

Part I. Fundamentals of Stream Processing with Apache Spark

The first part of this book is dedicated to building solid foundations on the concepts that underpin stream processing and a theoretical understanding of Apache Spark as a streaming engine.

We begin with a discussion on what motivating drivers are behind the adoption of stream-processing techniques and systems in the enterprise today ([Chapter 1](#)). We then establish vocabulary and concepts common to stream processing ([Chapter 2](#)). Next, we take a quick look at how we got to the current state of the art as we discuss different streaming architectures ([Chapter 3](#)) and outline a theoretical understanding of Apache Spark as a streaming engine ([Chapter 4](#)).

At this point, the readers have the opportunity to directly jump to the more practical-oriented discussion of Structured Streaming in [Part II](#) or Spark Streaming in [Part III](#).

For those who prefer to gain a deeper understanding before adventuring into APIs and runtimes, we suggest that you continue reading about Spark's Distributed Processing model in [Chapter 5](#), in

which we lay the core concepts that will later help you to better understand the different implementations, options, and features offered by Spark Streaming and Structured Streaming.

In Chapter 6, we deepen our understanding of the resilience model implemented by Spark and how it takes away the pain from the developer to implement robust streaming applications that can run enterprise-critical workloads 24/7.

With this new knowledge, we are ready to venture into the two streaming APIs of Spark, which we do in the subsequent parts of this book.

Chapter 1. Introducing Stream Processing

In 2011, Marc Andreessen famously said that “software is eating the world,” referring to the booming digital economy, at a time when 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 of the clerk, make up our mind, and finally buy a model that fulfilled our desires and expectations. After finishing our purchase, the shop would have registered a credit card transaction—or only a cash balance change in case of a cash payment—and the shop manager would that know they have one less inventory item of that particular camera model.

Now, let’s take that experience online: first, we begin searching the web. We visit a couple of online stores, leaving digital traces as we pass from one to another. Advertisements on websites suddenly begin 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 is registered and linked to the purchase. While we

complete our purchase, we are offered additional options that are allegedly popular with other people who bought the same camera. Each of our digital interactions, like searching for keywords on the web, clicking some link, or spending time reading a particular page generates a series of events that are collected and transformed into business value, like personalized advertisement or upsale recommendations.

Commenting on Andreessen's 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 in provides a technical and strategical advantage. Examples of this ongoing adoption include sectors such as internet commerce, continuously running data pipelines created by businesses that interact with customers on a 24/7 basis, or credit card companies, analyzing transactions as they happen in order 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, surveillance cameras,

thermostats, and so on. 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 facilitate the analysis of streams of data.

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

In this chapter, we 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 stream processing is, its applications, and its challenges. After we build that 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 on 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 follows:

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

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. We can count the number of elements in a bounded dataset.

Batch Versus 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 retrievable 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 need 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
- In motion, as continuously generated sequences 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 is 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, whether stored in files or contained in databases, were initially a stream of data collected over time into some storage. The user's database, all the orders from the last quarter, the GPS coordinates of taxi trips in a city, and so on all 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 datacenter, or much longer, like GPS data of a car traveling through a tunnel that is dispatched only when the vehicle reestablishes its wireless connectivity after 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 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 one another.

The Factor of Uncertainty

In a 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 once, or following a certain rhythm. This means that, because 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, our system will produce results as expected, but if we have not planned for such a burst of input data, some streaming systems might face delays, resource constriction, or 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 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, innovative applications of data. The following use cases, in which the authors have been involved in one way or another, are only a small sample that we use to illustrate the wide spectrum of application of stream processing:

Device monitoring

A small startup rolled out a cloud-based Internet of Things (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 minimum/maximum 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 moved its billing system to a streaming pipeline. Batch exports from its 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 it to enforce rules like

geographical limits and analyze driver behavior regarding speed limits.

Media recommendations

A national media company deployed a streaming pipeline to ingest new videos, such as news reports, into its recommendation system, making the videos available to its users' personalized suggestions almost as soon as they are ingested into the company's media repository. The company's 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 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: although *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 given 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 stream processing is and some examples of how it is being used today, it's time to delve 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 2003. Jeff Dean and Sanjay Gemawhat, after going through a couple of iterations of rewriting Google's crawling and indexing systems, began 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 of 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 filesystem, one chunk per executor machine. Then, all of 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 that we would want to do on a dataset.

The Lesson Learned: Scalability and Fault Tolerance

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

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

Although these characteristics make the model attractive, the main success of MapReduce is its ability to sustain growth. As data volumes increase and growing business requirements lead to more information-extraction jobs, the MapReduce model demonstrates 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. All data is replicated. If a data-carrying executor crashes, it is enough to relaunch the task that was running on the crashed executor. Because the master keeps track of that task, that does not pose any particular problem other than rescheduling.

These two characteristics combined result 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 although batch processing has access to the complete dataset, with streams, we see only a small portion of the dataset at any time.

This situation becomes aggravated in a distributed system; that is, 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 lets 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 classic batch approach would be to wait until all votes have been cast and then proceed to count them. Even though this approach produces a correct end result, it would make for very boring news over the day because 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 cast. At any moment, we have a partial count by participant that lets us see the current standing 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 cannot affect the final result. We can't risk declaring 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 burdens 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 you will see throughout 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 can 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 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 breakthrough 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

100 times faster processing than Hadoop MapReduce, the open source implementation of Google's MapReduce for batch workloads.

Its core abstraction, the *Resilient Distributed Dataset* (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`, and `filter` express the lazy conversion of the data from one internal representation to another, whereas 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 a schema. It makes it possible to query a comma-separated values (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; it was now accessible to data scientists, 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 while using fewer resources.

A Unified Engine

Nowadays, Spark is a unified analytics engine offering batch and streaming capabilities 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 equally advanced and is highly complementary to streaming applications. Spark's unified programming model means that developers need to learn only one new paradigm to address both batch and streaming workloads.

NOTE

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

Spark Components

Figure 1-1 illustrates how 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 that are within the scope of this book and grayed out those that are not covered. To learn

more about these other areas of Apache Spark, we recommend *Spark, The Definitive Guide* by Bill Chambers and Matei Zaharia (O'Reilly), and *High Performance Spark* by Holden Karau and Rachel Warren (O'Reilly).

