, we consider *bounded data* as a dataset of known size. The number of elements in a *bounded* dataset can be counted.

Batch vs. Stream Processing

How do we process both types of datasets? With *batch processing* we refer to the computational analysis of *bounded* datasets. In practical terms, this means that those datasets are available and retrivable as a whole from some form of storage. We know the size of the dataset at the start of the computational process and the duration of that process is limited in time.

In contrast, with *stream processing* we are concerned with the processing of data as it arrives to the system. Given the unbounded nature of data streams, the stream processors needs to run constantly for as long as the stream is delivering new data, that, as we learned, might be -- theoretically-- forever.

Stream processing systems apply programming and operational techniques to make possible the processing of potentially infinite data streams with a limited amount of computing resources.

The Notion of Time in Stream Processing

Data can be encountered in two forms:

- at rest, in the form of a file, the contents of a database, or some other kind of record, or
- in motion: as continuously generated sequence of signals, like the measurement of a sensor or GPS signals from moving vehicles.

We discussed already that a stream processing program is a program that assumes its input will be potentially infinite in size. More specifically, a stream processing program assumes that its input is a sequence of signals of indefinite length, *observed over time*.

From the point of view of a timeline, *data at rest* is data from the past: arguably, all bounded datasets, stored in files or contained in databases, were initially a stream of data, collected over time into some storage. The users database, all the orders from the last quarter, the gps coordinates of taxi trips in a city, etc., started as individual events collected in a repository.

Trying to reason about *data in motion* is more challenging. There is a time difference between the moment data is originally generated and when it becomes available for processing. That time delta might be very short, like web log events generated and processed within the same data center, or much longer, like GPS data of a car crossing through a tunnel, that only gets dispatched when the vehicle gets back its wireless connectivity as it leaves the tunnel.

We can observe that there's a timeline when the events were produced and another for when the events are handled by the stream processing system. These timelines are so significant, that we give them specific names:

Event-Time

: the time when the event was created. The time information is provided by the local clock of the device generating the event.

Processing Time

: the time when the event is handled by the stream processing system. This is the clock of the server running the processing logic. It's usually relevant for technical reasons like computing the processing lag or as a criteria to determine duplicated output.

The differentiation among these timelines becomes very important when we need to correlate, order, or aggregate the events with respect to each other.

The Factor of Uncertainty

In the timeline, *data at rest* relates to the past and *data in motion* can be seen as the present. But what about the future? One of the most subtle aspects of this discussion is that it makes no assumptions on the throughput at which the system receives events.

In general, streaming systems do not require the input to be produced at regular intervals, all at a time, or following a certain rhythm. This means that, since computation usually has a cost, it's a challenge to predict peak load: matching the sudden arrival of input elements with the computing resources necessary to process them.

If we have the computing capacity needed to match a sudden influx of input elements then our system will produce results as expected, but if we have not planned for such a burst of input data, some streaming systems may face delays, resource constriction, of failure.

Dealing with uncertainty is an important aspect of stream processing.

In summary, stream processing lets us extract information from infinite data streams observed as events delivered over time. Nevertheless, as we receive and process data, we need to deal with the additional complexity of event-time and the uncertainty introduced by an unbounded input.

Why would we want to deal with the additional trouble? In the next section we are going to glance over a number of use cases that illustrate the value added by stream processing and how it delivers on the promise of providing faster actionable insights, and hence business value, on data streams.

Some Examples of Stream Processing

The use of stream processing goes as wild as our capacity to imagine new real-time, innovate applications of data. The following use cases, where the authors have been involved in one way or another, are only a small sample we use to illustrate the wide spectrum of application of stream processing:

Device Monitoring

> A small startup rolled out a cloud-based IoT device monitor able to collect, process, and store data from up to 10 million devices. Multiple stream processors were deployed to power different parts of the application, from real-time dashboard updates using in-memory stores, to continuous data aggregates, like unique counts and min/max measurements.

Fault Detection

> A large hardware manufacturer applies a complex stream processing pipeline to receive device metrics. Using time-series analysis, potential failures are detected and corrective measures are automatically sent back to the device.

Billing Modernization

> A well-established insurance company migrated their billing system to a streaming pipeline. Batch exports from their existing mainframe infrastructure are streamed through this system to meet the existing billing processes while allowing new real-time flows from insurance agents to be served by the same logic.

Fleet Management

> A fleet management company installed devices able to report real-time data from the managed vehicles, such as location, motor parameters, and fuel levels, allowing them to enforce rules like geographical limits and analyse driver behavior for speed limits.

Media Recommendations

> A national media company deployed a streaming pipeline to ingest new videos, such as news reports, into their recommendation system, making them available to their users' personalized suggestions almost as soon as they are ingested in their media repository. Their previous system would take hours to do the same.

Faster Loans

> A bank active in loan services was able to reduce loan approval from hours to seconds by a combining several data streams into a streaming application.

A common thread among those use cases is the need of the business to process the data and create actionable insights in a short period of time from when the data was received. This time is relative to the use case: while *minutes* is a very fast turn-around for a loan approval, a milliseconds response is probably necessary to detect a device failure and issue a corrective action within a give service level threshold.

In all cases, we can argue that *data* is better when consumed as fresh as possible.

Now that we have an understanding of what is stream processing and some examples of how it is being used today, it's time to start delving into the concepts that underpin its implementation.

Scaling Up Data Processing

Before we discuss the implications of distributed computation in stream processing, let's take a quick tour through *MapReduce*, a computing model that laid the foundations for scalable and reliable data processing.

MapReduce

The history of programming for distributed systems experienced a notable event in February of 2003. Jeff Dean and Sanjay Gemawhat, after going through a couple iterations of rewriting Google's crawling and indexing systems, started noticing some operations that they could expose through a common interface. This led them to develop *MapReduce*, a system for distributed processing on large clusters at Google.

"Part of the reason we didn't develop MapReduce earlier was probably because when we were operating at a smaller scale, then our computations were using fewer machines, and therefore robustness wasn't quite such a big deal: it was fine to periodically checkpoint some computations and just restart the whole computation from a checkpoint if a machine died. Once you reach a certain scale, though, that becomes fairly untenable since you'd always be restarting things and never make any forward progress."
Jeff Dean, email to Bradford F. Lyon, August 2013

MapReduce is a programming API, first, and a set of components, second, that make programming for a distributed system a relatively easier task than all its predecessors.

Its core tenets are two functions:

Map

> The map operation takes as an argument a function to be applied to every element of the collection. The collection's elements are read in a distributed manner, through the distributed file system, one chunk per executor machine. Then, all the elements of the

collection that reside in the local chunk see the function applied to them, and the executor emits the result of that application, if any.

Reduce

The reduce operation takes two arguments: One is a neutral element, which is what the *reduce*operation would return if passed an empty collection. The other is an aggregation operation, that takes the current value of an aggregate, a new element of the collection, and lumps them into a new aggregate.

Combinations of these two higher-order functions are powerful enough to express every operation one would want to do on a dataset.

The Lesson Learned: Scalability and Fault-Tolerance

From the programmer's perspective, the main advantages of *MapReduce* are:

- It had a simple API
- It offers very high expressivity
- It significantly offloaded the difficulty of distributing a program, from the shoulders of the programmer to those of the library designer. In particular, resilience is built-in into the model.

While these characteristics make the model attractive, the main success of *MapReduce* was that its ability to sustain growth. As data volumes increased and growing business requirements lead to more information-extraction jobs, the *MapReduce* model demonstrated demonstrated two crucial properties:

Scalability

As datasets grow, it is possible to add more resources to the cluster of machines in order to preserve a stable processing performance.

Fault-Tolerance

The system can sustain and recover from partial failures. For any data-carrying executor that may crash, the data it contained when it does crash has a copy on another machine. As a result, it is enough to re-launch the task that was running on this executor. Since the master keeps track of that task, that does not pose any particular problem other than rescheduling.

These two characteristics combined, resulted in a system able to constantly sustain workloads in an environment fundamentally unreliable, *properties that we also require for stream processing*.

Distributed Stream Processing

One fundamental difference of stream processing with the *MapReduce* model, and with batch processing in general, is that while batch processing can assume to have access to the complete dataset, with streams, we only see a small portion of the dataset at any time.

This situation gets aggravated in a distributed system, as in an effort to distribute the processing load among a series of executors, we further split up the input stream into partitions. Each executor gets to see only a partial view of the complete stream.

The challenge for a distributed stream processing framework is to provide an abstraction that hides this complexity from the user and let us reason about the stream as a whole.

Stateful Stream Processing in a Distributed System

Let's imagine that we are counting the votes during a presidential election. The classical batch approach would be to wait until all votes have been casted to then proceed to count them. While this approach produces a correct end result, it would made up for very boring news over the day, as no (intermediate) results are known until the end of the electoral process.

A more exciting scenario is when we can count the votes per candidate as each vote is casted. At any moment, we have a partial count by participant that lets us see the current stand as well as the voting trend. We can probably anticipate a result.

To accomplish this scenario, the stream processor needs to keep an internal register of the votes seen so far. To ensure a consistent count, this register must recover from any partial failure. Indeed, we can't ask the citizens to issue their vote again due to a technical failure.

Also, any eventual failure recovery may not impact the final result. We can't risk to declare the wrong winning candidate as a side-effect of an ill-recovered system.

This scenario illustrates the challenges of stateful stream processing running in a distributed environment. Stateful processing poses additional concerns on the system:

- We need to ensure that the state is preserved over time.
- We require data consistency guarantees, even in the event of partial system failures.

As we will see through the course of this book, addressing these concerns is an important aspect of stream processing.

Now that we have a better sense of the drivers behind the popularity of stream processing and the challenging aspects of this discipline, we will introduce Apache Spark. As a unified data analytics engine, Spark offers data processing capabilities for both batch and streaming, making it an excellent choice to satisfy the demands of the data intensive applications, as we are going to see next.

Introducing Apache Spark

Apache Spark is a fast, reliable, and fault-tolerant distributed computing framework for large-scale data processing.

The First Wave: Functional APIs

In its early days, Spark's break-through was driven by its novel use of memory and expressive functional API. The Spark memory model uses RAM to cache data as it is being processed, resulting in up to 100x faster processing than Hadoop *MapReduce*, the open source implementation of Google's *MapReduce* for batch workloads.

Its core abstraction, the *Resilient Distributed Dataset*, or *RDD*, brought a rich functional programming model that abstracted out the complexities of distributed computing on a cluster. It introduced the concepts of *transformations* and *actions* that offered a more expressive programming model than the *map* and *reduce* stages that we discussed in the *MapReduce* overview. In that model, many available *transformations*, like `map`, `flatmap`, `join`, `filter`,... express the lazy conversion of the data from one internal representation to another while eager operations called *actions* materialize the computation on the distributed system to produce a result.

The Second Wave: SQL

The second game changer in the history of the Spark project was the introduction of Spark SQL and *DataFrames* (and later, *Dataset*, a strongly typed *DataFrame*) . From a high-level perspective, Spark SQL adds SQL support to any dataset that has an schema. It makes possible to query a CSV, Parquet, or JSON dataset in the same way that we used to query a SQL Database.

This evolution also lowered the threshold of adoption for users. Advanced distributed data analytics were no longer the exclusive realm of software engineers, but it was now accessible to Data Scientist, Business Analysts and other professionals familiar with SQL. From a performance point of view, SparkSQL brought a query optimizer and a physical execution engine to Spark, making it even faster and use less resources.

A Unified Engine

Nowadays, Spark is a unified analytics engine offering batch and streaming capabities that is compatible with a polyglot approach to data analytics, offering APIs in Scala, Java, Python, and the R language.

While in the context of this book, we are going to focus our interest on the streaming capabilities of Apache Spark, its batch functionality is equaly advanced and is highly complementary to streaming applications. Spark's unified programming model means that developers only need to learn one new paradigm to address both batch and streaming workloads.

NOTE

In the course of the book, we will use *Apache Spark* and *Spark* interchangeably. We will use *Apache Spark* when we want to make emphasis on the project or open source aspect of it, while we will use *Spark* to refer to the technology in general.

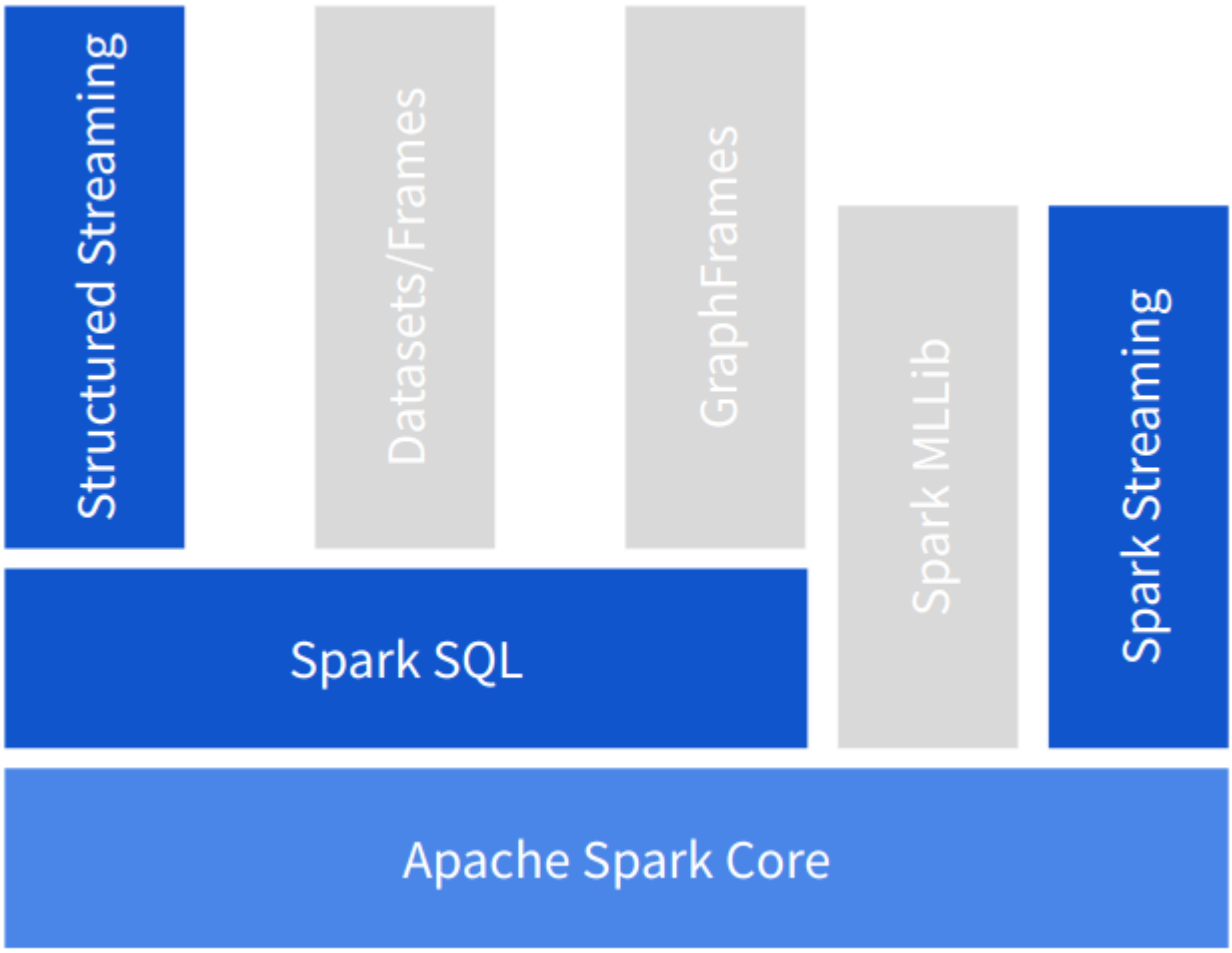


Figure 1-1. Abstraction layers (horizontal) and libraries (vertical) offered by Spark

As we can see in Figure 1-1, Spark consists of a core engine, a set of abstractions built on top of it (represented as horizontal layers), and libraries that use those abstractions to address a particular area (vertical boxes). We have highlighted the areas in scope of this book and grayed out those that are not covered. To learn more about these other areas of Apache Spark, we recommend the books: *"Spark, The Definitive Guide"* by Bill Chambers and Matei Zaharia and *"High Performance Spark"* by Holden Karau and Rachel Warren.

As abstraction layers in Spark, we have:

Spark core engine

> The core engine uses low-level functional APIs to distribute computations to a cluster of computing resources, called *executors* in Spark lingo. Its cluster abstraction allows it to submit workloads to YARN, MESOS, and Kubernetes as well as use its own standalone cluster mode, where Spark runs as a dedicated service in a cluster of machines. Its datasource abstraction enables the integration of many different data providers, such as files, block stores, databases, and event brokers.

Spark SQL

Implements the higher-level *Dataset* and *Dataframe* APIs of Spark and adds SQL support on top of arbitrary data sources. It also introduces a series of performance improvements through the Catalyst query engine and the code generation and memory management from project Tungsten.

The libraries built on top of these abstractions address different areas of large-scale data analytics: *MLLib* for machine learning, *GraphFrames* for graph analysis and the two APIs for stream processing that are the focus of this book: *Structured Streaming* and *Spark Streaming*.

Spark Streaming

Spark Streaming was the first stream processing framework built on top of the distributed processing capabilities of the core Spark engine. It was introduced in the Spark 0.7.0 release in February of 2013 as an alpha release that evolved over time to become today a mature API that's widely adopted in the industry to process large scale data streams.

Spark Streaming is conceptually built on a simple, yet powerful premise: Apply Spark's distributed computing capabilities to stream processing by transforming continuous streams of data into discrete data collections that Spark could operate on. This approach to stream processing is called the *micro-batch* model, that is in contrast with the *element-at-time* model that dominates in most other stream processing implementations.

Spark Streaming uses the same functional programming paradigm than the Spark core, but introduces a new abstraction, the *Discreatized Stream* or *DStream* that exposes a programming model to operate on the underlying data in the stream.

Structured Streaming

Structured Streaming is a stream processor built on top of the Spark SQL abstraction. It extends the *Dataset* and *DataFrame* APIs with streaming capabilities. As such, it adopts the schema-oriented transformation model, which confers the *structured* part of its name and inherits all the optimizations implemented in *Spark SQL*.

Structured Streaming was introduced as an experimental API with Spark 2.0 in July of 2016. A year later, it reached *general availability* with the Spark 2.2 release becoming elegible for production deployments. As a relatively new development, Structured Streaming is still evolving fast with each new version of Spark.

Structured Streaming uses a declarative model to acquire data from a stream or set of streams. In order to use the API to its full extent, it requires the specification of a schema for the data in the stream. In addition to supporting the general transformation model provided by the *Dataset* and *DataFrame*APIs, it introduces stream-specific features, such as support for event-time, streaming joins, and separation from the underlying runtime. That last feature opens the door for the implementation of runtimes with different execution models. The default implementation uses the classical micro-batch approach while a more recent *continuous processing* backend brings experimental support for near-real-time continuous execution mode.

Structured Streaming delivers a unified model that brings stream processing to the same level of batch-oriented applications, removing a lot of the cognitive burden of reasoning about stream processing.

Where Next?

If you are feeling the urge to learn any of these two APIs right away, you could directly jump to Part II. If you are not familiar with stream processing, we recommend to continue through this initial part of the book, as we build the vocabulary and common concepts that we will use in the discussion of the specific frameworks.

Chapter 2. Stream Processing Model

In this chapter, we will bridge the notion of a data stream — an operational building block of computation — with the programming language primitives and constructs that allow us to express stream processing.

We want to describe simple, fundamental concepts first before moving on to how Apache Spark represents them. Specifically, we will go over: - data sources, - stream processing pipelines, and - data sinks as components of stream processing, and then show how those concepts map to the specific stream processing model implemented by Apache Spark.

We then characterize a type of stream processing pipelines, called stateful stream processing, where it needs to do some bookkeeping on computations that it has performed in the past and store it in memory. We will see when those are appropriate. Finally, we will consider streams of time-stamped events and basic notions involved in addressing concerns such as "what do I do if the order and timeliness of the arrival of those events do not match expectations?"

Sources and Sinks

As we mentioned before, Apache Spark, in each of its two streaming systems — Structured Streaming and Spark Streaming — is a programming framework with APIs in the Scala, Java, Python, and R programming languages. It can only operate on data that enters the runtime of programs using this framework and it ceases to operate on the data as soon as it is being sent to another system.

This is a concept that you are probably already familiar with in the context of data at rest: to operate on data stored as a file of records, we need to read that file into memory where we can manipulate it, and once we have produced an output by computing on this data, we have the ability to write that result to another file. The same principle applies to databases, another example of data at rest.

Similarly, data streams can be made accessible as such, in the streaming framework of Apache Spark using the concept of streaming *data sources*. In the context of stream processing, accessing data from a stream is often refered to as *consuming the stream*. This abstraction is presented as an interface that allows the implementation of instances aimed to connect to specific systems: Apache Kafka, Flume, Twitter, a TCP socket, etc.

Likewise, we call the abstraction used to write a data stream outside of Apache Spark's control a *streaming sink*. Many connectors to various, specific systems are provided by the Spark project itself as well as by a rich ecosystem of open-source and commercial 3rd party integrations.

Figure 2-1. Simplified Streaming Model

In Figure 2-1 we illustrate this concept of *sources* and *sinks* in a stream processing system. Data is consumed from a source by a processing component and the eventual results are produced to a sink.

The notion of sources and sinks represents the system's boundary. This labeling of system boundaries makes sense given that a distributed framework can have a highly complex footprint among our computing resources. For example, it is possible to connect an Apache Spark cluster to another Apache Spark cluster, or to another distributed system of which apache Kafka is a frequent example. In that context, one framework's sink is the downstream framework's source. This chaining is commonly known as a *pipeline*. The name of sources and sinks is useful to both describe data passing from one system to the next and which point of view we are adopting when speaking about each system independently.

Immutable Streams Defined From Each Other

Between sources and sinks lies the programmable constructs of a stream processing framework. We do not want to get into the details of this subject yet — you will see them appear later in Part II and Part III for Structured Streaming and Spark Streaming, respectively. But we can introduce a few notions that will be useful to understand how we express stream processing.

Both stream APIs in Apache Spark take the approach of functional programming: they declare the transformations and aggregations they operate on data streams, assuming that those streams are immutable. As such, for one given stream, it is impossible to mutate one or several of its elements. Instead, we use transformations to express how to process the contents of one stream to obtain a derived data stream. This makes sure that at any given point in a program, any data stream can be traced to its inputs by a sequence of transformations and operations that are explicitly declared in the program. As a consequence, any particular process in a Spark cluster can reconstitute the content of the data stream using only the program and the input data, making computation unambiguous and reproducible.

Transformations and Aggregations

Spark makes extensive uses of transformations and aggregations. Transformations are computations that express themselves in the same way for every element in the stream. For example, creating a derived stream that doubles every element of its input stream corresponds to a transformation. Aggregations, on the other hand, produce results that depend on many elements and potentially every element of the stream observed until now. For example, collecting the top five largest numbers of an input stream corresponds to an aggregation. Computing the average value of some reading every 10 minutes is also an example of an aggregation.

Another way to designate those notions is to say that transformation have *narrow dependencies* (to produce one element of the output, you only need one of the elements of the input), while aggregations have *wide dependencies* (to produce one element of the output you would need to

observe many elements of the input stream encountered so far). This distinction is useful. It lets us envision a way to express basic functions that produces results using higher order functions.

While Spark Streaming and Structured Streaming have distinct ways of representing a data stream, the APIs they operate on are similar in nature. They both present themselves under the form of a series of transformations applied to immutable input streams and produce an output stream, either as a bona fide data stream or as an output operation (data sink).

Window Aggregations

Stream processing systems often feed themselves on actions that occur in real-time: those were tweets in our example, but purchases, financial events or sensor readings are also frequently encountered examples of such events. Our streaming application often has a centralized view of the logs of several places, whether those are retail locations or simply web servers for a common application. While seeing every transaction individually may not be useful or even practical, we may be interested in seeing the properties of events seen over a recent period of time. For example, the last 15 minutes or the last hour, or maybe even both.

Moreover, the very idea of stream processing is that the system is supposed to be long-running, dealing with a continuous stream of data. As these events keep coming in, the older ones usually become less and less relevant to whichever processing you are trying to accomplish.

We find many applications of regular and recurrent time-based aggregations that we call *windows*.

Tumbling Windows

The most natural notion of a window aggregation is that of "a grouping function each x period of time". For instance, "the maximum and minimum ambient temperature each hour" or "the total energy consumption (kW) each 15 minutes" are examples of window aggregations. Notice how the time periods are inherently consecutive and non-overlapping. We call *tumbling windows* a grouping of a fixed size, where each group follows the previous and do not overlap.

Tumbling Windows are the norm when we require to aggregate our data evenly over a period of time, independently from previous periods.

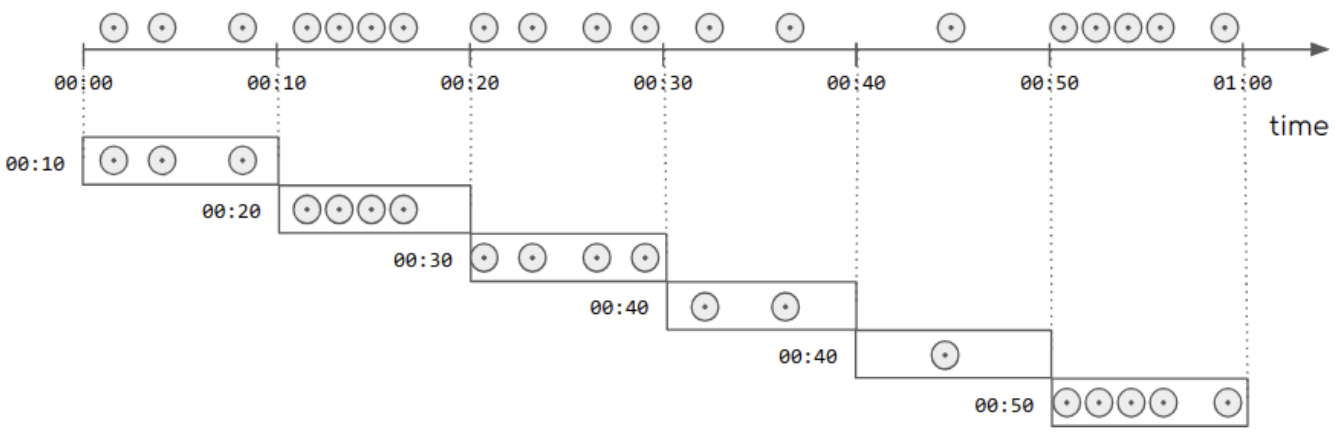


Figure 2-2. Tumbling Windows

In Figure 2-2 we show a *tumbling window* of 10 seconds over a stream of elements. In the illustration, we can appreciate the *tumbling* nature of *tumbling windows*.

Sliding Windows

There are cases when we are interested in historical trends that include the most recent data. As we hinted previously, when processing potentially infinite streams of data, older data might become less relevant over time while newer data may include drastical changes that might require additional data points to understand.

Sliding windows are aggregates over a period of time that are reported at a higher frequency than the aggregation period itself. As such, *sliding windows* refer to an aggregation with two time specifications: The window length and the reporting frequency. It usually reads like "a grouping function over a time interval x reported each y frequency". For example, "the average share price over the last day reported hourly". As you might have noticed already, this combination of a *sliding window*with the *average* function is the most widely known form of a *sliding window*, commonly known as *moving average*.

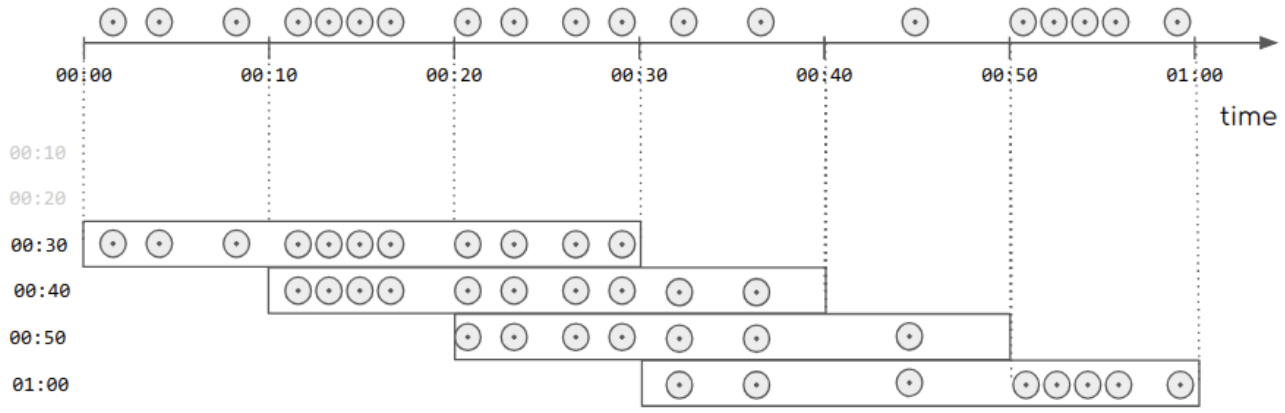


Figure 2-3. Sliding Windows

Figure 2-3 shows a sliding window with a window size of 30 seconds and a reporting frequency of 10 seconds. In the illustration we can observe and important characteristic of *sliding windows*: they are not defined for periods of time smaller than the size of the window. We can see that there are no windows reported for time `00:10` and `00:20`.

Although it cannot be seen in the final illustration, the process of drawing the chart reveals an interesting feature: We can construct and maintain a *sliding window* by adding the most recent data and removing the expired elements, while keeping all other element in place.

It's worth noting that tumbling windows are a particular case of sliding windows where the frequency of reporting is equal to the window size.

Stateless and Stateful processing

Now that we have a better notion of the programming model of the streaming systems in Apache Spark, we can look at the nature of the computations we want to apply on data streams. In our context, data streams are fundamentally a long collections of elements, observed over time. In fact, Structured Streaming pushes this logic by considering a data stream as a virtual table of records, where each row corresponds to an element.

Stateful Streams

Whether streams are viewed as a continuously extended collection or as a table, this approach gives us some insight on the kind of computation we may find interesting. In some cases, the emphasis is put on the continuously refreshed aspect: those are the cases where we want to filter some elements based on a well-known heuristic, such as alert messages coming from a log of events. This focus is perfectly valid but hardly requires an advanced analytics system such as Apache Spark. More often, we are interested in a real-time reaction to new elements based on an analysis that depends on the whole stream, such as detecting outliers in a collection or compute recent aggregate statistics from a data table. For example, it may be interesting to find higher than usual vibration patterns in a stream of airplane engine readings, which requires understanding the regular vibration measurements for the kind of engine we are interested in. This approach, in which we are simultaneously trying to understand our data at the same time we are trying to query it to affect a fast reaction on the arrival of new elements, often leads us to stateful stream processing. Stateful stream processing is the discipline by which we both compute something out of the new elements of data observed in our input data stream and refresh internal data that helps us perform this computation. For example, if we are trying to do anomaly detection, the internal state we want to update with every new stream element would be a machine learning model, whereas the computation we want to perform is to say whether an input element should be classified as an anomaly or not.

This pattern of computation is supported by a distributed streaming system such as Apache Spark because it can leverage a large amount of computing power and is an exciting new way of reacting to real-time data. For example, we could compute the running mean and standard deviation of the elements seen as input numbers and output a message if a new element is further away than 5 standard deviations from this mean. This is a simple, but useful way of marking particular outliers raising the alarm on particular extreme outliers of the distribution of our input elements [1]. In this case, the result of stream processing only stores the running mean and standard deviation of our stream — that is a couple of numbers.

BOUNDING THE SIZE OF STATE

One of the common pitfalls of practiciones new to the field of stream processing, is the temptation to store an amount of internal data that is proportional to the size of the input data stream. For example, if we would like to remove duplicate records of a stream, a naive way of approaching that problem would be to store every message already seen and compare new messages to them. That not only increases computing time with each incoming record, but also has an unbounded memory requirement that will eventually outgrow any cluster.

This is a common mistake, since the premise of stream processing is that there is no limit to the number of input events and, while your available memory in a distributed Spark cluster may be large, it is always limited. As such, intermediary state representations can be very useful to express

computation that operates on elements relative to the global stream of data they are observed on, but it is a somewhat unsafe approach. If you choose to have intermediate data, you need to make absolutely sure that the amount of data you may be storing at any given time is strictly bounded to a certain upper limit which is less than your available memory, independent of the amount of data you may encounter as input.

An example: Local Stateful Computation in Scala

To gain intuition in the statefulness concept without having to go into the details of distributed stream processing using Spark, we'll start with simple non-distributed streams in Scala.

The Fibonacci sequence is classicaly defined as a stateful stream: it's the sequence starting with 0 and 1, and thereafter composed of the sum of its two previous elements. Is it possible to define it without referring to its prior values, though, purely statelessly?

A stateless definition of the Fibonacci sequence as a stream transformation

To express this computation as a stream, taking as input the integers and outputting the Fibonacci Sequence, we express this as a stream transformation.

Example 2-1. A stateless computation of the Fibonacci elements

```
scala> import scala.math.sqrt
import scala.math.sqrt

scala> val phi = (sqrt(5)+1) / 2
phi: Double = 1.618033988749895

scala> import scala.math.pow
import scala.math.pow


scala> def fibonacci(s : Stream[Int]): Stream[Int] =
          s.map(x => ((pow(phi,x) - pow(-phi,-x))/sqrt(5)).toInt)
fibonacci: (s: `scala.collection.immutable.Stream[Int]):
          scala.collection.immutable.Stream[Int]

scala> val integers = Stream.from(0)
integers: scala.collection.immutable.Stream[Int] = Stream(0, ?)
scala> integers.take(10).print
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, empty

scala> fibonacci(integers).take(8).print
0, 1, 1, 2, 3, 5, 8, 13, empty
```

This counter-intuitive definition uses a stream of integers, starting from the single integer (0) to then define the Fibonacci sequence as a computation that takes as input an integer *n* received over the stream and returns the *n-th* element of the Fibonacci sequence as a result. This uses a floating point number formula known as the *Binet formula* to compute the n-th element of the sequence directly.

We notice how we take a limited number of elements of this sequence and print them in Scala, as an explicit operation. This is because the computation of elements in our stream is executed lazily,

which calls the evaluation of our stream only when required, considering the elements needed to produce them from the last materialization point to the original source.

A stateful definition of the Fibonacci sequence

The more intuitive way of defining the Fibonacci sequence from a stream of integers consists in defining it from two of its prior elements.

Example 2-2. A Stateful computation of the Fibonacci elements

```
scala> val ints = Stream.from(0)
ints: scala.collection.immutable.Stream[Int] = Stream(0, ?)

scala> val fibs = (ints.scanLeft((0, 1)){ case ((previous, current), index) =>
        (current, (previous + current))})

fibs: scala.collection.immutable.Stream[(Int, Int)] = Stream((0,1), ?)

scala> fibs.take(8).print
(0,1), (1,1), (1,2), (2,3), (3,5), (5,8), (8,13), (13,21), empty

Scala> fibs.map{ case (x, y) => x}.take(8).print
0, 1, 1, 2, 3, 5, 8, 13, empty
```

Stateful stream processing refers to any stream processing that refers to past information to obtain its result. They require to maintain some *state* information in the process of computing the next element of the stream.

Here, this is held in the recursive argument of the `scanLeft` function, where we can see `fibs` having a tuple of two elements for each element — the sought result, and the next value. We can apply a simple transformation to the list of tuples `fibs` to retain only the leftmost element, and thus obtain the clasical Fibonacci sequence.

STATELESS OR STATEFUL STREAMING

Though we mention the general — and state-less — formula for finding the n-th number in the Fibonacci sequence, there are algorithms that we only know how to express in a stateful form.

For example, Jeremy Gibbons ([Gibbons2004]) defines a streaming algorithm for receiving a stream of digits in one particular base, and returning a stream of digits for the number formed by those input digits in another base. It is easy to see that any algorithm doing this work will always have to store some sort of “carry"in its computation, before even checking the publication to see that the suggested algorithm does.

We’ll see that stateful computations are more general, but that they carry their own constraints as well in a later chapter — in particular, we’ll insist on the need for bounding the memory use of stateful computation.

The Effect of Time

So far, we have had a high-level look at Apache Spark's programming model, which can be expressed as operations on a streaming pipeline in between a data source and a data sink. We have noticed how those operations can be grouped into transformations and aggregations, and how those can often be expressed as higher order functions, a subject we will explore in more detail in Structured Streaming and Spark Streaming. We have considered how there is an advantage in keeping track of intermediary data as we produce results on each element of data stream because it allows us to analyze each of those elements relative to the data stream that they are part of as long as we keep this intermediary data of a bounded and reasonable size. Now, we want to consider another issue unique to stream processing, which is the operation on time-stamped messages.

Computing on time-stamped events

Elements in a data stream always have a *processing time*. That is, by definition, the time at which the stream processing system observes a new event from a data source. That time is entirely determined by the processing runtime and completely independent of the content of the stream's element. However, for most data streams, we also speak of a notion of *event time*, which is the time stamp contained as part of the message payload in the stream. Time stamping is an operation which consists in adding a register of time at the moment of the generation of the message which will become a part of the data stream. It is a ubiquitous practice that is present in both the most humble embedded devices, provided they have a clock, as well as the most complex weblogs in financial transaction systems.

Timestamps as the providers of the notion of time

The reason *time stamping* is so frequent is that it allows users to reason on their data considering the moment at which it was generated. So, since event logs form a larger proportion of the data streams being analyzed today, those timestamps help make sense of what happened to a particular system at a given time. This complete picture is something that is often made more elusive by the fact that transporting the data from the various systems or devices that have created it to the Spark cluster that processes it is a hazardous operation in which some events could be delayed, reordered, or lost.

Often, users of a framework such as Apache Spark want to compensate for those hazards without having to compromise on the reactivity of their system. Out of this desire was born a discipline for producing:

- clearly marked correct and reordered results,
- or intermediary prospective results,

With the classification reflecting the best knowledge that a stream processing system has of the time-stamped events represented by the data stream under the proviso that this view could be completed by the late arrival of delayed stream elements: events-time processing. In Spark, this feature is only offered natively by Structured Streaming. While Spark Streaming lacks built-in support for event-time, it is a question of development effort and some data consolidation processes to manually implement the same sort of primitives, as we will see in a later chapter.

Event Time vs Processing Time

We recognize that there is a timeline in which the events are created and a different one when they are processed:

- *event time* refers to the timeline when the events where originally generated. Typically, a clock available at the generating device places a timestamp in the event itself, meaning that all events from the same source could be chronologically ordered even in the case of transmission delays.
- *processing time* is the time when the event is handled by the stream processing system. This time is only relevant at the technical or implementation level. For example, it could be used to add a processing timestamp to the results and in that way, differentiate duplicates, as being the same output values with different processing times.

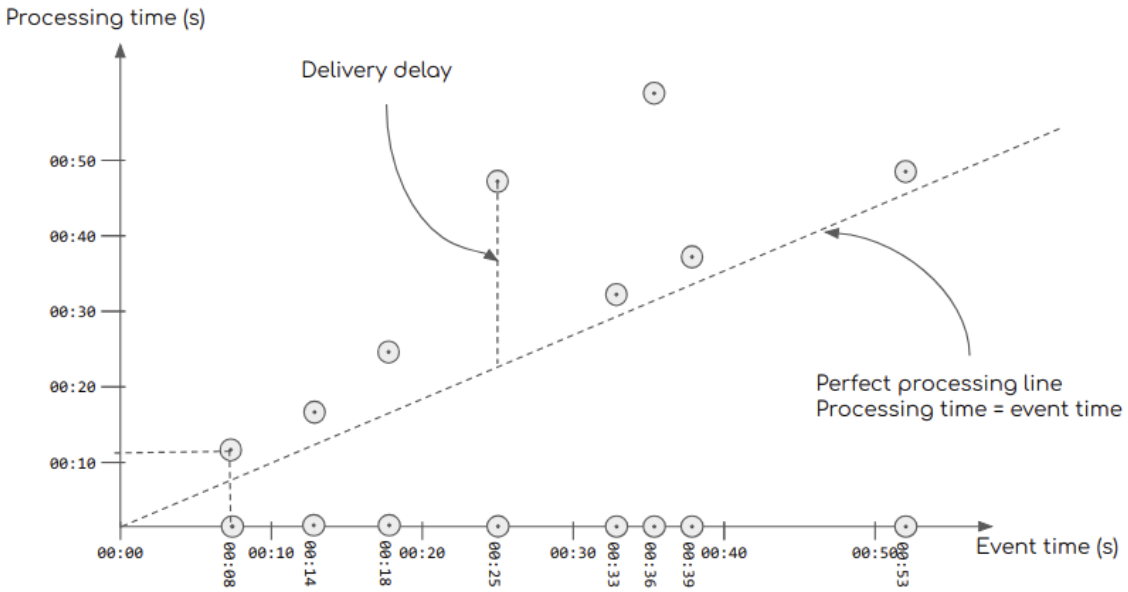


Figure 2-4. Event vs Processing Time

Imagine that we have a series of events produced and processed over time, as illustrated in Figure 2-4:

- The x-axis represents the event timeline and the dots on that axis denote the time at which each event was generated.
- The y-axis is the processing time. Each dot on the chart area corresponds to when the corresponding event in the x-axis is processed. For example, the event created at `00:08` (first on the x-axis) is processed at aproximately 00:12, the time that corresponds to its mark on the y-axis.
- The diagonal line represents the perfect processing time. In an ideal world, using a network with zero delay, event are processed immediately as they are created. Note that there can be no processing events below that line as it would mean that events get processed before they are created.
- The vertical distance between the diagonal and the processing time is the *delivery delay*: the time elapsed between the production of the event and its eventual consumption.

With this framework in mind, lets now consider a 10-second window aggregation.

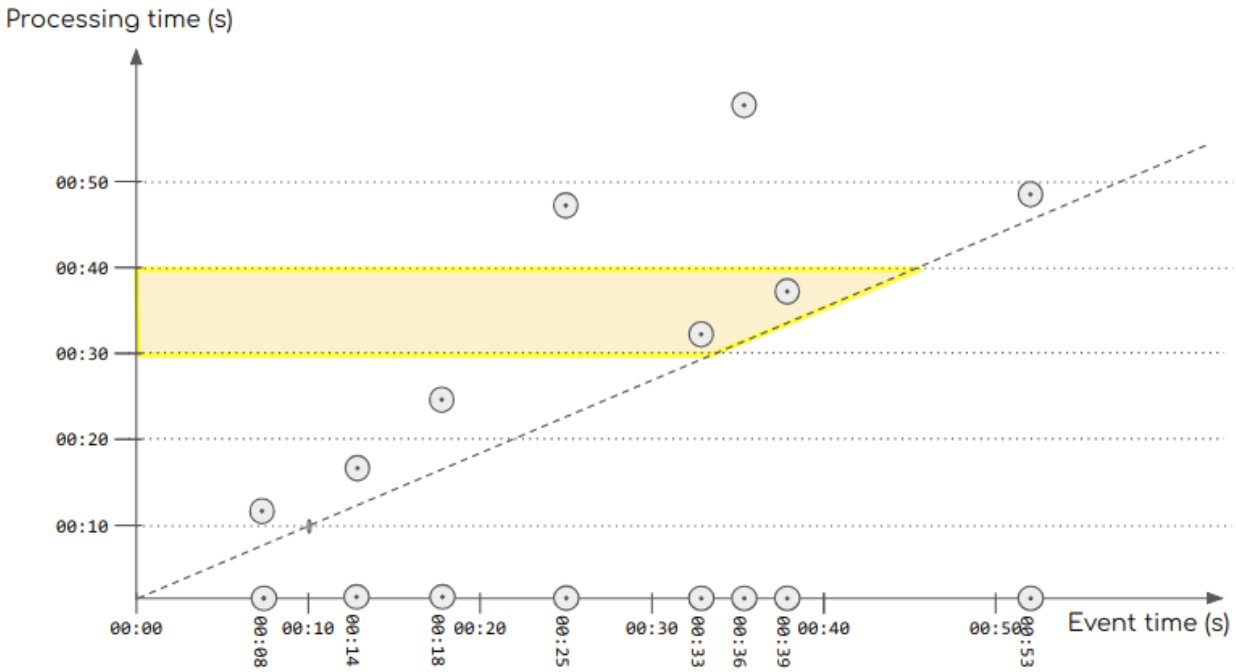


Figure 2-5. Processing-Time Windows

We start by considering windows defined on processing time:

- The stream processor uses its internal clock to measure 10-second intervals.
- All events that fall in that time interval belong to the window.
- In Figure 2-5, the horizontal lines define the 10-second windows.

We have also highlighted the window corresponding to the time interval `00:30-00:40`. It contains two events with event time `00:33` and `00:39`.

In this window, we can appreciate two important characteristics:

- The window boundaries are well-defined as we can see in the highlighted area. This means that the window has a defined start and end. We know what's in and what's out by the time the window closes.
- Its contents are arbitrary. They are unrelated to when the events were generated. For example, while we would assume that a `00:30-00:40` window would contain the event `00:36`, we see that it has fallen out of the resulting set because it was late.

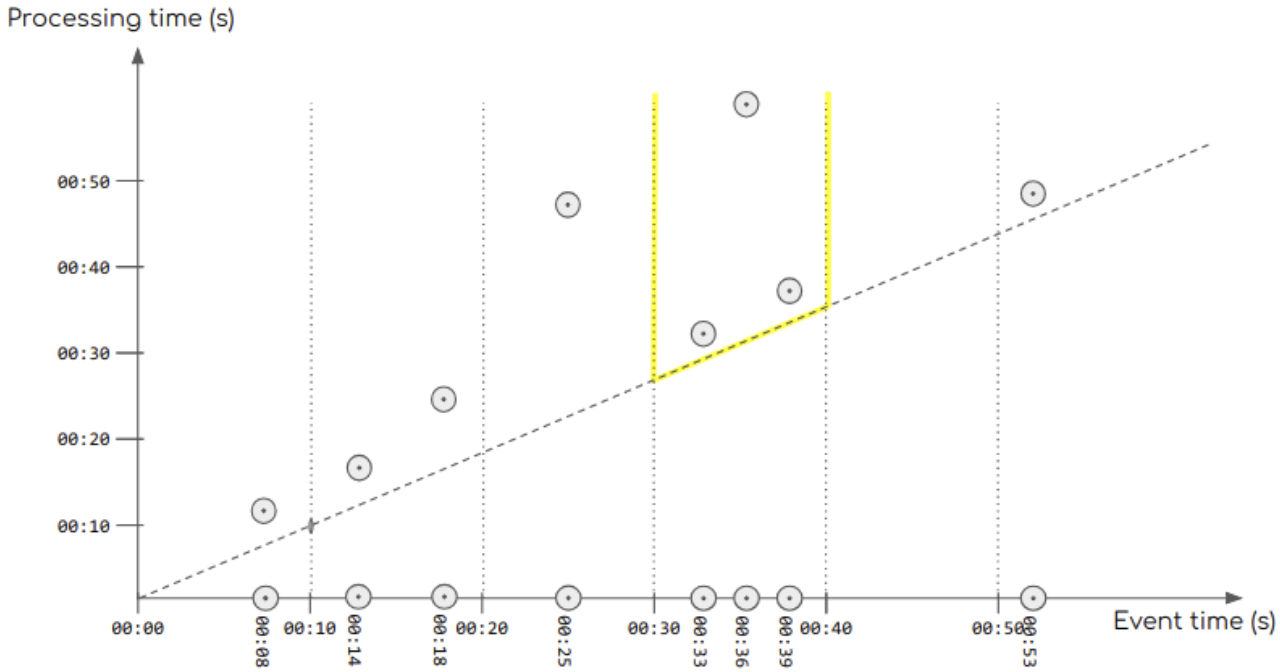
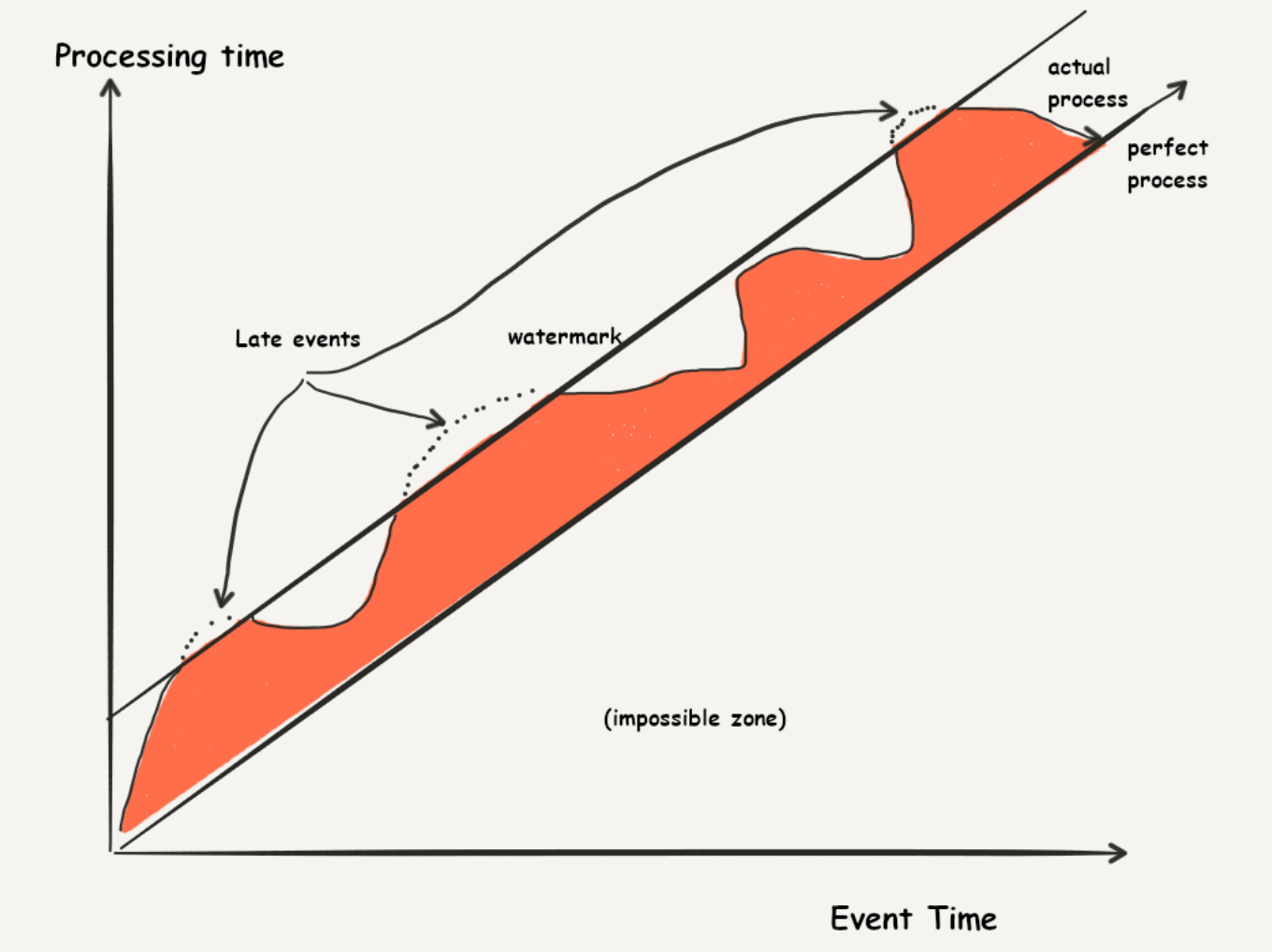


Figure 2-6. Event-Time Windows

Let's now consider the same 10-second window defined on event-time. For this case, we use the *event creation time* as the window aggregation criteria. As we can see in Figure 2-6, these windows look radically different to the processing time windows we saw before. In this case, the window `00:30-00:40` contains all the events that were *created* in that period of time. We can also see that this window has no natural upper boundary that defines when the window ends. The event created at `00:36` was late for more than 20 seconds. As a consequence, to report the results of the window `00:30-00:40`, we need to wait at least until `01:00` What if an event gets dropped by the network and never arrives? How long do we wait? To resolve this problem, we introduce an arbitrary deadline, called *watermark* to deal with the consequences of this open boundary, like lateness, ordering and deduplication.

Computing with a watermark

As we have noticed, stream processing produces periodic results, out of the analysis of the events observed in its input. When equipped with the ability to decode the timestamp contained in those messages, Spark's *Structured Streaming* is able to bucket those messages in two categories based on a notion known as a *watermark*. The *watermark* is at any given moment, *the oldest timestamp that Spark expects to observe on the data stream*. Any events that are older than this expectation are purely and simply ignored. However, to account for possibly delayed events, this watermark is usually much older than the average timestamp of the latest few elements. Note also that this watermark is a fluid value as it evolves over time, sliding a window of delayed tolerance as the time observed in the data stream progress.



Once this notion of watermark is firmly set, Structured Streaming can classify its output in two categories. Either it is producing output relative to events that are all older than the watermark, in which case the output is final because all of those elements have been observed so far, and no further event that old will ever be considered, or it is producing an output relative to the data that is before the watermark and a new delayed element newer than the watermark could arrive on the stream at any moment and can change the outcome. In this latter case, Structured Streaming considers its output as provisional. We will see in detail how to express and operate this sort of computation in Chapter 12.

CAUTION

We want to outline event-time processing as it is an exception to the general rule we have given in Chapter 1 of making no assumptions on the throughput of events observed in its input stream.

With events-time processing, we make the assumption that setting our watermark is appropriate. That is, we can only expect the results of a streaming computation based on events-time processing to be meaningful if the watermark allows for the delays that messages of our stream will actually encounter between their creation-time and their order of arrival on the input data stream. In summary, this only works if the watermark allows to catch all delayed events.

Note finally that with provisionary results, we are storing intermediate values and in one way or another, we require a method to revise their computation upon the arrival of delayed events. This process requires some amount of memory space. As such, events-time processing is another form of stateful computation, and is subject to the same limitation: to handle watermarks, Structured Streaming needs to store a lot of intermediate data, and as such consume a significant amount of memory, that roughly correspond to *the length of the watermark x the rate of arrival x message size*. It is left to the users to make sure they choose a watermark suitable for the events time processing they require and appropriate for the computing resources they have available as well.

Summary

In this chapter, we explored main notions unique to the stream processing programming model:

- Data sources and sinks
- Stateful processing, and
- Events-time processing.

We will explore the implementation of these concepts in the streaming APIs of Apache Spark in subsequent chapters.

[1]Thanks to the Chebycheff inequality, we know that alerts on this data stream should occur with less than 5% probability

Chapter 3. Distributed Stream Processing

In this book, we are about to spend a significant time exposing Stream Processing with Spark for practicing programmer, focusing on the API, operations, standard library and so on. But before we do that, we would like to highlight two choices that will happen at the start of every stream processing application with Spark, dictating the environment in which it will run.

The working engineer using Apache Spark for stream processing will have to:

- choose a cluster manager, and therefore live under the *fault tolerance guarantees* offered by this resource management system,
- tune his systems for the constraints of micro-batching.

This chapter aims at giving a high-level view of those early choices, before later revisiting the nitty-gritty details of cluster configuration and deployment in Chapter 6, and introducing the exception to the rule of micro-batching, continuous processing in Chapter 15. With the present chapter we lay the groundwork that lets each of these later chapters go in depth, without fear that the reader would lose the big picture.

We are first going to look at the discipline of distributing streaming computation on a set of machines that collectively form a *cluster*. This set of machines has a general purpose, and needs to receive the streaming application's runtime binaries and launching scripts - something known as *provisioning*. Indeed, modern clusters are managed automatically and include a large number of machines in a situation of *multi-tenancy*, which means that many stakeholders want to access and use the same cluster at various times in the day of a business. The clusters are therefore managed by *cluster managers*.

Cluster managers are a piece of software that receives utilization requests from a number of users, matches them to some resources, reserves the resources on behalf of the users for a given duration, and places user applications onto a number of resources for them to use. The challenges of the cluster managers' role include non-trivial tasks, such as figuring out the best placements of users requests among a pool of available machines, or securely isolating the user applications if several share the same physical infrastructure. Some considerations where these managers may shine or break includes fragmentation of tasks, optimal placement, availability, preemption, and prioritization. cluster management is therefore a discipline in itself,beyond the scope of Apache Spark. Instead, Apache Spark takes advantage of existing *cluster managers* to distribute its workload over a cluster.

Examples of cluster managers include:

- Apache YARN which is a relatively mature cluster manager born out of the Apache Hadoop project,
- Apache Mesos which is a cluster manager based on Linux's container technology, and which was originally the reason of existence of Apache Spark, and
- Kubernetes which is a modern cluster manager born out of service-oriented deployment APIs, originated in practice at Google and developed in its modern form under the flag of the Cloud Native Computing Foundation.

Now, where Spark can deter and sometimes confuse people is that Apache Spark, as a distribution, includes a cluster manager of its own, meaning Apache Spark has the ability to serve as its own particular deployment orchestrator.

In the rest of this chapter we will look at: - Spark's own cluster managers and how their **special purpose** means they take on less responsability in the domain of fault tolerance or multi-tenancy than production cluster managers like Mesos, YARN or Kubernetes. - how there is a **standard language of delivery guarantees** expected out of a distributed streaming application, how they differ from each other, and how Spark meets those guarantees, - and how micro-batching, a distinctive factor of Spark's approach to stream processing, comes from the decade-old model of **bulk-synchronous processing**, and enlightensthe evolution from Spark Streaming to Structured Streaming.

Spark's own cluster manager

Spark has two internal cluster managers:

The *local* cluster manager

> emulates the function of a cluster manager (or cluster manager) for testing purposes. It reproduces the presence of a cluster of distributed machines using a threading model that only relies on your local machine having a few available cores. This mode is usually not very confusing, since it executes only on the user's laptop.

The *stand alone* cluster manager

> A relatively simple, Spark-only cluster manager which is rather limited in its availability to slice and dice resource allocation. The *standalone* cluster manager holds and makes available the whole worker node on which a Spark executor is deployed and started. It also expects the executor to have been pre-deployed there, and the actual shipping of that jar to a new machine is not within its scope. It has the ability to take over a specific number of executors, which are part of its deployment of worker nodes, and execute a task on it. This cluster manager is extremely useful for the Spark developers to provide a barebones resource management solution, that allows to focus on improving Spark in an environment without any bells and whistles. It's not recommended for production deployments.

As a summary, Apache Spark is a *task scheduler*, in that what it schedules is *tasks*, a unit of distribution of computation that has been extracted from the user program. Spark also communicates and is deployed through *cluster managers* including Apache Mesos, YARN, Kubernetes, or allowing for some cases its own stand-alone cluster manager. The purpose of that communication is to reserve a number of *executors*, which are the unit to which Spark understands equal-sized amounts of computation resources, a virtual "node" of sorts. The reserved resources in question could be provided by the cluster manager as:

- limited processes (e.g. in some basic use cases of YARN), where processes have heir resource consumption metered but are not prevented from accessing each other's resource by default.
- *containers* (e.g. in the case of Mesos or Kubernetes) where containers are a relatively light-weight resource reservation technology that is born out of the cgroups and name spaces of the Linux kernel and have known their most popular iteration with the Docker project.

- they also could be one of the above deployed on *virtual machines*, themselves coming with specific cores and memory reservation.

Detailing the different levels of isolations entailed by these three techniques is beyond the scope of this book, but well worth exploring for productions setups.

Note that in an enterprise-level production cluster management domain we also encounter notions such as job queues, priorities, multi-tenancy options, preemption, that are properly the domain of that cluster manager and therefore not something that is very frequently talked about in material that is focused on Spark.

However, it will be essential for the developers to have a firm grasp of the specifics of their cluster manager setup to understand how to be a “good citizen” on a cluster of machines which are often shared by several teams. There are many good practices on how to run proper cluster manager while many teams compete for its resources. And for those recommendations, you should consult both the references listed at the end of this chapter and your local DevOps (developer operations) team.

Understanding Resilience and Fault Tolerance in a Distributed System

Resilience and fault tolerance are absolutely essential for a distributed application: they are the condition by which we will be able to perform the user’s computation to completion. Nowadays, clusters are made of commodity machines that are operated ideally near peak capacity over their lifetime.

To put it mildly, hardware breaks quite often. A *resilient* application can make progress with its computation despite latencies and non-critical faults in its distributed environment. A *fault-tolerant*applicaiton is able to succeed and complete computation despite the unplanned terminateion of one or several of its nodes.

This sort of resiliency is especially relevant in stream processing, since the applications that we scheduling is supposed to live for an undetermined amount of time. That undetermined amount of time is often correlated with the life-cycle of the data source. For example, if we are running a retail website and we are analyzing transactions and website interactions as they come in the system, to the actions and clicks and navigation of users visiting the site then we potentially have a data source that will be available for the whole duration of the lifetime of our business, which we hope to be very long, if our business is going to be successful.

As a consequence, a system that will process our data in a streaming fashion should run uninterrupted for long periods of time.

This "*show must go on*" approach of stremaing computation makes the resiliency and fault tolerance characteristice of our applications more important. For a batch job, we could launch it, hope it would succeed, and relaunch if we needed to change it or in case of failure. For an online streaming Spark pipeline, this is not a reasonable assumption.

Fault Recovery

In the context of fault tolerance, we are also interested in understanding how long it takes to recover from failure of one particular node. Indeed, streaming computation has a particular aspect : data continues being generated by the data source in real time. To deal with a batch computing failure, we always have the opportunity to restart from scratch and accept that obtaining the results of computation will take longer. A very primitive form of fault tolerance is hence detecting the failure of a particular node of our deployment, stopping the computation, and restarting from scratch. That process may take more than twice the original duration that we had budgeted for that computation, but if we are not in a hurry, this still acceptable.

For streaming computations, *we need to keep receiving data* and thus potentially storing it, if the recovering cluster is not ready to assume any processing yet. This may pose a problem at a high throughput: if we try restarting from scratch, we will need not only to re-process all the data that we have observed since the beginning of the application — which in itself may be a challenge — but we will, during that re-processing of historical data, need it to continue receiving and thus potentially storing new data that was generated while we were trying to catch up. This pattern of restarting from scratch is something so untractable for streaming that we will pay special attention to Spark's ability to restart only *minimal* amounts of computation in the case that a node becomes unavailable or non-functional.

Cluster Manager Support for Fault Tolerance

We want to highlight why it is still important to understand Spark's fault tolerance guarantees, even if there are similar features present in the cluster managers of YARN, Mesos or Kubernetes. To understnad this, we can consider that cluster managers help with fault tolerance when they work hand in hand with a framework that is able to report failures and request new resources to cope with those exceptions. Spark posesses such capabilities.

For example, *production* cluster managers such as YARN, Mesos or Kubernetes have the ability to detect a node's failure by inspecting endpoints on the node, and asking the node to report on its own health state. If they detect a failure, and they have spare capacity, they will replace that node with another, made available to Spark. That particular action implies that the Spark executor code will start anew in another node, and attempt to join the existing Spark cluster.

The *cluster manager*, by definition, does not have introspection capabilities into the applications being run on the nodes that it reserves. As such, it is not able to determine if that node contains some state that should be reproduced in the form of checkpointed files or restore backup data to "revive" a node. It is not able to understand at which stage of the job a node should rejoin the computation, and it is not able to understand what that node should do next in terms of its oncoming task. All of those actions are responsibility of the *application*, like Spark in our case. The goal here is for us to explore that, if a node is being replaced by the cluster manager, Spark has modes that allow it dynamically to take advantage of this new node and to distribute computation onto it.

In this book, we will focus on Spark's responsibilities as an application and underline the capabilities of a cluster manager only when necessary: for instance, a node could be replaced because of a harware failure, or because its work was simply preempted by a higher-priority job.

Apache Spark is blissfuly unaware of the *why*, and focuses on the *how*. The cluster manager would know, but we won't lose sleep over it.

Delivery guarantees

As we have seen in the streaming model, the fact that streaming jobs act on the basis of data that is generated in real time means that intermediate results need to be provided to the *consumer* of that streaming pipeline on a regular basis.

Those results are being produced by some part of our cluster. Ideally we would like those observable results to be coherent, in line and in real-time with respect to the arrival of data. This means that we want results that are exact and we want them as soon as possible. However, distributed computation has its own challenges in that it sometimes includes not only individual nodes failing, as we have mentioned, but it also encounters situations like *network partitions*, where some parts of our cluster are not able to communicate with other parts of that cluster as illustrated in Figure 3-1.

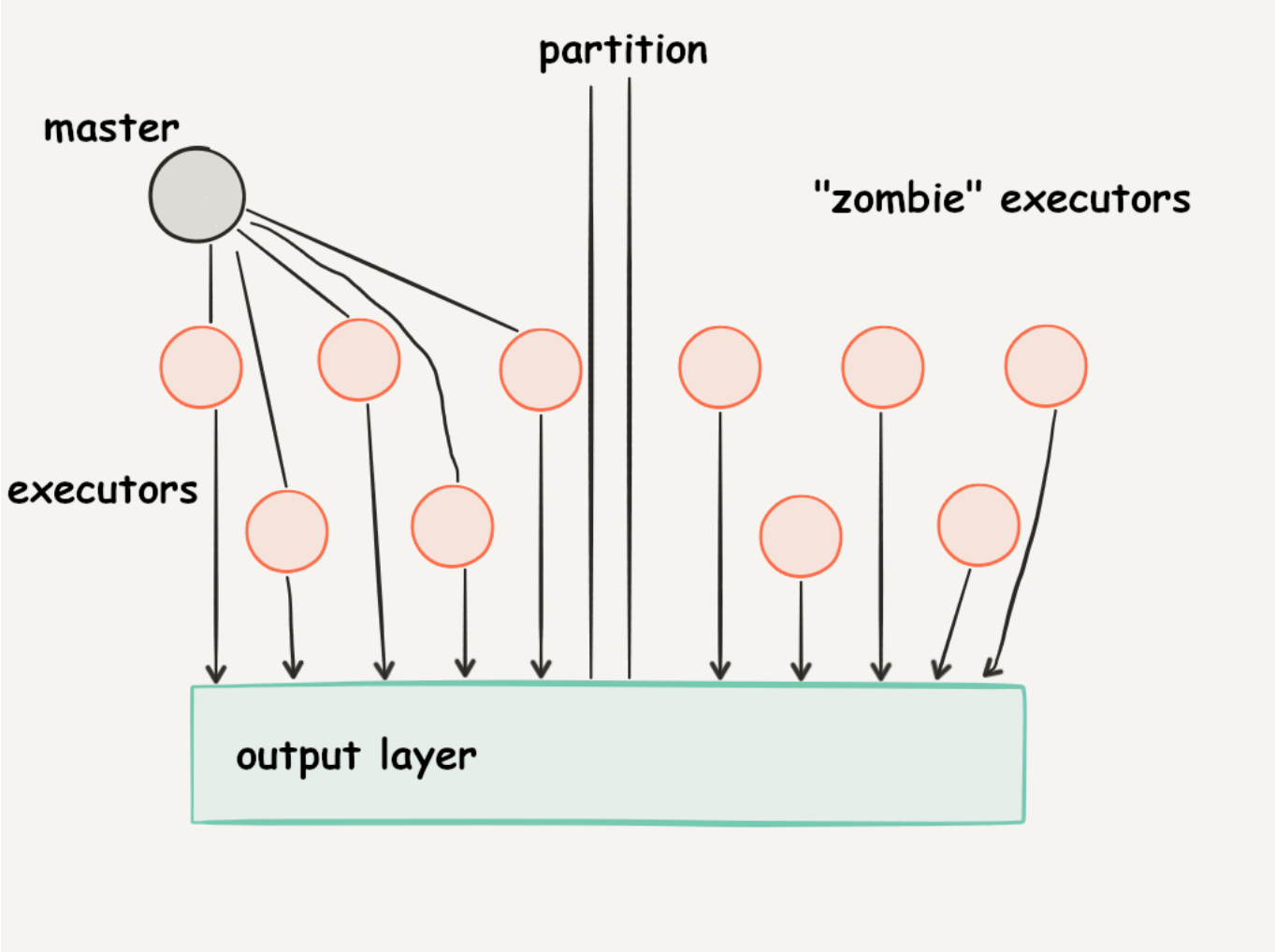


Figure 3-1. A Network partition

Spark organizes itself in a *master/worker* architecture. A specific machine, the master, is tasked with keeping track of the *job progression* along the job submissions of a user, and the computation of that program occurs as the data arrives. However if the network partitions separate some part of the cluster, the master may be able to keep track of only the part of the executors that form up the initial cluster. In the other section of our partition, we will find nodes that are entirely able to function, but will simply unable to account for the proceedings of their computation to the master.

This creates an interesting case in which those "*zombie*" nodes do not receive new tasks, but may well be in the process of completing some fragment of computation they were previously given. Being unaware of the partition, they will report their results as any executor would. And since this reporting of results sometimes does not go through the master (for fear of making the master a bottleneck), the reporting of these “zombie” results could succeed.

Because the master, a single point of bookkeeping, does not know that those “zombie” executors are still functioning and reporting results, it will reschedule the same tasks that the lost executors had to accomplish on new nodes. This creates a *double-answering* problem, in which the zombie machines lost through partition and the machines bearing the rescheduled tasks both report the same results. This bears real consequences: one example of stream computation that we previously mentioned is routing tasks for financial transactions. A double withdrawal, in that context, or double stock purchase orders, could have tremendous consequences.

This challenge has therefore lead to a distinction between *at least once* processing and *at most once*processing.

- “at least once” processing ensures every element of a stream has been processed once or more,
- “at most once” processing ensures every element of the stream is processed once or less,
- “exactly once” is the compbination of both of the above.

At least once processing is the notion that we want to make sure that every chunk of initial data has been dealt with — it deals with the node failure we were talking about earlier. As we’ve mentioned, when streaming computation suffers a partial failure where some nodes need to be replaced or some data needs to be recomputed, we need to re-process the lost units of computation, while keeping the ingestion of data going. That requirement means that if you do not respect *at least once* processing then there is a chance for you, under certain conditions, to loose data.

The antisymmetric notion is called *at most once* processing. *At most once* processing systems guarantee that the *zombie nodes* repeating the same results as a rescheduled node are treated in a coherent manner, in which we keep track of only one set of results. By keeping track of *what data*their results *were about*, we’re able to make sure we can discard repeated results, yielding *at most once*processing guarantees. The way in which we achieve this relies on the notion of *idempotence* applied to the “last mile” of result reception. Idempotence qualifies a function such that if we apply it twice (or more) to any data, we will get the same result as the first time. This can be achieved by keeping track of the data that we are reporting a result for, and having a bookeeping system at the output of our streaming computation.

Micro-batching and one-element-at-a time

In this section, we would like to take the time to acknowledge two important approaches to stream processing: bulk-synchronous processing, and one-at-a-time record processing.

The objective of this is to connect those two ideas to the two APIs that Spark possesses for stream processing: Spark Streaming and Structured Streaming.

Spark Streaming, the more mature model of streaming computation in Spark, is roughly approximated by what's called a Bulk Synchronous Parallelism (BSP) system.

The gist of BSP, is that it includes two things:

- a split distribution of asynchronous work and
- a synchronous barrier, coming in at fixed intervals.

The split is the idea that each of the successive steps of work to be done in streaming is separated in a number of parallel chunks that are roughly proportional to the number of executors available to perform this task. Each executor receives their own chunk (or chunks) of work and works separately until the second element comes in. A particular resource is tasked with keeping track of the progress of computation. With Spark Streaming, this is a synchonization point at the "driver", that allows the work to progress to the next step. In-between those scheduled steps, all the executors on the cluster are doing the same thing.

Note that what is being passed around in this scheduling process is the functions that describe the processing that the user wants to execute. The data is already on the various executors, most often being delivered directly to these resources over the lifetime of the cluster.

This was coined "function-passing style" by Heather Miller in 2016 (and formalized in [Miller2016]): asynchronously pass safe functions to distributed, stationary, immutable data in a stateless container, and use lazy combinators to eliminate intermediate data structures.

The rate at which further rounds of data processing are scheduled is dictated by time, only in Spark Streaming and other BSP-like systems. The batch interval, as we will see, is an arbitrary duration that is measured in batch processing time — i.e. what you would expect to see in a "wall clock" sitting in your cluster. The idea of this batch processing is that, as we are observing a stream of data — and therefore observing data from a metaphorical wire rather than separating that data into atomic elements — and counting elements which would be efficient in determining the size of the work to be done, but very inefficient for the purpose of determining the rate and the throughput at which data writes over the wire. We choose to implement barriers at fixed intervals that correspond more to the real time notion of data processing that Spark Streaming embodies.

NOTE

Bulk-synchronous parallelism (BSP) is a very generic model for thinking about parallel processing, introduced by Leslie Valiant in the 1990s. Intended as an abstract (mental) model above all else, it was meant to provide a pendant to the Von Neumann model of computation for parallel processing.

It introduces three key concepts:

1. A number of components, each performing processing and/or memory functions.
2. A router that delivers messages point to point between components.
3. Facilities for synchronizing all or a subset of the component at a regular time interval L, where L is the periodicity parameter.

The purpose of the bulk-synchronous model is to give clear definitions that allow thinking of the moments where a computation can be performed by agents each acting separately, while pooling their knowledge together on a regular basis to obtain one single, aggregate result. Valiant introduces the notion:

A computation consists of a sequence of supersteps. In each superstep, each component is allocated a task consisting of some combination of local computation steps, message transmissions and (implicitly) message arrivals from other components. After each period of L time units, a global check is made to determine whether the superstep has been completed by all the components. If it has, the machine proceeds to the next superstep. Otherwise, the next period of L units is allocated to the unfinished superstep.

This model also proceeds to give guarantees about the scalability and cost of this mode of computation — to learn more about this, consult [Valiant1990]. It was influential in the design of modern graph processing systems such as Google's Pregel. Here, we will use it as a way to speak of the timing of synchonization of parallel computation in Spark's `DStream`s.

By contrast, one-record-at-a-time processing functions by *pipelining*: it analyses the whole computation as described by user-specified functions and deploys it as pipelines using the resources of the cluster. Then the only remaining matter is to flow data through the various resources, following the prescribed pipeline. Note that in this later case, each steps of the computation is materialized at some place in the cluster at any given point.

Systems which function mostly according to this paradigm include Apache Flink, Naiad, Storm and IBM Streams. This does not necesarily mean that those systems are incapable of micro-batching, but rather characterizes their major or most native mode of operation, and makes a statement on their dependency on the process of pipelining, often at the heart of their processing.

The minimum latency, or time needed for the system to react to the arrival of one particular event, is very different between those two: minimum latency of the micro-batching system is therefore the time needed to complete the reception of the current micro-batch (the batch interval) plus the time needed to start a task at the executor where this data falls (also called scheduling time). On the other hand, a system processing records one by one can react as soon as it meets the event of interest.

Despite their higher latency, micro-batching systems are not without advantages.

- First, they are able to *adapt* at the synchronization barrier boundaries. That adaptation might represent the task of recovering from failure, if a number of executors have been shown to become deficient or lose data. The periodic synchonization may also give us an opportunity to add or remove executor nodes, giving us the possibility to grow or shrink our resources depending on what we're seeing as the cluster load, observed through the throughput on the data source.

- Second, our bulk synchronous processing systems can sometimes have an easier time providing *strong consistency*, because their batch determinations — that indicate the beginning and the end of a particular batch of data — are deterministic and recorded. Thus any kind of computation can be redone and produce the same results the second time.
- Finally, having data available *as a set* that we can probe or inspect at the beginning of the micro-batch allows us to perform efficient optimizations, that can provide ideas on the way to compute on the data, exploiting that on *each* micro-batch, we can consider the specific case rather than the general processing which is used for all possible input.

More importantly, the simple presence of the micro-batch as a well-identified element also allows an efficient way of specifying programming for both batch processing — where the data is at rest and has been saved somewhere — and streaming — where the data is in flight. The micro-batch, even for mere instants, *looks* like data at rest.

The marriage between micro-batching and one-record-at-a-time processing as is implemented in systems like Flink or Naiad is still a subject of research.[1] >>]

But while not solving every issue, Structured Streaming, while still backed by a main implementation that relies on micro-batching, allows for an evolution that is independent on a fixed batch interval. In fact, it could be said that the main current internal functioning of Structured Streaming is that of micro-batchign with a dynamic batch interval but it does not surface to the API.[2]

What is this notion of dynamic batch interval, which is central to the function of a stricter streaming model?

The dynamic batch interval is the notion that the recomputation of data in a streaming data frame or data set consists in an update of existing data with the new elements seen over the wire. This update is occurring based on a trigger and the usual basis of this would be time duration. That time duration is still determined based on a fixed world clock signal that we expect to be synchronized within our entire cluster and that represents a single synchronous source of time that, shared among every executor.

However, this trigger can also be the statement of “as often as possible”. That statement is simply the idea that a new batch should be started as soon as the previous one has been processed, given a reasonable initial duration for the first batch. This means that the system will launch batches as often as possible. In this situation, the latency that can be observed is closer to that of one element at a time processing. The idea here is that the micro-batches produced by this system will converge to the smallest manageable size, making our stream flow faster through the executor computations that are necessary to produce a result. As soon as that result will have been produced, a new query will be started and scheduled by the Spark Master.

The main steps in Structured Streaming processing are that:

- when the Spark driver triggers a new batch, processing starts with updating the account of data read from a data source, in particular, getting data offsets for the beginning and the end of the latest batch which.
- this is followed by logical planning, the construction of successive steps to be executed on data, followed by query planning (intra-step optimization)

- and then the launch and scheduling of the actual computation by adding a new batch to update the continuous query that we're trying to refresh.

Hence, from the point of view of the computation model, we will see that the API is significantly different from Spark Streaming.

We will now dive deeper into what Structured Streaming batches mean and what they mean with respect to operations. One of the first and most interesting aspects of this is that the batch interval that we are using is no longer a computation budget. The idea with Spark Streaming was that if we produce data every 2 minutes, and flow data into Spark's memory every 2 minutes, then we should produce the results of computation on one batch of data every 2 minutes, to clear the memory from our cluster. Ideally, as much data flows out as flows in, and the collective memory of our cluster is not overloaded.

Without this fixed synchronization, we note that our ability to see performance issues in our cluster is more complex: a cluster that is unstable — that is, unable to "clear out" data by finishing to compute on it as fast as new data flows in — will see ever-growing batch processing times, with an accelerating growth. We can expect that keeping a hand on this batch processing time will be pivotal.

However, if we have a cluster that is correctly-sized with respect to the throughput of our data, there are a lot of advantages to have an *as often as possible* update. In particular, we should expect to see very frequent results from our Structured Streaming cluster with much more granularity than we used to in the time of a conservative batch interval. One of the APIs that will logically see the most consequences from that change in computation model is the Windowing API by which we will be able to group the result of our queries in consistent tumbling or sliding with those as we are about to see.

[1]One interesting Spark-related project that recently came out of the University of Berkeley is called Drizzle and uses "group scheduling" to form a sort longer-lived pipeline that persists across several batches, for the purpose of creating near-continuous queries. See << Venkataraman2016, [Venkataraman2016

[2]Structured Streaming is also implementing continuous processing for some operators, something we wil touch upon in Chapter 15.