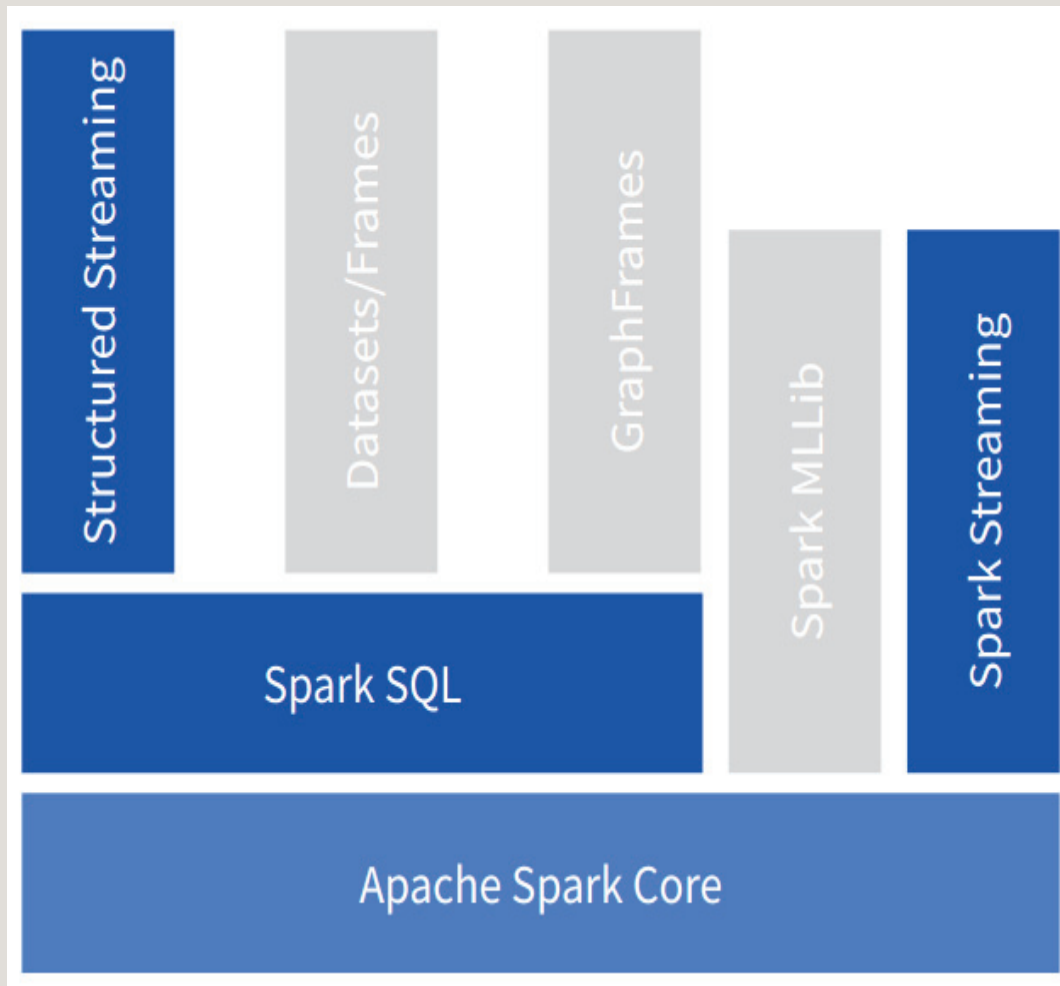


Figure 1-1. Abstraction layers (horizontal) and libraries (vertical) offered by Spark

As abstraction layers in Spark, we have the following:

Spark Core

Contains the Spark core execution engine and a set of low-level functional APIs used 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, in which 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 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: Spark Streaming and Structured 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 on which Spark could operate. This approach to stream processing is called the *microbatch* model; this 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 as the Spark core, but it introduces a new abstraction, the *Discretized Stream* or *DStream*, which 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 eligible 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. 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 microbatch approach, whereas 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 either of these two APIs right away, you could directly jump to Structured Streaming in [Part II](#) or Spark Streaming in [Part III](#).

If you are not familiar with stream processing, we recommend that you continue through this initial part of the book because we build the vocabulary and common concepts that we use in the discussion of the specific frameworks.

Chapter 2. Stream-Processing Model

In this chapter, we bridge the notion of a data stream—a source of data “on the move”—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 want to cover the following as components of stream processing:

- Data sources
- Stream-processing pipelines
- Data sinks

We then show how those concepts map to the specific stream-processing model implemented by Apache Spark.

Next, we characterize stateful stream processing, a type of stream processing that requires bookkeeping of past computations in the form of some intermediate state needed to process new data. Finally, we consider streams of timestamped 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 earlier, 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 as soon as 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 referred 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, and so on.

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 third-party integrations.

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.

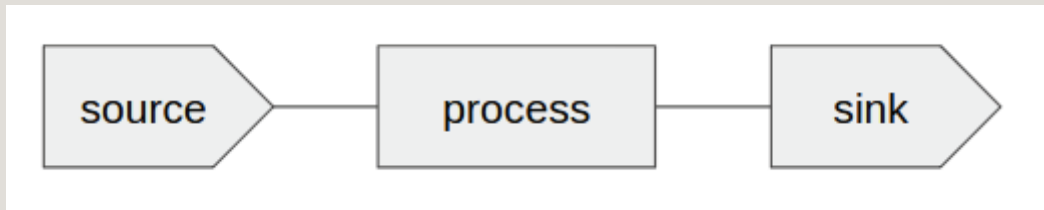


Figure 2-1. Simplified streaming model

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 One Another

Between sources and sinks lie 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 use 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 transformations have *narrow dependencies* (to produce one element of the output, you need only one of the elements of the input), whereas 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.

Although 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: social media messages, clicks on web pages, ecommerce transactions, 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. Even though seeing every transaction individually might not be useful or even practical, we might 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 nonoverlapping. We call this grouping of a fixed time period, in which each group follows the previous and does not overlap, *tumbling windows*.

Tumbling windows are the norm when we need to produce aggregates of our data over regular periods of time, with each period independent from previous periods. [Figure 2-2](#) shows a tumbling window of 10 seconds over a stream of elements. This illustration demonstrates the tumbling nature of tumbling windows.

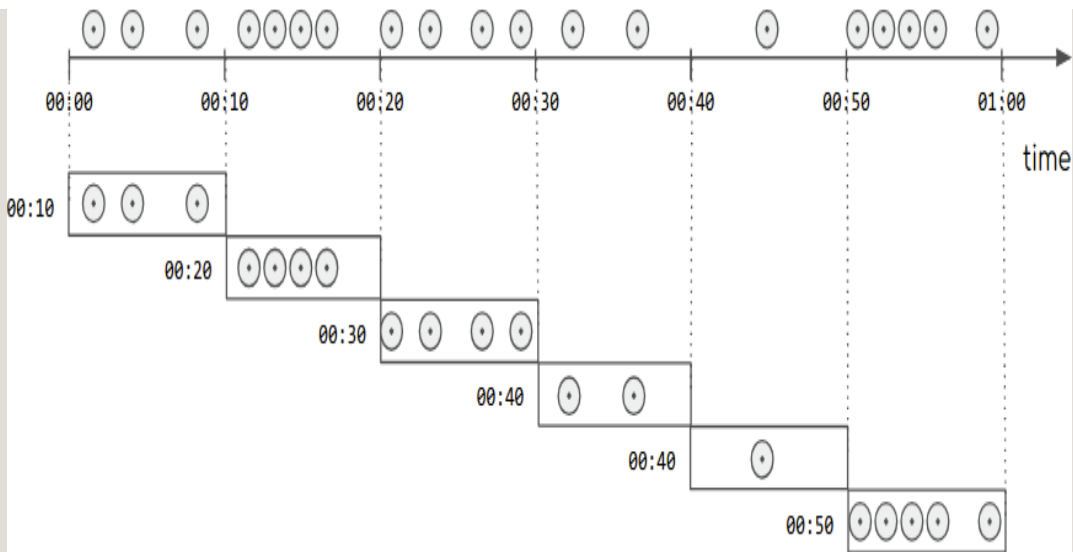


Figure 2-2. Tumbling windows

Sliding Windows

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 a *moving average*.

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 an 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.

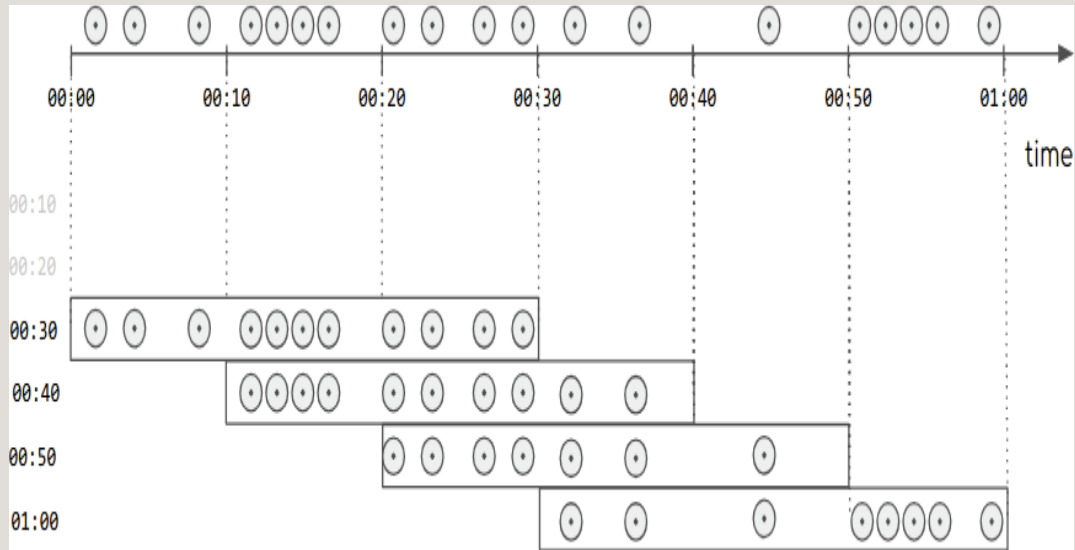


Figure 2-3. Sliding windows

Although you cannot see it 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 elements in place.

It's worth noting that tumbling windows are a particular case of sliding windows in which 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 that we want to apply on data streams. In our context,

data streams are fundamentally 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 in which 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 into the kind of computation that we might find interesting. In some cases, the emphasis is put on the continuous and independent processing of elements or groups of elements: those are the cases for which we want to operate on 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 computing recent aggregate statistics from event data. For example, it might 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 new data in the context of data already seen, often leads us to *stateful stream processing*. Stateful stream processing is the discipline by which we 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 that 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 take advantage of 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 five standard deviations from this mean. This is a simple, but useful, way of marking particular extreme outliers of the distribution of our input elements.¹ In this case, the internal state of the stream processor only stores the running mean and standard deviation of our stream—that is, a couple of numbers.

BOUNDING THE SIZE OF THE STATE

One of the common pitfalls of practitioners 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 you 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 because 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 might 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 on which they are observed, 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 might be storing at any given time is strictly bounded to a certain upper limit that is less than your available memory, independent of the amount of data that you might encounter as input.

An Example: Local Stateful Computation in Scala

To gain intuition into the concept of statefulness without having to go into the complexity of distributed stream processing, we begin with a simple nondistributed stream example in Scala.

The Fibonacci Sequence is classically 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, as shown in [Example 2-1](#).

Example 2-1. 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 looks to past information to obtain its result. It's necessary 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, in which 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 classical Fibonacci Sequence.

The important point to highlight is that to get the value at the *n*th place, we must process all *n-1* elements, keeping the intermediate $(i-1, i)$ elements as we move along the stream.

Would it be 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 that uses a stateless map function to transform each number to its Fibonacci value. We can see the implementation of this approach in [Example 2-2](#).