Chapter 4. Streaming Architectures

So far, we have seen:

- the definitions of a stream processing system, and its natural synergy with both incremental algorithms and the modern appetite for processing data as it is being produced.
- the sequencing of stream processing pipelines, in successions of transformations in between source and sink boundaries,
- the definition and usage of stateful stream processing, and event-time processing

Now we turn to the link between stream processing an batch processing in a concrete architecture. In particular, we're going to ask ourselves the question of whether batch processing is still relevant if we have a system that can do stream processing, and if so why?

In this chapter, we contrast two conceptions of streaming application architecture: the *Lambda architecture*, which suggests duplicating a streaming application with a batch counterpart running in parallel to obtain complementary results, and the *Kappa architecture*, which purports that if two versions of an application have to be compared, those should both be streaming applications. We are going to see in detail what those architectures mean to achieve, and we examine that while the Kappa architecture is easier and lighter to implement in general, there may be cases where a Lambda architecture is still needed, and why.

The use of a batch processing component in a streaming application

Often, if we evolve a batch application that ran on a periodic interval into a streaming application, we are provided with batch datasets already — and a batch program representing this periodic analysis as well. In this evolution use case, as described in the prior chapters, we want to evolve to a streaming application to get the benefits of a lighter, simpler application that gives faster results.

In a greenfield application, we may also be interested in creating a reference batch dataset: most data engineers don't work on merely solving a problem once, but revisit their solution, and continuously improve it, especially if value or revenue is tied to the performance of their solution. For this purpose, a batch data set has the advantage of setting a benchmark: once collected, it does not change anymore, and can be used as a "test set".

We can indeed replay a batch dataset to a streaming system to compare its performance to prior iterations, or to a known benchmark. Three uses of have therefore evolved with Spark Streaming, from the least to the most mixed with batch processing:

- *code reuse*, often born out of a reference batch implementation, seeks to re-employ as much of it as possible, so as not to duplicate efforts. It is a subject where Spark shines, since it is particularly easy to call functions that transform `RDD` s and Dataframes — they share most of the same APIs, and only the setup of the data input and output is distinct.
- *data reuse*, where a streaming application feeds itself from a feature or data source prepared, at regular intervals, from a batch processing job. This is a frequent pattern: for example, some international applications have to deal time conversions, and a frequent

pitfall is that daylight saving rules change on a more frequent basis than expected. In this case, it is good to be thinking of this data as a new dependent source that our streaming application feed itself off.
- *mixed processing*, where the application itself is understood to have both a batch and a streaming component during its lifetime. This pattern does happen relatively frequently, out of a will to manage both the precision of insights provided by an application, and as a way to deal with the versioning and the evolution of the application itself.

The first two uses are uses of convenience, but the last one introduces a new notion: using a batch data set as a benchmark. We'll see in the next subsections how this impacts the architecture of a streaming application.

The Lambda and Kappa Architectures, or how to compare streaming applications

In the world of replay-ability and performance analysis over time, there are two historical but conflicting recommendations. Remember our main concern was about how to measure and test the performance of a Streaming Application. When we do so, there are two things that can change in our setup: the nature of our model itself, as a result of our attempt at improving it, and the data that the model operates on, as a result of organic change. For instance, if we are processing data from weather sensors, we can expect a seasonal pattern of change in the data.

To make sure we compare apples to apples, we have already established that replaying a *batch dataset*to the two versions of our streaming application is useful: it lets us make sure that we are not seeing a change in performance that is really reflecting a change in the data. Ideally, in the case above, we would test our improvements in yearly data, making sure we're not over-optimizing for the current season at the detriment of performance 6 months after.

But we want to contend that a comparison with a *batch analysis* is necessary as well, beyond the use of a benchmark dataset — and this is where the architecture comparison helps. The Lambda Architecture (Figure 4-1) suggests taking a batch analysis performed on a periodic basis — say, nightly — and to supplement the model thus created with streaming refinements as data comes, until we are able to produce a new version of the batch analysis based on the whole day's data.

It was introduced as such by Nathan Marz in a blog post, How to beat the CAP Theorem.[1] It proceeds from the idea that we want to emphasize two novel points beyond the precision of the data analysis:

- the historical replay-ability of data analysis is important,
- the availability of results proceeding from fresh data is also a very important point.

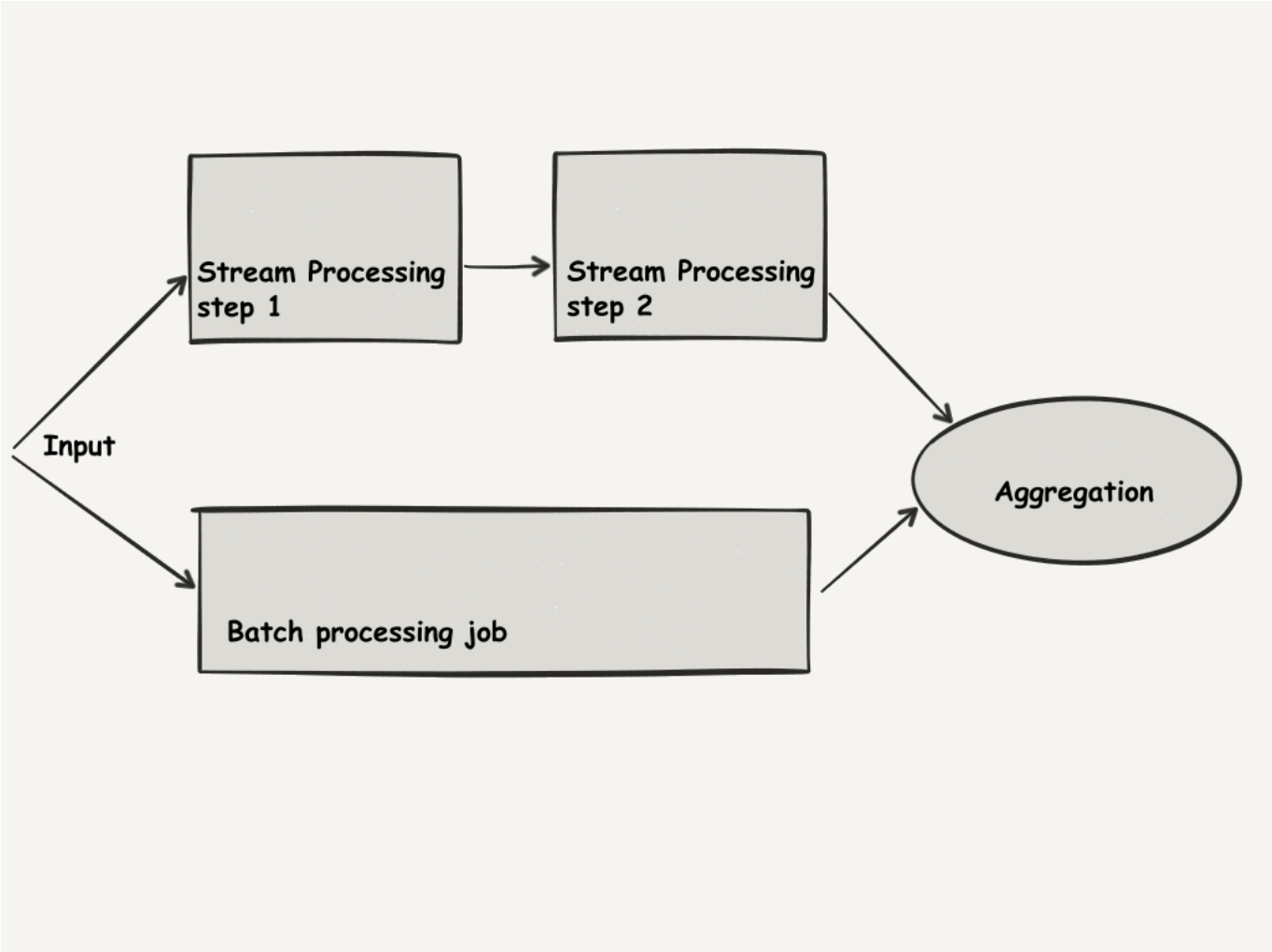


Figure 4-1. The Lambda Architecture

This is a useful architecture, but its drawbacks seem obvious as well: such a setup is complex, and requires maintaining two versions of the same code, for the same purpose. Even if Spark helps in letting use reuse most of our code between the batch and streaming versions of our application, the two versions of the application are distinct in lifecycles, which may seem complicated.

Some have suggested that it would be enough to keep the ability to feed the same dataset to two versions of a streaming application (the new, improved experiment, and the older, stable workhorse). This helps maintainability of our solution, and is outlined in the figure below.

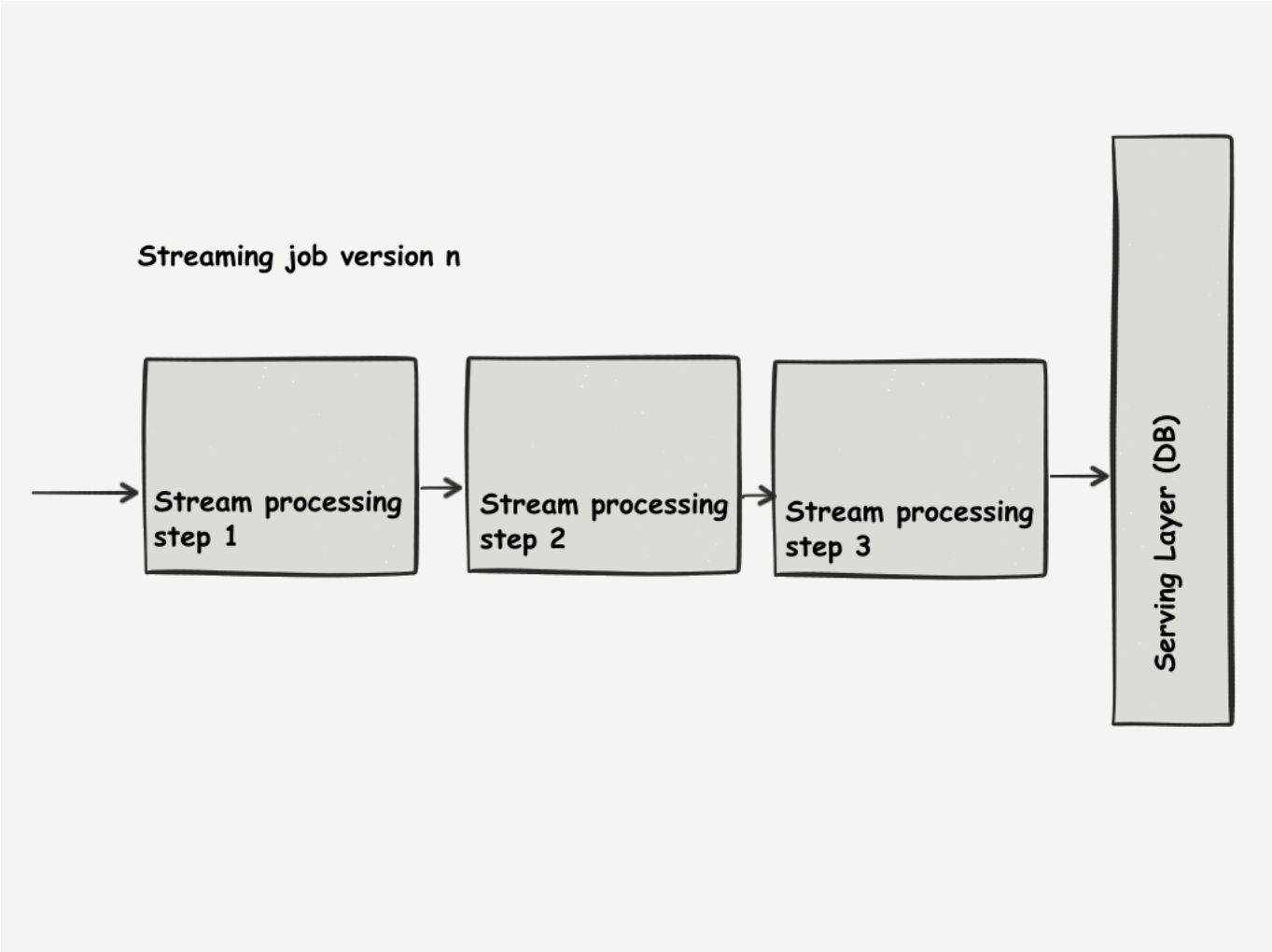


Figure 4-2. The Kappa Architecture

This architecture compares two streaming applications, and does away with any batching, noting that if reading a batch file is needed, a simple component can replay the contents of this file, record by record, as a streaming data source. This simplicity is still a great benefit, since even the code that consists in feeding data to the two versions of this application can be reused. In this paradigm, called the **Kappa architecture** ([Kreps2014]) here is no deduplication, and the mental model is simpler.

This begs the question: is batch computation still relevant? Should we convert our applicaitons to be all streaming, all the time.

We think the Lambda architecture is not dead, and in fact vitally useful in some cases — though those are not always easy to figure out.

There are some use cases in which it is still useful to go through the effort of implementing a batch version of our analysis, and compare it to our streaming solution. We want to paint this case in the next two subsections.

Streaming algorithms are sometimes completely different in nature

Sometimes, it is difficult to deduce batch from streaming, or the reverse, and those two classes of algorithms have different characteristics. This means at first glance that we may not be able to reuse code between both approaches, but also, and more importantly that relating the performance characteristics of those two modes of processing should be done with high care.

To make things more precise, let's look at an example: the buy or rent problem. In this case, we decide to go skiing. We can buy skis for 500$ or rent them for 50$. Should we rent or buy?

Our intuitive strategy is to first rent, to see if we like skiing. But suppose we do: in this case, we will eventually realize we will ahve spent more money than we would have if we had bought the skis in the first place.

In the batch version of this computation, we proceed "in hindsight", being given the total number of times we will go skiing in a lifetime. In the streaming, or online verison of this problem, we are asked to make a decision (produce an output) on each discrete skiing event, as it happens. The strategy is fundamentally different.

In this case, we can consider the competitive ratio of a streaming algorithm. We run the algorithm on the worst possible input, and then compare its "cost" to the decision that a batch algorithm would have taken, "in hindsight".

In our buy-or-rent problem, let's consider the following streaming strategy: we rent until renting makes our total spending as much as buying, in which case we buy.

If we go skiing 9 or less times, we are optimal, because we spend as much as what we would have in hindsight. The competitive ratio is one. If we go skiing 10 times or more, we pay 450$ + 500$ = The worst input is to receive ten "ski trip" decision events, in which case the batch algrithm, in hindsight, would have paid 500$. The competitive ratio of this strategy is (2 - 1/10).

If we were to choose another algorithm, say "always buy on the first occasion", then the worst possible input s to go skiing only once, which means that the competitive ratio is 500$ / 50$ = 10.

A better competitive ratio is smaller, whereas a competitive ratio above one shows that the streaming algorithm performs measurably worse on some inputs. It is easy to see that with the worst input condition, the batch algorithm, which proceeds in hindsight with strictly more information, is always expected to perform better (the competitive ratio oany stremaing algorithm is greater than one).

Streaming algorithms can't be guaranteed to measure well against batch algorithms

Another example of those unruly cases is the bin-packing problem. The bin-packing problem, with an input of a set of objects of different sizes or different weights, which are fit them into a number of bins or containers, each of them having a set volume or set capacity in terms of weight or size, such that this assignment of objects into bins minimizes the number of containers used. In computational complexity theory, the offline ration of that algorithm is known to be NP-hard. The simple variant of the problem, is the *decision* question: knowing if that set of objects will fit into a

specified number of bins. It is itself NP-complete, meaning — for our purposes here — computationally very difficult in and of itself.

In practice, this algorithm is used very frequently, from the shipment of actual goods in containers, to the way operating systems match memory allocation requests to blocks of free memory of various sizes.

There are many variations of these problems, but we want to focus on the distinction between online versions — where the algorithm does not have the full range of the input stream of objects — and offline versions — where the algorithm can examine the entire set of input objects before it even starts computation.

The offline version of that particular algorithm always has an optimal solution. This is very straight-forward to see, meaning that the greedy approximation algorithm always allows placing the input objects into a set number of bins that is, at worst, sub-optimal; meaning we use more bins than necessary. The algorithm processes the items in arbitrary order, then places each item in the first bin that can accommodate it and if no such bin exists, it opens a new bin and puts the item within that new bin.

That algorithm is particularly inefficient, but shows that there exists an optimal towards which we might want to strive.

A better algorithm, which is still relatively intuitive to understand, is the first fit decreasing strategy, which operates by first sorting the items to be inserted in decreasing order of their sizes. Then inserting each item into the first bin in the list with sufficient remaining space. That algorithm was proven in 2007 to be much closer to the optimal algorithm producing the absolute minimum number of bins ([Dosa2007])).

First fit decreasing, however, hinges on the idea that we can first sort the items in decreasing order of sizes before we start processing them and packing them into bins. In the case of the online bin-packing problem, the situation is completely different in that there is no access to that sorting. Not only do they not have access to that, but a potentially large number of possible bins must be considered as places where an item could possibly fit. Intuitively, it is thus easy to understand that the online bin packing problem — which by nature lacks foresight when it operates — is much more difficult than the offline bin packing problem.

WARNING

That intuition is in fact supported by proof if we consider the *competitive ratio* of streaming algorithms. This is the ratio of resources consumed by the online algorithm, to those used by an online optimal algorithm delivering the minimal number of bins by which the input set of objects encountered so far can be packed. This competitive ratio for the knapsack (or bin-packing) problem is in fact arbitrarily bad (that is, large — see [Sharp2007]), meaning that it is always possible to encounter a “bad” sequence in which the performance of the online algorithm will be arbitrarily far from that of the optimal algorithm.

NOTE

The performance ratio or competitive ratio is a measure of how far from the optimal the values returned by an algorithm are, given a measure of optimality (here the number of bins utilized). an algorithm is Formally, ρ-competitive if its objective value is no more than ρ times the optimal off-line value for all instances.

The larger issue presented in this section is that there is no guarantee that a streaming algorithm will perform better than a batch algorithm, since those algorithms have to function without foresight. In particular, some online algorithms, including the knapsack problem, have been proven to have an arbitrary large performance ratio when compared to their offline algorithms.

What this means, to use an analogy, is that is on the one hand we have one worker that receives the data as batch, as if it was all in a **storage room** from the beginning, and the other receiving the data in a streaming fashion, as if it was on a **conveyor belt**, then, *no matter how clever our streaming worker is, there is always a way to place items on the conveyor belt in such a pathological way that he will finish his task with an arbitrarily worse result than the batch worker*.

The take-away message for the lambda architecture is twofold:

- streaming systems are indeed "lighter": their semantics can express a lot of low-latency analytics in expressive terms.
- streaming APIs invite us to implement analytics using streaming or online algorithms which heuristics are sadly limited as we've seen below.

As a consequence, it's useful to have two different abstract implementations: one of them from functioning off a posit that it works on total input, and another processing with the idea that it observes new inputs over the wire as they come.

In conclusion, the news of batch processing's demise are overrated: batch processing is still relevant, at least to provide a baseline of performance for a streaming problem. Any responsible engineer should have a good idea of the performance of a batch algorithm operating "in hindsight" on the same input as their application.

- if there is a known competitive ratio for the streaming algorithm at hand, and the resulting performance is acceptable, then running just the stream processing may be enough,
- if there is no known competitive ratio between the implemented Stream processing and a batch version, then running a batch computation on a regular basis is a valuable benchmark to hold one's application to.

[1]We invite the reader to consult the original article if you want to know more about the link with the CAP theorem (also called Brewer's theorem). The idea was that it concentrated some limitations fundamental to distributed computing described by the theorem, to a limited part of the data processing system. In our case, we will focus on the practical implications of that constraint.

Chapter 5. Apache Spark

include::Ch01-06-BigDataParts.asciidoc[] // === A Data Processing engine include::Ch01-09-RealTimeProcessingPartIntro.asciidoc[] // === A Real Time processing engine include::Ch01-11-Spark-MemCache.asciidoc[] //=== Spark’s Memory Usage include::Ch01-14-StreamingByTime.asciidoc[] // === A strong Streaming characteristic

include::Ch01-15-MinimalLatency.asciidoc[] // === A minimal delay include::Ch01-16-ThroughputOriented.asciidoc[] // === Throughput-oriented tasks

include::Ch01-07-SparkAPI-MoreMapred.asciidoc[] // === A polyglot API include::Ch01-18-productive-deployment.asciidoc[] // === Fast implementation of data analysis include::Ch01-19-Conclusion.asciidoc[] // === To learn more about Spark

Chapter 6. Spark in Practice: How to leverage Apache's Spark rock-solid fault tolerance for stream processing

In most cases, streaming job is a long-running job. By definition, streams of data observed and computed on over time tend to lead to jobs that run for longer, accumulate intermediary results that are harder to reproduce, and therefore where the cost of failure is larger. We therefore want to look at reducing the chance of failure of a streaming job by leveraging all the tools that the Apache Spark platform gives us, starting with a deep understanding of the flow of data through the Apache Spark system in the lifetime of a job.

Thinking about reliability: Closures

We can start with the notion of a *closure*: the pair formed by a function and its arguments, along with the variable definitions necessary for its evaluation. A **closure** is conceptually what Spark is meant to package (serialize) and distribute across a cluster of executors. We've seen in "Micro-batching and one-element-at-a time" how this discipline of moving a user-provided computation, in the form of a closure, to the data for execution is sometimes called "function-passing style".

The arguments of the function are, in the context of big data and distributed computing, URLs and pointers to big data repositories that are hard to move across the network. But since we have access to separate fallible computation units, usually separated by a computer network, we want to move functions to the date. Making sure that closures reach their execution environment safely and reliably, and report the result of their execution faithfully is actually a non-trivial task. Hence, Spark has several measures aimed at helping with fault tolerance, and we'll address them in this chapter.

DISTRIBUTED COLLECTIONS IN SPARK, OR HOW SPARK TRACKS PROGRESS

Apache Spark depends builds its data representations on RDDs, or Resilient Distributed Datasets. This data structure is the foundational data structure in Spark, and offers strong fault tolerance guarantees. This RDD is represented in partitions, which are units stored on individual nodes, and tracked centrally by Spark to form a unique data structure from the user's point of view.

As such, RDDs have enjoyed a wide popularity in the Spark world, and we give them a brief introduction in <<#rdd-primer>>. Many other streaming abstractions, including `DataFrame`s and `DataSet`s are built using facilities created for `RDD`s, and more importantly, they reuse the same fault tolerance facilities.

We mention those structures here to present the idea that Spark tracks progress of the user's computation through modifications of the data. Indeed, knowing how far along we are in what the user wants to do through inspecting the control flow of his program (including loops and potential recursive calls) can be a daunting and error-prone task. It is much more reliable to define types of distributed data collections, and let the user create one from another, or from other data sources.

As the system tracks the ordered creation of these distributed data collections, it tracks the work done, and that left to accomplish, in the way we'll outline below.

To understand at what level fault tolerance operates in Spark Streaming, it's useful to go through a reminder of the nomenclature of some basic concepts in Spark. The user provides a program that ends up being divided into chunks, and executed on various machines. Let's run down those steps, which define the vocabulary of the Spark runtime:

User Program

: the user application in Spark Streaming is composed of user-specified *function calls* operating on a resilient data structure (`RDD`, `DStream`, streaming `DataSet` ...), categorized as *actions* and *transformations*.

Transformed User Program

: the user program may undergo adjustments that modify some of the specified calls to make them simpler, the most approachable and understandable of which is map-fusion. [1]

Stages

: the user's operations are then grouped into *stages*, which boundary separates user operations into steps that have to be executed separately. For example, operations that require a shuffle of data across multiple nodes, such as a join between the results of two distinct upstream operations, mark a distinct stage. Stages in Apache Spark are the unit of sequencing: they are executed one after the other. At most one interdependent stage

Jobs

: Once these *stages* are defined, what internal actions Spark should take is clear. Indeed, at this stage, a set of interdependent *jobs* is defined. And jobs, precisely, are the vocabulary for a unit of scheduling. They describe the work at hand from the point of view of an entire Spark cluster, whether it's waiting in a queue or currently being run across many machines.

Tasks

: Jobs can then, depending on where their source data is on the cluster, be cut into *tasks*, crossing the conceptual boundary between distributed and single-machine computing : a task is a unit of local computation, the name for the local, executor-bound part of a job.

Spark aims to make sure that all of these steps are safe from harm, and to recover quickly in the case of any incident occurring in any stage of this process. This concern is reflected in fault-tolerance facilities that are structured by the notions above: restart and checkpointing operations that occur at the task, job, stage, or program level.

Spark's Reliability primitives

- First, Spark saves the user-provided data flow of the application in a directed acyclic graph (DAG), a data structure that lets Spark represent the inter-dependency between some steps of the computation. That DAG is stored on the driver machine.
- Second, Spark offers the possibility to store the data of RDDs at a configurable storage level, which lets users persist the results of an intermediate stage of computation. However, since the user operations in Spark are lazy and grouped into conglomerates of operations, there is

no particular reason for Spark to store the result of every function call. However, when an intermediate state in computation needs to be materialized and passed on — such as a shuffle boundary — Spark will use the replication instructions inherent in the storage level setting.

- Third, while by default, during a shuffle operation, Spark is moving data around in a pull mode. That is, the Spark executor first writes its own map outputs locally to disk, and then acts as the server for those files when other executors attempt to fetch them.

Spark's Fault Tolerance Guarantees

In this section, we'll see Spark fault tolerance guarantees organized by "increasing blast radius", from the more modest to the larger failure. We'll investigate: - how Spark mitigates Task failure through restarts, - how Spark mitigates stage failure through the shuffle service, - how Spark mitigates the disparition of the *orchestrator* of the user program, through driver restarts, - and how we run this "driver" reliably through cluster modes in the first place, - we'll also see how Spaek addresses the long-range dependencies of Streaming jobs through checkpointing, - and we'll see how to address the most major failure, that of the master node, through a hot-swappable replacement.

Once through this section, we will have a clear mental picture of the guarantees Spark affords us at runtime, letting us understand the failure scenarios that a well-configured Spark job can deal with. We will the be able to move on to the more specific concerns like receiver reliability, zero data loss configurations, and cluster-specific setups in the next section.

Task failure

If the input data of the task was stored, through a call to `cache()` or `persist()` and if the chosen storage level implies a replication of data (look out for a storage level which setting ends in `_2`, such as `MEMORY_ONLY_SER_2`), the task does not need to have its input recomputed, as a copy of it exists in complete form on another machine of the cluster. We can then use this input to re-start the task.

Level	Uses Disk	Uses Memory	Uses Off-heap storage	Object (deserialized)	# of replicated copies
`NONE`					1
`DISK_ONLY`	X				1
`DISK_ONLY_2`	X				2
`MEMORY_ONLY`		X		X	1
`MEMORY_ONLY_2`		X		X	2

Level	Uses Disk	Uses Memory	Uses Off-heap storage	Object (deserialized)	# of replicated copies
MEMORY_ONLY_SER		X			1
MEMORY_ONLY_SER_2		X			2
MEMORY_AND_DISK	X	X		X	1
MEMORY_AND_DISK_2	X	X		X	2
MEMORY_AND_DISK_SER	X	X			1
MEMORY_AND_DISK_SER_2	X	X			2
OFF_HEAP			X		1

If however there was no persistence, or if the storage level does not guarantee the existence of a copy of the task's input data, then the Spark driver will have to consult the DAG that stores the user-specified computation, to figure out which segments of the job need to be recomputed.

Consequently, without enough precautions to save either on the caching or on the storage level, the failure of a task can trigger the re-computation of several others, up to a stage boundary.

Indeed, stage boundaries imply a shuffle, and a shuffle implies that intermediate data will somehow be materialized: as we discussed, the shuffle transforms executors into data servers that can provide the data to any other executor serving as a destination.

As a consequence, these executors have a copy of the map operations that led up to the shuffle. Hence, executors that participated in a shuffle have a copy of the map operations that led up to it. But that's a life saver if you have a dying downstream executor, able to rely in the upstream servers of the shuffle (which serve the output of the map-like operation). What if it's the contrary: you need to face the crash of one of the upstream executors?

The External shuffle service

This brings us to the second facility that allows Spark to resist arbitrary execution failures: the *shuffle service*. We've seen that task failure (possibly due to executor crash) was the most frequent incident happening on a cluster, and hence the most important unusual event to prevent.

When this failure occurs, it always means some roll-back of the data, but a shuffle operation, by definition, depends on all of the prior executors involved in the step that precedes it.

As a consequence, Spark 1.3 has introduced the *shuffle service*, which lets users work on map data that is persisted and distributed through the cluster with a good locality, but more importantly through a server that is not a Spark task. It's an external file exchange service in Java, which has no dependency on Spark, and is made to be a much-longer running service that a Spark executor. This

additional service attaches as a separate process in all cluster modes of Spark, and simply offers a data file exchange for executors to transmit data reliably, right before a shuffle. It is highly optimized through the use of a netty back-end, to allow a very low overhead in transmitting data. This way, an executor can shut down after the execution of its map task, as soon as the shuffle service has a copy of its data. And because data transfers are faster, this transfer time is also highly reduced, reducing the vulnerable time in which any executor could face an issue.

Driver failure

Having seen how Spark recovers from the failure of a particular task, we can now look at the facilities Spark offers to recover from the failure of the driver program. The driver in Spark has an essential role: it is the depository of the Block manager, which knows where each block of data resides in the cluster. It is also the place where the DAG — the graph of Spark "actions" that represent the user's program — lives.

Finally, it is where the scheduling state of the job, its metadata and logs resides. Hence, if the driver is lost, a Spark cluster as a whole may well have lost which stage it has reached in computation, what the computation actually consists of, and where the data that serves it can be found, in one fell swoop.

Cluster-mode deployment

Spark has implemented what's called the cluster deployment mode, which allows the driver program to be hosted on the cluster, as opposed to the user's computer.

The deployment mode is one of two options: in client mode, the driver is launched in the same process as the client that submits the application. In cluster mode, however, the driver is launched from one of the Worker processes inside the cluster, and the client process exits as soon as it fulfills its responsibility of submitting the application without waiting for the application to finish.

This, in sum, allows Spark to operate an automatic driver restart, so that the user can start a job in a "fire and forget fashion," starting the job and the closing their laptop to catch the next train. Every cluster mode of Spark offers a Web UI that will let the user access the log of their application. Another advantage is that driver failure does not mark the end of the job, as the driver process will be relaunched by the cluster manager. But this only allows recovery from scratch, as the temporary state of the computation — previously stored in the driver machine — may have been lost.

Checkpointing

To avoid this, Spark offers the option of check-pointing, that is recording periodically a snapshot of the application's state to disk. The setting of the `sparkContext.setCheckpointDirectory()` option should point to reliable storage (HDFS) because having the driver try to reconstruct the state of intermediate RDDs from its local file system makes no sense: those intermediate RDDs are being created on the executors of the cluster and should as such not require any interaction with the driver for backing them up.

We will come back to the subject of checkpointing later. In the meantime, there is still one component of any Spark cluster whose potential failure we have not yet addressed: the master node.

A hot-swappable master through Zookeeper

In Spark, failure of the master node is addressed through a recovery mode, which consists in having a high availability piece of software that maintains a live hot-swappable copy of the state of the master node, making sure it has the same interactions with the rest of the cluster as the "main" machine does. This high-availability manager, in practice, turns out to be Apache Zookeeper.

This tour of Spark-core's fault tolerance and high-availability modes should have given us an idea of the main primitives and facilities offered by Spark and of their defaults. Note that none of this is so far specific to Spark Streaming, but that all these lessons apply to Spark Streaming in that they are required to deliver long-running, fault-tolerant and yet performant application.

Note also that these facilities reflect different concerns in the frequency of faults for a particular cluster: while facilities as a spare master node kept up-to-date through Zookeeper are really about avoiding a single point of failure in the design of a Spark distributed application, the Spark shuffle service is here to avoid any problems with the end of a long string of map-like applications ending in failure because the executor serving these results is about to die — a much more frequent occurrence. The first is about dealing with every possible condition, the second is more about ensuring smooth performance and efficient recovery.

Fault-tolerance in Spark Streaming: the context of the Receiver model

The previous reminders have set up a rich context for the taxonomy of running parts in Spark. We can thus get an idea of how the details of running a Spark application translate into fault-tolerance aspects. Let's start analyzing this with the Receiver model — the model that has the most impact on fault-tolerance.

The Receiver model, as we've mentioned before, is the ability for Spark Streaming to run data ingestion as a job. The data flow of a Spark application in this case looks like the following figure:

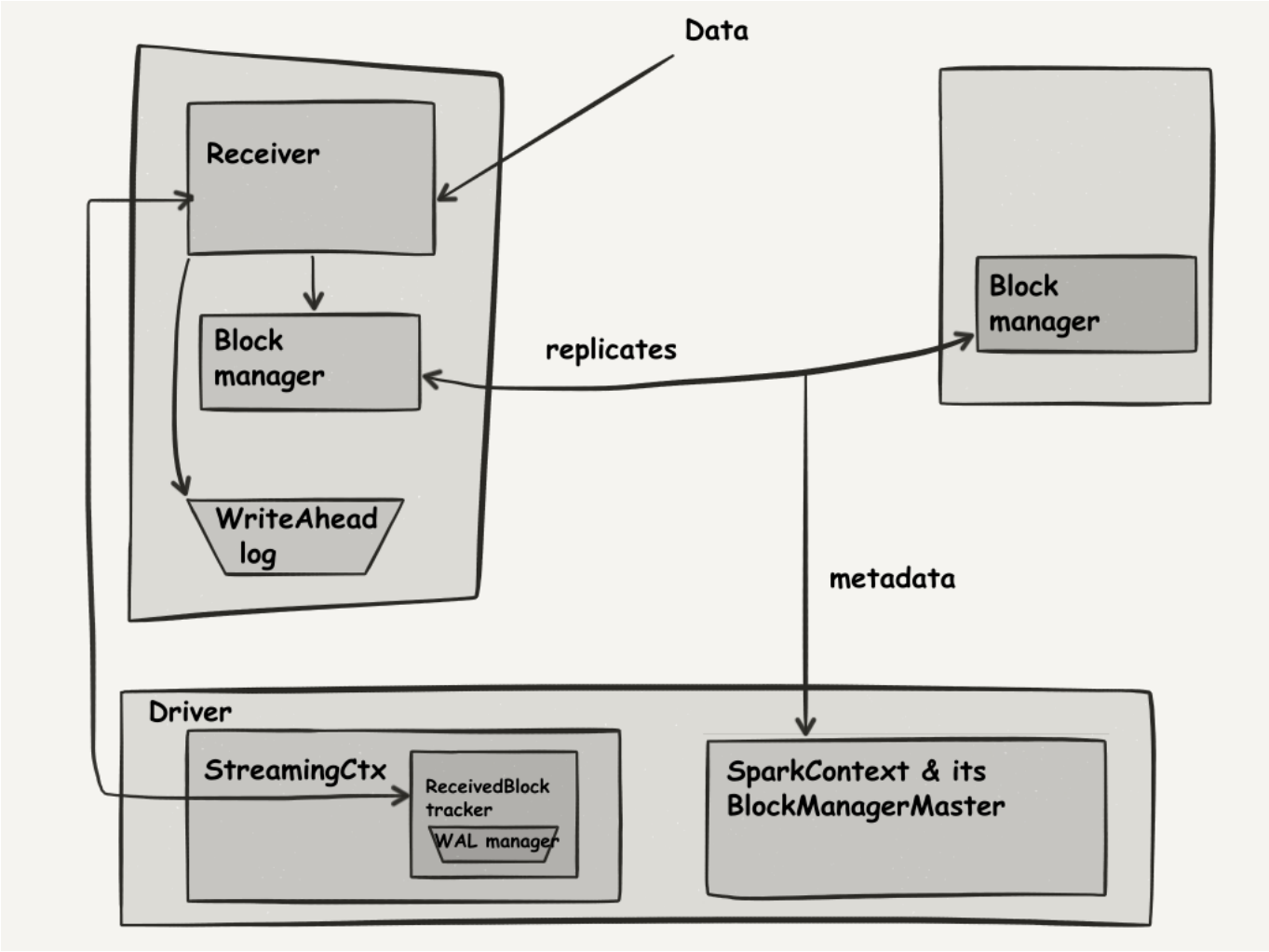


Figure 6-1. The Data flow of Spark's Receiver

In this figure, we can see that data ingestion occurs as a job, that gets translated into a single task on one executor. This task deals with connecting to a data source, and actually initiates the data transfer. It is managed from the Spark Context, a bookkeeping object which live inside the driver machine.

On each block interval tick (as measured on the executor that is running the Receiver), this machine groups the data received for the previous block interval into a block. The block is then registered with the Block Manager, also present in the bookkeeping of the Spark Context, on the driver. This can initiate replication of the data that this block represents, to ensure that the source data in Spark Streaming is replicated the number of times indicated by the storage level.

On each batch interval tick (as measured on the driver), the driver groups the data received for the previous batch interval, which has been replicated the correct number of times, into an RDD. This RDD gets registered with the JobScheduler. This will initiate the scheduling of a job on said RDD — in fact, the whole point of Spark Streaming's micro-batching model consists in repeatedly scheduling the user-defined program on successive batched RDDs of data.

SYNCHRONIZING EXECUTOR CLOCKS

The machines in a Spark cluster may have different clocks, and may be impacted by clock drift between two machines. To ensure the correct processing of data at reliable and even intervals, there should be a synchronization of the clocks of the cluster's machines, as well as the one on the driver (particularly in client deployment mode). It is highly advised to run a Networked Time Protocol (NTP) client on all cluster machines, before and during the execution of a Spark applicaiton.

RECEIVER AND LOCALITY

One other aspect of note in this data flow diagram is that the first copy of any piece of data on the cluster arrives in a single machine, often dubbed "the Receiver" — though the Receiver can, as a full-fledged Spark executor, run any other task the scheduler submits to it. As a consequence, Spark users will be keen to chose storage levels that imply a replication of the data received through the cluster. In fact, even with a storage level which implies replication (e.g. `MEMORY_ONLY_SER_2`), the probability that the Receiver will be assigned most of the tasks in an RDD is still inordinately high.

When accessing a single data source, it has become a standard best practice to not create a single `DStream` for this data source. If it is at all possible to serve this data in parallel, users concerned with locality often create several Receivers that share a connection pool with the data source. Spark Streaming will in turn create several Receivers, which will share the load of ingesting this data. The`union` function in the `DStream` API then lets Spark consider these several DStreams be one single merged stream, allowing the user to proceed as if this was a single entry point all along.

THE RECEIVER MODEL IN APACHE SPARK STREAMING TODAY

Today the Receiver mode of data flow has fallen out of favor, as new "direct" modes in which the data is read directly from the executors that will process it, in chunks. Those modes of data ingestion put the responsibility of fault tolerance on the external data source, that is labeled **reliable** if it is able to index data elements and replay them on demand. The receiver model remains available to Spark Streaming because it is the model which puts the most responsibility for data ingestion and division in parallel chunks upon Spark — as opposed to the *direct* or *reliable source* modes where splitting data in distributed chunks is the responsability of another framework. As such, it is less demanding of external systems, and it's a useful "fallback solution", if one needs to read data from an exotic data source, or one that does not integrate natively with distributed processing. We now know how to integrate to those data sources, and what the impact on fault tolerance is.

Spark Streaming's Zero Data Loss guarantees

To solve the problem of data loss, Spark introduced a high-availability option called the WriteAhead log. Spark can optionally write all the data it receives — as it's receiving it and in a blocking fashion — to a log file stored in a safe location such as HDFS[1]. Since Spark Streaming takes this Write-Ahead Log into account during the recovery of the driver, this means that a surviving log guarantees that no data has been lost during a Receiver failure. This Zero-data loss result is obtained, however, at the very high performace cost of writing to HDFS in a blocking fashion. As a consequence, few users really need to implement it. But this setting can be particularly important for some applications where the loss of even a little data could be catastrophic, such as anomaly detection.