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

```
scala> import scala.math.{pow, sqrt}
import scala.math.{pow, sqrt}

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

scala> def fibonacciNumber(x: Int): Int =
  ((pow(phi,x) - pow(-phi,-x))/sqrt(5)).toInt
fibonacciNumber: (x: Int)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> val fibonacciSequence = integers.map(fibonacciNumber)
fibonacciSequence: scala.collection.immutable.Stream[Int] =
Stream(0, ?)

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

This rather counterintuitive 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, without requiring the previous elements; that is, without requiring knowledge of the state of the stream.

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.

Stateless or Stateful Streaming

We illustrated the difference between stateful and stateless stream processing with a rather simple case that has a solution using the two approaches. Although the stateful version closely resembles the definition, it requires more computing resources to produce a result: it needs to traverse a stream and keep intermediate values at each step.

The stateless version, although contrived, uses a simpler approach: we use a stateless function to obtain a result. It doesn't matter whether we need the Fibonacci number for 9 or 999999, in both cases the computational cost is roughly of the same order.

We can generalize this idea to stream processing. Stateful processing is more costly in terms of the resources it uses and also introduces concerns in face of failure: what happens if our computation fails halfway through the stream? Although a safe rule of thumb is to choose for the stateless option, if available, many of the interesting

questions we can ask over a stream of data are often stateful in nature. For example: how long was the user session on our site? What was the path the taxi used across the city? What is the moving average of the pressure sensor on an industrial machine?

Throughout the book, we will see that stateful computations are more general but they carry their own constraints. It's an important aspect of the stream-processing framework to provide facilities to deal with these constraints and free the user to create the solutions that the business needs dictate.

The Effect of Time

So far, we have considered how there is an advantage in keeping track of intermediary data as we produce results on each element of the 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 Timestamped 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 when the event actually happened. When the capabilities of the system sensing the event allow for it, this time is usually added as part of the message payload in the stream.

Timestamping is an operation that consists of 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 logs in financial transaction systems.

Timestamps as the Provider of the Notion of Time

The importance of time stamping is that it allows users to reason on their data considering the moment at which it was generated.

For example, if I register my morning jog using a wearable device and I synchronize the device to my phone when I get back home, I would like to see the details of my heart rate and speed as I ran through the forest moments ago, and not see the data as a timeless sequence of values as they are being uploaded to some cloud server. As we can see, timestamps provide the context of time to data.

So, because event logs form a large 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 cluster that processes it is an operation prone to different forms of failure 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 the following:

- Clearly marked correct and reordered results
- Intermediary prospective results

With that classification reflecting the best knowledge that a stream-processing system has of the timestamped events delivered by the data stream and under the proviso that this view could be completed by the late arrival of delayed stream elements. This process constitutes the basis for *event-time processing*.

In Spark, this feature is offered natively only by Structured Streaming. Even though Spark Streaming lacks built-in support for event-time processing, it is a question of development effort and some data consolidation processes to manually implement the same sort of primitives, as you will see in [Chapter 22](#).

Event Time Versus 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 were 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 relevant only 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.

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

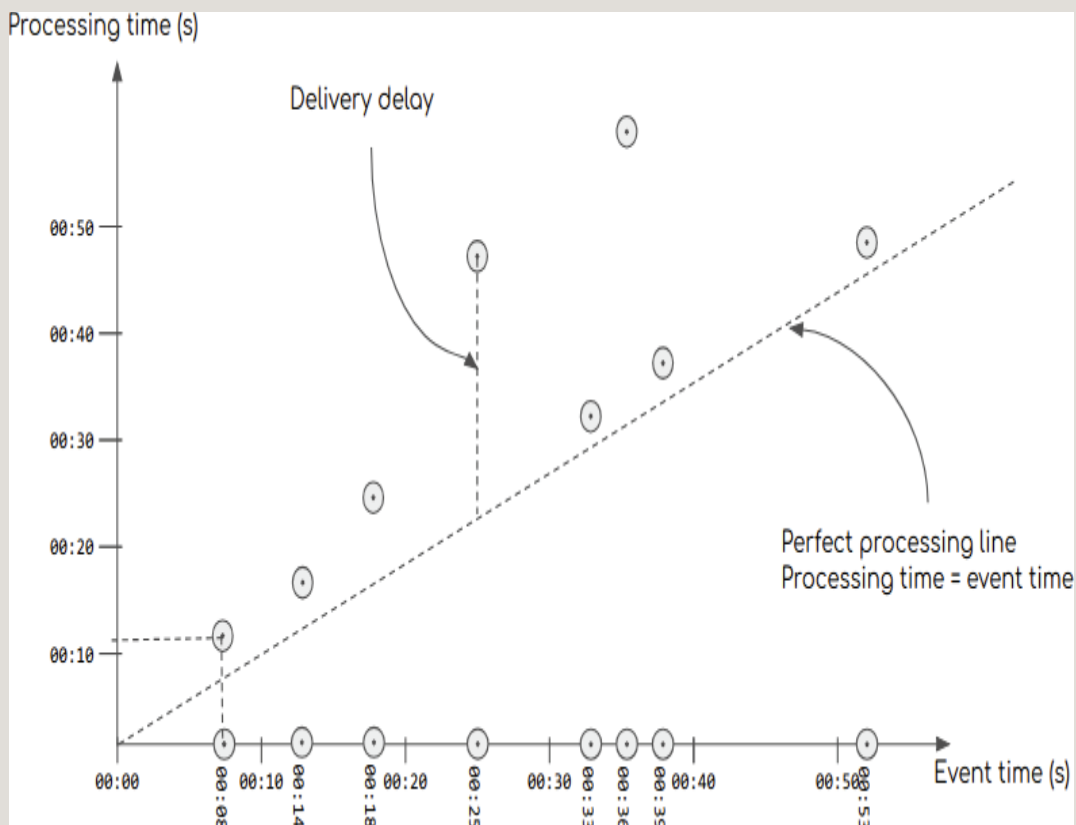


Figure 2-4. Event versus processing time

Let's look at this more closely:

- 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 approximately 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, events are processed immediately as they are created. Note that there can be no processing events below that line, because it would mean that events are 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, let's now consider a 10-second window aggregation, as demonstrated in [Figure 2-5](#).

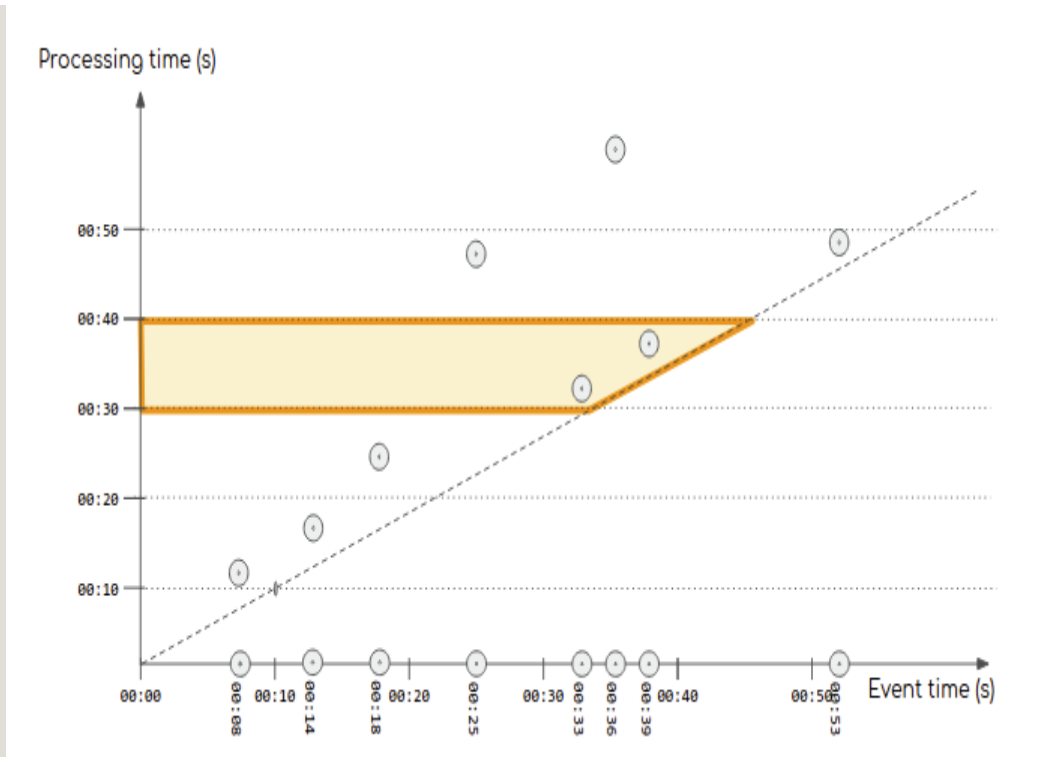


Figure 2-5. Processing-time windows

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, although we would assume that a 00:30–00:40 window would contain the event 00:36, we can see that it has fallen out of the resulting set because it was late.

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. [Figure 2-6](#) illustrates how these windows look radically different from the processing-time windows we saw earlier. 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 is dropped by the network and never arrives? How long do we wait? To resolve this problem, we introduce an arbitrary deadline called a *watermark* to deal with the consequences of this open boundary, like lateness, ordering, and deduplication.

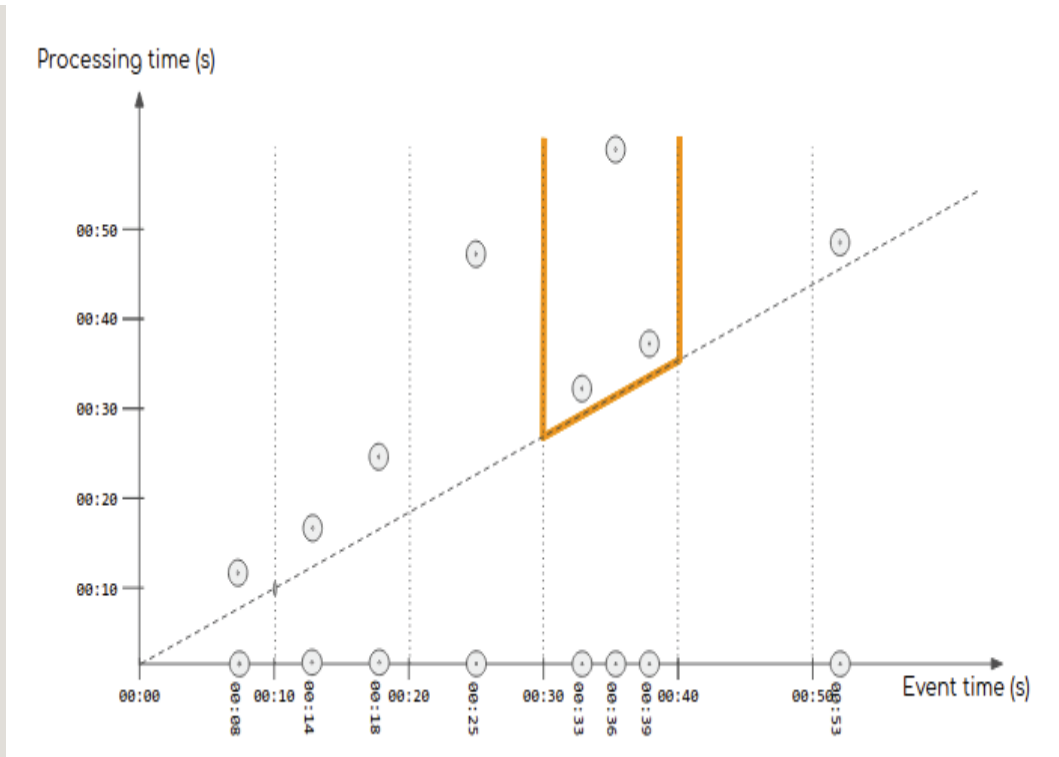


Figure 2-6. Event-time windows

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 use the timestamp contained in the event messages, the stream processor is able to bucket those messages into two categories based on a notion of a watermark.

The watermark is, at any given moment, *the oldest timestamp that we will accept on the data stream*. Any events that are older than this expectation are not taken into the results of the stream processing. The streaming engine can choose to process them in an alternative way, like report them in a *late arrivals* channel, for example.

However, to account for possibly delayed events, this watermark is usually much larger than the average delay we expect in the delivery of the events. Note also that this watermark is a fluid value that monotonically increases over time,² sliding a window of delay-tolerance as the time observed in the data-stream progresses.

When we apply this concept of watermark to our event-time diagram, as illustrated in Figure 2-7, we can appreciate that the watermark closes the open boundary left by the definition of event-time window, providing criteria to decide what events belong to the window, and what events are too late to be considered for processing.

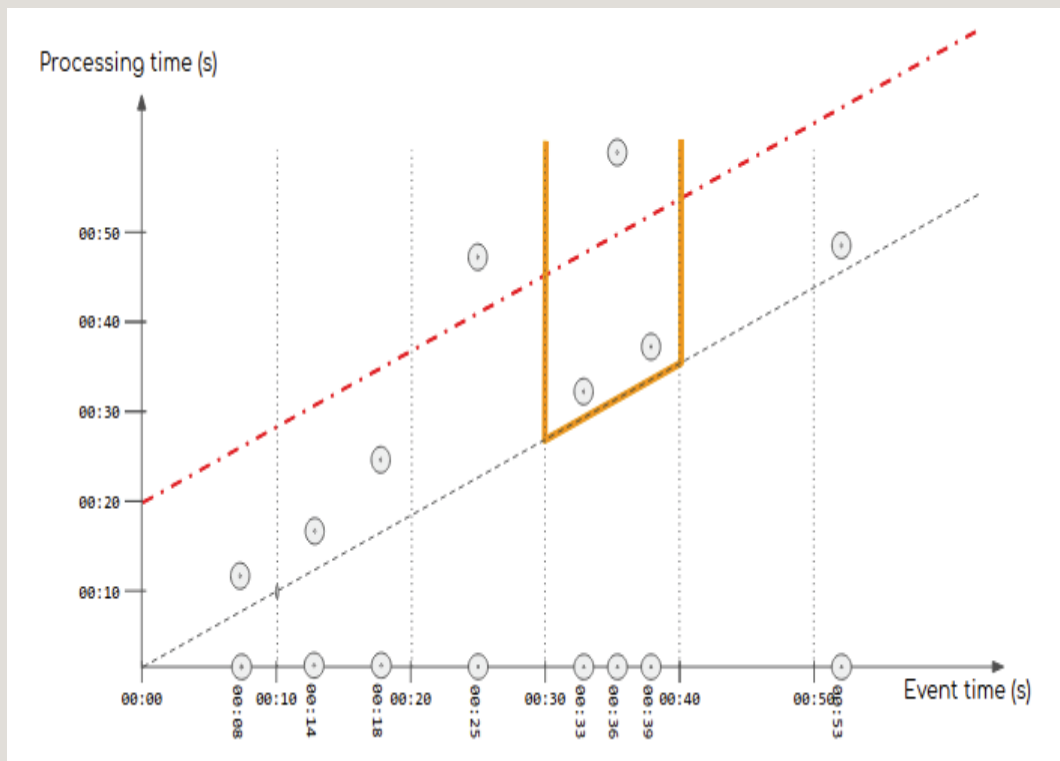


Figure 2-7. Watermark in event time

After this notion of watermark is defined for a stream, the stream processor can operate in one of two modes with relation to that

specific stream: 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 older than that 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, we can consider the output as provisional because newer data can still change the final outcome, whereas in the former case, the result is final and no new data will be able to change it.

We examine in detail how to concretely express and operate this sort of computation in [Chapter 12](#).

Note finally that with provisional 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, event-time processing is another form of stateful computation and is subject to the same limitation: to handle watermarks, the stream processor needs to store a lot of intermediate data and, as such, consume a significant amount of memory that roughly corresponds to *the length of the watermark \times the rate of arrival \times message size*.

Also, since we need to wait for the watermark to expire to be sure that we have all elements that comprise an interval, stream processes that use a watermark and want to have a unique, final result for each interval, must delay their output for at least the length of the watermark.

CAUTION

We want to outline event-time processing because 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 event-time processing, we make the assumption that setting our watermark to a certain value is appropriate. That is, we can expect the results of a streaming computation based on event-time processing to be meaningful only 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.

A too small watermark will lead to dropping too many events and produce severely incomplete results. A too large watermark will delay the output of results deemed complete for too long and increase the resource needs of the stream processing system to preserve all intermediate events.