NOTE

The assumption, made several times in this book, that a distributed file system is a safe place to write data to requires the probability of failure of HDFS data nodes to be independent from that of failure of Spark executors. The reader should beware that Spark and HDFS nodes are not always independent, for example in some cases where HDFS nodes and Spark nodes are colocated on the same hardware. HDFS by default replicates data twice, choosing replication location with a logic completely independent from Spark. This means that in most cases, and with a big enough cluster, it's an acceptable risk to deploy Spark executor nodes co-located with HDFS data nodes: the probability of a Spark in-memory replication location falling on the same node as an HDFS replication location is 1/(n^2) for a cluster with n nodes. However the Spark and HDFS master, which failure is more consequential, should host no other functionality. And one should also note that an entire Spark streaming cluster showing a pattern of cascading failure, such as the later cluster running out of memory, may bring down your HDFS cluster as well, encurring the loss of data recently sent to the HDFS cluster.

CAUTION

The notion that a RDD of intermediate results is not available unless the user has specified whether it should be kept in memory is in fact an oversimplification. Spark does try not to delete data that it could need, so that RDDs are in fact cached. Until then, a particular RDD is kept in memory is dictated by `spark.cleaner.ttl`. However, since it is often difficult to make a link between the expected runtime of a task leading an RDD in the operation DAG to the next, and how much cache lifetime allows the Spark cluster's memory not to blow up is difficult to establish. Consequently, we would advise users that are serious about fault tolerance to pay attention to strategically caching the most difficult to recompute results of their application.

We can now consider a summary of the fault-tolerance measures we have discussed, informed by the data flow diagram we have described above.

Name	Settings to activate	Goal
The Write-Ahead log	`spark.streaming.receiver.writeAheadLog.enable`	Zero data loss in data ingestion
The Shuffle service	`spark.shuffle.service.enabled`, `spark.shuffle.service.port`	Efficient exchange of shuffle files, recovery from late mapper failure

Name	Settings to activate	Goal
High availability of the master (with Zookeeper)	`spark.deploy.recoveryMode`, `spark.deploy.recoveryDirectory`, `spark.deploy.zookeeper.url`	Avoiding a single point of failure at master
Checkpointing	call `sparkContext.setCheckpointDir(checkpointDirectory: String)`, create context w/`StreamingContext.getOrCreate(checkPointDirectory)`	Limiting RDD Lineage
Persist & Storage Level	call `myRDD.persist(StorageLevel.CHOOSE_A_STORAGE_LEVEL)`	Avoiding prohibitively long recovery
Cluster deployment mode	pass `--depoy-mode cluster` to spark submit, consider `--supervise`	Recovery from driver failure

Note that the other measures we have talked about — such as separating data ingestion in different streams to finalize them — may also relate to distinct concepts, such as performance. We have tried to be as exhaustive as possible while building this table.

Cluster managers and driver restart

Running Spark can be done in one of four cluster managers. Three of these cluster managers offer the possibility to run Spark in a "real" distributed Spark context. The other is the local mode, which emulates the behavior of executors in the distributed context using threading. We have mentioned them in Chapter 3. Now we would like to compare their capabilities to help you choose one for your purposes.

Comparing cluster managers

Preparing robust Spark deployment requires a bit of understanding of the different cluster managers, even though they are quite similar in their features, from the point of view of the Spark user. In particular, it requires understanding what can — from this very point of view — affect the running of a Spark application.

A comparison from the point of view of fault tolerance is done in the following table:

Cluster mode	Driver resiliency	Master resiliency	Receiver resiliency
Standalone	`--deploy-mode cluster --supervise`	set `spark.deploy.recoveryMode`, `spark.deploy.zookeeper.url` and `spark.deploy.zookeeper.dir`	`spark.streaming.receiver.writeAheadLog=true` and activate checkpointing
YARN	`--master yarn-cluster` or `--master yarn --deploy-mode cluster`	set `spark.deploy.recoveryMode`, `spark.deploy.zookeeper.url` and `spark.deploy.zookeeper.dir`	`spark.streaming.receiver.writeAheadLog=true` and activate checkpointing
Mesos	post Spark 1.4.0, `--deploy-mode cluster --supervise`	set `spark.deploy.recoveryMode`, `spark.deploy.zookeeper.url` and `spark.deploy.zookeeper.dir`	`spark.streaming.receiver.writeAheadLog=true` and activate checkpointing

NOTE

The default storage level for elements of a `DStream` is MEMORY_ONLY_SER, except for input DStreams which are MEMORY_AND_DISK_SER_2

More settings to ensure efficient failure recovery and decent parallelism in Data ingestion can be seen in the table below:

Cluster mode	Efficient executor failure recovery for a long lineage	Executor failure recovery at the shuffle	Limited parallelism of the receiver
Standalone	`sparkContext.checkpoint("hdfs://myDirectory")` and persist the `DStream` strategically	launch the shuffle service on each executor	Launch several receivers and union them,
YARN	`sparkContext.checkpoint("hdfs://myDirectory")` and persist the `DStream` strategically	set the shuffle service in the YARN configuration	Launch several receivers and union them,
Mesos	`sparkContext.checkpoint("hdfs://myDirectory")` and persist the `DStream` strategically	configure and launch the shuffle service on each executor	Launch several receivers and union them,

Moreover, an interesting recent addition to Spark can be seen as a fault tolerance addition: the *dynamic allocation mode*. In fact, two runtime scenarios can affect the quality of a Spark Streaming deployment from the moment of the assignation of executors:

- if the long-running Spark Streaming application *starts losing nodes* — for example if the application faces serious trouble in a multi-tenant cluster, it may become progressively unstable. Indeed, we have already mentioned that the executors Spark Streaming has available allow it to keep a job stable only if it keeps the execution time for each given RDD below the batch interval, on average. A consequence of that is that jobs that lose executors, since those executors never reconnect to the application (especially in cluster modes that assign a set of executors once and for all, such as YARN and Mesos) often go even more unstable because they start overwhelming the job completion capabilities — and therefore soon, the memory — of their executors.
- if the long-running Spark execution *has a bad executor-to-physical-node distribution*. This is pretty much limited to the Mesos coarse deployment mode, which has strong constraints on having a fixed set of executors negotiated at the launch of the application. Nonetheless, if for example all the Receivers get concentrated on a single physical machine, these containers end up competing for network resources.

Making the set of executors used by the applicaiton dynamic has great benefits in terms of failure recovery as well as redistribution of resources. As of Spark 1.5, all cluster modes support dynamic allocation, a mode that allows Spark to incorporate new executors being made available to the

Spark cluster manager, when the application requires it and they are assigned by the cluster resource manager. As of Spark 2.0, dynamic allocation works with Spark Streaming, as we'll see below.

In this mode, re-provisioned machines of the cluster, that could have suffered the consequences of say a hardware failure, can again be part of your Spark application. They can also register spontaneously in case your application needs resources, and somewhat alleviate the performance limitations witnessed when a bad initial assignation occurs on the moment where the application is launched.

Spark Streaming's dynamic allocation works slightly differently from the dynamic allocation of regular batch Spark jobs. Spark Streaming's dynamic allocation works based on the ratio of batch job duration to batch interval. Since the average batch job duration is how fast data moves "out" of the system, and the batch interval is how frequently jobs move "in" the system, then the ratio of batch job duration over batch interval is how "late" the system is (if > 1) or "early" (if < 1). Spark compares this ratio at regular intervals with five configuration parameters:

Parameter	Default Value	Meaning
`spark.streaming.dynamicAllocation.scalingInterval`	60	the length of the interval
`spark.streaming.dynamicAllocation.scalingUpRatio`	0.9	the threshold ratio above which Spark will try to add executors to this Spark Streaming configuration
`spark.streaming.dynamicAllocation.scalingDownRatio`	0.3	the threshold ratio below which Spark will try to shut down executors that are not carrying a Receiver
`spark.streaming.dynamicAllocation.minExecutors`	0	the minimum number of non-Receiver executors Spark Streaming will always maintain
`spark.streaming.dynamicAllocation.maxExecutors`	**infinity**	the maximum number of non-Receiver executors Spark Streaming will

Parameter	Default Value	Meaning
		never try to go above of.

Note that those parameters, as their name indicates, are specific to Streaming.

NOTE

Mesos is a cluster manager that has two distinct modes for launching an application: the (default) coarse mode, negotiates a number of executors for the duration of the application — in effect creating a mini cluster-within-cluster. The other mode is the (now deprecated) fine-grained mode, in which resources are requested on a per-task basis. The advantage of this second mode is that it is less dependent on a fixed assignation of executors, since this is being renegotiated on each job, but it has a longer overhead when launching each task. The arrival of dynamic allocation in Spark 1.5 has in effect made the fine-grained mode obsolete, in that allocation works much more efficiently and quickly with this new mode, on top of a minimum allocation of a "base" number of executors in coarse mode. Our discussion of dynamic allocation mode therefore describes the "modern way" of setting up a Mesos cluster that adapts its resources based on the incoming jobs.

Bibliography

- [Gibbons2004] J. Gibbons. *An unbounded spigot algorithm for the digits of π.* American Mathematical Monthly, 113(4):318-328, 2006. URL
- [Miller2016] H. Miller; P. Haller; N. Müller; J. Boullier *Function Passing: A Model for Typed, Distributed Functional Programming.* 2016. ACM SIGPLAN Conference on Systems, Programming, Languages and Applications: Software for Humanity, Onward! Research Papers, Amsterdam, Netherlands, November 2-4, 2016. p. 82-97. DOI : 10.1145/2986012.2986014.
- [Greenberg2015] David Greenberg. *Building Applications on Mesos : Leveraging Resilient, Scalable, and Distributed Systems.* O'Reilly Media. 2015 ISBN 1-4919-2652-X
- [Valiant1990] Valiant, L.G. *Bulk-synchronous parallel computers*, Communications of the ACM 33:8, August 1990 *http://web.mit.edu/6.976/www/handout/valiant2.pdf*
- [Vavilapalli2013] Vavilapalli et al. *Apache Hadoop YARN: yet another resource negotiator.*SOCC'13, ACM, NY, 2013 doi:10.1145/2523616.2523633
- [Venkataraman2016] Shivaram Venkataraman; Aurojit Panda; Kay Ousterhout; Ali Ghodsi; Michael J. Franklin; Benjamin Recht; Ion Stoica. *Drizzle: Fast and Adaptable Stream Processing at Scale* Tech Report, UC Berkeley, 2016 URL
- [White2010] White, T. *Hadoop: The Definitive Guide* O'Reilly Media. 2010. ISBN-13 9781449396893
- [Dosa2007] Gÿorgy Dósa, The Tight Bound of First fit Decreasing Bin-Packing Algorithm Is FFD(I)≤(11/9)OPT(I)+6/9 in *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, Springer-Verlag, 2007
- [Kleppmann2015] Martin Kleppmann, *A critique of the CAP theorem* Technical Report, arXiv:1509.05393, Sep 2015 URL

- [Kreps2014] Jay Kreps, *Questioning the Lambda Architecture.* O'Reilly Radar July 2, 2014. URL
- [Marz2011] Nathan Marz. *How to beat the CAP theorem.* Blog post, October 13, 2011. URL
- [Marz2015] Nathan Marz and Jamez Warren. *Big Data: Principles and best practices of scalable realtime data systems.* Manning Publications, May 2015. ISBN 1-6172-9034-3
- [Sharp2007] Alexa Megan Sharp. *Incremental algorithms: solving problems in a changing world.*PhD thesis, Cornell University, Aug., 2007 URL
- [Shapira2014] Gwen Shapira. *Building The Lambda Architecture with Spark Streaming.* Cloudera Engineering Blog. Aug 29, 2014 URL
- [Chambers2018] Chambers, B.; Zaharia, M., *Spark : The Definitive Guide*, O'Reilly Media, February 2018, ISBN 1491912219. URL
- [Dean2004] Jeff Dean and Sanjay Ghemawat, *MapReduce :Simplified Data Processing on Large Clusters .* OSDI'04 San Francisco, CA, December, 2004. URL
- [Halevy2009] Alon Halevy, Peter Norvig, and Fernando Pereira. *The Unreasonable Effectiveness of Data*. IEEE Intelligent Systems, March/April 2009 URL
- [Lyon2013] Brad F Lyon. *Musings on the Motivations for Map Reduce*. URL
- [Karau2015] Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia. *Learning Spark : Lightning-Fast Big Data Analysis*. O'Reilly Media. 2015. ISBN 1-4493-5862-4
- [Lin2010] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce* Morgan & ClayPool. 2010. URL
- [Armbrust2018] Armbrust, M.; Das, T.; Torres, J.; Yavuz, B; Zhu, S.; Xin, R.; Ghodsi, A.; Stoica, I.; Zaharia, M; *Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark* May 27, 2018. URL
- [Bhartia2016] Bhartia, R. *Optimize Spark-Streaming to Efficiently Process Amazon Kinesis Streams*, AWS Big Data Blog, February 26, 2016. URL
- [Chintapalli2015] Chintapalli, S; Dagit, D; Evans, B; Farivar, R; Graves, T; Holderbaugh, M; Liu, Z; Musbaum, K; Patil, K; Peng, B; Poulosky, P. *Benchmarking Streaming Computation Engines at Yahoo!* Yahoo! Engineering. December 18, 2015. URL
- [Das2013] Tathagata Das. *Deep Dive With Spark Streaming*. Spark meetup, June 17, 2013. URL
- [Das2014] Tathagata Das; Yuan Zhong. *Adaptive Stream Processing using Dynamic Batch Sizing*2014 ACM Symposium on Cloud Computing.November 3-5,2014. URL
- [Das2015] Tathagata Das. *Improved Fault Tolerance And Zero Data Loss in Spark Streaming.*Databricks Engineering Blog. January 15, 2015. URL
- [Doley1987] Doley, D; Dwork, C; StockMeyer, L. *On the minimal synchronism needed for distributed consensus*. Journal of the ACM. 34 (1):77-97; 1987; URL
- [Dunner2016] Dünner, C; Parnell, T; Atasu, K; Sifalakis, M; Pozidis, H. *High-Performance Distributed Machine Learning using Apache SPARK* December 2016. URL
- [Fischer1985] Fischer, M. J.; Lynch, N. A.; Paterson, M. S.*Impossibility of distributed consensus with one faulty process*. Journal of the ACM. 32 (2): 374–382. 1985. URL
- [Kestelyn2015] Kestelyn, J. *Exactly-once Spark Streaming from Apache Kafka* Cloudera Engineering Blog. March 16, 2015. URL
- [Koeninger2015] Cody Koeninger, Davies Liu and Tathagata Das. *Improvements to Kafka integration of Spark Streaming.* Databricks Engineering Blog. March 30, 2015 URL
- [Lamport1998] Lamport, Leslie. *The Part-Time Parliament* ACM Transactions on Computer Systems. 16 (2): 133–169. URL
- [Mas2014] Gérard Maas. *Tuning Spark Streaming for Throughput.* Virdata Engineering Blog. December 22, 2014. URL

- [Venkat2015] Venkat, B; Padmanabhan, P; Arokiasamy, A; Uppalapati, R. *Can Spark Streaming survive Chaos Monkey?* The Netflix Tech Blog. March 11, 2015. URL
- [Nasir2016] Nasir, M.A.U. *Fault Tolerance for Stream Processing Engines*. arXiv preprint arXiv:1605.00928, May 2016 URL
- [Zaharia2012] Matei Zaharia, Tathagata Das et al. *Discretized Streams: A Fault-Tolerant Model for Scalable Stream Processing* UCB/EECS-2012-259 URL

[1]The process by which `l.map(foo).map(bar)` is changed into `l.map((x) => bar(foo(x)))`

Part II. Structured Streaming

In this part, we are going to learn about Structured Streaming.

We start our journey in Structured Streaming by exploring a practical example that should help us build our intuition for the model. From there, we examine the API and get into the details of the different aspects of stream processing:

- Consuming data using sources
- Buildingdata processing logic using the rich Streaming Dataframe/Dataset API
- Understanding and work with event time.
- Dealing with state in streaming applications.
- Learning about arbitrary stateful transformations.
- Writing the results to other systems using sinks.

Before closing, we provide an overview of the operational aspects of Structured Streaming.

Finally, we account for the current developments in this exciting new streaming API and provide insights in current experimental areas, like machine learning applications and near-real-time data processing with continuous streaming, and provide insights in the current roadmap.

Chapter 7. Introducing Structured Streaming

In data-intensive enterprises, we find many large datasets: log files from internet-facing servers, tables of shopping behavior, noSQL databases with sensor data just to name a few examples. All these datasets share the same fundamental lifecycle: They started empty at some point in time and were progressively filled by arriving data points that were directed to some form of secondary storage. This process of data arrival is nothing more that a *data stream* being materialized onto secundary storage. We can then apply our favorite analytics tools on those datasets *at rest*, using techniques known as *batch processing* because they take large chunks of data at once and usually take considerable amounts of time to complete, ranging from minutes to days.

The `Dataset` abstraction in *Spark SQL* is one such way of analysing data at rest. It is particularly useful for data that is *structured* in nature, that is, it follows a certain schema. `Dataset`s in Spark combine the expressivity of a SQL-like API with type-safe collection combinators that are reminiscent of the Scala collections and the `RDD` programming model. At the same time, the `Dataframe` API, which is in nature similar to Python Pandas and R Dataframes, widens the audience of Spark users beyond the initial core of Data Engineers used to develop in a functional paradigm. This higher level of abstraction intends to support modern big data and data science practices by enabling a wider range of professionals to jump onto the Big Data analytics train using a familiar API.

What if, instead of having to wait for the data to *settle down*, we could apply the same `Dataset`concepts to the data in its original stream form?

The Structured Streaming model is an extension of the `Dataset` SQL-oriented model to handle data on the move:

- The data arrives from a *source* stream and is assumed to have a defined schema.
- The stream of events can be seen as rows that are appended to an unbounded table.
- To obtain results from the stream, we express our computation as queries over that table.
- By continuosly applying the same query to the updating table we create an output stream of processed events.
- The resulting events are offered to an output *sink*.
- The *sink* could be a storage system, another streaming backend or an application ready to consume the processed data.

In this model, our theoretically **unbounded** table must be implemented in a physical system with defined resource constraints. Therefore, the implementation of the model requires certain considerations and restrictions to deal with a potentially infinite data inflow. To address these challenges, Structured Streaming introduces additional concepts such as support for event time, *watermarking*, and different output modes that determine how long past data is actually stored for.

Conceptually, the Structured Streaming model blurs the line between batch and streaming processing, removing a lot of the burden of reasoning about fast moving data.

First Steps with Structured Streaming

In the introduction, we learnt about the high level concepts comprising Structured Streaming such as sources, sinks, and queries. We are now going to explore the Structured Streaming from a practical perspective, using a simplified web log analytics use case as example.

But before we start delving into our first streaming application, we are going to see how classical batch analysis in Apache Spark can be applied to the same use case. This has two main goals:

- First, most, if not all, streaming data analytics start by studying a static data sample. It is far easier to start a study with a file of data, gain intuition on how the data looks like, what kind of patterns it shows and define the process that we require to extract the intended knowledge from that data. Typically, it's only after we have defined and tested our data analytics job that we proceed to transform it into a streaming process that can apply our process to data on the move.
- Second, from a practical perspective, we can appreciate how Apache Spark simplifies many aspects of transitioning from a batch exploration to a streaming application through the use of uniform APIs for both batch and streaming analytics.

This exploration will allow us to compare and contrast the batch and streaming APIs in Spark and show us the necessary steps to move from one to the other.

NOTE

For this example, we will use Apache Web Server logs from the public 1995 NASA Apache web logs, originally from *http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html*

For the purpose of our exercise, the original log file has been split in daily files and each log line has been formatted as JSON. The compressed file can be downloaded from the online book resources at:*https://github.com/StreamProcessingWithSpark/datasets/blob/master/NASA-weblogs/nasa_dataset_july_1995.tgz* Download this dataset and place on a folder on your computer.

Batch Analytics

Given that we are working with archive log files, we have access to all the data at once. Before we start building our streaming application, let's take a brief intermezzo to have a look at how a classical batch analytics job would look like.

First, we load the log files, encoded as JSON, from the directory where we unpacked them:

```
// This is the location of the unpackaged files. Update accordingly
val logsDirectory = ???
val rawLogs = sparkSession.read.json(logsDirectory)
```

Next, we declare the schema of the data as a `case class`, in order to use the typed `Dataset` API. Following the formal description of the dataset (at: [NASA-HTTP](*http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html*)), the log is structured as follows:

The logs are an ASCII file with one line per request, with the following columns: - host making the request. A hostname when possible, otherwise the Internet address if the name could not be looked up. - timestamp in the

format "DAY MON DD HH:MM:SS YYYY", where DAY is the day of the week, MON is the name of the month, DD is the day of the month, HH:MM:SS is the time of day using a 24-hour clock, and YYYY is the year. The timezone is -0400. - request given in quotes. - HTTP reply code. - bytes in the reply.
NASA, http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html

Translating that schema to Scala, we have the following `case class` definition:

```
import java.sql.Timestamp
case class WebLog(host:String,
                  timestamp: Timestamp,
                  request: String,
                  http_reply:Int,
                  bytes: Long
                 )
```

NOTE

We use `java.sql.Timestamp` as the type for the timestamp as it's internally supported by Spark as a `Timestamp` type and does not require any additional `cast` that other options may require.

We convert the original JSON to a typed data structure using the previous schema definition

```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.types.IntegerType
// we need to narrow the `Interger` type because the JSON representation is
interpreted as `BigInteger`
val preparedLogs = rawLogs.withColumn("http_reply",
$"http_reply".cast(IntegerType))
val weblogs = preparedLogs.as[WebLog]
```

Now that we have the data in a structured format, we can start asking the questions that interest us. As a first step, we would like to know how many records are contained in our dataset.

```
val recordCount = weblogs.count
>recordCount: Long = 1871988
```

A common question would be: "what was the most popular URL per day?" To answer that, we first reduce the timestamp to the day of the month. We then group by this new *day of month* column and the request url and we count over this aggregate. We finally order using descending order to get this top URLs first.

```
topDailyURLs.show()
+----------+-----------------------------------------+-----+
|dayOfMonth|                                  request|count|
+----------+-----------------------------------------+-----+
|        13|GET /images/NASA-logosmall.gif HTTP/1.0 |12476|
|        13|GET /htbin/cdt_main.pl HTTP/1.0         | 7471|
|        12|GET /images/NASA-logosmall.gif HTTP/1.0 | 7143|
|        13|GET /htbin/cdt_clock.pl HTTP/1.0        | 6237|
|         6|GET /images/NASA-logosmall.gif HTTP/1.0 | 6112|
|         5|GET /images/NASA-logosmall.gif HTTP/1.0 | 5865|
       ...
```

Top hits are all images. What now? It's not unusual to see that the top URLs are images commonly used across a site. Our true interest lies in the content pages generating most traffic. To find those, we will first filter out `html` content and then proceed to apply the top aggregation we just learned.

As we can see, the request field is a quoted sequence of `[HTTP_VERB] URL [HTTP_VERSION]`. We will extract the url and preserve only those ending in .html, .htm or no extension (directories). This is a simplification for the purpose of this example.

```
val urlExtractor = """^GET (.+) HTTP/\d.\d""".r
val allowedExtensions = Set(".html",".htm", "")
val contentPageLogs = weblogs.filter {log =>
  log.request match {
    case urlExtractor(url) =>
      val ext = url.takeRight(5).dropWhile(c => c != '.')
      allowedExtensions.contains(ext)
    case _ => false
  }
}
```

With this new dataset that contains only `html`, `htm` and directories, we proceed to apply the same *top-k* function as above.

```
val topContentPages = contentPageLogs
  .withColumn("dayOfMonth", dayofmonth($"timestamp"))
  .select($"request", $"dayOfMonth")
  .groupBy($"dayOfMonth", $"request")
  .agg(count($"request").alias("count"))
  .orderBy(desc("count"))

topContentPages.show()
+----------+--------------------------------------------------+-----+
|dayOfMonth|                                           request|count|
+----------+--------------------------------------------------+-----+
|        13| GET /shuttle/countdown/liftoff.html HTTP/1.0"    | 4992|
|         5| GET /shuttle/countdown/ HTTP/1.0"                | 3412|
|         6| GET /shuttle/countdown/ HTTP/1.0"                | 3393|
|         3| GET /shuttle/countdown/ HTTP/1.0"                | 3378|
|        13| GET /shuttle/countdown/ HTTP/1.0"                | 3086|
|         7| GET /shuttle/countdown/ HTTP/1.0"                | 2935|
|         4| GET /shuttle/countdown/ HTTP/1.0"                | 2832|
|         2| GET /shuttle/countdown/ HTTP/1.0"                | 2330|
       ...
```

We can see that the most popular page that month was `liftoff.html` corresponding to the coverage of the launch of the Discovery shuttle, as documented on the NASA archives: *https://www.nasa.gov/mission_pages/shuttle/shuttlemissions/archives/sts-70.html*. It's closely followed by `countdown/` the days prior ot the launch.

Streaming Analytics

In the previous section we explored historical NASA weblog records. We found trending events in those records, but much later than the actual event happen.

One key driver for streaming analytics comes from the increasing demand of organizations to have timely infomation that can help them make decision at many different levels.

We can use the lessons we have learnt while exploring the archived records using a batch oriented approach and create a streaming job that will provide us with trending information as it happens.

The first difference we observe with the batch analytics is the source of the data. For our streaming exercise we will use a TCP server to simulate a web system that delivers its logs in real time. The simulator will use the same dataset, but will feed it through a TCP socket connection that will embody the stream that we will be analysing.

NOTE

For this example, we will use the `weblog_TCP_server` notebook in the online resources for the book. It simulates a TCP log producer that will serve as the source of streaming data. It's important to have this notebook running before continuing with the Structured Streaming notebook we will be building at this stage.

Connecting to a Stream

If we recall from the introduction, Structured Streaming defines the concepts of Sources and Sinks as the key abstractions to consume a stream and produce a result. We are going to use the `TextSocketSource` implementation to connect to the server through a TCP socket. Socket connections are defined by the host of the server and the port where it is listening for connections. These two configuration elements are required to create the `socket` source.

```
val stream = sparkSession.readStream
  .format("socket")
  .option("host", host)
  .option("port", port)
  .load()
```

Note how the creation of a stream is quite similar to the declaration of a static datasource in the batch case. Instead of using the `read` builder, we use the `readStream` construct and we pass to it the parameters required by the streaming source. As we will see during the course of this exercise and later on as we go into the details of Structured Streaming, the API is basically the same Structured API for static data, but with some modifications and limitations that we will will learn in detail.

Preparing the Data in the Stream

The `socket` source produces a streaming `DataFrame` with one column, `value` which contains the data received from the stream. See [TODO: REF: Sources#Socket] for additional details.

In the batch analytics case we could load the data directly as JSON records. In the case of the `Socket`source, that data is plain text. To transform our raw data to `WebLog` records, we first require a schema. The schema provides the necessary information to parse the text to a JSON object. It's the *structure*when we talk about *structured* streaming.

After defining a schema for our data, we will:

- Transform the text `value` to JSON using the JSON support built in the structured API of Spark
- Use the `Dataset` API to transform the JSON records to `WebLog` objects

As result of this process, we will obtain a `Streaming Dataset` of `WebLog` records.

```
import java.sql.Timestamp
case class WebLog(host:String,
                  timestamp: Timestamp,
                  request: String,
                  http_reply:Int,
                  bytes: Long
                 )

val webLogSchema = Encoders.product[WebLog].schema
val jsonStream = stream.select(from_json($"value", webLogSchema) as "record")
val webLogStream: Dataset[WebLog] = jsonStream.select("record.*").as[WebLog]
```

Operations on Streaming Dataset

The `webLogStream` we just obtained is of type `Dataset[WebLog]` like we had in the batch analytics job. The difference between this instance and the batch version is that `webLogStream` is a streaming `Dataset`.

We can observe this by querying the object.

```
webLogStream.isStreaming
> res: Boolean = true
```

At this point in the batch job, we were creating the first query on our data: How many records are contained in our dataset? This is a question that we can answer easily when we have access to all the data. But how to count records that are constantly arriving? The answer is that some operations we consider usual on a static `Dataset`, like counting all records, do not have a defined meaning on a streaming Dataset.

As we can observe, attempting to execute the `count` query below will result in an `AnalysisException`.

```
val count = webLogStream.count()
> org.apache.spark.sql.AnalysisException: Queries with streaming sources must
be executed with writeStream.start();;
```

This means that the direct queries we used to do on a static `Dataset` or `DataFrame` now need two levels of interaction. First, we need to declare the transformations of our stream and then we need to start the stream process with the `writeStream.start()` function.

Creating a Query

What are popular URLs? In what timeframe? Now that we have immediate analytic access to the stream of weblogs, we don't need to wait for a day or a month (or more than 20 years in the case of these NASA web logs) to have a rank of the popular URLs. We can have that information as trends unfold on much shorter windows of time.

First, to define the period of time of our interest, we create a window over some timestamp. An interesting feature of Structured Streaming is that we can define that time interval on the timestamp when the data was produced, also known as *event time* as opposed to the time when the data is being processed.

Our window definition will be of 5 minutes of event data. Given that our timeline is simulated, the 5 minutes might happen much faster or slower than the clock time. In this way we can clearly appreciate the how structured streaming uses the timestamp information in the events to keep track of the event timeline.

As we learnt from the batch analytics, we should extract the URLs and only select content pages, like `html`, `htm` or directories. Let's apply that acquired knowledge first before proceeding to define our `window` query.

```
// A regex expression to extract the accessed URL from weblog.request
val urlExtractor = """^GET (.+) HTTP/\d.\d""".r
val allowedExtensions = Set(".html", ".htm", "")

val contentPageLogs: String => Boolean = url => {
  val ext = url.takeRight(5).dropWhile(c => c != '.')
  allowedExtensions.contains(ext)
}

val urlWebLogStream = webLogStream.flatMap { weblog =>
  weblog.request match {
    case urlExtractor(url) if (contentPageLogs(url)) =>
      Some(weblog.copy(request = url))
    case _ => None
  }
}
```

We have converted the request to only contain the visited URL and filtered out all non-content pages. We will now define the windowed query to compute the top trending URLs

```
val rankingURLStream = urlWebLogStream
    .groupBy($"request", window($"timestamp", "5 minutes", "1 minute"))
    .count()
```

Start the Stream Processing

All the steps we have followed so far have been to define the process that the stream will undergo. But no data has been processed yet.

To start a Structured Streaming job, we need to specify a `sink` and an `output mode`. These are two new concepts introduced by Structured Streaming:

- A `sink` defines where we want to materialize the resulting data, like to a file in a file system, to an in-memory table or to another streaming system such as Kafka.
- The `output mode` defines how we want the results to be delivered: Do we want to see all data every time, only updates or just the new records?

These options are given to a `writeStream` operation. It creates the streaming query that starts the stream consumption, materializes the computations declared on the query and produces the result to the output `sink`.

We will visit all these concepts in detail later on. For now, we will use them empirically and observe the results.

For our query, we will use the `memory sink` and output mode `complete` to have a fully updated table each time new records are added to the result of keeping track of the URL ranking.

```
val query = rankingURLStream.writeStream
  .queryName("urlranks")
  .outputMode("complete")
  .format("memory")
  .start()
```

The memory sink outputs the data to a temporary table of the same name given in the queryName option. This can be observed by querying the tables registered on `Spark SQL`

```
sparkSession.sql("show tables")

## TODO Add output
```

In the expression on listing XXX `query` is of type `StreamingQuery` and it's a handler to control the query lifecycle. We will learn more about query management in chapter [TODO: REF: Query Management]

Exploring the Data

Given that we are accelerating the log timeline on the producer side, after few seconds, we can execute the next command to see the result of the first windows. Note how the processing time (few seconds) is decoupled from the event time (tens of minutes of logs). We will explore event time in detail in chapter [TODO: REF: Event Time]

```
urlRanks.select($"request", $"window", $"count").orderBy(desc("count"))
```

[[URL Rank Query Results by Window]] image::images/url-rank-query-results.png[]

Summary

In these first steps into Structured Streaming, you have seen the though process behind the development of a streaming application. By starting with a batch version of the process, we gained intuition about the data and using those insights, we created a streaming version of the job. In the process we could appreciate how close the *structured* batch and the streaming APIs are, albeit we also observed that some usual batch operations do now apply in a streaming context.

With this exercise, we hope to have increased your curiosity about Structured Streaming and that you are ready for the learning path through this section.

Chapter 8. The Structured Streaming Programming Model

Structured Streaming builds on the foundations laid on top of Spark's Structured APIs. By extending the `Dataset` API to support streaming workloads, Structured Streaming inherits the traits of the high-level language introduced by Spark SQL as well as the underlying optimizations. [1] At the same time, Structured Streaming becomes available in all the supported language bindings for Spark SQL. These are: Scala, Java, Python and R, although some of the advanced state management features are only currently available on Scala.

When compared to the more mature Spark Streaming API, Structured Streaming introduces support for event-time across all windowing and aggregation operations, making it easy to program logic that uses the time when events were generated, as opposed to the time when they enter the processing engine, also known as *processing time*.

With the availability of Structured Streaming in the Spark ecosystem, Spark manages to unify the development experience between *classical* batch and stream-based data processing.

In this chapter, we are going to learn the programming model of Structured Streaming by following the sequence of steps that are usually required to create a streaming job with Structured Streaming:

- Initializing Spark,
- Sources: acquiring streaming data,
- Declaring the operations we want to apply to the streaming data,
- Sinks: output the resulting data

Initializing Spark

Part of the visible unification of APIs in Spark, is that `SparkSession` becomes the single entry point for *batch* and *streaming* applications that use Structured Streaming.

Therefore, our entry point to create a Spark job is the same as when using the Spark batch API: We instantiate a `SparkSession`.

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession
  .builder()
  .appName("StreamingProgramming")
  .master("local[*]")
  .getOrCreate()
```

USING THE SPARK SHELL

When using the Spark shell to explore Structured Streaming, the `SparkSession` is already provided. We don't need to create any additional context to use Structured Streaming.

Sources: acquiring streaming data

In Structured Streaming, a *source* is the abstraction that lets us consume data from a streaming data producer. Sources are not directly created. The spark session provides a builder method, `readStream`, that exposes the API to specify a streaming source and its configuration.

For example, the code in Example 8-1 creates a `File` streaming source. We specify the specific type of *source* using the `format` method. The method `schema` lets us provide a schema for the data stream, which in mandatory for certain *source* types.

Each *source* implementation has different options and some have tunable parameters. In the example Example 8-1, we are setting the *option* `mode` to `DROPMALFORMED`. This option indicates the JSON stream processor to drop any line that does not comply to the JSON format or does not match the provided schema.

Example 8-1. File Streaming Source

```
val fileStream = spark.readStream
  .format("json")
  .schema(schema)
  .option("mode","DROPMALFORMED")
  .load("/tmp/datasrc")

>fileStream: org.apache.spark.sql.DataFrame = [id: string, timestamp: timestamp ... 2
more fields]
```

Behind the scenes, the call to `spark.readStream` creates a `DataStreamBuilder` instance. This instance is in charge of managing the different options provided through the builder method calls. Calling `load(...)` on this `DataStreamBuilder` instance, validates the options provided to the builder and, if everything checks, it returns a streaming `DataFrame`.

In our example, this streaming `DataFrame` represents the stream of data that will result in monitoring the provided *path* and processing each new file in that *path* as *JSON* encoded data, parsed using the *schema* provided. All malformed will be dropped from this data stream.

Loading a streaming source is lazy. What we get is a representation of the stream, embodied in the streaming `Dataframe` instance, that we can use to express the series of transformations we want to apply to it in order to implement our specific business logic. Creating a streaming `DataFrame` does not result in any data actually being consumed or processed until the stream is materialized. This requires a *query*, as we will see further on.

Available `Sources`

As of Spark v2.3.0, the following streaming sources are supported:

json, orc, parquet, csv, text, textFile

> These are all file-based streaming sources. The base functionality it to monitor a path (folder) in a filesystem and consume files atomically placed in it. The files found will then be parsed by the formatter specified. For example, if `json` is provided, the Spark `json` reader will be used to process the files, using the *schema* information provided.

socket

Establishes a client connection to a TCP server that is assumed to provide text data through a socket connection. The options `host` and `port` are mandatory. There's an optional `timestamp`flag that can be provided as an option. When `true`, the resulting records will be timestamped with the time of arrival. The data from the TCP connection is assumed to be in text format.

kafka

Creates Kafka consumer able to retrieve data from Kafka.

rate

Generates a stream of rows at the rate given by the `rowsPerSecond` option. It's mainly intended as a testing source.

Transforming Streaming Data

As we saw in the previous section, the result of calling `load` is a streaming `DataFrame`. Once we have created our streaming `DataFrame` using a `source`, we can use the `Dataset` API to express the logic we want to apply to the data in the stream in order to implement our specific use case.

NOTE

Let's recall that `DataFrame` is an alias for `Dataset[Row]`. Although this seems a small technical distinction, when used from a typed language such as `scala` the `Dataset` API presents a typed interface while the `DataFrame` usage is untyped. When the structured API is used from a dynamic language such as Python, the `DataFrame` API is the only available API.

```
val highTempSensors = sensorStream
  .select($"deviceId", $"timestamp", $"sensorType", $"value")
  .where($"sensorType" === "temperature" && $"value" > threshold)
```

Using a theoretical sensor network, in this example we are selecting the fields `deviceId`, `timestamp`, `sensorType`, `value` from a `sensorStream` and filtering to only those records where the sensor `value` is higher than the given `threshold`.

Likewise, we can aggregate our data and apply operations to the groups over time.

```
val avgBySensorType = sensorStream.groupBy($"sensorType")
                                  .agg(avg("value"))
```

In this example, we are interested in the average of the sensor `value`s grouped by `sensorType`. As this represents a stateless aggregation, it will result in the average values per `sensorType` that were received together.

A more interesting question to ask this stream is likely the evolution of that average over time.

```
val avgBySensorTypeOverTime = sensorStream
  .select($"timestamp", $"sensorType", $"value")
```

```
    .groupBy(window($"timestamp", "5 minutes", "1 minute"), $"sensorType")
    .agg(avg($"value")
```

As we see here, we can use `timestamp` information from the event itself to define a time window of 5 minutes that will slide every minute. What is important to grasp here is that the Structured Streaming API is practically the same as the `Dataset` API for batch analytics.

If you are not familiar with the structured APIs of Spark, we suggest that you familiarize yourself with it. Covering this API in detail is beyond the scope of this book. We recommend "Spark: The Definitive Guide" by Bill Chambers and Matei Zaharia as a comprehensive reference.

Streaming API Restrictions

As we hinted during the introduction, some operations that are offered by the standard API do not make sense on a streaming context.

For example, consider `stream.count`. What would be the count of a stream that might contain infinite values over time? We can say that a stream count is undefined.

```
val count = stream.count()
org.apache.spark.sql.AnalysisException: Queries with streaming sources must be
    executed with writeStream.start();
```

If we try to execute such operation, Spark will complain. What this error message is saying (in a cryptical way for the uninitiatied) is that `count` would require the materialization of the stream and that the operation is only possible by issuing an `output` query. We will learn about `output` in the next section.

API operations not directly supported on streams:

- `count`
- `show`
- `describe`
- `limit`
- `take(n)`
- `distinct`
- `foreach`
- `sort`
- multiple stacked aggregations

Next to these operations, `join` is partially supported.

In general, operations that require immediate materialization of the underlying dataset are not allowed. These include: `take`, `show`, `count`, `limit`, `describe`.

Some stream operations are computationally hard. `distinct` is one of them. To filter duplicates in an arbitrary stream, it would require to remember all the data seen so far and compare each new record with all records already seen. The first condition would require infinite memory and the second has a computational complexity of O(n^2) which becomes prohibitive as the number of elements (n) increases.

OPERATIONS ON AGGREGATED STREAMS

Some of the unsupported operations become defined after we apply an aggregation function to the stream. While we can't `count` the stream, we could `count` messages received per minute or `count`the number of devices of a certain type.

In the code sample Example 8-2 we define a `count` of events per `sensorType` per minute.

Example 8-2. Count of sensor types over time

```
val avgBySensorTypeOverTime = sensorStream
  .select($"timestamp", $"sensorType")
  .groupBy(window($"timestamp", "1 minutes", "1 minute"), $"sensorType")
  .count()
```

Likewise, it's also possible to define a `sort` on aggregated data, although it's further restricted to queries with output mode `complete`.

STREAM DEDUPLICATION

We discussed that `distinct` on an arbitrary stream is computationally hard to implement. But if we can define a key that tells us when an element in the stream has already been seen, we can use it to remove duplicates.