It is left to the users to ensure they choose a watermark suitable for the event-time processing they require and appropriate for the computing resources they have available, as well.

Summary

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

- Data sources and sinks
- Stateful processing
- Event-time processing

We explore the implementation of these concepts in the streaming APIs of Apache Spark as we progress through the book.

-
- 1 Thanks to the Chebycheff inequality, we know that alerts on this data stream should occur with less than 5% probability.
 - 2 Watermarks are nondecreasing by nature.

Chapter 3. Streaming Architectures

The implementation of a distributed data analytics system has to deal with the management of a pool of computational resources, as in-house clusters of machines or reserved cloud-based capacity, to satisfy the computational needs of a division or even an entire company. Since teams and projects rarely have the same needs over time, clusters of computers are best amortized if they are a shared resource among a few teams, which requires dealing with the problem of multitenancy.

When the needs of two teams differ, it becomes important to give each a fair and secure access to the resources for the cluster, while making sure the computing resources are best utilized over time.

This need has forced people using large clusters to address this heterogeneity with modularity, making several functional blocks emerge as interchangeable pieces of a data platform. For example, when we refer to database storage as the functional block, the most common component that delivers that functionality is a relational database such as PostgreSQL or MySQL, but when the streaming application needs to write data at a very high throughput, a scalable column-oriented database like Apache Cassandra would be a much better choice.

In this chapter, we briefly explore the different parts that comprise the architecture of a streaming data platform and see the position of a processing engine relative to the other components needed for a complete solution. After we have a good view of the different elements in a streaming architecture, we explore two architectural styles used to approach streaming applications: the Lambda and the Kappa architectures.

Components of a Data Platform

We can see a data platform as a composition of standard components that are expected to be useful to most stakeholders and specialized systems that serve a purpose specific to the challenges that the business wants to address.

Figure 3-1 illustrates the pieces of this puzzle.

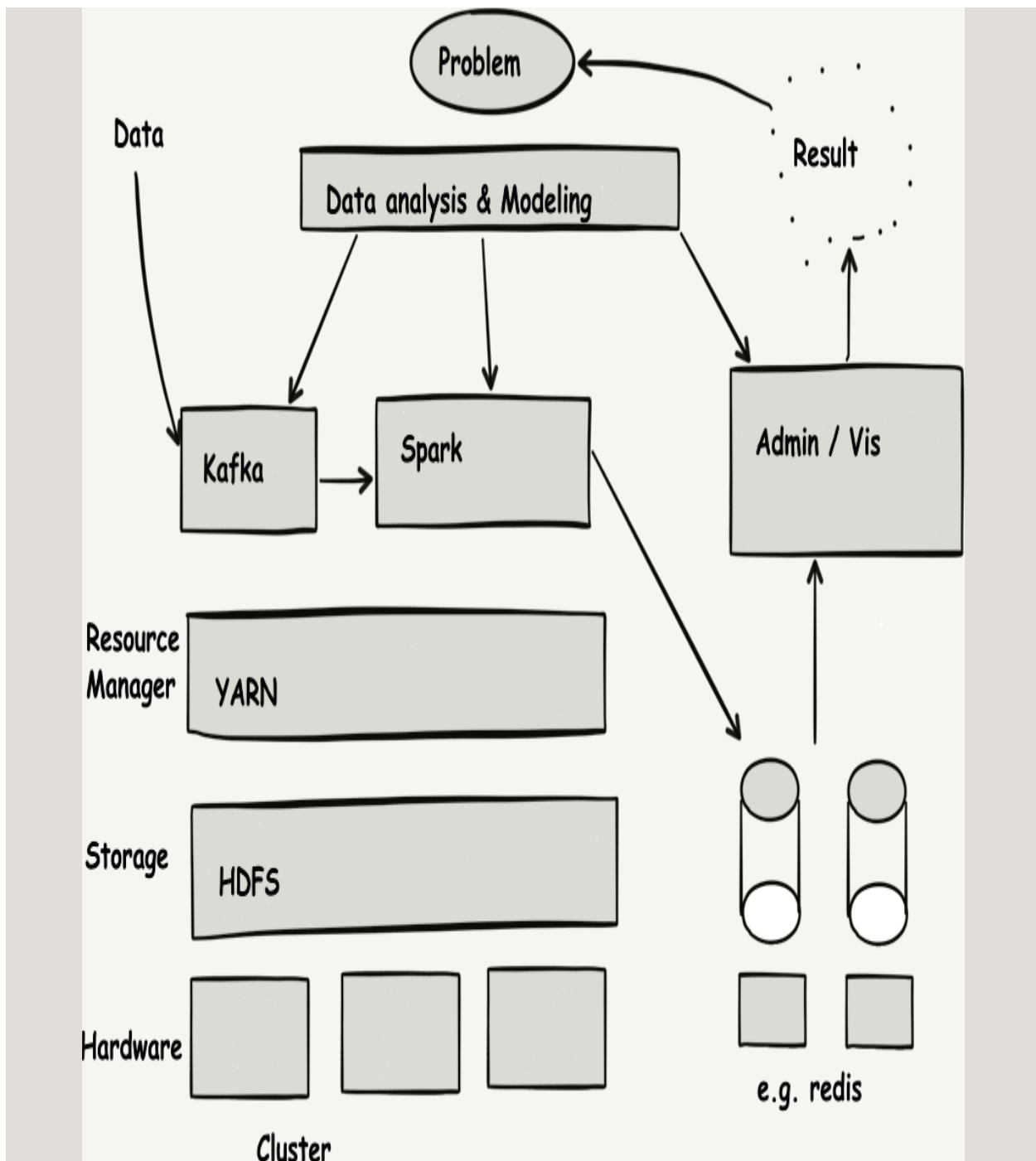


Figure 3-1. The parts of a data platform

Going from the bare-metal level at the bottom of the schema to the actual data processing demanded by a business requirement, you could find the following:

The hardware level

On-premises hardware installations, datacenters, or potentially virtualized in homogeneous cloud solutions (such as the T-shirt size offerings of Amazon, Google, or Microsoft), with a base operating system installed.

The persistence level

On top of that baseline infrastructure, it is often expected that machines offer a shared interface to a persistence solution to store the results of their computation as well as perhaps its input. At this level, you would find distributed storage solutions like the Hadoop Distributed File System (HDFS)—among many other distributed storage systems. On the cloud, this persistence layer is provided by a dedicated service such as Amazon Simple Storage Service (Amazon S3) or Google Cloud Storage.

The resource manager

After persistence, most cluster architectures offer a single point of negotiation to submit jobs to be executed on the cluster. This is the task of the resource manager, like YARN and Mesos, and the more evolved *schedulers* of the *cloud-native* era, like Kubernetes.

The execution engine

At an even higher level, there is the execution engine, which is tasked with executing the actual computation. Its defining characteristic is that it holds the interface with the programmer's input and describes the data manipulation. Apache Spark, Apache Flink, or MapReduce would be examples of this.

A data ingestion component

Besides the execution engine, you could find a data ingestion server that could be plugged directly into that engine. Indeed, the old practice of reading data from a distributed filesystem is often

supplemented or even replaced by another data source that can be queried in real time. The realm of messaging systems or log processing engines such as Apache Kafka is set at this level.

A processed data sink

On the output side of an execution engine, you will frequently find a high-level data sink, which might be either another analytics system (in the case of an execution engine tasked with an *Extract, Transform and Load* [ETL] job), a NoSQL database, or some other service.

A visualization layer

We should note that because the results of data-processing are useful only if they are integrated in a larger framework, those results are often plugged into a visualization. Nowadays, since the data being analyzed evolves quickly, that visualization has moved away from the old static report toward more real-time visual interfaces, often using some web-based technology.

In this architecture, Spark, as a computing engine, focuses on providing data processing capabilities and relies on having functional interfaces with the other blocks of the picture. In particular, it implements a cluster abstraction layer that lets it interface with YARN, Mesos, and Kubernetes as resource managers, provides connectors to many data sources while new ones are easily added through an easy-to-extend API, and integrates with output data sinks to present results to upstream systems.

Architectural Models

Now we turn our attention to the link between stream processing and 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 need to be compared, those should both be streaming applications. We are going to see in detail what those architectures intend to achieve, and we examine that although the Kappa architecture is easier and lighter to implement in general, there might be cases for which a Lambda architecture is still needed, and why.

The Use of a Batch-Processing Component in a Streaming Application

Often, if we develop a batch application that runs 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 reap the benefits of a lighter, simpler application that gives faster results.

In a greenfield application, we might 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 dataset has the advantage of setting a benchmark: after it's 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.

In this context, we identify three levels of interaction between the batch and the stream-processing components, from the least to the most mixed with batch processing:

Code reuse

Often born out of a reference batch implementation, seeks to reemploy as much of it as possible, so as not to duplicate efforts. This is an area in which Spark shines, since it is particularly easy to call functions that transform Resilient Distributed Databases (RDDs) and DataFrames—they share most of the same APIs, and only the setup of the data input and output is distinct.

Data reuse

Wherein 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 must handle 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 feeds itself off.

Mixed processing

Wherein 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 dataset as a benchmark. In the next subsections, we see how this affects the architecture of a streaming application.

Referential Streaming Architectures

In the world of replay-ability and performance analysis over time, there are two historical but conflicting recommendations. Our main concern is 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 (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 ensure we compare apples to apples, we have already established that replaying a *batch dataset* to 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 this case, we would test our improvements in yearly data, making sure we're not overoptimizing for the current season at the detriment of performance six months after.

However, 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

The Lambda architecture (Figure 3-2) 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 entire 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

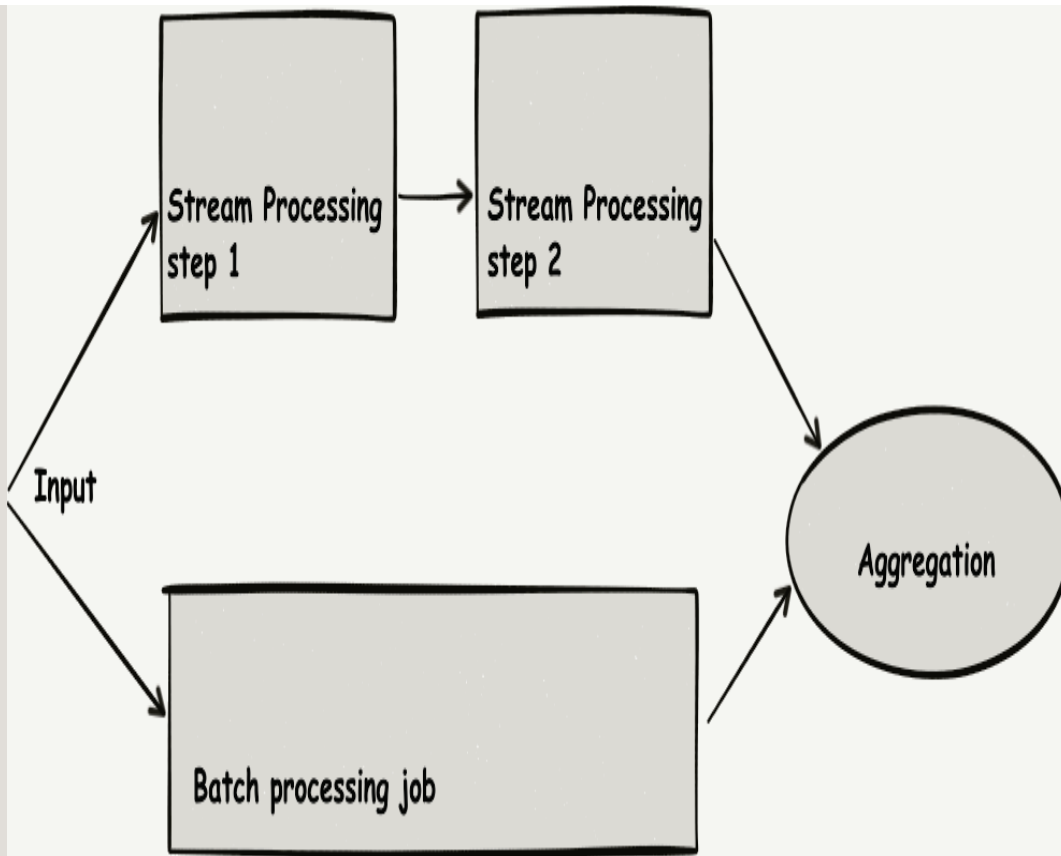


Figure 3-2. The Lambda architecture

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 us reuse most of our code between the batch and streaming versions of our application, the two versions of the application are distinct in life cycles, which might seem complicated.

An alternative view on this problem suggests 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), helping with the maintainability of our solution.

The Kappa Architecture

This architecture, as outlined in [Figure 3-3](#), 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* ([\[Kreps2014\]](#)), there is no deduplication and the mental model is simpler.

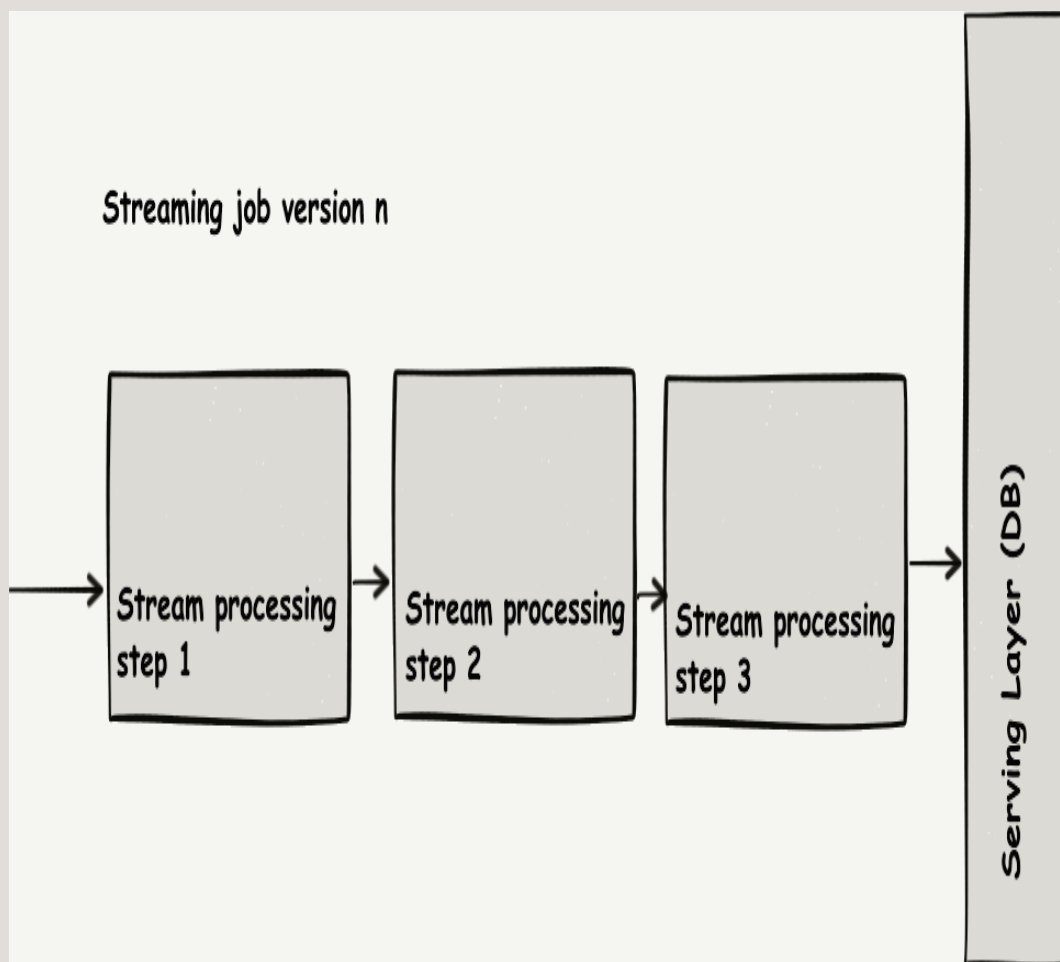


Figure 3-3. The Kappa architecture

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

We think some concepts stemming from the Lambda architecture are still relevant; in fact, they're vitally useful in some cases, although those are not always easy to figure out.