```
stream.dropDuplicates("key") ...
```

WORKAROUNDS

Although some operations are not supported in the exact same way as in the *batch* model, there are alternative ways to achieve the same functionality.

foreach

: Although `foreach` cannot be directly used on a stream, there's a *foreach sink* that provides the same functionality.

  Sinks are specified in the output definition of a stream.

show

: While `show` requires an immediate materialization of the query, and hence it's not possible on a streaming `Dataset`, we can use the `console` sink to output data to the screen.

Sinks: output the resulting data

All operations that we have done so far — such as creating a stream and applying transformations on it — have been declarative. They define where to consume the data from and what operations we want to apply to it. But up to this point, there is still no data flowing through the system.

Before we can initiate our stream, we need to first define *where* and *how* we want the output data to go:

- *Where* relates to the streaming *sink*: the receiving side of our streaming data.
- *How* refers to the *output mode*: how to treat the resulting records in our stream.

From the API perspective, we materialize a stream by calling `writeStream` on a streaming `DataFrame` or `Dataset`.

NOTE

We can appreciate the symmetry in the Spark API: While `readStream` provides the options to declare the source of the stream, `writeStream` lets us specify the output sink and the output mode required by our process. They are the counterparts of `read` and `write` in the `DataFrame` APIs. As such, they provide an easy way to remember see the execution mode used in a Spark program:

- `read`/`write` : Batch operation
- `readStream`/`writeStream` : Streaming operation

Calling `writeStream` on a streaming `Dataset` creates a `DataStreamWriter`. This is a builder instance which provides methods to configure the output behavior of our streaming process.

Example 8-3. File Streaming Source

```
val query = stream.writeStream
  .format("json")
  .queryName("json-writer")
  .outputMode("append")
  .option("path", "/target/dir")
  .option("checkpointLocation", "/checkpoint/dir")
  .trigger(ProcessingTime("5 seconds"))
  .start()

>query: org.apache.spark.sql.streaming.StreamingQuery = ...
format
```

The `format` method lets us specify the output sink by providing the name of a built-in *sink* or the fully qualified name of a custom *sink*.

As of Spark v2.3.0, the following streaming sinks are available:

console sink

: A sink that prints to the standard output. It shows a number of rows configurable with the option *numRows*.

file sink

: File-based and format-specific sink that write the results to a file system. The format is specified by providing the format name: csv, hive, json, orc, parquet, text.

kafka sink

: A Kafka-specific producer sink that is able to write to one or more Kafka topics.

memory sink

Creates an in-memory table using the provided query name as table name. This table receives continuous updates with the results of the stream.

foreach sink

Provides a programmatic interface to access the stream contents.

`outputMode`

The `outputMode` specifies the semantics of how records are added to the output of the streaming query. The supported modes are `append`, `update` and `complete`:

`append`

(default mode) Adds only *final* records to the output stream. A record is considered *final* when no new records of the incoming stream can modify its value. This is always the case with linear transformations, like those resulting from applying projection, filtering and mapping. This mode guarantees that each resulting record will be output only once.

When the streaming query contains aggregations, the definition of *final* becomes non-trivial. In an aggregated computation, new incoming records might change an existing aggregated value when they comply with the aggregation criteria used. Following our definition, we cannot output a record using `append` until we know that its value is *final*. Therefore, the use of `append` output mode in combination with aggregate queries is restricted to queries where the aggregation is expressed using event-time and it defines a *watermark*. In that case, `append` will output an event once the `watermark`has expired and hence it's considered that no new record can alter the aggregated value. As a consequence, output events in `append` mode will be delayed by the aggregation time window plus the watermark offset.

`update`

Adds new and updated records since the last trigger to the output stream. `update` is only meaningful in the context of an aggregation, where aggregated values change as new records arrive. If more than one incoming record changes a single result, all changes between trigger intervals are collated into one output record.

`complete`

`complete` mode outputs all records seen until now. This mode also relates to aggregations, as for non-aggregated streams, we would need to remember all records seen so far, which is unrealistic. From a practical perspective, `complete` mode is recommended only when we are aggregating values over low-cardinality criteria, like *count of visitors by country*, where we know that the number of countries is bounded.

`queryName`

With `queryName`, we can provide a name for the query which is used by some *sinks* and also presented in the *job* description in the Spark Console.

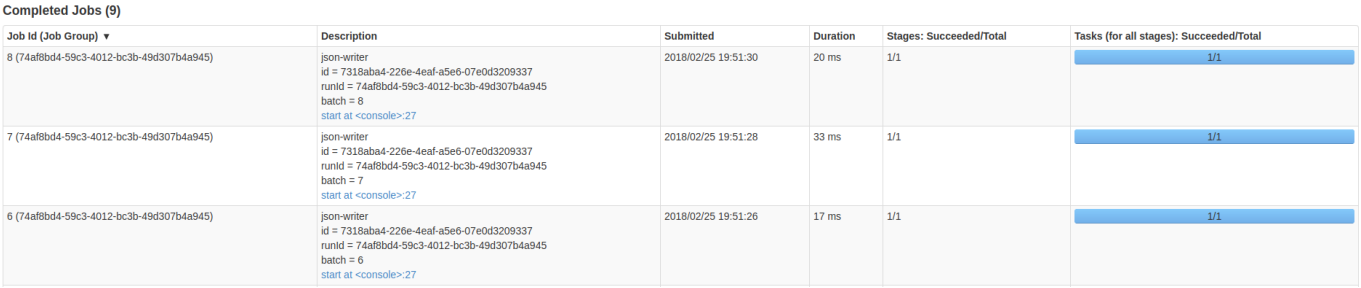


Figure 8-1. Completed Jobs in the Spark UI showing the query name in the job description

`option`

With the `option` method we can provide specific key-value pairs of configuration to the stream, akin to the configuration of the *source*. Each *sink* may have specific configuration that can be customized using this method.

We can add as many `.option(...)` calls as necessary to configure the sink.

`options`

`options` is an alternative to `option` that takes a `Map[String, String]` containing all the key-value configuration parameters that we want to set. This alternative is more friendly to an externalized configuration model, where we don't know *a priori* the settigs to be passed to the sink's configuration.

`trigger`

The optional `trigger` option lets us specify the frequency at which we want the results to be produced. By default, Structured Streaming will process the input and produce a result as soon as possible. When a trigger is specified, output will be produced at each trigger interval.

To materialize the streaming computation, we need to `start` the streaming process.
Finally, `start()` materializes the complete job description into a streaming computation and initiates the internal scheduling process that results in data being consumed, processed and outputted. `start()` returns a `StreamingQuery` object which is a handle to manage the individual lifecycle of each query. This means that multiple queries can be started and stopped simultaneously and independently of each other within the same `sparkSession`.

Summary

After reading this chapter, you should have a decent knowledge about the Structured Streaming programming model and API. In this chapter, we learned:

- That each streaming program starts by defining a *source* and what sources are currently available
- That we can reuse most of the familiar `Dataset` and `DataFrame` API for transforming the streaming data.
- That some common operations from the `batch` API do not make sense in streaming mode.

- That sinks are the configurable definition of the stream output.
- The relation between output modes and aggregation operations in the stream.
- That all transformations are lazy and that we need to `start` our stream to get data flowing through the system.

In the next chapter, we are going to apply our newly acquired knowledge to create a comprehensive stream processing program. After that, we will zoom into specific areas of the Structured Streaming API such as event-time handing, window definitions, the concept of watermarks and arbitrary state handling.

[1]This includes the use of the Catalyst query optimizer and the low overhead memory management and code generation delivered by Project Tunsgten.

Chapter 9. Structured Streaming in Action

Now that we have a better understanding of the Structured Streaming API and programming model, we are going to create a small but complete IoT-inspired streaming program. Our use case will be to consume a stream of sensor readings from a Apache Kafka as the streaming source.

NOTE

Apache Kafka is a distributed server for data streams, which includes many advanced functionalities interesting in a production envrionment, from stream replay to stream processing. You'll find it at `http://kafka.apache.org`.

We will correlate the incoming IoT sensor data with a reference file that contains all known sensors with their configuration. That way, we will enrich each incoming record with specific sensor parameters that allow us to interpret the reported data. We will then save all correctly processed records to a file in Parquet Format.

NOTE

Consuming a Streaming Source

The first part of our program deals with the creation of the streaming `Dataset`:

```
val rawData = sparkSession.readStream
      .format("kafka")
      .option("kafka.bootstrap.servers", kafkaBootstrapServer)
      .option("subscribe", topic)
      .option("enable.auto.commit", true)
      .option("group.id", "iot-data-consumer")
      .option("startingOffsets", "earliest")
      .load()

> rawData: org.apache.spark.sql.DataFrame
```

The entry point of Structured Streaming is an existing `Spark Session`. As we can appreciate on the first line, the creation of a `Streaming Dataset` is almost identical to the creation of a static `Dataset` that would use a `read` operation instead. `sparkSession.readStream` returns a `DataStreamReader`, a class that implements the builder pattern to collect the information needed to construct the streaming source using a *fluid* API. In that API, we find next the `format` option, that lets us specify our *source* provider, which, in our case, is `kafka`. The options that follow it are provider-specific:

- `kafka.bootstrap.servers`: indicates the set of bootstrap servers to contact as a comma-separated list of `host:port` addresses
- `subscribe`: specify the topic or topics to subcribe to
- `enable.auto.commit`: whether or not to enable auto committing the consumed offsets
- `group.id`: the identity of this application as a consumer group
- `startingOffsets`: the offset reset policy to apply when this application starts out fresh

We will cover the details of the Kafka streaming provider later in this book.

```
> rawData: org.apache.spark.sql.DataFrame
```

The `load()` method evaluates the `DataStreamReader` builder and creates a `DataFrame` as a result. A `DataFrame` is an alias for `Dataset[Row]` with a known schema. After creation, streaming `Dataset`s can be used just like regular `Dataset`s. This makes it possible to use the full-fledged `Dataset` API with Structured Streaming, albeit some exceptions apply as not all operations, such as `show()` or `count()`, make sense in a streaming context.

To programmatically differentiate a *streaming* `Dataset` from a static one, we can ask a `Dataset`whether it is of the *streaming* kind:

```
rawData.isStreaming
res7: Boolean = true
```

And we can also explore the schema attached to it, using the existing `Dataset` API:

```
rawData.printSchema()

root
 |-- key: binary (nullable = true)
 |-- value: binary (nullable = true)
 |-- topic: string (nullable = true)
 |-- partition: integer (nullable = true)
 |-- offset: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- timestampType: integer (nullable = true)
```

In general, Structured Streaming requires the explicit declaration of a schema for the consumed stream. In the specific case of `kafka`, the schema for the resulting `Dataset` is fixed and is independent of the contents of the stream. It consists of a set of fields specific to the Kakfa `source`: `key`, `value`, `topic`, `partition`, `offset`, `timestamp` and `timestampType`. In most cases, applications will be mostly interested in the contents of the `value` field, where the actual payload of the stream resides.

Application Logic

As we recall, the intention of our job is to correlate the incoming IoT sensor data with a reference file that contains all known sensors with their configuration. That way, we would enrich each incoming record with specific sensor parameters that would allow us to interpret the reported data. We would then save all correctly processed records to a Parquet file. The data coming from unknown sensors would be saved to a separate file for later analysis.

Using Structured Streaming, our job can be implemented in terms of `Dataset` operations:

```
val iotData = rawData.select($"value").as[String].flatMap{record =>
  val fields = record.split(",")
  if (fields.size == 3) {
    Try {
      SensorData(fields(0).toInt, fields(1).toLong, fields(2).toDouble)
```

```
    }.toOption
  } else { None }
}

val sensorWithInfo = sensorRef.join(iotData, Seq("sensorId"), "inner")

val knownSensors = sensorWithInfo
  .withColumn("dnvalue", $"value"*($"maxRange"-$"minRange")+$"minRange")
  .drop("value", "maxRange", "minRange")
```

In the first step, we transform our csv-formatted records back into `SensorData` entries. We apply Scala functional operations on the typed `Dataset[String]` that we obtained from extracting the `value` field as a `String`. The rest of the operation is identical to the Spark Streaming version, where we used `flatmap` as a means to prune invalid records that do not match our parsing logic.

Then, we use a streaming `Dataset` to static `Dataset inner join` to correlate the sensor data with the corresponding reference using the `sensorId` as key.

To complete our application, we compute the real values of the sensor reading using the min-max ranges in the reference data.

Let's recall our Spark Streaming implementation to see the commonalities as well as where the two APIs differ:

```
val schemaStream = rawDStream.flatMap{record =>
  val fields = record.split(",")
  if (fields.size == 3) {
    Try {
      SensorData(fields(0).toInt, fields(1).toLong, fields(2).toDouble)
    }.toOption
  } else {
   None
  }
}

schemaStream.foreachRDD{rdd =>
  val sensorDF = rdd.toDF()
  val sensorWithInfo = sensorDF.join(sensorRef, "sensorId")
  val dnSensorData = sensorWithInfo
    .withColumn("dnvalue", $"value"*($"maxRange"-$"minRange")+$"minRange")
  val sensorRecords = dnSensorData.drop("value", "maxRange", "minRange")
  // ...
}
```

The transformation of the raw data into records is very similar in both approaches: we use the functional `flatMap` combinator together with an `Option` type to filter our invalid records. Both the `DStream` and the `Dataset` APIs provide us with a `flatMap` implementation over the collections they represent. This lets us apply the same parsing function: `String -> SensorData` to reconstruct our stream of `SensorData` records.

The next part is where we see clear differences between the two streaming models. In Spark Streaming, we use the `foreachRDD` operation to get access to the underlying `RDD`. Within this operation, we also have access to the `Spark Context` and this enables us to transform the `RDD` to a `DataFrame` in order to gain access to Spark SQL's column-oriented functions.

In Structured Streaming, our incoming stream is already available as a `Dataset`, giving us first-class access to the Spark SQL facilities we wish to use. If we wanted to use Spark SQL functions from the beginning, the Structured Streaming API provides us direct access. On the other hand, if we wanted to combine Spark SQL and other libraries directly operating on data collections, Spark Streaming gives us a more generic programmatic access to the streaming data, without prescribing a programming model like SQL.

Writing to a Streaming Sink

The final step of our streaming application is to write the enriched IoT data to a *Parquet* formatted file. In the Structured Streaming version, the `write` operation is crucial: it declares the completion of the query definition on the stream, defines a *write mode* and upon calling *start*, the continuous query will start. Just like with Spark Streaming, in Structured Streaming all operations are lazy declarations of what we want to do with the streaming data. Only when we call `start`, the actual consumption of the stream will start and the query operations on the data will materialize into actual results.

```
val knownSensorsQuery = knownSensors.writeStream
  .outputMode("append")
  .format("parquet")
  .option("path", "/tmp/learning-spark-streaming/stst-known_sensors")
  .option("checkpointLocation", "/tmp/checkpoint")
  .start()
```

Let's break this operation down:

- Like `write` in the static `Dataset` API, `writeStream` creates a builder object where we can configure the options for the desired write operation.
- With `format` we specify the sink that will materialize the result downstream. In our case, we use the built-in `FileStreamSink` with *Parquet* format.
- `mode` is a new concept in Structured Streaming: given that we, theoretically, have access to all the data seen in the stream so far, we also have the option to produce different views of that data. The `append` mode, used here, is the equivalent of the Spark Streaming model: the new records affected by our streaming computation are produced to the output.

Another notable difference with the Spark Streaming model, is that the result of the `start` call is a`StreamingQuery` instance. This object provides methods to control the execution of the query and request information about the status of our running streaming query. Here we can see the `StreamingQueryProgress` as a result of calling `knownSensorsQuery.recentProgress`.

```
knownSensorsQuery.recentProgress

res37: Array[org.apache.spark.sql.streaming.StreamingQueryProgress] =
Array({
  "id" : "6b9fe3eb-7749-4294-b3e7-2561f1e840b6",
  "runId" : "0d8d5605-bf78-4169-8cfe-98311fc8365c",
  "name" : null,
  "timestamp" : "2017-08-10T16:20:00.065Z",
  "numInputRows" : 4348,
  "inputRowsPerSecond" : 395272.7272727273,
  "processedRowsPerSecond" : 28986.666666666668,
```

```
"durationMs" : {
  "addBatch" : 127,
  "getBatch" : 3,
  "getOffset" : 1,
  "queryPlanning" : 7,
  "triggerExecution" : 150,
  "walCommit" : 11
},
"stateOperators" : [ ],
"sources" : [ {
  "description" : "KafkaSource[Subscribe[iot-data]]",
  "startOffset" : {
    "iot-data" : {
      "0" : 19048348
    }
  },
  "endOffset" : {
    "iot-data" : {
      "0" : 19052696
    }
  },
  "numInputRow...
```


Summary

Hopefully, this initial hands-on chapter has shown you how to create your first non-trivial application using Structured Streaming.

After reading this chapter, you should have a better understanding of the structure of a Structured Streaming program and how to approach a streaming application, from consuming the data, to processing it using the `Dataset` and `DataFrames` APIs, to producing the data to an external output. At this point, you could already adventure yourself into creating your own streaming processing jobs.

In the next chapters are going to learn in depth the different aspects of Structured Streaming.

Chapter 10. Structured Streaming Sources

The previous chapters should have given you a good overview of the Structured Streaming programming model and how to apply it in a practical way. We also saw how *sources* are the starting point of each Structured Streaming program. In this chapter, we are going to study the general characteristics of a *source* and review the available *sources* in greater detail, including their different configuration options and modes of operation.

Understanding Sources

In Structured Streaming, a *source* is the abstraction that represents streaming data providers. The concept behind the *source* interface is that streaming data is a continuous flow of events over time that can be seen as a sequence, indexed with a monotonously incrementing counter.

As we can see in Figure 10-1, each event in the stream is considered to have an ever increasing offset.

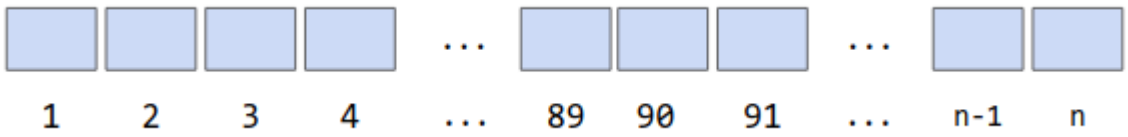


Figure 10-1. A stream seen as an indexed sequence of events

Offsets are used to request data from the external source and to indicate what data has already been consumed. Structured Streaming knows when there is data to process by asking the current offset from the external system and comparing it to the last processed offset. The data to be processed is requested by getting a *batch* between two offsets `start` and `end`. The *source* is informed that data has been processed by committing a given offset. The *source* contract guarantees that all data with an offset less than or equal to the committed offset has been processed and that subsequent requests will only request offsets greater than that committed offset. Given these guarantees, *sources* might opt to discard the processed data to free up system resources.

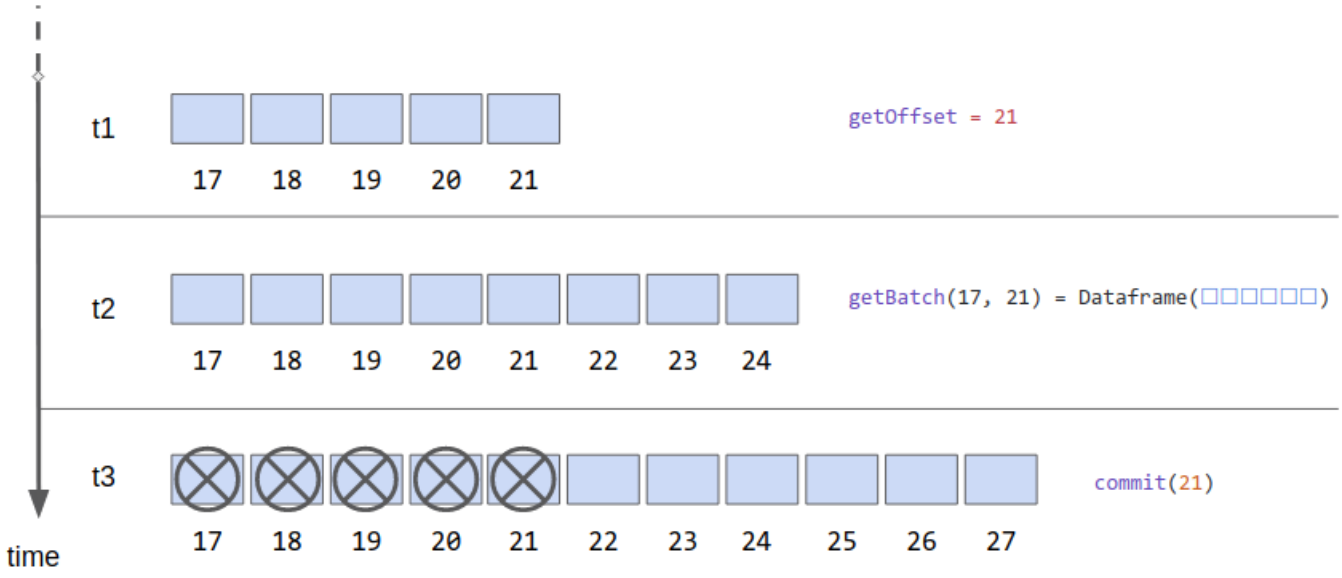


Figure 10-2. Offset process sequence

The illustration Figure 10-2 shows the dynamics of the offset-based processing:

1. At *t1* time, the system calls `getOffset` and obtains the current offset for the *source*.
2. At *t2* time, the system obtains the batch up to the last known offset by calling `getBatch(start, end)`. Note that new data might have arrived in the meantime.
3. At *t3* time, the system `commits` the offset and the source drops the corresponding records.

This process repeats constantly, ensuring the acquisition of streaming data. To recover from eventual failure, offsets are often *checkpointed* to external storage.

Besides the offset-based interaction, *sources* must fulfill two requirements: they must be replayable and they must provide a schema.

Sources must be replayable

In Structured Streaming, *replayability* is the capacity to request a part of the stream that had been already requested but not committed yet. Just like we can rewind that Netflix serie we are watching to see a piece we just missed because of a distraction, *sources* must offer the capability to replay a piece of the stream that was already requested but not committed. This is done by calling `getBatch` with the offset range that we want to receive again.

A *source* is considered to be *reliable* when it can produce an uncommitted offset range even after a total failure of the Structured Streaming process. In this failure recovery process, offsets are restored from their last known *checkpoint* and are requested again from the *source*. This requires the actual streaming system that is backing the *source* implementation to store data safely outside the `streaming` process. By requiring *replayability* from the *source*, Structured Streaming delegates recovery responsibility to the *source*. This implies that only *reliable* sources work with Structured Streaming to create strong end-to-end delivery guarantees.

Sources must provide a schema

A defining characteristic of the *structured* APIs of Spark is that they rely on schema information to handle the data at different levels. As opposed to processing opaque *Strings* or *Byte Array* blobs, *Schema* information provides insights on how the data is shaped in terms of fields and types. Schema information can be used to drive optimizations at different levels in the stack — from query planning to the internal binary representation of data, storage and access to it.

Sources must provide schema information that describes the data they produce. Some *source*implementations allow this schema to be configured and use this configuration information to automatically parse incoming data and transform it into valid records. In fact, many file-based streaming sources such as `json` or `csv` files follow this model, where we must provide the schema used by the file format to ensure proper parsing. Some other *sources* use a fixed internal schema that express the meta-data information of every record and leaves the parsing of the payload to the application.

From an architectural perspective, creating schema-driven streaming applications is desirable since it facilitates the global understanding of how data flows through the system and drives the formalization of the different stages of a multi-process streaming pipeline.

Available Sources

The following are the sources currently available in the Spark distribution of Structured Streaming:

File Source

: Allows the ingestion of data stored as files. In most cases, the data is transformed in records that are further processed in streaming mode. It supports these formats: JSON, CSV, Parquet, ORC, and plain text.

Kafka Source

: Allows the consumption of streaming data from Apache Kafka.

Socket Source

: A TCP socket client able to connect to a TCP server and consume a text-based data stream. The stream must be encoded in the UTF-8 charset

Rate Source

: Produces internally generated stream of a `(timestamp, value)` records with a configurable production rate. It's normally used for learning and testing purposes.

As we discussed in "Understanding Sources", sources are considered *reliable* when they provide replay capabilities from an offset, even when the structured streaming process fails. Using this criteria, we can classify the available sources in:

- *reliable*: File Source, Kafka Source
- *unreliable*: Socket Source, Rate Source

The *unreliable* sources may only be used in a production system when the loss of data can be tolerated.

WARNING

The Streaming source API is currently under evolution. At the time of writing, there is no stable public API to develop custom sources. This is expected to change in the near future.

In the next part of this chapter we are going to explore in detail the *sources* currently available. As production-ready sources, the *File* and the *Kafka* sources have many options that we will discuss in detail. The *Socket* and the *Rate* source are limited in features, which will be evident in their concise coverage.

The File Source

The file *source* is a simple streaming data source that reads files from a monitored directory in a file system. A file-based hand-over is a commonly used method to bridge a batch-based process with a streaming system. The batch process produces its output in a file format and drops it in a common directory where a suitable implementation of the *File source* can pick these files up and transform their contents into a stream of records for further processing in streaming mode.

Specifying a File Format

The files are read using a specified format, which is provided with the `.format(<format_name>)` method in the `readStream` builder or by using the dedicated methods in the `DataStreamReader`that indicate the format to use, e.g.: `readStream.parquet('/path/to/dir/')`. When using the dedicated methods corresponding to each supported format, the method call should be done as the last call of the builder.

For example, these three forms are equivalent.

```
// Use format and load path
val fileStream = spark.readStream
  .format("parquet")
  .schema(schema)
  .load("hdfs://data/exchange")

// Use format and path options
val fileStream = spark.readStream
  .format("parquet")
  .option("path", "hdfs://data/exchange")
  .schema(schema)
  .load()

// Use dedicated method
val fileStream = spark.readStream
  .schema(schema)
  .parquet("hdfs://data/exchange")
```

As of Spark v2.3.0, the following file-based formats are supported by Structured Streaming. These are the same file formats supported by the static `DataFrame`, `Dataset` and SQL APIs:

- CSV
- JSON
- Parquet
- ORC
- text
- textFile

Common Options

Regardless of the specific format, the general functionality of the *File source* is to monitor a *directory*in a shared *file system* identified by its specific `url`. All file formats support a common set of options that control the file inflow and define the aging criteria of the files.

WARNING

As Apache Spark is a fast evolving project, APIs and their options may change in future versions. Also, in this section, we will only cover the most relevant options that apply to streaming workloads. For the most up-to-date information, always check the API documentation corresponding to your Spark version at *http://spark.apache.org/*.

These options can be set for all file-based *sources*:

`maxFilesPerTrigger` (Default: unset)

> Indicates how many files will be consumed at each query trigger. This setting limits the number of files processed at each trigger and in doing so, it helps controlling the data inflow in the system.

`latestFirst` (Default: `false`)

> When this flag is set to `true`, newer files are elected for processing first. Use this option when the most recent data has higher priority over older data.

`maxFileAge` (Default: `7 days`)

> Defines an age threshold for the files in the directory. Files older than the threshold will not be elegible for processing and will be effectively ignored. This threshold is relative to the most recent file in the directory and not to the system clock. For example, if `maxFileAge` is `2 days` and the most recent file is from yesterday, the threshold to consider a file too old will be *older than 3 days ago*.

`fileNameOnly` (Default: `false`)

> When set to `true`, two files will be considered the same if they have the same name, otherwise, the full path will be considered.

NOTE

When `latestFirst` is set to `true` and `maxFilesPerTrigger` option is configured, `maxFileAge` will be ignored as there might be a condition where files that are valid for processing become older than the threshold as the system gives priority to the most recent files found. In such cases, no aging policy can be set.

Common Text Parsing Options (*CSV*, *JSON*)

Some file formats, such as CSV and JSON use a configurable parser to transform the text data in each file into structured records. It's possible for upstream processes to create records that do not fulfill the expected format. These records are considered corrupted.

Streaming systems are characterized for running continuosly. A streaming process should not fail when bad data is received. Depending of the business requirements, we can either drop the invalid records or route the data considered corrupted to a separate error-handling flow.

HANDING PARSING ERRORS

The following options allow for the configuration of the parser behavior to handle those records that are considered corrupted:

`mode` (default `PERMISSIVE`)

Controls the way that corrupt records are handled during parsing. Allowed values are: `PERMISSIVE`, `DROPMALFORMED`, and `FAILFAST`.

- `PERMISSIVE`: The value of the corrupt record is inserted in a special field configured by the option `columnNameOfCorruptRecord` that must exist in the *schema*. All other fields are set to `null`. If the field does not exist, the record is dropped (same behavior as `DROPMALFORMED`).
- `DROPMALFORMED`: corrupted records are dropped.
- `FAILFAST` : an *exception* is thrown when corrupted records is found. This method is not recommended in a streaming process as the propagation of the *exception* will potentially make the streaming process fail and stop.

`columnNameOfCorruptRecord` (default: "_corrupt_record")

Permits the configuration of the special field that contains the string value of malformed records. This field can also be configured by setting `spark.sql.columnNameOfCorruptRecord` in the Spark configuration. If both `spark.sql.columnNameOfCorruptRecord` and this option are set, this option takes precedence.

SCHEMA INFERENCE

`inferSchema` (default: `false`)

schema inference is not supported. Setting this option is ignored. Providing a *schema* is mandatory for the *CSV* and *JSON* file sources.

DATE AND TIME FORMATS

`dateFormat` (default "yyyy-MM-dd")

Configures the pattern used to parse `date` fields. Custom patterns follow the formats defined at java.text.SimpleDateFormat.

`timestampFormat` (default "yyyy-MM-dd'T'HH:mm:ss.SSSXXX")

Configures the pattern used to parse `timestamp` fields. Custom patterns follow the formats defined at java.text.SimpleDateFormat.

```
[[json_file_source_format]]

==== _JSON_ File Source Format
```

The *JSON* format support for the *File* source lets us consume text files encoded as *JSON*, where each line in the file is expected to be a valid *JSON* object. The `JSON` records are parsed used the provided schema. Records that do not follow the schema are considered invalid and there are several options available to control the handling of invalid records.

JSON PARSING OPTIONS

By default, the *JSON File Source* expects the file contents to follow the JSON Lines specification [1]. That is, each independent line in the file corresponds to a valid JSON document that complies with the specified schema. Each line should be separated by a *new line* (`\n`) character. A `CRLF` character (`\r\n`) is also supported as trailing white spaces are ignored.

We can tweak the tolerance of the JSON parser to process data that does not fully comply to the standard. It's also possible to change the behavior to handle those records that are considered corrupted. The following options allow for the configuration of the parser behavior:

`allowComments` (default `false`)

> When enabled, comments in Java/C++ style are allowed in the file and the corresponding line will be ignored. e.g.:

```
// Timestamps are in ISO 8601 compliant format
{"id":"x097abba", "timestamp": "2018-04-01T16:32:56+00:00"}
{"id":"x053ba0bab", "timestamp": "2018-04-01T16:35:02+00:00"}
```

Otherwise, comments in the JSON file are considered corrupt records and handled following the `mode`setting.

`allowNumericLeadingZeros` (default `false`)

> when enabled, leading zeros in numbers are allowed. (e.g.: 00314). Otherwise the leading zeros are considered invalid numeric values, the corresponding record is deemned corrupted and it is handled following the `mode` setting.

`allowSingleQuotes` (default `true`)

> allows the use of single quotes to demark fields. When enabled, both single quotes and double quotes are allowed. Regardless of this setting, quotes cannot be nested and must be properly escaped when used within a value.

e.g.:

```
// valid record
{"firstname":"Alice", 'lastname': 'Wonderland'}
// invalid nesting
{"firstname":'Elvis "The King"', 'lastname': 'Presley'}
// correct escaping
{"firstname":'Elvis \"The King\"', 'lastname': 'Presley'}
```

`allowUnquotedFieldNames` (default `false`)

Allows unquoted JSON field names. (e.g.: `{firstname:"Alice"}`). Note that it's not possible to have spaces in field names when using this option. (e.g.: `{first name:"Alice"}` is considered corrupt even when the field name matches the schema). Use with caution.

`multiLine` (default `false`)

When enabled, instead of parsing JSON Lines, the parser will consider the contents of each file as a single valid JSON document and will attempt to parse its contents as records following the defined schema. e.g.:

```
[

  {"firstname":"Alice", "last name":"Wonderland", "age": 7},
  {"firstname":"Coraline", "last name":"Spin"   , "age":15}
]
```

Use this option when the producer of the file can only output complete JSON documents as files. In such cases, use a top-level array to group the records as shown here.

`primitivesAsString` (default `false`)

When enabled, primite value types are considered strings. This allows to read documents having fields of mixed types but all values are read as a `String`. e.g.:

```
{"firstname":"Coraline", "last name":"Spin", "age": 15}
{"firstname":"Diana", "last name":"Prince", "age": "unknown"}
```

In this example, the resulting `age` field is of type `String`, containing values `age="15"` for *"Coraline"* and `age="unknown"` for *"Diana"*

CSV File Source Format

The *comma separated values* (CSV) is a popular tabular data storage and exchange format widely supported by enterprise applications. The `File Source` *CSV* format support allows us to ingest and process the output of such applications in a Structured Streaming. Although the name "CSV" originally indicated that values are separated by commas, often the separation character can be freely configured. There are many configuration options available to control the way data is transformed from plain text to structured records.

In the rest of this section we cover the most common options and, in particular, those that are relevant for the streaming process. For formatting-related options, we refer the reader to the lastest documentation, which is found under the "DataFrameReader#csv" documentation at https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrameReader#csv

CSV PARSING OPTIONS

`comment` (default: "" (disabled))

Configures a character that marks lines considered as comments. e.g. when using `option("comment","#")` we can parse the following CSV with a comment in it:

```
#Timestamps are in ISO 8601 compliant format

x097abba, 2018-04-01T16:32:56+00:00, 55

x053ba0bab, 2018-04-01T16:35:02+00:00, 32
```

`header` (default: `false`)

The first non-comment line of each file is considered to be contain the name for each column. When a schema is provided, the header line is ignored and has no effect. Remember that providing a schema is the recommended practice when developing streaming applications.

`multiline` (default: `false`)

consider each file as one record spawing all the lines in the file.

`quote` (default: " (double quote))

configures the character used to enclose values that contain a column separator.

`sep` (default: ",")

configures the character that separates the fields in each line.

Parquet File Source Format

Apache Parquet is a column-oriented file-based data store format. The internal representation splits original rows into chunks of columns that are stored using compression techniques. As a consequence, queries that require specific columns do not need to read the complete file. Instead, the relevant pieces can be independently addressed and retrieved. Parquet supports complex nested data structures and preserves the schema structure of the data. Due to its enhanced query capabilities, efficient use of storage space and the preservation of schema information, Parquet is a popular format for storing large, complex datasets.

SCHEMA DEFINITION

To create a streaming source from Parquet files, it is sufficient to provide the schema of the data and the directory location. The schema provided during the streaming declaration is fixed for the duration of the streaming source definition.

The following example shows the creation of a Parquet-based File *Source* from a folder in `hdfs://data/folder` using the provided schema.

```
// Use format and load path
val fileStream = spark.readStream
  .schema(schema)
  .parquet("hdfs://data/folder")
```

Text File Source Format

The *text* format for the *File Source* supports the ingestion of plain text files. Using configuration options, it's possible to either ingest the text line by line or the whole file as a single text blob. The schema of the data produced by this source is naturally `StringType` and does not need to be

specified. This is a generic format that can be used to ingest arbitrary text for further programmatic processing, from the famous *word count* to custom parsing of proprietary text formats.

TEXT INGESTION OPTIONS.

Next to the common options for the *File Source* that we saw in "Common Options", the *text* file format supports reading text files as a whole using the `wholetext` option:

`wholetext` (default `false`)

: If true, read the complete file as a single text blob. Otherwise, split the text in lines using the standard line separators (`\n`, `\r\n`, `\r`) and consider each line a record.

`TEXT` AND `TEXTFILE`

The *text* format support offers two API alternatives:

`text`

: Returns a dynamically typed `DataFrame` with a single `value` field of type `StringType`

`textFile`

: Returns a statically typed `Dataset[String]`

The `text` format specification can be used as a terminating method call or as a `format` option. To obtain a statically typed `Dataset`, we must use `textFile` as the last call of the stream builder call as shown next.

```
// Text specified as format
>val fileStream = spark.readStream.format("text").load("hdfs://data/folder")
fileStream: org.apache.spark.sql.DataFrame = [value: string]

// Text specified through dedicated method
>val fileStream = spark.readStream.text("hdfs://data/folder")
fileStream: org.apache.spark.sql.DataFrame = [value: string]

// TextFile specified through dedicated method
val fileStream = spark.readStream.textFile("/tmp/data/stream")
fileStream: org.apache.spark.sql.Dataset[String] = [value: string]
```

The *Kafka Source*

Apache Kafka is a Publish/Subscribe (pub/sub) system based on the concept of a distributed log. Kafka is highly scalable and offers high-throughput, low-latency handling of the data, both at the consumer and producer sides. In Kafka, the unit of organization is a topic. Publishers send data to a topic and subscribers receive that data from the topic they subscribed to, both in a realiable way. Apache Kafka has become a popular choice of messaging infrastructure for a wide range of streaming use cases.

The Structured Streaming *source* for Kafka implements the subcriber role and can consume data published to one or several topics. This is a reliable source. As we recall for our discussion in "Understanding Sources", this means that data delivery semantics are guaranteed even in case of partial or total failure and restart of the streaming process.

Setting up a Kafka Source

To create a Kafka Source, we use the `format("kafka")` method with the `createStream` builder on the Spark *Session*. We need two mandatory parameters to connect to Kafka: the addresses of the Kafka brokers the topic(s) we want to connect to.

The address of the Kafka brokers to connect to is provided throug the option `kafka.bootstrap.servers` as a `String` containing a comma-separated list of `host:port` pairs.

Example 10-1. Creating a Kafka Source

```
>val kafkaStream = spark.readStream
  .format("kafka")
  .option("kafka.bootstrap.servers", "host1:port1,host2:port2,host3:port3")
  .option("subscribe", "topic1")
  .option("checkpointLocation", "hdfs://spark/streaming/checkpoint")
  .load()

kafkaStream: org.apache.spark.sql.DataFrame =
  [key: binary, value: binary ... 5 more fields]

>kafkaStream.printSchema

root
 |-- key: binary (nullable = true)
 |-- value: binary (nullable = true)
 |-- topic: string (nullable = true)
 |-- partition: integer (nullable = true)
 |-- offset: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- timestampType: integer (nullable = true)

>val dataStream = kafkaStream.selectExpr("CAST(key AS STRING)", "CAST(value AS
STRING)")
                            .as[(String, String)]

dataStream: org.apache.spark.sql.Dataset[(String, String)] =
  [key: string, value: string]
```

In Example 10-1 we see the a simple Kafka source definition. It subscribes to a single topic “topic1” by connecting to the brokers located at `host1:port1,host2:port2` and `host3:port3`.

The result of that call is a `DataFrame` with five fields: `key,value,topic,partition,offset,timestamp,timestampType`. This the schema is fixed the *Kafka source*. It provides the raw `key` and `values` from Kafka and the metadata of each consumed record.

Usually, we are only interested in the *key* and *value* of the message. Both the `key` and the `value`contain a binary payload, represented internally as a `Byte Array` When the data is written to Kafka using a `String` serializer, we can read that data back by casting the values to `String` as we have done in the last expression in the example. Although text-based encoding is a common practice, it’s not the most space-efficient way of exchanging data. Other encodings, such as the schema-aware AVRO format, may offer a better space efficiency with the added benefit of embedding the schema information.

The additional metadata in the message, such as *topic*, *partition* or *offset* may be used in more complex scenarios. For example, the `topic` field will contain the *topic* that produced the record and could be used as a label or discriminator, in case we subscribe to several topics at the same time.

Selecting a Topic Subscription Method

There are three different ways to specify the topic or topics we want to consume:
* `subscribe` * `subscribePattern` * `assign`

The Kafka *source* setup must contain one and only one of these subscription options. They provide different levels of flexibility to select the topic(s) and even the partitions to subscribe to:

`subscribe`

> Takes a single topic or a list of comma separated topics: `topic1, topic2, ..., topicn`. This method will subscribe to each topic and create a single, unified stream with the unioned data to all the topics. E.g.: `.option("subscribe", "topic1,topic3")`

`subscribePattern`

> Is similar to `subscribe` in behaviour, but the topics are specified with a regex pattern. For example, if we have topics: `'factory1Sensors', 'factory2Sensors', 'street1Sensors', 'street2Sensors'`, we can subscribe to all "factory" sensors with the expression `.option("subscribePattern", "factory[\\d]+Sensors")`

`assign`

> Allows the fine grained specification of specific partitions per topic to consume. This is known as `TopicPartition`s in the Kafka API. The partitions per topic are indicated using a JSON object, in which each _key_ is a topic and its _value_ is an array of partitions. For example, the option definition: `.option("assign", """{"sensors":[0,1,3]}""")` would subscribe to the partitions 0,1, and 3 of topic *"sensors"*. To use this method we need information about the topic partitioning. The partition information can be obtained programmatically, using the Kafka API, or through configuration.

Configuring *Kafka Source* Options

There are two categories of configuration options for the *Structured Streaming Source for Kafka*: dedicated *source* configuration and pass-through options that are given directly to the undelying Kafka consumer.

KAFKA SOURCE SPECIFIC OPTIONS

The following options configure the behavior of the `Kafka Source`. They relate in particular to how offsets are consumed

`startingOffsets` (default: "latest")

> Accepted values are: "earliest", "latest", or a JSON Object representing an association between topics, their partitions and a given offset. Actual offset values are always positive numbers. There are two special offset values: `-2` denotes "earliest" and `-1` means "latest". E.g.: `""" {"topic1": { "0": -1, "1": -2, "4":1024 }} """

`startingOffsets` are only used the first time a query is started. All subsequent restarts will use the*checkpoint* information stored. To restart a streaming job from a specific offset, we need to remove the contents of the *checkpoint*

`failOnDataLoss` (default: `true`)

> This flag indicates whether to fail the restart of a streaming query in case data might be lost. This is usually when offsets are out of range, topics are deleted or rebalanced. We recommend to set this option to `false` during the develop-test cycle as stop/restart of the query side with a continuous producer will ofter trigger a failure. Enable back to `true` for production deployment.

`kafkaConsumer.pollTimeoutMs` (default: `512`)

> The poll timeout (in milliseconds) to wait for data from Kafka in the distributed consumers running on the Spark executors.

`fetchOffset.numRetries` (default: `3`)

> The of retries before failing the fetch of Kafka offsets.

`fetchOffset.retryIntervalMs` (default: `10`)

> delay between offset fetch retries in milliseconds.

`maxOffsetsPerTrigger` (default: not set)

> This option allows to set a rate limit to the number of total records to be consumed at each query trigger. The limit configured will be equally distributed among the set of partitions of the subscribed topics.

Kafka Consumer Options

It's possible to pass configuration options through to the underlying Kafka *consumer* in the this *source*. This is done adding a `'kafka.'` prefix to the configuration key we want to set.

For example, to configure *transport level security* (TLS) options for the Kafka Source, we can set the *Kafka consumer* configuration option `security.protocol` by setting `kafka.security.protocol` in the *source* configuration.

In Example 10-2, we can see how to configure TLS for the Kafka source using this method.

Example 10-2. Kafka Source TLS Configuration Example

```
val tlsKafkaSource = spark.readStream.format("kafka")
  .option("kafka.bootstrap.servers", "host1:port1, host2:port2")
  .option("subscribe", "topsecret")
  .option("kafka.security.protocol", "SSL")
  .option("kafka.ssl.truststore.location", "/path/to/truststore.jks")
  .option("kafka.ssl.truststore.password", "truststore-password")
  .option("kafka.ssl.keystore.location", "/path/to/keystore.jks")
  .option("kafka.ssl.keystore.password", "keystore-password")
  .option("kafka.ssl.key.password", "password")
  .load()
```

NOTE

For an exhaustive listing of the Kafka consumer configuration options, we refer the reader to the official Kafka Documentation

BLACKLISTED CONFIGURATION OPTIONS

Not all options from the standard *consumer* configuration can be used because they conflict with the internal process of the *source*, which is controlled with the settings we saw in "*Kafka Source* specific options".

These options are black-listed. This means that attempting to use any of them will result in an `IllegalArgumentException`.

option	Reason	Alternative
`auto.offset.reset`	Offsets are managed within Structured Streaming	use `startingOffsets` instead
`enable.auto.commit`	Offsets are managed within Structured Streaming	
`group.id`	an unique group id is managed internally per query	
`key.deserializer`	payload is always represented as a `Byte Array`	Deserialization into specific formats is done programmatically
`value.deserializer`	payload is always represented as a `Byte Array`	Deserialization into specific formats is done programmatically
`interceptor.classes`	A Consumer interceptor might break the internal data representation	

Table 10-1. Blacklisted kafka options

The *Socket* Source

The Transmission Control Protocol or TCP is a connection-oriented protocol that enables bi-directional communication between a client and a server. This protocol underpins many of the higher-level communication protocols over the Internet, such as FTP, HTTP, MQTT and many others. While application layer protocols, like HTTP, add additional semantics on top of a TCP connection, there are many applications that offer a vanilla, text-based TCP connection over a UNIX socket to deliver data.

The *Socket source* is a TCP socket client able to connect to a TCP server offering a UTF-8 encoded text-based data stream. It connects to a TCP server using a `host` and `port` provided as mandatory `options`.

Configuration

To connect to a TCP server, we need the address of the host and a port number. It's also possible to configure the *Socket* source to add the timestamp at which each line of data received.

These are the configuration options:

`host` (mandatory)

: The DNS hostname or IP Address of the TCP server to connect to.

`port` (mandatory)

: The port number of the TCP server to connect to.

`includeTimestamp` (default: `false`)

: When enabled, the Socket source adds the timestamp of arrival to each line of data. It also changes the schema produced by this *source*, adding the `timestamp` as an additional field.

Example 10-3. `Socket` source examples

```
// Only using host and port

>val stream = spark.readStream
  .format("socket")
  .option("host", "localhost")
  .option("port", 9876)
  .load()

18/04/14 17:02:34 WARN TextSocketSourceProvider:
The socket source should not be used for production applications! It does not support
recovery.

stream: org.apache.spark.sql.DataFrame = [value: string]

// With added timestamp information

val stream = spark.readStream
  .format("socket")
  .option("host", "localhost")
  .option("port", 9876)
  .option("includeTimestamp", true)
  .load()

18/04/14 17:06:47 WARN TextSocketSourceProvider:
The socket source should not be used for production applications! It does not support
recovery.

stream: org.apache.spark.sql.DataFrame = [value: string, timestamp: timestamp]
```

In Example 10-3 we observe the two modes of operation that this *source* offers. With the `host`,`port`configuration, the resulting streaming `DataFrame` has a single field named `value` of `String` Type. When we add the flag `includeTimestamp` set to `true`, the schema of the resulting streaming `DataFrame` contains the fields `value` and `timestamp`, where `value` is of

type `String` as before and `timestamp` is of type `Timestamp`. Also, observe the log warning this source prints at creation.

Operations

The *Socket* source creates a TCP client that connects to a TCP server specified in the configuration. This client runs on the Spark Driver. It keeps the incoming data in memory until it is consumed by the query and the corresponding offset is committed. The data from committed offsets is evicted, keeping the memory usage stable under normal circumstances.

As we recall from the discussion in "Understanding Sources", a *source* is considered *reliable* if it can replay an uncommitted offset even in the event of failure and restart of the streaming process. This *source* is not considered to be reliable, as a failure of the Spark driver will result in losing all uncommitted data in memory.

This *source* may only be used in case data loss is acceptable for the use case.

NOTE

A common architectural alternative to directly connecting to a TCP server using the *socket source* is to use Kafka as a reliable intermediate storage. A robust microservice can be used to bridge the TCP server and Kafka. This microservice collects the data from the TCP server and delivers it atomically to Kafka. Then, we can use the reliable Kafka *source*to consume the data and further process it in Structured Streaming.

The *Rate* Source

The *Rate* source is a internal stream generator that produces a sequence of records at a configurable frequency, given in `records/second`. The output is a stream of records `(timestamp, value)` where `timestamp` corresponds to the moment of generation of the record and `value` is an ever increasing counter.

```
> val stream = spark.readStream.format("rate").load()

stream: org.apache.spark.sql.DataFrame = [timestamp: timestamp, value: bigint]
```

It is intended for benchmarks and for exploring Structured Streaming, given that it does not rely on external systems to work. As we can appreciate in the previous example, it is very easy to create and completely self-contained.

Example 10-4. Rate Source Example

```
> val stream = spark.readStream.format("rate")
  .option("rowsPerSecond", 100)
  .option("rampUpTime",60)
  .load()
stream: org.apache.spark.sql.DataFrame = [timestamp: timestamp, value: bigint]
```

The code in Example 10-4 creates a *rate* stream of 100 rows per seconds with a ramp up time of 60 seconds. The schema of the resulting `Dataframe` contains two fields: `timestamp` of

type `Timestamp` and `value`, which is a `BigInt` at schema level and a `Long` in the internal representation.

Options

The *Rate* source supports few options that control the its throughput and level of parallelism:

`rowsPerSecond` (default: 1)

: The number of rows to generate each second

`rampUpTime` (default: 0)

: At the start of the stream, the generation of records will increase progressively until this time has been reached. The increase is linear.

`numPartitions` (default: default spark parallelism level)

: The number of partitions to generate. More partitions increase the parallelism level of the record generation and downstream query processing.

[1] *http://jsonlines.org/*

Chapter 11. Structured Streaming Sinks

In the previous chapter, we learnt about *Sources*, the abstraction that allows Structured Streaming to acquire data for processing. Once that data has been processed, we would want to do something with it. We might want to write it to a database for later querying, a file for further (batch) processing, or another streaming backend to keep the data in movement.

In Structured Streaming, *sinks* are the abstraction that represents how to produce data to an external system. Structured Streaming comes with several built-in *sources* and defines an API that let us create custom *sinks* to other systems that are not natively supported.

In this chapter we are going to learn how a sink works, review the details of the sinks provided by Structured Streaming and explore how to create custom sinks to write data to systems not supported by the default implementations.

Understanding *Sinks*

Sinks serve as output adaptors between the internal data representation in Structured Streaming and external systems. They provide a write path for the data resulting from the stream processing. Additionally, they must also close the loop of reliable data delivery.

In order to participate in the end-to-end reliable data delivery, *sinks* must provide an *idempotent* write operation. *Idempotent* means that the result of executing the operation two or more times is equal to executing the operation once. When recovering from a failure, Spark might reprocess some data that was partially processed at the time the failure occured. At the side of the *source*, this is done by using the *replay* functionality. As we can recall from “Understanding Sources”, reliable *sources* must provide means of replaying uncommitted data, based on a given offset. Likewise, *sinks* must provide the means of removing duplicated records before those are written to the external source.

The combination of a *replayable source* and an *idempotent sink* is what grants Structured Streaming its *effectively exaclty once* data delivery semantics. *Sinks* that are not be able to implement the idempotent requirement will result in end-to-end delivery guaranties of at most “at least once” semantics. *Sinks*that are not able to recover from the failure of the streaming process are deemned “unreliable” as they might lose data.

In the next section we go in detail over the available *sinks* in Structured streaming.

NOTE

3rd party vendors might provide custom Structured Streaming *sinks* for their products. When integrating one of these external *sinks* in your project, consult their documentation to determine the data delivery warranties they support.

Available Sinks

Structured Streaming comes with several *sinks* that match the supported *sources* as well as *sinks* that let us output data to temporary storage or to the console. In rough terms, we can divide the provided *sinks* into: realiable and learning/experimentation support. In addition, it also offers a programmable interface that allows us to interface with arbitrary external systems.

Reliable Sinks

The sinks considered reliable or *production-ready* provide well-defined data delivery semantics and are resilient to total failure of the streaming process.

The following are the provided *reliable* sinks:

The File Sink

writes data to files in a directory in the filesystem. It supports the same file formats than the *File Source*: JSON, Parquet, CSV and Text.

The Kafka Sink

writes data to Kafka, effectively keeping the data "on the move". This is an interesting option to integrate the results of our proceess with other streaming frameworks that rely on Kafka as a the data backbone.

Sinks for Experimentation

The following sinks are provided to support interaction and experimentation with Structured Streaming. They do not provide failure recovery and therefore, their use in production is discourage as it may result in data loss.

The following are non-reliable sinks:

The Memory Sink

creates a temporary table with the results of the streaming query. The resulting table can be queried within the same JVM process, which allows in-cluster query to access the results of the streaming process.

The Console Sink

prints the results of the query to the console. This is useful at development time to visually inspect the results of the stream process.

The Sink API

Next to the built-in sinks, we also have the possibility to create a *sink* programmatically. This is achieved with the *foreach* operation that, as its name implies, offers access to each individual resulting record of the output stream. Finally, it is possible to develop our own custom *sinks* using the *sink* API directly.

Exploring Sinks in Detail

In the rest of this chapter we are going to explore the configuration and options available for each sink. We present an in-depth coverage of the reliable sinks that should provide a thorough view of their applicability and may serve as reference when you start developing your own applications.

The experimental *sinks* are limited in scope and that is also reflected in the level of coverage that follows.

Towards the end of this chapter, we will explore the custom *sink* API options and review the considerations we need to take when developing our own sinks.

TIP

If you are in your initial exploration phase of Structured Streaming, you may want to skip this section and come back to it later, when you are busy developing your own Structured Streaming jobs.

The `File` Sink

Files are a common inter-system boundary. When used as a sink for a streaming process, they allow the data to become *at rest* after the stream-oriented processing. Those files can become part of a *data lake* or can be consumed by other (batch) processes as part of a larger processing pipeline that combines streaming and batch modes.

Scalable, reliable and distributed file systems — such as HDFS or object stores, like Amazon's S3 — make it possible to store large datasets as files in arbitrary formats. When running in local mode, at exploration or development time, it's possible to use the local file system for this sink.

The `File Sink` supports the same formats as the `File Source`:

- CSV
- JSON
- Parquet
- ORC
- Text

NOTE

Structured Streaming shares the same File *Data Source* implementation used in batch mode. The write options offered by the `DataFrameWriter` for each File format are also available in streaming mode. In this section, we highlight the most commonly used options. For the most up-to-date list, always consult the online documentation for your specific Spark version.

Before going into the details of each format, let's explore a general *file sink* example:

Example 11-1. File Sink Example

```
// assume an existing streaming dataframe df
```

```
val query = stream.writeStream
  .format("csv")
  .option("sep", "\t")
  .outputMode("append")
  .trigger(Trigger.ProcessingTime("30 seconds"))
  .option("path","<dest/path>")
  .option("checkpointLocation", "<checkpoint/path>")
  .start()
```

In this example, we are using the `csv` format to write the stream results to the `<dest/path>`destination directory using `TAB` as the custom separator. We also specify a `checkpointLocation`, where the checkpoint metadata is stored at regular intervals.

The *File Sink* only supports `append` as `outputMode` and it can be safely omited in the `writeStream`declaration. Attempting to use another mode will result in the following exception when the query starts: `org.apache.spark.sql.AnalysisException: Data source ${format} does not support ${output_mode} output mode;`

Using Triggers with the File Sink

One additional parameter we see in the listing Example 11-1 is the use of a `trigger`. When no trigger is specified, Structured Streaming will start the processing of a new batch as soon as the previous one is finished. In the case of the *File sink*, and depending on the throughput of the input stream, this might result in the generation of many small files. This might be detrimental for the file system storage capacity and performance.

Consider the example Example 11-2:

Example 11-2. Rate Source with File Sink

```
val stream = spark.readStream.format("rate").load()
val query = stream.writeStream
  .format("json")
  .option("path","/tmp/data/rate")
  .option("checkpointLocation", "/tmp/data/rate/checkpoint")
  .start()
```

If we let this query run for a little while, and then we check the target directory, we should observe a large number of small files.

```
$ ls -1
part-00000-03a1ed33-3203-4c54-b3b5-dc52646311b2-c000.json
part-00000-03be34a1-f69a-4789-ad65-da351c9b2d49-c000.json
part-00000-03d296dd-c5f2-4945-98a8-993e3c67c1ad-c000.json
part-00000-0645a678-b2e5-4514-a782-b8364fb150a6-c000.json
...

# Count the files in the directory
$ ls -1 | wc -l
562

# A moment later
$ ls -1 | wc -l
608

# What's the content of a file?
```

```
$ cat part-00007-e74a5f4c-5e04-47e2-86f7-c9194c5e85fa-c000.json
{"timestamp":"2018-05-13T19:34:45.170+02:00","value":104}
```

As we learnt in the *Sources* chapter, the *rate* source generates one record per second by default. When we see the data contained in one file, we can indeed see that single record. The query is, in fact, generating one file each time new data is available. Although the contents of that file are not large, file systems occur in an overhead to keep track of the number of files in the file system. Even more, in the Hadoop File System, HDFS, each file will occupy a block and be replicated `n` times regardless of the contents. Given that the typical HDFS block is 128MB, we can see how our naive query that uses a *File sink* can quickly deplete our storage.

The `trigger` configuration is there to help us avoid this situation. By providing time triggers for the production of files, we can ensure that we have a reasonably sufficient amount of data in each file.

We can observe the effect of a time `trigger` by modifying our previous example as following:

```
import org.apache.spark.sql.streaming.Trigger

val stream = spark.readStream.format("rate").load()
val query = stream.writeStream
  .format("json")
  .trigger(Trigger.ProcessingTime("1 minute")) // <-- Add Trigger configuration
  .option("path","/tmp/data/rate")
  .option("checkpointLocation", "/tmp/data/rate/checkpoint")
  .start()
```

Let's issue the query and wait for a couple of minutes. When we inspect the target directory, we should see considerably less number of files than before and each file should contain more records. The number of records per file will depend of the `DataFrame` partitioning.

```
$ ls -1
part-00000-2ffc26f9-bd43-42f3-93a7-29db2ffb93f3-c000.json
part-00000-3cc02262-801b-42ed-b47e-1bb48c78185e-c000.json
part-00000-a7e8b037-6f21-4645-9338-fc8cf1580eff-c000.json
part-00000-ca984a73-5387-49fe-a864-bd85e502fd0d-c000.json
...

# Count the files in the directory
$ ls -1 | wc -l
34

# Some seconds later
$ ls -1 | wc -l
42

# What's the content of a file?

$ cat part-00000-ca984a73-5387-49fe-a864-bd85e502fd0d-c000.json
{"timestamp":"2018-05-13T22:02:59.275+02:00","value":94}
{"timestamp":"2018-05-13T22:03:00.275+02:00","value":95}
{"timestamp":"2018-05-13T22:03:01.275+02:00","value":96}
{"timestamp":"2018-05-13T22:03:02.275+02:00","value":97}
{"timestamp":"2018-05-13T22:03:03.275+02:00","value":98}
{"timestamp":"2018-05-13T22:03:04.275+02:00","value":99}
{"timestamp":"2018-05-13T22:03:05.275+02:00","value":100}
```

If you are trying this example on a personal computer, the number of partitions defaults to the number of cores present. In our case, we have eight cores, and we observe 7 or 8 records per partition. That's still very few records, but it shows the principle that can be extrapolated to real scenarios.

While a `trigger` based on the number of records or the size of the data would arguably be more interesting in this scenario, currently, only time-based triggers are supported. This might change in the future, as Structured Streaming evolves.

Common Configuration Options Across All Supported File Formats

We have already seen in previous examples the use of the the method `option` that takes a *key* and a *value* to set configuration options in a sink.

All supported file formats share the following configuration options:

`path`

A directory in a target filesystem where the streaming query writes the data files.

`checkpointLocation`

A directory in a resilient filesystem where to store the checkpointing metadata. New checkpoint information is written with each query execution interval. `compression` (default: None): All supported file formats share the capability to compress the data, although the available compression `codecs` might differ from format to format. The specific compression algorithms are shown for each format in their corresponding section.

NOTE

When configuring options of the File Sink, it's often useful to remember that any file written by the File Sink can be read back using the corresponding File Source. For example, in the previous chapter we saw that it normally expects each line of the file to be a valid JSON document. Likewise, the JSON sink format will produce files containing one record per line.

Common Time and Date Formatting (CSV, JSON)

Text-based files formats, such as CSV and JSON, accept custom formatting for *date* and *timestamp*data types.

`dateFormat` (default: `yyyy-MM-dd`)

Configures the pattern used to format `date` fields. Custom patterns follow the formats defined at java.text.SimpleDateFormat.

`timestampFormat` (default "yyyy-MM-dd'T'HH:mm:ss.SSSXXX")

Configures the pattern used to format `timestamp` fields. Custom patterns follow the formats defined at java.text.SimpleDateFormat.

`timeZone` (default: local timezone)

Configures the time zone to use to format timestamps.

The *CSV* Format of the File Sink

Using the CSV File format, we can write our data in a ubiquitous tabular format that can be read by many programs, from spreadsheet applications to a wide range of enterprise software.

OPTIONS

The CSV support in Spark supports many options to control the field separator, quoting behavior and the inclusion of a header. In addition to that, the common *File Sink* options and the date formatting options apply to the CSV sink.

NOTE

In this section, we list the most commonly used options. For a comprehensive list, check the online documentation at: *https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrameWriter*

`header` (default: `false`)

A flag to indicate whether we should include a header to the resulting file. The header consists of the name of the fields in this streaming dataframe.

`quote` (default: `"` (double quote))

sets the character used to quote records. Quoting is necessary when records may contain the separator character, which, without quoting, would result in corrupt records.

`quoteAll` (default: `false`)

A flag used to indicate whether all values should be quoted or only those that contain a separator character. Some external systems require all values to be quoted. When using the CSV format to import the resulting files in an external systems, check that system's import requirements to correctly configure this option.

`sep` (default: `,` (comma))

configures the separator character used between fields. The separator must be a single character. Otherwise the query will throw a `IllegalArgumentException` at runtime when it starts.

The *JSON* File Sink Format

The *JSON File Sink* lets us write output data to files using the JSON lines format. This format transforms each record in the output dataset into a valid JSON document written in a line of text. The *JSON* File Sink is symmetrical to the *JSON Source*. As we would expect, files written with this format can be read again using the *JSON Source*.

NOTE

When using 3rd party JSON libraries to read the resulting files, we should take care of first reading the file(s) as lines of text and then parse each line as a JSON document representing one record.

OPTIONS

Next to the common file and text format options, the *JSON* sink suport these specific configurations:

`encoding` (default: `UTF-8`)

Configures the charset encoding used to write the JSON files

`lineSep` (default `\n`)

Sets the line separator to be used between JSON records.

Supported `compression` options (default: `none`): `none`, `bzip2`, `deflate`, `gzip`, `lz4`, and `snappy`.

The *Parquet* File Sink Format

The Parquet file sink supports the common *File sink* configuration and does not have format-specific options.

Supported `compression` options (default: `snappy`): `none`, `gzip`, `lzo` and `snappy`

The *Text* File Sink Format

The text file sink writes plain text files. While the other file formats would perform a conversion from the streaming `DataFrame` or `Dataset` schema to the particular file format structure, the *text* sink expects either a flattened streaming `Dataset[String]` or a streaming `DataFrame` with a schema containing a single `value` field of `StringType`.

The typical use of the text file format is to write custom text-based formats not natively supported in Structured Streaming. To achieve this goal, we first transform programmatically the data into the desired text representation. After that, we use the *Text format* to write the data to files. Attempting to write any complex schema to a text sink will result in an error.

OPTIONS

Next to the common options for sinks and text-based formats, the *text* sink supports the following configuration option:

`lineSep` (default: `\n`)

Configures the line separator used for terminating each text line.

The Kafka Sink

As we discussed in “The *Kafka Source*”, Kafka is a *Publish/Subscribe* (pub/sub) system. While the *Kafka Source* functions as a subscriber, the *Kafka* sink is the *publisher* counterpart. The *Kafka Sink*allows us to write data (back) to Kafka, which can be then consumed by other subscriber to continue a chain of streaming processors.

Downstream consumers might be other streaming processors, implemented using Structured Streaming or any of the other streaming frameworks available, or (micro) services that consume the streaming data to fuel applications in the enterprise ecosystem.

Understanding Kafka Publish model

In Kafka, data is represented as *(key,value)* records exchanged over a topic. Topics are composed of distributed partitions. Each partition maintains the messages in the order they were received. This ordering is indexed by an offset, which is, in turn, is used by the consumer to indicate the record(s) to read. When a record is published to a topic, it's placed in a partition of a topic. The choice of the partition depends on the key. The ruling principle is that a records with the same key will land in the same partition. As a result, the ordering in Kafka is partial. Sequence of records from a single partition will be ordered sequentially by arrival time but there are no ordering warranties among partitions.

This model is highly scalable and Kafka implementation ensures low-latency reads and writes, making it an excellent carrier for streaming data.

Using the Kafka Sink

After learning about the Kafka publishing model, we can look at the practical side of producing data to Kafka. We just saw that the Kafka records are structured as key-value pairs. We need to structure our data in the same shape.

In a minimal implementation, we must ensure that our streaming `DataFrame` or `Dataset` has a `value` field of `BinaryType` or `StringType`. The implication of this requirement is that we usually need to encode our data into a transport representation before sending it to Kafka.

When the key is not specified, Structured Streaming will replace the `key` with `null`. This makes the *Kafka sink* use a round-robin assigment of the partition for the corresponding topic.

If we want to preserve control over the key assignment, we must have a `key` field, also of `BinaryType` or `StringType`. This `key` will be used for the partition assigment, resulting in a guaranteed ordering between records with equal keys.

Optionally, we can control the destination topic at record level by adding a `topic` field. If present, the `topic` value must correspond to an existing Kafka topic. The related record will be published to that topic. This option is useful when implementing a fan-out pattern in which incoming records are sorted in different dedicated topics for further processing. Think for example about classifying incoming support tickets into dedicated *sales*, *technical*, and *troubleshooting* topics that are consumed downstream by their corresponding (micro) service.

Once we have the data in the right shape, we also need the address of the target bootstrat servers broker in order to connect to the brokers.

In practical terms, this generally involves two steps: 1. Transform each record as a single field called *value* and optionally assign a key and a topic to each record. 1. Declare our stream sink using the `writeStream` builder.

Let's use these steps in an example:

Example 11-3. Kafka Sink Example

```
// Assume an existing streaming dataframe 'sensorData'
// with schema: id: String, timestamp: Long, sensorType: String, value: Double

// Create a key and a value from each record:

val kafkaFormattedStream = sensorData.select(
  $"id" as "key",
  to_json(
    struct($"id", $"timestamp", $"sensorType", $"value")
  ) as "value"
)

// In step two, we declare our streaming query:

val kafkaWriterQuery = kafkaFormat.writeStream
  .queryName("kafkaWriter")
  .outputMode("append")
  .format("kafka") // determines that the kafka sink is used
  .option("kafka.bootstrap.servers", kafkaBootstrapServer) // comma-separated list of
host:port
  .option("topic", targetTopic)
  .option("checkpointLocation", "/path/checkpoint")
  .option("failOnDataLoss", "false") // use this option when testing
  .start()
```

When we add `topic` information at record level, we must omit the *topic* configuration option.

In this next example, we modify the previous code to write each record to a dedicated topic matching the *sensorType*. That is, all `humidity` records go to the *humidity* topic, all `radiation` records go to the *radiation* topic, etc.

Example 11-4. Kafka Sink to Different Topics

```
// assume an existing streaming dataframe 'sensorData'
// with schema: id: String, timestamp: Long, sensorType: String, value: Double

// Create a key, value and topic from each record:

val kafkaFormattedStream = sensorData.select(
  $"id" as "key",
  $"sensorType" as "topic",
  to_json(struct($"id", $"timestamp", $"value")) as "value"
)

// In step two, we declare our streaming query:

val kafkaWriterQuery = kafkaFormat.writeStream
  .queryName("kafkaWriter")
  .outputMode("append")
  .format("kafka") // determines that the kafka sink is used
  .option("kafka.bootstrap.servers", kafkaBootstrapServer) // comma-separated list of
host:port
  .option("checkpointLocation", "/path/checkpoint")
  .option("failOnDataLoss", "false") // use this option when testing
  .start()
```

Note how we have removed the option `("topic", targetTopic)` and added a `topic` field to each record. This will result in each record being routed to the topic corresponding to its `sensorType`

CHOOSING AN ENCODING

When we look closely at the code in Example 11-3 we see that we create a single *value* field by converting the existing data into its JSON representation. In Kafka, each record consist of a *key* and a *value*. The *value* field contains the payload of the record. In order to send or receive a record of arbitrary complexity to/from Kafka, we need to convert such record into single field representation that we can *fit* into this *value* field. In Structured Streaming, the conversion to/from this transport *value* representation to the actual record must be done through user code. Ideally, the encoding we choose can be easily transformed to an *structured* record to take advantage of the Spark capabilities to manipulate data.

A common encoding format is *JSON*. *JSON* has native support in the structured APIs of Spark and that extends to Structured Streaming. As we saw in our example above, we write JSON by using the SQL function `to_json`: `to_json(struct($"id", $"timestamp", $"value")) as "value")`.

Binary representations such as AVRO and ProtoBuffers are also possible. In such cases, we treat the *value* field as a `BinaryType` and use 3rd party libraries to do the encoding/decoding.

At the moment of writing, there is no built-in support for binary encodings but AVRO support has been announced for an upcoming version.

WARNING

An important factor to consider when choosing an encoding format is schema support. In a multi-service model that uses Kafka as the communications backbone it's typical to find services producing data that use a different programming model, language and/or framework than a streaming processor or other service that consumes it.

To ensure interoperability, schema-oriented encodings are the prefered choice. Having a schema definition allows for the creation of artifacts in different languages and ensures that produced data can be consumed later on.

The Memory Sink

The memory sink is a non-reliable sink that persist the results of the stream processing in an in-memory temporary table. It is consider non-reliable because all data will be lost in the case the streaming process ends but it's certainly useful in scenarios where low-latency access to the streaming results are is required.

The temporary table created by this sink is named after the query name. This table is backed up by the streaming query and will be updated at each trigger following the semantics of the chosen `outputMode`.

The resulting table contains an up-to-date view of the query results and can be queried using classical Spark SQL operations. The query must be executed in the same process (JVM) where the Structured Streaming query is started.

The table maintained by the `Memory Sink` can be accessed interactively. That property makes it an ideal interface with interactive data exploration tools, like the Spark REPL or a notebook.

Another common use is to provide a query service on top of the streaming data. This is done by combining an server module, like an HTTP server with the Spark driver. Calls to specific HTTP endpoints can then be served with data from this in-memory table.

The following example assumes a `sensorData` streaming dataset. The result of the stream processing is materialized in this in-memory table, which is available in the SQL context as `sample_memory_query`

Example 11-5. Memory Sink Example

```
val sampleMemoryQuery = sensorData.writeStream
  .queryName("sample_memory_query")    // this query name will be the SQL table name
  .outputMode("append")
  .format("memory")
  .start()

// After the query starts we can access the data in the temp table
val memData = session.sql("select * from sample_memory_query")
memData.count() // show how many elements we have in our table
```

Output Modes

The *Memory Sink* supports all output modes: `Append`, `Update`, and `Complete`. Hence it can be used with all queries, including aggregations. The combination of the *Memory Sink* with the *Complete* mode is particularly interesting, as it provides a fast, in-memory queriable store for the up-to-date computed complete state. Note that for a query to support *Complete* state, it must aggregate over a bounded-cardinality key. That is to ensure that the memory requirement to handle the state are likewise bounded within the system resources.

The Console Sink

For all of us who love to print *"hello world"* to the screen output, we have the *Console Sink*. Indeed, the *Console Sink* lets us print a small sample of the results of the query to the standard output.

Its use is limited to debugging and data exploration in an interactive shell-based environment, such as the *spark-shell*. As we would expect, this sink is not reliable as does not commit any data to another system.

The use of the *console sink* in production environments should be avoided, much like `println`s are frown upon from an operational code base.

Options

The optiond configurable for the Console Sink are:

`numRows` (default: `20`)

The maximum number of rows to show at each query trigger.

`truncate` (default: `true`)

A flag that indicates whether the output of each cell in a row should be truncated.

Output Modes

As of Spark 2.3, the *Console Sink* supports all output modes: `Append`, `Update` and `Complete`.

The `Foreach` Sink

There are times when we need to integrate our stream processing applications with legacy systems in the enterprise. Also, as a young project, the range of available sinks in Structured Streaming is rather limited.

The `foreach` sink consists of an API and sink definition that provides access to the results of the query execution. It extends the writing capabilities of Structured Streaming to any external system that provides a JVM client library.

The `ForeachWriter` Interface

To use the *Foreach Sink* we must provide an implementation of the `ForeachWriter` interface. The `ForeachWriter` controls the lifecycle of the writer operation. It's execution takes place distributed on the executors and the methods are called for each partition of the streaming `DataFrame` or `Dataset`.

Example 11-6. The API Definition of the ForeachWriter

```
abstract class ForeachWriter[T] extends Serializable {

  def open(partitionId: Long, version: Long): Boolean

  def process(value: T): Unit

  def close(errorOrNull: Throwable): Unit

}
```

As we can see in Example 11-6, the `ForeachWriter` is bound to a type `[T]` which corresponds to the type of the streaming `Dataset` or to `spark.sql.Row` in case of a streaming `DataFrame`. Its API consists of three methods: `open`, `close`, and `process`:

`open`

is called at every trigger interval with the `partitionId` and a unique `version` number. Using these two parameters, the `ForeachWriter` must decide whether or not to process the partition being offered. Returning `true` will lead to the processing of each element using the logic in the `process` method. If the method returns `false`, the partition will be skipped for processing.

`process`

provides access to the data, one element at the time. The function applied to the data must produce a side-effect, such as inserting the record in a database, calling a REST API or using a networking library to communicate the data to another system.

`close`

is called to notify the end of writing a partition. The `error` object will be null when the output operation terminated successfully for this partition or will contain an `Throwable` otherwise.`close` is called at the end of every partition writing operation, even when `open` returned `false`(to indicate that the partition should not be processed).

This contract is part of the data delivery semantics, as it allows to remove duplicated partitions that might have been already sent to the sink but are re-processed by Structured Streaming as part of a recovery scenario. For that mechanism to properly work, the sink must implement some persistent way to remember the `partition/version` combinations that it has already seen.

Once we have our `ForeachWriter` implementation, we use the customary `writeStream` method of declaring a sink and we call the dedicated `foreach` method with the `ForeachWriter` instance.

The `ForeachWriter` implementation must be `Serializable`. This is mandatory because the `ForeachWriter` is executed distributedly on each node of the cluster that contains a partition of the streaming `Dataset` or `DataFrame` being processed. At runtime, a new deserialized copy of the provided `ForeachWriter` instance will be created for each each partition of the `Dataset` or `DataFrame`. As a consequence, we may not pass any state in the initial constructor of the `ForEachWriter`.

Let put this all together in a small example that shows how the *foreach sink* works and illustrates the subtle intricacies of dealing with the state handling and *serialization* requirements.

TCP Writer Sink: A Practical `ForeachWriter` example

For this example, we are going to develop a text-based TCP Sink that transmits the resulst of the query to an external TCP Socket receiving server. In this example, we will be using the `spark-shell`utility that comes with the Spark installation.

In Example 11-7 we create a simple *TCP* client that can connect and write text to a server socket, provided its `host` and `port`. Note that this class is not `Serializable`. `Socket`s are inherently non-serializable, as they are dependent of the underlying system IO ports.

Example 11-7. TCP Socket Client

```
class TCPWriter(host:String, port: Int) {
  import java.io.PrintWriter
  import java.net.Socket
  val socket = new Socket(host, port)
  val printer = new PrintWriter(socket.getOutputStream, true)
  def println(str: String) = printer.println(str)
  def close() = {
    printer.flush()
    printer.close()
    socket.close()
  }
}
```

Next, in Example 11-8, we use this `TCPWriter` in a `ForeachWriter` implementation.

Example 11-8. **`TCPForeachWriter`** *implementation*

```
import org.apache.spark.sql.ForeachWriter
class TCPForeachWriter(host: String, port: Int)
    extends ForeachWriter[RateTick] {

  @transient var writer: TCPWriter = _
  var localPartition: Long = 0
  var localVersion: Long = 0

  override def open(
      partitionId: Long,
      version: Long
    ): Boolean = {
    writer = new TCPWriter(host, port)
    localPartition = partitionId
    localVersion = version
    println(
      s"Writing partition [$partitionId] and version[$version]"
    )
    true // we always accept to write
  }

  override def process(value: RateTick): Unit = {
    val tickString = s"${v.timestamp}, ${v.value}"
    writer.println(
      s"$localPartition, $localVersion, $tickString"
    )
  }

  override def close(errorOrNull: Throwable): Unit = {
    if (errorOrNull == null) {
      println(
        s"Closing partition [$localPartition] and version[$localVersion]"
      )
      writer.close()
    } else {
      print("Query failed with: " + errorOrNull)
    }
  }
}
```

Pay close attention to how we have declared the `TCPWriter` variable: `@transient var writer:TCPWriter = _`. `@transient` means that this reference should not be serialized. The initial value is `null` (using the empty variable initialization syntax _). It's only in the call to `open` that we create an instance of `TCPWriter` and assign it to our variable for later use.

Also note how the `process` method takes an object of type `RateTick`. Implementing a `ForeachWriter` is easier when we have a typed `Dataset` to start with as we deal with a specific object structure instead of `spark.sql.Row`s, which are the generic data container for *streaming DataFrames*. In this case, we transformed the initial streaming `DataFrame` to a typed `Dataset[RateTick]` before proceeding to the sink phase.

Now, to complete our example, we will create a simple `Rate` source and write the produced stream directly to our newly developed `TCPForeachWriter`.

```
case class RateTick(timestamp: Long, value: Long)

val stream = spark.readStream.format("rate")
                  .option("rowsPerSecond", 100)
                  .load()
                  .as[RateTick]

val writerInstance = new TCPForeachWriter("localhost", 9876)

val query = stream
      .writeStream
      .foreach(writerInstance)
      .outputMode("append")
```

Before starting our query, we run a simple TCP server to observe the results. For this purpose, we use a `nc`, a useful *nix command to create TCP/UDP clients and servers in the command line. In this case, we will use a TCP server listening to *port* `9876`.

```
nc -lk 9876
```

Finally, we start our query:

```
val queryExecution = query.start()
```

In the shell running the `nc` command we should see output like:

```
5, 1, 1528043018, 72

5, 1, 1528043018, 73

5, 1, 1528043018, 74

0, 1, 1528043018, 0

0, 1, 1528043018, 1

0, 1, 1528043018, 2

0, 1, 1528043018, 3

0, 1, 1528043018, 4

0, 1, 1528043018, 5

0, 1, 1528043018, 6

0, 1, 1528043018, 7

0, 1, 1528043018, 8

0, 1, 1528043018, 9
```

```
0, 1, 1528043018, 10

0, 1, 1528043018, 11

7, 1, 1528043019, 87

7, 1, 1528043019, 88

7, 1, 1528043019, 89

7, 1, 1528043019, 90

7, 1, 1528043019, 91

7, 1, 1528043019, 92
```

In the output, the first column is the `partition`, and the second is the `version`, followed by the data produced by the `Rate` source. It's interesting to note that the data is ordered within a partition, like `partition 0` in our example, but there is no ordering guarantees among different partitions. Partitions are processed in parallel in different machines of the cluster. There's no guarantee which one comes first.

Finally, to stop the query execution, we call the `stop` method:

```
queryExecution.stop()
```

The Moral of this Example

In this example, we have seen how to correctly use a minimalistic `socket` client to output the data of a streaming query with the *foreach sink*. Socket communication is the underlying interaction mechanism of most database drivers and many other application clients in the wild. The method that we have illustrated here is a common pattern that can be effectively applied to write to a variety of external systems that offer a JVM-based client library. In a nutshell, this pattern can be summarized as follows:

1. Create a `@transient` mutable reference to our driver class in the body of the `ForeachWriter`
2. In the `open` method, initialize a connection to the external system. Assign this connection to the mutable reference. It's warranteed that this reference will be used by a single thread.
3. In `process`, publish the prodided data element to the external system.
4. Finally, in `close` we terminate all connections and clean up any state.

Troubleshooting **`ForeachWriter`** Serialization Issues

In the Example 11-8 listing we saw how we needed an uninitialized mutable reference to the `TCPWriter`: `@transient var writer:TCPWriter = _` This seemingly elaborated construct is required to ensure that we instantiate the non-serializable class only when the `ForeachWriter` is already deserialized and running remotely, in a executor.

If we wanted to explore what happens when we attempt to include a non-serializable reference in a `ForeachWriter` implementation, we could declare our `TCPWriter` instance like this instead:

```
import org.apache.spark.sql.ForeachWriter
class TCPForeachWriter(host: String, port: Int) extends ForeachWriter[RateTick] {

  val nonSerializableWriter:TCPWriter = new TCPWriter(host,port)
  // ... same code as before ...
}
```

Although this looks simpler and more familiar, when we attempt to run our query with this `ForeachWriter` implementation, we will get a `org.apache.spark.SparkException: Task not serializable`. This produces a very long *stack trace* that contains a best-effort attempt at pointing out the offending class. We must follow the stack trace until we find the `Caused by`statement, like shown in the following trace:

```
Caused by: java.io.NotSerializableException: $line17.$read$$iw$$iw$TCPWriter
Serialization stack:
  - object not serializable (class: $line17.$read$$iw$$iw$TCPWriter,
      value: $line17.$read$$iw$$iw$TCPWriter@4f44d3e0)
  - field (class: $line20.$read$$iw$$iw$TCPForeachWriter,
      name: nonSerializableWriter, type: class $line17.$read$$iw$$iw$TCPWriter)
  - object (class $line20.$read$$iw$$iw$TCPForeachWriter,
      $line20.$read$$iw$$iw$TCPForeachWriter@54832ad9)
  - field (class: org.apache.spark.sql.execution.streaming.ForeachSink, name:
      org$apache$spark$sql$execution$streaming$ForeachSink$$writer,
      type: class org.apache.spark.sql.ForeachWriter)
```

As this example was running in the `spark-shell`, we find some weird `$$`-notation, but removing that noise, we can see that the non-serializable object is: `object not serializable (class: TCPWriter)` and the reference to it is the field `field name: nonSerializableWriter, type: class TCPWriter`.

Serialization issues are common in `ForeachWriter` implementations. Hopefully, with the tips in this section, you will be able to avoid any trouble in your own implementation. But in cases when this happens, Spark makes a best-effort attempt at determining the source of the problem. This information, provided in the stack trace, is very valuable to debug and solve these *Serialization* issues.

Chapter 12. Event Time Based Stream Processing

In "The Effect of Time", we discussed the effect of time in stream processing from a general perspective.

As we recall, *event-time processing* refers to looking at the stream of events from the timeline at which they were produced and applying the processing logic from that perspective. When we are interested in analysing the patterns of the event data over time, it is necessary to process the events as if we are observing them at the time they were produced. To do this, we require the device or system that produces the event to "stamp" the events with the time of creation. Hence the usual name "timestamp" to refer to a specific event-bound time. We use that time as our frame of reference of how time evolves.

To illustrate this concept, lets explore a familiar example. Consider a network of weather stations used to monitor local weather conditions. Some remote stations are connected through the mobile network, while others, hosted at volunteering homes, have access to Internet connections of varying quality. The weather monitoring system cannot rely on the arrival order of the events as that order is mostly dependent of the speed and reliability of the network they are connected to. Instead, the weather application relies on each weather station to timestamp the events delivered. Our stream processing then uses these timestamps to compute the time-based aggregations that feed the weather forecasting system.

The capability of a stream processing engine to use *event time* is important, because we are usually interested in the relative order that events were produced, and not in the sequence the events are processed. In this section we will learn how Structured Streaming provides seamlessly support for *event time* processing.

Understanding Event Time in Structured Streaming

At the server side, the notion of time is ruled by the internal clock of the computers running any given application. In the case of distributed applications running on a cluster of machines, it is a mandatory practice to use a common clock synchronization technique and protocol,such as NTP, to align all clocks to the same time. This prevents issues of the notion of time moving forward or backward as events move from one machine to another.

But, when data is coming from external devices, such as sensor networks, other datacenters, mobile phones or connected cars, just to name few examples, we have no guarantees that their clocks are aligned with our cluster of machines. We need to interprete the timeline of the incoming events from the perspective of the producing system and not in reference to the internal clock of the processing system.

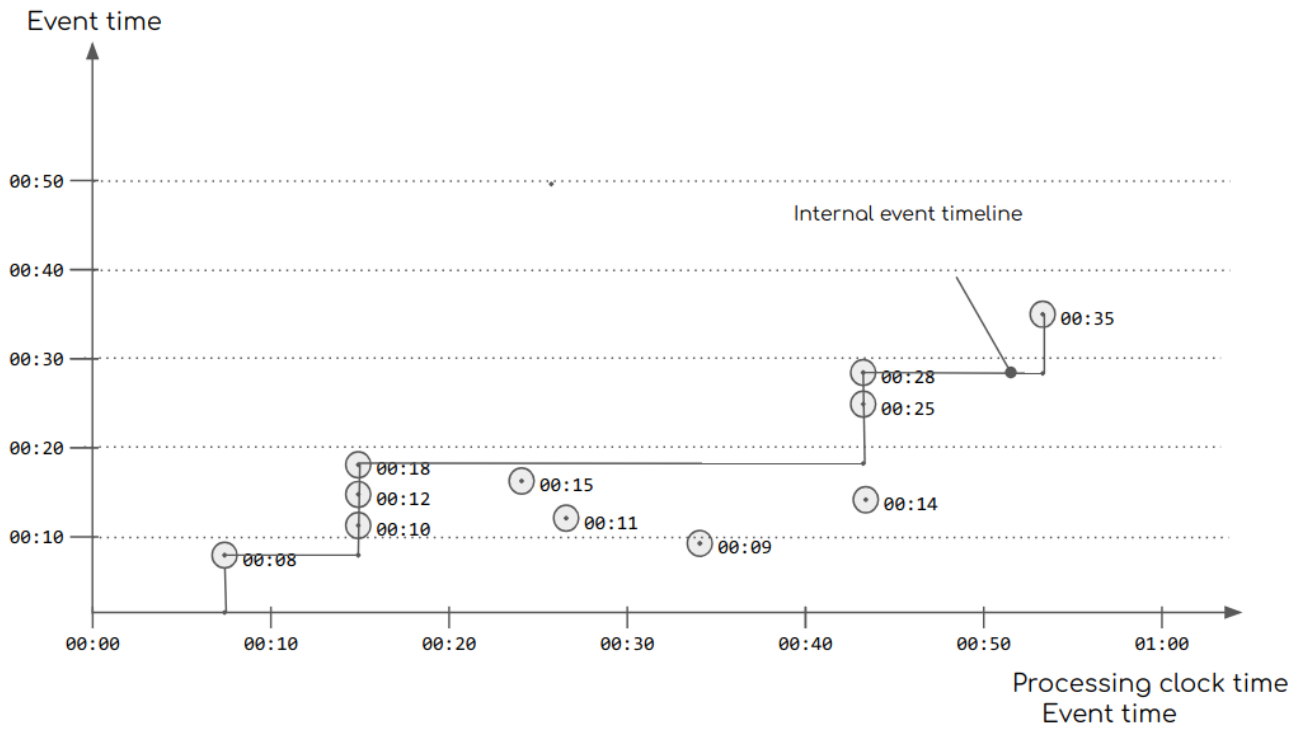


Figure 12-1. Internal Event timeline

In Figure 12-1, we visualize how the time in handled in Structured Streaming:

- In the x-axis we have the *processing time*, the clock time of the processing system.
- The y-axis represents the internal representation of the *event time* timeline.
- Events are represented with a circle with their corresponding *event time* label next to them.
- The event arrival time corresponds to the time on the x-axis.

As events arrive in the system, our internal notion of time progresses:

1. The first event, `00:08` arrives into the system at `00:07`, "early" from the point of view of the machine clock. We can appreciate that the internal clock time does not affect our perception of the event timeline.
2. The event timeline advances to `00:08`.
3. The next batch of events, `00:10`, `00:12`, `00:18` arrive for processing. The event timeline moves up to `00:18` as it's the maximun observed time so far.
4. Thereafter, `00:15` enters the system. The event timeline remains in its current value of `00:18` as `00:15` is earlier than the current internal time.
5. Likewise, `00:11` and `00:09` are received. Should we process these events or are they too late?
6. When the next set of events are processed, `00:14`, `00:25`, `00:28`, the streaming clock increases up to their maximum of `00:28`.

In general, Structured Streaming infers the timeline of the events processed with *event time* by keeping a monotonically increasing upper bound of the field declared as *time stamp* in the events. This non-linear timeline is the ruling clock used for the time-based processing features in this chapter. The ability of Structured Streaming of understanding the time flow of the event source

decouples the event generation from the event processing time. In particular, we can replay a sequence of past events and have Structured Streaming produce the correct results for all event-time aggregations. We could, for example, replay a week worth of events in few minutes and have our system produce results consistent with a week period. This would be impossible if time was governed by the computer clock.

Using *Event Time*

In Structured Streaming, we can take advantange of the built-in support for *event-time* in two areas: time-based aggregation and state management.

In both cases, the first step is to have a field in our data in the right format for Structured Streaming to understand it as a timestamp.

In the example Example 12-1, the initial `ts` field contains the reported timestamp. Notice the different methods used to obtain a `Timestamp` for the Structured Streaming application depending on the origina format of the time field.

Example 12-1. Obtaining a timestamp field

```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.types._
// Lets assume an existing streaming dataframe of weather station readings
// We will vary the type of the 'ts' field to demonstrate the various options
// (id: String, ts: <see-below>, pressure: Double, temperature: Double)

// 'ts: Long' - time is epoch time in milliseconds
// we can cast it directly to TimestampType
val timeStampEvents =
  raw.withColumn("timestamp", $"ts".cast(TimestampType))

// 'ts: java.sql.Timestamp' - time is java.sql.Timestamp object
// 'java.sql.Timestamp' directly represents the Spark TimestampType
val timeStampEvents = raw.withColumn("timestamp", $"ts")

// 'ts: String' String timestamp with default format 'yyyy-MM-dd HH:mm:ss'
// we can cast it directly to TimestampType
val timeStampEvents =
  raw.withColumn("timestamp", $"ts".cast(TimestampType))

// 'ts: String' String timestamp with default format 'yyyy-MM-dd HH:mm:ss'
// We can also use the 'to_timestamp' function
val timeStampEvents =
  raw.withColumn("timestamp", to_timestamp($"ts"))

// 'ts: String' String timestamp with a custom format. eg.'dd-MM-yyyy HH:mm:ss'
// we use the 'to_timestamp' function, specifying the the custom format
val timeStampEvents = raw
  .withColumn(
    "timestamp",
    to_timestamp($"ts", "dd-MM-yyyy HH:mm:ss")
  )
```

Processing Time

As we discussed in the introduction of this section, we make a distinction between *event-time* and *processing-time* processing. Event-time relates to the timeline at which events were produced and is independent of the time of processing. In contrast, processing-time is the timeline when events are ingested by the engine and it is based on the clock of the computers processing the event stream. It's is the "now" when the events enter the processing engine.

There are cases where the event data does not contain time information but we still want to take advantage of the native time-based functions offered by Structured Streaming. In those cases, we can add the a *processing-time* timestamp to the event data and use that timestamp as the event-time.

Continuing with the same example as above, we can add *processing time* information using the `current_timestamp` SQL function.

```
// Lets assume an existing streaming dataframe of weather station readings
// (id: String, pressure: Double, temperature: Double)

// we add a processing-time timestamp
val timeStampEvents = raw.withColumn("timestamp", current_timestamp())
```

Watermarks

At the beginning of the chapter, we learned that external factors may affect the delivery of event messages and hence, when using *event time* for processing, we didn't have a guarantee of order or delivery. Events might be late or never arrive at all. How late is too late? For how long do we hold partial aggregations before considering them complete? To answer these questions, the concept of *watermarks* was introduced in Structured Streaming: A *watermark* is a time threshold that dictates how long do we wait for events before declaring that they are too late. Events that are considered late beyond the watermark are discarded.

Watermarks are computed as a threshold based on the internal time representation. As we can appreciate in Figure 12-2, the watermark line is a shifted line from the *event time* timeline infered from the event's time information. In this chart, we can observe that all events falling in the "gray area" demarked below the *watermark* line are considered "too late" and will not be taken into consideration in the computations consuming this event stream..

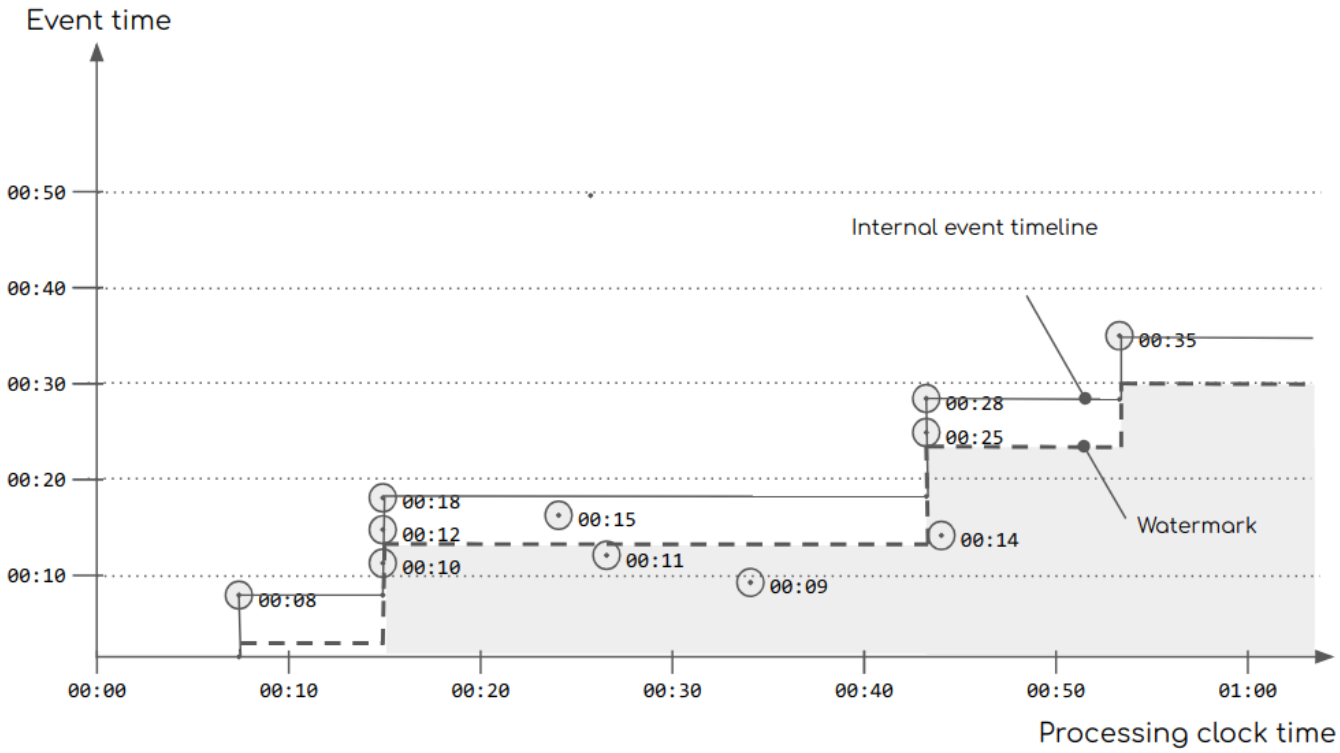


Figure 12-2. Watermark with the Internal Event timeline

We declare a *watermark* by linking our *timestamp* field with the time threshold corresponding to the watermark. Continuing with the Example 12-1, we declare a *watermark* like this:

```
// Lets assume an existing streaming dataframe of weather station readings
// (id: String, ts:Long, pressure: Double, temperature: Double)

val timeStampEvents = raw.withColumn("timestamp", $"ts".cast(TimestampType))
                         .withWatermak("timestamp", "5 minutes")
```

Time-based Window Aggregations

A natural question we want to ask to streams of data is aggregated information at regular intervals of time. As streams are potentially never-ending, instead of asking "how many X are there?", in a stream processing context, we are more interested in knowing "how many X were there in 15-minute intervals".

With the use of *event time* processing, Structured Streaming removes the usual complexity of dealing with intermediate state in the face of the event delivery challenges that we have motivated in this chapter. Structured Streaming takes care of keeping partial aggregates of data and of updating the downstream consumer using the semantics corresponding to the chosen output mode.

Defining Time-based Windows

We discussed the concept of window-based aggregations in "Window Aggregations", where we presented the definitions of *tumbling* and *sliding* windows. In Structured Streaming, the built-in event-time support makes it easy to define and use such window-based operations.

From the API perspective, window aggregations are declared using a `window` function as grouping criteria. The window function must be applied to the field we want to use as *event time*.

Continuing with our weather station example, we can compute the average pressure each 10 minutes totalized across all reporting stations:

Example 12-2. Computing Totalized Averages

```
$>val perMinunteAvg = timeStampEvents
  .withWatermak("timestamp", "5 minutes")
  .groupBy(window($"timestamp", "1 minute"))
  .agg(avg($"pressure"))

$>perMinunteAvg.printSchema // let's inspect the schema of our window aggregation

root
 |-- window: struct (nullable = true)
 |    |-- start: timestamp (nullable = true)
 |    |-- end: timestamp (nullable = true)
 |-- pressureAvg: double (nullable = true)
 |-- tempAvg: double (nullable = true)

$>perMinunteAvg.writeStream.outputMode("append").format("console").start()
// after few minutes
+---------------------------------------------+-------------+-------------+
|window                                       |pressureAvg  |tempAvg      |
+---------------------------------------------+-------------+-------------+
|[2018-06-17 23:27:00.0,2018-06-17 23:28:00.0]|101.515516867|5.19433723603|
|[2018-06-17 23:28:00.0,2018-06-17 23:29:00.0]|101.481236804|13.4036089642|
|[2018-06-17 23:29:00.0,2018-06-17 23:30:00.0]|101.534757332|7.29652790939|
|[2018-06-17 23:30:00.0,2018-06-17 23:31:00.0]|101.472349471|9.38486237260|
|[2018-06-17 23:31:00.0,2018-06-17 23:32:00.0]|101.523849943|12.3600638827|
|[2018-06-17 23:32:00.0,2018-06-17 23:33:00.0]|101.531088691|11.9662189701|
|[2018-06-17 23:33:00.0,2018-06-17 23:34:00.0]|101.491889383|9.07050033207|
+---------------------------------------------+-------------+-------------+
```

In this example, we observe that the resulting schema of a windowed aggregation contains the window period, indicated with `start` and `end` timestamp for each resulting window, together with the corresponding computed values.

Understanding How Intervals are Computed.

The window intervals are aligned to the start of the second/minute/hour/day that corresponds to the next upper time magnitude of the time unit used. For example, a `window($"timestamp", "15 minutes")` will produce 15-minute intervals aligned to the start of the hour.

The start time of the first interval is in the past to adjust the window aligment without any data loss. That implies that the first interval might contain only a fraction of the usual interval worth of data. So, if we are receiving 100 messages per second, we expect to see 90k messages in 15 minutes while our first window might be just a fraction of that.

The time intervals in a window are inclusive at the start and exclusive at the end. In set notation, this is written as `[start-time, end-time)`. Using the 15-minute intervals as defined above, a data point that arrives with with timestamp `11:30:00.00` will belong to the `11:30-11:45` window interval.

Using Composite Aggregation Keys

In Example 12-2, we calculated globally aggregated values for the *presssure* and *temperature* sensors. We are also interested in computing aggregated values for each weather stations. We can achieve that by creating a composite aggregation key where we add the `stationId` to the aggregation criteria in the same way that we would do that with the static `DataFrame` API.

Example 12-3. Computing Averages per Station

```
$>val minuteAvgPerStation = timeStampEvents
  .withWatermak("timestamp", "5 minutes")
  .groupBy($"stationId", window($"timestamp", "1 minute"))
  .agg(avg($"pressure") as "pressureAvg", avg($"temp") as "tempAvg")

// The aggregation schema now contains the station Id
$>minuteAvgPerStation.printSchema
root
 |-- stationId: string (nullable = true)
 |-- window: struct (nullable = true)
 |    |-- start: timestamp (nullable = true)
 |    |-- end: timestamp (nullable = true)
 |-- pressureAvg: double (nullable = true)
 |-- tempAvg: double (nullable = true)

$>minuteAvgPerStation.writeStream.outputMode("append").format("console").start

+---------+------------------------------------------+-----------+------------+
|stationId|window                                    |pressureAvg|tempAvg     |
+---------+------------------------------------------+-----------+------------+
|d60779f6 |[2018-06-24 18:40:00,2018-06-24 18:41:00]|101.2941341|17.305931400|
|d1e46a42 |[2018-06-24 18:40:00,2018-06-24 18:41:00]|101.0664287|4.1361759034|
|d7e277b2 |[2018-06-24 18:40:00,2018-06-24 18:41:00]|101.8582047|26.733601007|
|d2f731cc |[2018-06-24 18:40:00,2018-06-24 18:41:00]|101.4787068|9.2916271894|
|d2e710aa |[2018-06-24 18:40:00,2018-06-24 18:41:00]|101.7895921|12.575678298|
   ...
|d2f731cc |[2018-06-24 18:41:00,2018-06-24 18:42:00]|101.3489804|11.372200251|
|d60779f6 |[2018-06-24 18:41:00,2018-06-24 18:42:00]|101.6932267|17.162540135|
|d1b06f88 |[2018-06-24 18:41:00,2018-06-24 18:42:00]|101.3705194|-3.318370333|
|d4c162ee |[2018-06-24 18:41:00,2018-06-24 18:42:00]|101.3407332|19.347538519|
+---------+------------------------------------------+-----------+------------+
// ** output has been edited to fit into the page
```

Tumbling and Sliding Windows

`window` is a SQL function that takes a `timeColumn` of `TimestampType` type and additional parameters to specify the duration of the window:

```
window(timeColumn: Column,
       windowDuration: String,
       slideDuration: String,
       startTime: String)
```

Overloaded definitions of this method make `slideDuration` and `startTime` optional.

This API lets us specify two kinds of windows: tumbling and sliding windows. The optional `startTime` can delay the creation of the window, for example, when we want

TUMBLING WINDOWS

Tumbling windows segment the time in non-overlapping, contiguous periods. They are the natural window operation when we refer to a "total count each 15 minutes" or "production level per generator each hour" We specify a tumbling window by only providing the `windowDuration` parameter:

```
window($"timestamp", "5 minutes")
```

This `window` definition will produce one result each 5 minutes.

SLIDING WINDOWS

In contrast with tumbling windows, sliding windows are overlapping intervals of time. The size of the interval is determined by the `windowDuration` time. All values from the stream in that interval come into consideration for the aggregate operation. For the next *slice*, we add the elements arriving during `slideDuration`, remove the elements corresponding to the oldest *slice* and apply the aggregation to the data within the window, producing a result at each `slideDuration`.

```
window($"timestamp", "10 minutes", "1 minute")
```

This window definition uses 10 minutes worth of data to produce a result every minute.

It's worth noting that a *tumbling* window is a particular case of a *sliding* window, where `windowDuration` and `slideDuration` have equal values:

```
window($"timestamp", "5 minutes", "5 minutes")
```

It is illegal to use a `slideInterval` larger than the `windowDuration`. Structured Streaming will throw an `org.apache.spark.sql.AnalysisException` error if such case occurs.

INTERVAL OFFSET

The third parameters in the window definition, called `startTime` provides a way to offset the window alignment. In "Understanding How Intervals are Computed." we saw that the window intevals are aligned to the upper next time magnitude. `startTime` (a misnomer in our opinion) lets us offset the window intervals by the indicated time.

In the following window definition, we offset a 10-minute window with a slide duration of 5 minutes by 2 minutes, resulting in time intervals like: `00:02-00:12, 00:07-00:17, 00:12-00:22, ...`

```
window($"timestamp", "10 minutes", "5 minute", "2 minutes")
```

`startTime` must strictly less than `slideDuration`. Structured Streaming will throw an `org.apache.spark.sql.AnalysisException` error if an invalid configuration is provided. Intuitively, given that `slideDuration` provides the periodicity at which the window is reported, we can only offset that period for a time lesser than the period itself.

Chapter 13. Advanced Stateful Operations

In Chapter 8, we saw how easy it is to express an aggregation in Structured Streaming using the existing aggregation functions in the *structured* Spark APIs. In Chapter 12, we learned the effectiveness of Spark's built-in support for using the embedded time information in the event stream.

However, there are cases when we need to meet custom aggregation criteria that are not directly supported by the built-in models. In this chapter, we explore how to conduct advanced stateful operations to address these situations. For those cases, Structured Streaming offers an API to implement arbitrary stateful processing. This API is represented by two operations: `mapGroupsWithState` and `flatMapGroupsWithState`. Both operations allow us to create a custom definition of a state, set up the rules of how this state evolves as new data comes in over time, determine when it expires, and provides us with a method to combine this state definition with the incoming data to produce results.

The main difference between `mapGroupsWithState` and `flatMapGroupsWithState` is that the former must produce a single result for each processed group, while the latter may produce zero or more results.

Internally, Structured Streaming takes care of managing state between operations and ensures its availability and fault-tolerant preservation during and across executions of the streaming process over time.

Starting with an Example

Let's imagine a car fleet management solution where the vehicles of the fleet are enabled with wireless network capabilities. Each vehicle regularly reports its geographical location and many operational parameters, like fuel level, speed, acceleration, bearing, engine temperature, etc. The stakeholders would like to exploit this stream of telemetry data to implement a range of applications to help them manage the operational and financial aspects of the business.

Using the Structured Streaming features we know so far, we could implement already many use cases, like monitoring kilometers driven per day using event-time windows or finding vehicles with a low-fuel warning by applying filters.

Now, we would like to have the notion of a trip: the driven road segment from a start to a stop. Individually, the notion of a trip is useful to compute fuel efficiency or monitor compliance to geo-fencing agreements. When analysed in groups, trip information might reveal transportation patters, traffic hotspots and when combined with other sensor information, they can even report road conditions. From our stream processing perspective, we could see trips as an arbitrary window that opens when the vehicle starts moving and closes when it finally stops. The event-time window aggregations we saw in Chapter 12 use fixed time intervals as windowing criteria, so they are of no help to implement our trip analysis.

We can appreciate that we need a more powerful definition of state that is not purely based on time, but also on arbitrary conditions. In our example, this condition is that the vehicle is driving.

Understanding *group with state* operations

The arbitrary state operations, `mapGroupsWithState` and `flatMapGroupWithState` work exclusively on the typed `Dataset` API using either the Scala or the Java bindings.

Based on the data that we are processing and the requirements of our stateful transformation, we need to provide three type definitions, typically encoded as a `case class` (Scala) or a `Java Bean` (Java):

- the input event (I),
- the arbitrary state to keep (S)
- the output (O) (this type might be the same as the state representation, if suitable)

All these types must be encodable into Spark SQL types. That means that there should be an `Encoder`available. The usual import statement:

```
`import spark.implicits._`
```

is sufficient for all basic types, tuples and `case class`es

With these types in place, we can formulate the state transformation function that implements our custom state handling logic.

`mapGroupsWithState` requires that this function returns a single mandatory value:

```
def mappingFunction(key: K, values: Iterator[I], state: GroupState[S]): O
```

`flatMapGroupsWithState` requires that this function returns an `Iterator`, that may contain zero or more elements:

```
def flatMappingFunction(
    key: K, values: Iterator[I], state: GroupState[S]): Iterator[O]
```

`GroupState[S]` is a wrapper provided by Structured Streaming and used internally to manage the state `S` across executions. Within the function, `GroupState` provides mutation access to the state and the ability to check and set timeouts.

WARNING

The implementation of the `mappingFunction`/`flatMappingFunction` **must be**`Serializable`.

At runtime, this function is distributed to the executors of the cluster using *Java Serialization*. This requirements also has the consequence that we **must not** include any local state, like counters or other mutable variables in the body of the function. All managed state **must be** encapsulated in the `State` representation class.

Internal State Flow

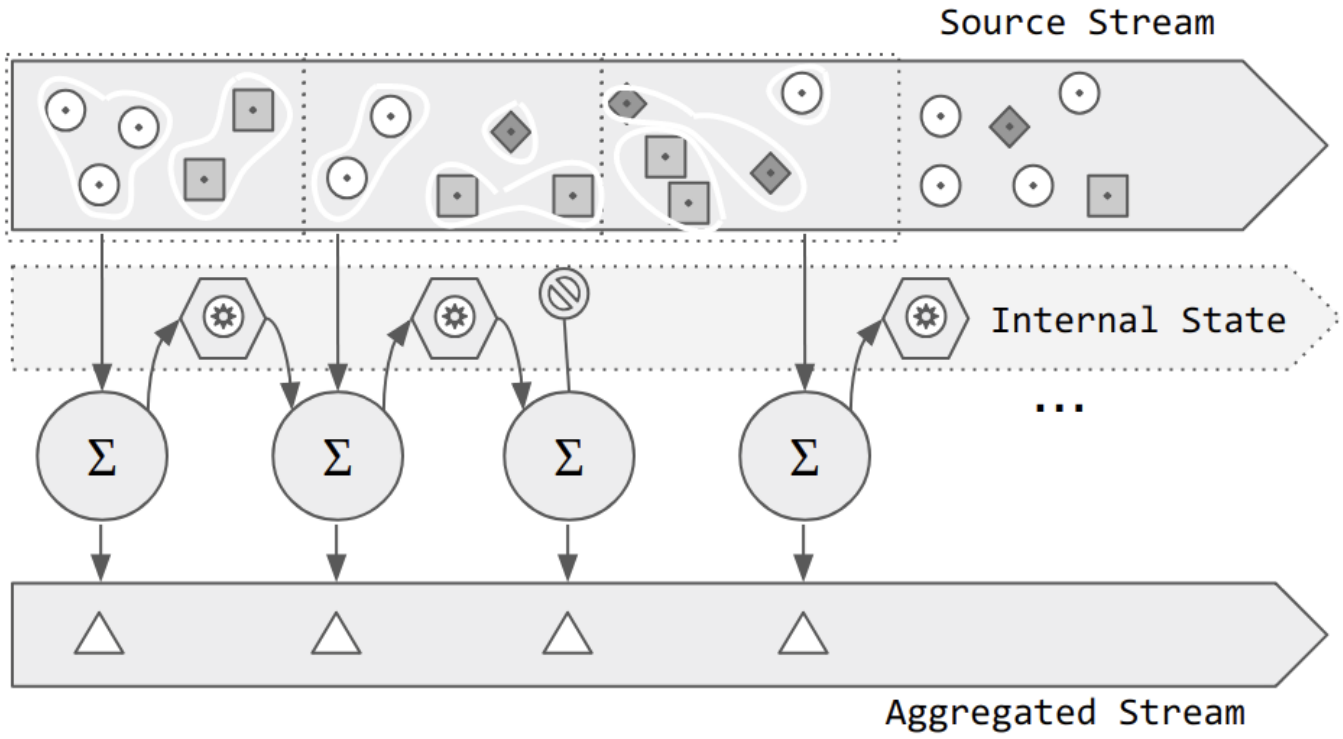


Figure 13-1. Map Groups With State Dynamics

In Figure 13-1 we illustrate the process that combines the incoming data, in the form of events, with the state maintained internall, to produce a result. In this chart, the `mappingFunction` (denoted with a `Σ`) uses the custom logic to process this group of elements using, that when combined with the state managed by `GroupState[S]`, leads to a result. In this illustration, we used the stop symbol to indicate a timeout. In the case of `MapGroupsWithState`, a timeout also triggers the production of an event and **should** evict the state. Given that the eviction logic is under the control of the programmed logic, the complete state management is under the responsibility of the developer. Structured Streaming only provides the building blocks.

Using MapGroupsWithState

In "Sliding Windows" we saw how we can compute a moving average based on a time window. This time-base window would produce a result independently of the number of elements found in the window.

Now, suppose that our requirement is to compute a moving average of the last 10 elements received per key. We cannot use a time window, because we don't know how long it will take us to have the number of elements we need. Instead, we can define our own count-based window using a custom state with `MapGroupsWithState`.

Let's start the same streaming `Dataset` we used in "Sliding Windows". The `WeatherEvent case class` becomes our input type (I).

```
// a representation of a weather station event
```

```
case class WeatherEvent(stationId: String,
  timestamp: Timestamp,
  location:(Double,Double),
  pressure: Double,
  temp: Double)

val weatherEvents: Dataset[WeatherEvents] = ...
```

Next, we define the *state* (S). What we want is to keep the latest *n* elements in our state and drop anything older. This seems a natural application of a FIFO (First In, First Out) collection, such as a `Queue`. Newer elements are added to the front of the queue, we keep the most recent *n* and drop any older element.

Our state definition becomes a `FIFOBuffer` backed by a `Queue` with few helper methods to facilitate its usage.

```
import scala.collection.immutable.Queue
case class FIFOBuffer[T](
    capacity: Int, data: Queue[T] = Queue.empty
    ) extends Serializable {

  def add(element: T): FIFOBuffer[T] =
    this.copy(data = data.enqueue(element).take(capacity))

  def get: List[T] = data.toList

  def size: Int = data.size
}
```

Then, we need to define the output type (O) that results of the stateful computation. As desired result of our stateful computation is the moving average of the sensor values present in the input `WeatherEvent`. We also would like to know the time span of the values used for the computation. With this knowledge, we design our output type: `WeatherEventAverage`

```
import java.sql.Timestamp
case class WeatherEventAverage(stationId: String,
                               startTime: Timestamp,
                               endTime:Timestamp,
                               pressureAvg: Double,
                               tempAvg: Double)
```

With these types defined, we can proceed to create the `mappingFunction` that combines the existing state and the new elements into a result. Remember that this function is also responsible of updating the internal state through the functions provided by the `GroupState` wrapper. It's important to note that the state cannot be updated with a `null` value. An attempt to do so will throw an `IllegalArgumentException`. To remove a state, use the method `state.remove()`

Example 13-1. Using **`mapGroupsWithState`** *for a count-based moving average*

```
import org.apache.spark.sql.streaming.GroupState
def mappingFunction(
    key: String,
    values: Iterator[WeatherEvent],
    state: GroupState[FIFOBuffer[WeatherEvent]]
  ): WeatherEventAverage = {

  // the size of the window, defined here for
```

```
  val ElementCountWindowSize = 10

  // get current state or create a new one if there's no previous state
  val currentState = state.getOption
    .getOrElse(
      new FIFOBuffer[WeatherEvent](ElementCountWindowSize)
    )

  // enrich the state with the new events
  val updatedState = values.foldLeft(currentState) {
    case (st, ev) => st.add(ev)
  }

  // update the state with the enriched state
  state.update(updatedState)

  // if we have enough data, create a WeatherEventAverage from the state
  // otherwise, make a zeroed record
  val data = updatedState.get
  if (data.size > 2) {
    val start = data.head
    val end = data.last
    val pressureAvg = data
      .map(event => event.pressure)
      .sum / data.size
    val tempAvg = data
      .map(event => event.temp)
      .sum / data.size
    WeatherEventAverage(
      key,
      start.timestamp,
      end.timestamp,
      pressureAvg,
      tempAvg
    )
  } else {
    WeatherEventAverage(
      key,
      new Timestamp(0),
      new Timestamp(0),
      0.0,
      0.0
    )
  }
}
```

Now, we use the `mappingFunction` to declare the stateful transformation of the streaming `Dataset`.

```
import org.apache.spark.sql.streaming.GroupStateTimeout
val weatherEventsMovingAverage = weatherEvents
  .groupByKey(record => record.stationId)
  .mapGroupsWithState(GroupStateTimeout.ProcessingTimeTimeout)(mappingFunction)
```

Note that we first create *groups* out of the key identifiers in our domain. In this example, this is the `stationId`. The `groupByKey` operation creates an intermediate structure, a `KeyValueGroupedDataset` that becomes the entry point for the `[map|flatMap]GroupWithState` operations.

Besides the mapping function, we also need to provide a timeout type. A timeout type may be either a `ProcessingTimeTimeout` or an `EventTimeTimeout`. As we are not relying on the timestamp of the events for our state management, we chose the `ProcessingTimeTimeout`. We will discuss timeout management in detail further in this chapter.

Finally, we can easily observe the results of the query by using the *console* sink.

```
val outQuery = weatherEventsMovingAverage.writeStream
  .format("console")
  .outputMode("update")
  .start()
```

```
+---------+-------------------+-------------------+------------+------------+
|stationId|startTime          |endTime            |pressureAvg |tempAvg     |
+---------+-------------------+-------------------+------------+------------+
|d1e46a42 |2018-07-08 19:20:31|2018-07-08 19:20:36|101.33375295|19.753225782|
|d1e46a42 |2018-07-08 19:20:31|2018-07-08 19:20:44|101.33667584|14.287718525|
|d60779f6 |2018-07-08 19:20:38|2018-07-08 19:20:48|101.59818386|11.990002708|
|d1e46a42 |2018-07-08 19:20:31|2018-07-08 19:20:49|101.34226429|11.294964619|
|d60779f6 |2018-07-08 19:20:38|2018-07-08 19:20:51|101.63191940|8.3239282534|
|d8e16e2a |2018-07-08 19:20:40|2018-07-08 19:20:52|101.61979385|5.0717571842|
|d4c162ee |2018-07-08 19:20:34|2018-07-08 19:20:53|101.55532969|13.072768358|
+---------+-------------------+-------------------+------------+------------+
// (!) output edited to fit in the page
```

Using FlatMapGroupsWithState

Our previous implementation has a flaw. Can you spot it?

When we start processing the stream, and before we have collected all the elements that we deem required to compute the moving average, the operation of `mapGroupsWithState` produces zeroed out values.

```
+---------+-------------------+-------------------+-----------+-------+
|stationId|startTime          |endTime            |pressureAvg|tempAvg|
+---------+-------------------+-------------------+-----------+-------+
|d2e710aa |1970-01-01 01:00:00|1970-01-01 01:00:00|0.0        |0.0    |
|d1e46a42 |1970-01-01 01:00:00|1970-01-01 01:00:00|0.0        |0.0    |
|d4a11632 |1970-01-01 01:00:00|1970-01-01 01:00:00|0.0        |0.0    |
+---------+-------------------+-------------------+-----------+-------+
```

As we mentioned earlier, `mapGroupsWithState` requires the state handling function to produce a single record for each group processed at every trigger interval. This is fine when the arrival of new data corresponding to each *key* naturally updates its state.

But there are cases when our state logic requires for a series of events to occur before we can produce a result. In our current example, we need *n* elements before we can start producing an average over them. In other scenarios, it might be that a single incoming event might complete several temporary states and therefore produce more than one result. For example, the arrival of a single mass transport to its destination might update the traveling state of all of its passengers, potentially producing a record for each of them.

`flatMapGroupsWithState` is a generalization of `mapGroupsWithState` where the state handling function produces an `Iterator` of results, which might contain zero or more elements.

Let's see how we can use this function to improve our *moving average* computation over n-elements.

We need to update the mapping function to return an `Iterator` of results. In our case, this `Iterator`will contain zero elements when we don't have enough values to compute the average and a value otherwise. Our changed function looks like:

Example 13-2. Using FlatMapGroupsWithState for a count-based moving average

```
import org.apache.spark.sql.streaming._
def flatMappingFunction(
    key: String,
    values: Iterator[WeatherEvent],
    state: GroupState[FIFOBuffer[WeatherEvent]]
  ): Iterator[WeatherEventAverage] = {

  val ElementCountWindowSize = 10

  // get current state or create a new one if there's no previous state
  val currentState = state.getOption
    .getOrElse(
      new FIFOBuffer[WeatherEvent](ElementCountWindowSize)
    )

  // enrich the state with the new events
  val updatedState = values.foldLeft(currentState) {
    case (st, ev) => st.add(ev)
  }

  // update the state with the enriched state
  state.update(updatedState)

  // only when we have enough data, create a WeatherEventAverage from the state
  // before that, we return an empty result.
  val data = updatedState.get
  if (data.size == ElementCountWindowSize) {
    val start = data.head
    val end = data.last
    val pressureAvg = data
      .map(event => event.pressure)
      .sum / data.size
    val tempAvg = data
      .map(event => event.temp)
      .sum / data.size
    Iterator(
      WeatherEventAverage(
        key,
        start.timestamp,
        end.timestamp,
        pressureAvg,
        tempAvg
      )
    )
  } else {
    Iterator.empty
  }
}
```

```
val weatherEventsMovingAverage = weatherEvents
  .groupByKey(record => record.stationId)
  .flatMapGroupsWithState(
    OutputMode.Update,
    GroupStateTimeout.ProcessingTimeTimeout
  )(flatMappingFunction)
```

Using `flatMapGroupsWithState`, we don't need to produce artificial zeroed records anymore. In addition to that, our state management definition is now strict in having *n* elements to produce a result.

Output Modes

Although the cardinality difference in the results between the `map` and the `flatMap`–`GroupsWithState` operations might seem like a small practical API difference, it has deeper consequences beyond the obvious variable production of results.

As we can appreciate in the example, `flatMapGroupsWithState` requires the additional specification of an *output mode*. This is needed to provide information about the record production semantics of the stateful operation to the downstream process. In turn, this helps Structured Streaming to compute the allowed *output operation* for the downstream sink.

The *output mode* specified in `flatMapGroupsWithState` may be:

`update`

: indicates that the records produced are non-final. They are intermediate results that might be updated with new information later on. In the previous example, the arrival of new data for a key will produce a new data point. The downstream sink must use `update` and no aggregations may follow the `flatMapGroupsWithState` operation.

`append`

: designates that we have collected all the information we need to produce a result for a group and no further incoming events will change that outcome. The downstream sink must use `append`mode to write. Given that the application of `flatMapGroupsWithState` produces a final record, it's possible to apply further aggregations to that result.

Managing Timeouts

A critical requirement of managing state over time is to ensure that we have a stable working set. [1]That is, the memory required by our process is bounded over time and remains at a safe distance below the available memory to allow for fluctuations.

In the managed stateful aggregations, such as the time-based windows that we saw in Chapter 12, Structured Streaming internally manages mechanisms to evict state and events that are deemed expired in order to limit the amount of memory used. When we use the custom state management capabilities offered by `[map|flatMap]GroupsWithState`, we must also asume the responsibility of removing old state.

Luckily, Structured Streaming exposes time and timeout information that we can use to decide when to expire certain state. The first step is to decide the time reference to use. Timeouts can be based on *event time* or *processing time* and the choice is global to the state handled by the particular `[map|flatMap]GroupsWithState` being configured.

The timeout type is specified when we call `[map|flatMap]GroupsWithState`. Recalling the moving average example, we configured the `mapGroupsWithState` function to use *processing time*like this:

```
import org.apache.spark.sql.streaming.GroupStateTimeout
val weatherEventsMovingAverage = weatherEvents
  .groupByKey(record => record.stationId)
  .mapGroupsWithState(GroupStateTimeout.ProcessingTimeTimeout)(mappingFunction)
```

To use *event time*, we also need to declare a *watermark* definition. This definition consists of the timestamp field from the event and the configured lag of the watermark. If we wanted to use *event time* with the previous example, we would declare it as:

```
val weatherEventsMovingAverage = weatherEvents
  .withWatermark("timestamp", "2 minutes")
  .groupByKey(record => record.stationId)
  .mapGroupsWithState(GroupStateTimeout.EventTimeTimeout)(mappingFunction)
```

The timeout type declares the global source of the time reference. There is also the option `GroupStateTimeout.NoTimeout` for the cases where we don't need timeouts. The actual value of the timeout is managed per individual group, using the methods available in `GroupState` to manage timeout: `state.setTimeoutDuration` or `state.setTimeoutTimestamp`.

To check whether a state has expired, we check `state.hasTimedOut`. When a *state* has timed out, the call to the `(flat)MapFunction` will be issued with an empty iterator of values for the group that has timed out.

Let's put the timeout feature to use. Continuing with our running example, the first we want to do is extract the transformation of state into event:

```
def stateToAverageEvent(
    key: String,
    data: FIFOBuffer[WeatherEvent]
  ): Iterator[WeatherEventAverage] = {
  if (data.size == ElementCountWindowSize) {
    val events = data.get
    val start = events.head
    val end = events.last
    val pressureAvg = events
      .map(event => event.pressure)
      .sum / data.size
    val tempAvg = events
      .map(event => event.temp)
      .sum / data.size
    Iterator(
      WeatherEventAverage(
        key,
        start.timestamp,
        end.timestamp,
        pressureAvg,
        tempAvg
```

```
        )
      )
    } else {
      Iterator.empty
    }
  }
```

Now, we can use that new abstraction to transform our state in the case of a timeout as well as in the usual scenario where data is coming in. Note how we use the timeout information to evict the expiring state.

Example 13-3. Using timeouts in **`flatMapGroupsWithState`**

```
import org.apache.spark.sql.streaming.GroupState
def flatMappingFunction(
    key: String,
    values: Iterator[WeatherEvent],
    state: GroupState[FIFOBuffer[WeatherEvent]]
  ): Iterator[WeatherEventAverage] = {
  // first check for timeout in the state
  if (state.hasTimedOut) {
    // when the state has a timeout, the values are empty
    // this validation is only to illustrate the point
    assert(
      values.isEmpty,
      "When the state has a timeout, the values are empty"
    )
    val result = stateToAverageEvent(key, state.get)
    // evict the timed-out state
    state.remove()
    // emit the result of transforming the current state into an output record
    result
  } else {
    // get current state or create a new one if there's no previous state
    val currentState = state.getOption.getOrElse(
      new FIFOBuffer[WeatherEvent](ElementCountWindowSize)
    )
    // enrich the state with the new events
    val updatedState = values.foldLeft(currentState) {
      case (st, ev) => st.add(ev)
    }
    // update the state with the enriched state
    state.update(updatedState)
    state.setTimeoutDuration("30 seconds")
    // only when we have enough data, create a WeatherEventAverage from the
accumulated state
    // before that, we return an empty result.
    stateToAverageEvent(key, updatedState)
  }
}
```

WHEN A TIMEOUT ACTUALLY TIMES OUT

The semantics of the timeouts in Structured Streaming gives the guarantee that no event will be timed out before the clock advances past the watermark. This follows our intuition of a timeout: our state does not timeout before the set expiration time.

Where the timeout semantics depart from the common intuition is **when** the timeout event actually happens after the expiration time has passed.

Currently, the timeout processing is bound to the receival of new data. So, a stream that goes *silent* for a while and does not generate new triggers to process will not generate timeouts either. The current timeout semantics are defined in terms of an eventuality: The timeout event will be eventually triggered after the state has expired, without any guaranties about how long the *timeout event* will fire after the actual timeout has happened. Stated formally: there is a no strict upper bound on when the timeout would occur.

WARNING

There is work in progress to make timeouts fire even when no new data is available.

Summary

In this chapter, we learned about the arbitrary stateful processing API in Structured Streaming. We explored the details of and differences between the `mapGroupsWithState` and `flatMapGroupsWithState` with relation to the events produced and the output modes supported. At the end, we also learned about the timeout settings and became aware of its semantics.

While this API is more complex to use than the regular *SQL-like* constructs of the *structured* APIs, it provides with a powerful toolset to implement arbitrary state management to address the implementation of the most demanding streaming use cases.

[1]A working set is a a concept that refers to the amount of memory used by a process to function over a period of time

Chapter 14. Monitoring Structured Streaming Applications

Application monitoring is an integral part of any robust deployment. Monitoring provides insights on the application performance characteristics over time by collecting and processing metrics that quantify different aspects of the application's performance, such as responsiveness, resource usage, and task specific indicators.

Streaming applications have strict requirements regarding response times and throughput. In the case of distributed applications, like Spark, the number of variables that we need to account for during the application's lifetime are multiplied by the complexities of running on a cluster of machines. In the context of a cluster, we need to keep tabs on resource usage, like CPU, memory, and secondary storage across different hosts, both from the perspective of each host, as well as a consolidated view of the running application.

For example, imagine an application running on 10 different executors. The total memory usage indicator shows a 15% increase, which might be within the expected tolerance for this application but then, we notice that the increase comes from a single node. Such imbalance needs investigation as it will potentially cause a failure once that node runs out of memory and also implies that there is potentially an unbalanced distribution of work that causes a bottleneck. Without proper monitoring, we would not observe such behavior in the first place.

The operational metrics of Structured Streaming can be exposed through three different channels:

- the Spark's metrics subsystem,
- the `StreamingQuery` instance returned by the `writeStream.start` operation, and
- the `StreamingQueryListener` interface

As we detail in the following sections, these interfaces offer different different levels of detail and exposure to cater for different monitoring needs.

The Spark's Metrics Subsystem

Available throught the Spark core engine, the Spark metrics subsystem offers a configurable metrics collection and reporting API with a pluggable `sink` interface — not to be confused with the streaming `sink`s that we discussed earlier in this book. Spark comes with several such `sink`s, including HTTP, JMX, and CSV files. In addition to that, there's a Ganglia sink that needs additional compilation flags due to licensing restrictions.

By default, the HTTP sink is enabled. It's implemented by a servlet that registers an endpoint on the driver host on same port as the Spark UI. The metrics are accessible at the `/metrics/json` endpoint. Other `sink`s can be enabled through configuration. The choice of a given `sink` is driven by the monitoring infrastructure we want to integrate with. For example, the JMX `sink` is a common option to integrate with Prometheus, a popular metric collector in the Kubernetes cluster scheduler.

Structured Streaming Metrics

To acquire metrics from a Structured Streaming job, we must first enable the internal reporting of such metrics. We achive that by setting the configuration flag `spark.sql.streaming.metricsEnabled` to `true`.

```
// at session creation time
val spark = SparkSession
   .builder()
   .appName("SparkSessionExample")
   .config("spark.sql.streaming.metricsEnabled", true)
   .config(...)
   .getOrCreate()

// by setting the config value
spark.conf.set("spark.sql.streaming.metricsEnabled", "true")

// or by using the SQL configuration
spark.sql("SET spark.sql.streaming.metricsEnabled=true")
```

With this configuration in place, the metrics reported will contain three additional metrics for each streaming query running in the same `SparkSession` context:

- inputRate-total:: The total number of messages ingested per trigger interval
- latency: the processing time for the trigger interval
- processingRate-total: the speed at which the records are being processed.

The `StreamingQuery` Instance

As we have seen many times through previous Structured Streaming examples, the call to start a *streaming query* produces a `StreamingQuery` result. Let's zoom in on the `weatherEventsMovingAverage` from the Example 13-1:

```
val query = scoredStream.writeStream
        .format("memory")
        .queryName("memory_predictions")
        .start()

query: org.apache.spark.sql.streaming.StreamingQuery =
org.apache.spark.sql.execution.streaming.StreamingQueryWrapper@7875ee2b
```

The `StreamingQuery` object is a handler to the actual streaming query that is running continuously in the background. This handler contains methods to inspect the execution of the query and control its lifecycle. In particular, `query.awaitTermination` blocks the query either by a failure or a call to `query.stop()`. `query.stop()` ends the query execution. `query.status` will show a brief snapshot of what the query is currently doing.

```
$query.status
res: org.apache.spark.sql.streaming.StreamingQueryStatus =
{
  "message" : "Processing new data",
  "isDataAvailable" : true,
  "isTriggerActive" : false
}
```

While the *status* information is not very revealing when everything is working correctly, it can be useful when developing a new job. `query.start()` is usually silent when an error occurs. Consulting `query.status()` might reveal the cause of a problem and help with troubleshooting.

In the output shown in Example 14-1, we used an incorrect schema as input for a Kafka sink. If we recall from "The Kafka Sink", a Kafka sink requires two mandatory fields in the output stream: `key`and `value`. In this case, `query.status` provided relevant feedback to solve that issue.

Example 14-1. **`query.status`** *shows reason form stream failure to start*

```
res: org.apache.spark.sql.streaming.StreamingQueryStatus =
{
  "message": "Terminated with exception: Required attribute 'value' not found",
  "isDataAvailable": false,
  "isTriggerActive": false
}
```

The methods in `StreamingQueryStatus` are *thread-safe*, meaning that they can be called concurrently from another thread without risking corruption of the query state.

Getting Metrics with `StreamingQueryProgress`

For the purpose of monitoring, we are more interested in a set of methods that provide insights in the query execution metrics. The `StreamingQuery` handlers offers two such methods:

`query.lastProgress`

: retrieves the most recent `StreamingQueryProgress` report.

`query.recentProgress`

: retrieves an array of the most recent `StreamingQueryProgress` reports. The maximum number of *progress* objects retrieved can be set using the configuration parameter `spark.sql.streaming.numRecentProgressUpdates` in the Spark Session. If this configuration is not set, it defaults to the last 100 reports.

Example 14-2. **`StreamingQueryProgress`** *sample*

```
{
  "id": "639503f1-b6d0-49a5-89f2-402eb262ad26",
  "runId": "85d6c7d8-0d93-4cc0-bf3c-b84a4eda8b12",
  "name": "memory_predictions",
  "timestamp": "2018-08-19T14:40:10.033Z",
  "batchId": 34,
  "numInputRows": 37,
  "inputRowsPerSecond": 500.0,
  "processedRowsPerSecond": 627.1186440677966,
  "durationMs": {
    "addBatch": 31,
    "getBatch": 3,
    "getOffset": 1,
    "queryPlanning": 14,
    "triggerExecution": 59,
    "walCommit": 10
  },
  "stateOperators": [],
```

```
  "sources": [
    {
      "description": "KafkaSource[Subscribe[sensor-office-src]]",
      "startOffset": {
        "sensor-office-src": {
          "0": 606580
        }
      },
      "endOffset": {
        "sensor-office-src": {
          "0": 606617
        }
      },
      "numInputRows": 37,
      "inputRowsPerSecond": 500.0,
      "processedRowsPerSecond": 627.1186440677966
    }
  ],
  "sink": {
    "description": "MemorySink"
  }
}
```


As we can appreciate in the listing Example 14-2, each `StreamingQueryProgress` instance offers a comprehensive snapshot of the query performance produced at each trigger.

From the perspective of monitoring the job's performance, we are particularly intersted in`numInputRows`,`inputRowsPerSecond` and `processedRowsPerSecond`. These self-describing fields provide key indicators about the job performance. If we have more data than our query can process, `inputRowsPerSecond` will be higher than `processedRowsPerSecond` for sustained periods of time. This may indicate that the cluster resources allocated for this job should be increased to reach a sustainable long-term performance.

The `StreamingQueryListener` Interface

Monitoring is a "day 2 operations" concern and we require to have an automated collection of performance metrics to enable other processes, such as capacity management, alerting and operational support.

The inspection methods made available by the `StreamingQuery` handler that we saw in the previous section are useful when we work on an interactive environment such as the *Spark Shell* or a notebook, like we use in the exercises of this book. In an interactive setting, we have the opportunity to manually sample the output the `StreamingQueryProgress` to get an initial idea about the performance characteristics of our job.

Yet, the `StreamingQuery` methods are not automation friendly. Given that a new progress record becomes available at each streaming trigger, automating a method to collect information from this interface needs to be coupled to the internal scheduling of the streaming job.

Luckily, Structured Streaming provides the `StreamingQueryListener`, an *listener-based* interface that provides asynchronous callbacks to report updates in the lifecycle of a streaming job.

Implementing a `StreamingQueryListener`

To hook up to the internal event bus, we must provide an implementation of the `StreamingQueryListener` interface and register it to the running `SparkSession`.

`StreamingQueryListener` consists of three methods:

`onQueryStarted(event: QueryStartedEvent)`

> Called when a streaming query starts. The `event` provides an unique `id` for the query and a `runId` that changes is the query is stopped and restarted. This callback is called synchronously with the start of the query and should not be blocked.

`onQueryTerminated(event: QueryTerminatedEvent)`

> Called when a streaming query is stopped. The `event` contains `id` and `runId` fields that correlate with the start event. It also provides an `exception` field that contains an `exception` if the query failed due to an error.

`onQueryProgress(event: StreamingQueryProgress)`

> Called at each query trigger. The `event` contains a `progress` field that encapsulates a `StreamingQueryProgress` instance that we know already from "Getting Metrics with`StreamingQueryProgress`". This callback provides us with the events that we need to monitor the query performance.

The listing in <<>> illustrates the implementation of a simplified version of such listener. This `chartListener`, when instantiated from a notebook, plots the input and processing rates per second

Example 14-3. Plotting Streaming Job Performance

```
import org.apache.spark.sql.streaming.StreamingQueryListener
import org.apache.spark.sql.streaming.StreamingQueryListener._
val chartListener = new StreamingQueryListener() {
  val MaxDataPoints = 100
  // a mutable reference to an immutable container to buffer n data points
  var data: List[Metric] = Nil

  def onQueryStarted(event: QueryStartedEvent) = ()

  def onQueryTerminated(event: QueryTerminatedEvent) = ()

  def onQueryProgress(event: QueryProgressEvent) = {
    val queryProgress = event.progress
    // ignore zero-valued events
    if (queryProgress.numInputRows > 0) {
      val time = queryProgress.timestamp
      val input = Metric("in", time, event.progress.inputRowsPerSecond)
      val processed = Metric("proc", time, event.progress.processedRowsPerSecond)
      data = (input :: processed :: data).take(MaxDataPoints)
      chart.applyOn(data)
    }
  }
}
```

Once a listener instance has been defined, it must be attached to the event bus, using the `addListener` method in the `SparkSession`:

```
sparkSession.streams.addListener(chartListener)
```

After running this `chartListener` against one of the notebooks included in the book online resources, we can visualize the input and processing rates, like Figure 14-1 shows.

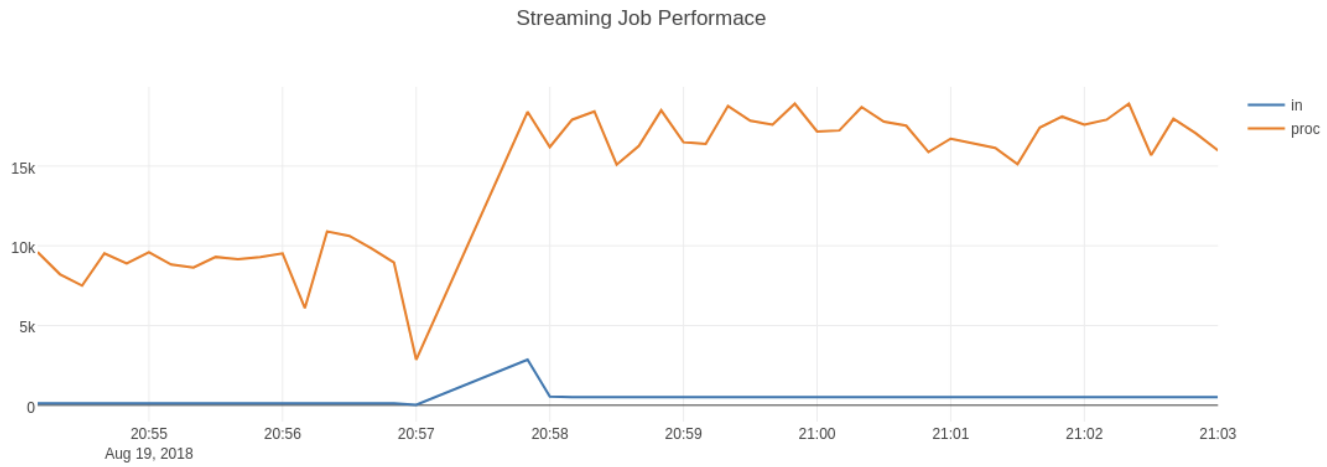


Figure 14-1. Input and Processing Streaming Rates

Similar listener implementations can be used to send metric reports to popular monitoring systems, such as Prometeus, Graphite, or queriable databases like InfluxDB, that can be easily integrated with dashboard applications such as Graphana.

Chapter 15. Experimental Areas: Continuous Processing and Machine Learning

Structured Streaming first appeared in Spark 2.0 as an experimental API, offering a new streaming model aimed at simplifying the way we think about streaming applications. In Spark 2.2, Structured Streaming "graduated" to *production-ready*, giving the signal that the this new model is ready for industry adoption. With Spark 2.3, we saw further improvements in the area of streaming joins and it also introduced a new experimental *continuous* execution model for low latency stream processing.

As with any new successful development, we can expect Structured Streaming to keep advancing at a fast pace. While industry adoption will contribute evolutionary feedback on important features, market trends such as the increasing popularity of machine learning will drive the roadmap of the releases to come.

In this chapter, we want to provide insights in some of the areas under development that will probably become mainstream in upcoming releases.

Continous Processing

Continous Processing is an alternative execution mode for Structured Streaming that allows for low-latency processing of individual events. It has been included as an experimental feature in Spark v2.3. It's still under active development, in particular in the areas of delivery semantics, stateful operation support and monitoring.

Understanding *Continuous Processing*

The initial streaming API for Spark, Spark Streaming, was conceived with the idea of reusing the batch capabilities of Spark. In a nutshell, the data stream gets splitted in small chunks that are given to Spark for processing, using the core engine in its native batch execution mode. Using a scheduled repetition of this process at short time intervals, the input stream is constantly consumed and results are produced in a streaming fashion. This is called the "micro-batch" model that we discussed early on in Chapter 3. We will study the application of this model in more detail when we talk about Spark Streaming in the next part of the book. The important part to remember for now, is that the definition of that interval of time, called *batch interval* is the keystone of the original *micro-batch*implementation.

MICRO-BATCH IN STRUCTURED STREAMING

When Structured Streaming was introduced, a similar evolution took place. Instead of introducing an alternative processing model, Structured Streaming got embedded into the `Dataset` API and reused the existing capabilities of the underlying Spark SQL engine. As a result, Structured Streaming offers an unified API with the more traditional *batch* mode and fully benefits from the performance optimizations introduced by Spark SQL, such as query optimization and Tungsten's code generation.

In that effort, the underlying engine got additional capabilities to sustain a streaming workload, like incremental query execution and the support of resilient state management over time.

At the API surface level, Structured Streaming avoided making the notion of time an explicit user-facing parameter. This is what allows for the implementation of event-time aggregations, as the notion of time is infered from the data stream instead. Internally, the execution engine still relies in a micro-batch architecture, but the abstraction of time allows for the creation of engines with different models of execution.

The first execution model that departs from the fixed-time micro-batch is the *best-effort* execution in Structured Streaming, which is the default mode when no `trigger` is specified. In *best effort* mode, the next micro-batch starts as soon as the previous one ends. This creates improves the observable continuity of the resulting stream and improves the usage of the underlying computing resources.

The micro-batch execution engine uses a task-dispatching model. Task dispatching and coordination over the cluster is rather expensive and the minimal is latency possible is ~100ms.

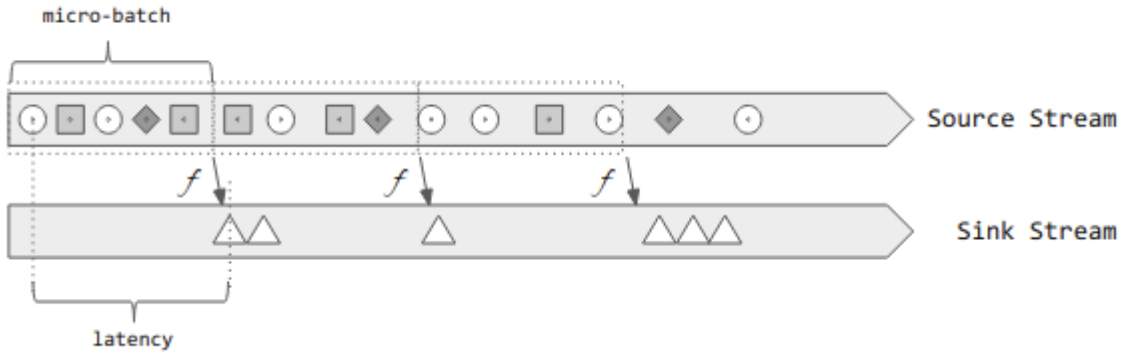


Figure 15-1. Micro-batch Latency

In Figure 15-1 we can appreciate how our process of filtering the incoming "circles" and transforming them to "triangles" works with the micro-batch model. We collect all elements that arrive in a certain interval and apply our function *f* to all of them at the same time.

The processing latency is the time duration between the arrival of the event in the *source* stream and the production of a result in the *sink*. As we can appreciate, in the micro-batch model, the latency upper limit is the batch interval + the time it takes to process the data, which consists in the computation itself and the coordination needed to execute such computation in some executor of the cluster.

INTRODUCING *CONTINUOUS PROCESSING*: A LOW LATENCY STREAMING MODE

Taking advantage of the time abstraction in Structured Streaming, it is possible to introduce new modes of execution without changing the user-facing API.

In the *Continuous Processing* execution mode, the data processing query is implemented as a long-running task that executes *continuously* on the executors. The parallelism model is simple: For each input partition, we will have such a task running on a node of the cluster This task will subscribe to

the input partition and continuously process incoming individual events. The deployment of a query under *continuous processing* creates a topology of tasks in the cluster.

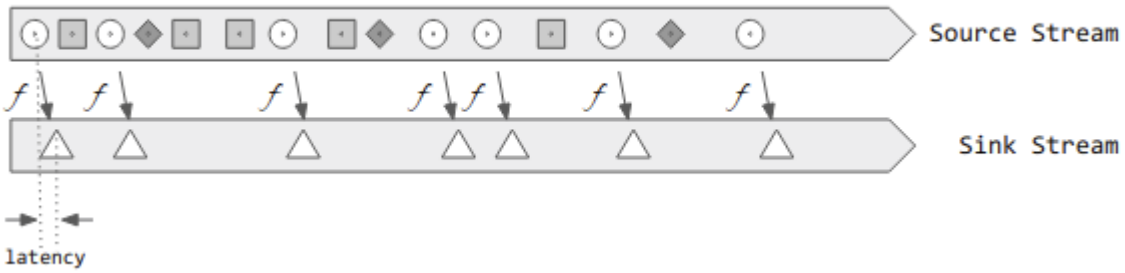


Figure 15-2. Continuous Processing Latency

As illustrated in Figure 15-2, this new execution model eliminates the micro-batch delay and produces a result for each element as soon as it's processed.

Using *Continuous Processing*

All that is required to use the *continuous processing* execution mode is to specify a `Trigger.Continuous` as `trigger` and provide it with the time interval for the checkpoint function, like we show in this minimalistics example:

```
import org.apache.spark.sql.streaming.Trigger

val stream = spark.readStream
    .format("rate")
    .option("rowsPerSecond", "5")
    .load()

val pairElements = stream.select($"timestamp", $"value").where($"value" % 2 === 0)

val query = pairElements.writeStream
    .format("console")
    .trigger(Trigger.Continuous("2 seconds"))
    .start()
```

Although there are no changes at the API level, there are a number of restrictions on the type of queries that are supported in *continous mode*. The intuition is that *contiuous mode* works with queries that can be applied to a per-element basis. In SQL terms, we can use selections, projections and transformations. In functional terms, we can use `filter`, `map`, `flatMap` and `mapPartitions`.

Continous mode does not make much sense on queries that require previous events, like windows. The very nature of a time period in a window and related concepts such as *watermarks* wouldn't benefit from the low-latency characteristcs of this execution model.

The support of arbitrary stateful processing such as `mapGroupsWithState` is currently under development.

Machine Learning

As the amount of available data and its rate of arrival increases, traditional techniques of understanding signals in the data become a major block to extract actionable insights from it.

Machine learning is, in essence, the combination of algorithms and statistical analysis techniques to *learn* from the data and use that learning to provide an answer to certain questions. Machine learning uses data to estimate a model, a mathematical representation of some aspect of the world. Once a model has been determined, it can be queried on existing or new data to obtain an answer.

The nature of the answer we want from the data, divides the aim of the machine learning algorithms in three groups:

- Regression: we want to predict a value in a continous range. Example: using data about student's number of absences and hours of study for a given class, predict the score in their final exam.
- Classification: we want to separate data points into one of several categories. Example: given a text sample, we want to estimate the language.
- Clustering: given a set of elements, we want to divide it into subsets using some notion of similarity. Example: in a online wine store, we want to group customers with similar purchase behavior.

In the learning process, we also have the notion of *supervision*. We talk about *supervised learning*when the algorithm being trained requires data that maps a number of observations to an outcome. Regression and classification techniques fall under *supervised learning*. Using our previous example of the exam score, to build or regression model, we will require a dataset of historical student performance that contains the exam scores along the number of absences and hours of study reported by the students. Obtaining *good* data is the most challenging aspect of a machine learning task.

Learning vs Exploiting

We can identify two phases in the application of machine learning techniques:

- a learning phase, where data is prepared and used to estimate a model. This is also known as *training* or *learning*
- an exploit phase, where the estimated model is queried on new data. This phase is known as *prediction* or *scoring*

The training phase in machine learning is typically done using historical datasets. These datasets are usually cleaned and prepared for the target application. The machine learning methodology also calls for a validation phase in which the resulting model is evaluated against a dataset of known results, usually called *testing* or *validation* set. The result of the testing phase are metrics that report how well the learned model performs on data that it didn't see during the training.

SPARK MLLIB: MACHINE LEARNING SUPPORT IN SPARK

As part of the effort to offer an unified data processing API based on the concepts of *structured data*, Spark MLLib, the machine learning library in Spark, also adopted the `Dataset` API, through the introduction of the *ML Pipelines* concept.

ML Pipelines is a high level API for MLLib built on top of `DataFrame`s. It combines the idea of dynamic `DataFrame` transformations with the schema-driven capability of addressing specific fields in the data as the values that participate in the machine learning process. This reduces the typical burden of preparing separate data artifacts in specific formats, such as *vectors*, used as input to the MLLib implementation of the algorithm used.

For an in-depth coverage of Machine Learning using Apache Spark, we recommend*Advanced Analytics with Spark, 2nd Edition* by Sean Owen, Sandy Ryza, et al

Applying a ML Model to a Stream

As we mentioned earlier, creating a machine learning model is usually a batch-based process that uses historical data to train a model. Once that model is available, it can be used to *score* new data to obtain an estimate of the specific aspect that the model was trained for.

The unified *structured* APIs of Apache Spark across batch, machine learning, and streaming make it straightforward to apply a trained model on a streaming `DataFrame`.

Assuming that the model is stored on disk, the process consist of two steps:

1. Load the model
2. Use its `transform` method to apply the model to the streaming `DataFrame`

Let's see the API in action with an example:

During the development of this section, we have been using sensor information as a running theme. Up to now, we have used the sensor data to explore the data processing and analytic capabilities of Structured Streaming. Now, imagine that we have such ambient sensors in a series of rooms, but instead of keeping track of temperature or humidity data over time, we want to use that information to drive a novel application. We would like to estimate whether or not the room is occupied at a certain moment by using the sensor data. While probably temperature or humidity alone are not sufficient to tell whether a room is in use, maybe a combination of these factors is able to predict occupancy to a certain degree of accuracy.

For this example, we are going to use an occupancy dataset that was collected with the intention of answering that question. The dataset, that can be downloaded from *https://archive.ics.uci.edu/ml/datasets/Occupancy+Detection+*, consist of the following schema:

```
|-- id: integer (nullable = true)

|-- date: timestamp (nullable = true)

|-- Temperature: double (nullable = true)
```

```
|-- Humidity: double (nullable = true)

|-- Light: double (nullable = true)

|-- CO2: double (nullable = true)

|-- HumidityRatio: double (nullable = true)

|-- Occupancy: integer (nullable = true)
```

The *occupancy* information in the training dataset was obtained using camera images of the room to detect with certainty the presence of people in it.

Using this data, we trained a *Logistic Regression* model that estimates the occupancy, represented by the binomial outcome [0,1], where `0 = not occupied` and `1 = occupied`.

NOTE

For this example, we assume that the trained model is already available on disk. The complete training phase of this example is available in the book's online resources.

The first step is to load the previously trained model.

```
$ import org.apache.spark.ml._
$ val pipelineModel = PipelineModel.read.load(modelFile)
>pipelineModel: org.apache.spark.ml.PipelineModel = pipeline_5b323b4dfffd
```

This call results in a `model` that contains information over the stages of our `Pipeline`.

With the call `model.stages`, we can visualize these stages:

```
$ model.stages
res16: Array[org.apache.spark.ml.Transformer] =
    Array(vecAssembler_7582c780b304, logreg_52e582f4bdb0)
```

Our *pipeline* consists of two stages: A `VectorAssembler` and a `LogisticRegression` classifier. The `VectorAssembler` is a transformation that selectively transforms the chosen fields in the input data into a numeric `Vector` that serves as input for the model. The *LogisticRegression* stage is the trained logistic regression classifier. It uses the learned parameters to transforms the input *Vector* into three fields that are added to the *Streaming* `DataFrame`: `rawPrediction`, `probability` and `prediction`.

For our application, we are interested in the `prediction` value that will tell us whether the room is in use (1) or not (0).

The next step is to apply the model to the *streaming* `DataFrame`.

Example 15-1. Using a trained ML model in Structured Streaming

```
// let's assume an existing sensorDataStream
$ val scoredStream = pipeline.transform(sensorDataStream)
```

```
// inspect the schema of the resulting DataFrame
$ scoredStream.printSchema
root
  |-- id: long (nullable = true)
  |-- timestamp: timestamp (nullable = true)
  |-- date: timestamp (nullable = true)
  |-- Temperature: double (nullable = true)
  |-- Humidity: double (nullable = true)
  |-- Light: double (nullable = true)
  |-- CO2: double (nullable = true)
  |-- HumidityRatio: double (nullable = true)
  |-- Occupancy: integer (nullable = true)
  |-- features: vector (nullable = true)
  |-- rawPrediction: vector (nullable = true)
  |-- probability: vector (nullable = true)
  |-- prediction: double (nullable = false)
```

At this point, we have a streaming `DataFrame` that contains the prediction of our original streaming data.

The final step in our streaming prediction is to do something with the prediction data. In this example, we are going to limit this step to querying the data using the *memory sink* to access the resulting data as a *SQL table*.

```
import org.apache.spark.sql.streaming.Trigger
val query = scoredStream.writeStream
        .format("memory")
        .queryName("occ_pred")
        .start()

// let the stream run for a while first so that the table gets populated
sparkSession.sql("select id, timestamp, occupancy, prediction from occ_pred")
            .show(10, false)

+---+-----------------------+---------+----------+
|id |timestamp              |occupancy|prediction|
+---+-----------------------+---------+----------+
|211|2018-08-06 00:13:15.687|1        |1.0       |
|212|2018-08-06 00:13:16.687|1        |1.0       |
|213|2018-08-06 00:13:17.687|1        |1.0       |
|214|2018-08-06 00:13:18.687|1        |1.0       |
|215|2018-08-06 00:13:19.687|1        |1.0       |
|216|2018-08-06 00:13:20.687|1        |0.0       |
|217|2018-08-06 00:13:21.687|1        |0.0       |
|218|2018-08-06 00:13:22.687|0        |0.0       |
|219|2018-08-06 00:13:23.687|0        |0.0       |
|220|2018-08-06 00:13:24.687|0        |0.0       |
+---+-----------------------+---------+----------+
```

Given that we are using a test dataset to drive our stream, we also have access to the original occupancy data. In this limited sample, we can observe that the actual *occupancy* and the *prediction*are accurate most but not all the time.

For real-world application, we will typically be interested in offering this service to other applications. Maybe in the form of an HTTP-based API or through pub/sub messaging interactions. We can use any of the available *sinks* to write the results to other system for further use.

THE CHALLENGE OF MODEL SERVING

Trained machine learning models are seldom perfect. There are always opportunities to train a model with more or better data, or tweak its parameters to improve the prediction accuracy. With ever evolving trained models, the challenge becomes to upgrade our streaming scoring process with a new model whenever it becomes available.

This process of managing the lifecycle of ML models from the training stage to their explotation in an application is usually known by the broad concept of *model serving*.

Model serving comprises the process of transitioning trained models into a production platform and keeping those online *serving* processes up to date with the most recent trained models.

MODEL SERVING IN STRUCTURED STREAMING

In Structured Streaming, updating a running query is not possible. Like we saw in Example 15-1, we include the model scoring step as a transformation in our streaming process. Once we start the corresponding streaming query, that declaration becomes part of the query plan that gets deployed and will run until the query is stopped. Hence, updating ML models in Structured Streaming is not directly supported. It is, nevertheless, possible to create a managing system that calls the Structured Streaming APIs to stop, update and restart a query for which a new model becomes available.

The topic of *model serving* is an ongoing discussion in the Spark community and will certainly see an evolution in future versions of Spark and Structured Streaming.

Online Training

In the machine learning process we described earlier, we made a distiction between the learning and scoring phases, where the learning step was mainly an offline process. In the context of a streaming application, it is possible to train a machine learning model as data arrives. This is also called *online learning*.

Online learning is particularly interesting when we want to adapt to evolving patterns in the data, such as the changing interests of a social network or trend analysis in finantial markets.

Online learning poses a new set of challenges, as its implementation mandates that each data point is observed only once and that should take into consideration that the total amount of data observed might be endless.

In its current form, Structured Streaming does not offer support for *online training*. There are efforts to implement some (limited) form of online learning on Structured Streaming, the most notable being: - Holden Karau and Seth Hendrickson [1] - Ram Sriharsha and Vlad Feinberg [2]

It would seem that early initiatives to implement online learning on top of Structured Streaming have lost momentum. This might change in the future, so check new releases of Structured Streaming for potential updates in this area.

[1] *https://www.oreilly.com/learning/extend-structured-streaming-for-spark-ml*

[2]*https://www.youtube.com/watch?v=r0hyjmLMMOc*