There are some use cases for which it is still useful to go through the effort of implementing a batch version of our analysis and then compare it to our streaming solution.

Streaming Versus Batch Algorithms

There are two important considerations that we need to take into account when selecting a general architectural model for our streaming application:

- Streaming algorithms are sometimes completely different in nature
- Streaming algorithms can't be guaranteed to measure well against batch algorithms

Let's explore these thoughts in the next two sections using motivating examples.

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 that at first glance we might not be able to reuse code between both approaches, but also, and more important,

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 have 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 version 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 nine times or fewer, 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 $\$450 + \$500 = \$950$. The worst input is to receive 10 “ski trip” decision events, in which case the batch algorithm, 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 is to go skiing only once, which means that the competitive ratio is $\$500 / \$50 = 10$.

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. An algorithm is formally ρ -competitive if its objective value is no more than ρ times the optimal offline value for all instances.

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 of any streaming 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. In the bin-packing problem, an input of a set of objects of different sizes or different weights must be fitted into a number of bins or

containers, each of them having a set volume or set capacity in terms of weight or size. The challenge is to find an assignment of objects into bins that minimizes the number of containers used.

In computational complexity theory, the offline version of that algorithm is known to be *NP-hard*. The simple variant of the problem is the *decision* question: knowing whether 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—for which the algorithm has as input a stream of objects—and offline versions—for which the algorithm can examine the entire set of input objects before it even starts the computing process.

The online algorithm processes the items in arbitrary order and 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. This greedy approximation algorithm always allows placing the input objects into a set number of bins that is, at worst, suboptimal; meaning we might use more bins than necessary.

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, and 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]).

The first fit decreasing strategy, however, relies on the idea that we can first sort the items in decreasing order of sizes before we begin processing them and packing them into bins.

Now, attempting to apply such a method in the case of the online bin-packing problem, the situation is completely different in that we are dealing with a stream of elements for which sorting is not possible. Intuitively, it is thus easy to understand that the online bin-packing problem—which by its 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.

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

What this means, to use an analogy, is that we have one worker that receives the data as batch, as if it were all in a *storage room* from the beginning, and the other worker receiving the data in a streaming fashion, as if it were 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 takeaway message from this discussion 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 in which heuristics are sadly limited, as we’ve seen earlier.

Summary

In conclusion, the news of batch processing’s demise is 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 streaming application:

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

¹ We invite you 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’re focusing on the practical implications of that constraint.

Chapter 4. Apache Spark as a Stream-Processing Engine

In [Chapter 3](#), we pictured a general architectural diagram of a streaming data platform and identified where Spark, as a distributed processing engine, fits in a big data system.

This architecture informed us about what to expect in terms of interfaces and links to the rest of the ecosystem, especially as we focus on stream data processing with Apache Spark. Stream processing, whether in its Spark Streaming or Structured Streaming incarnation, is another *execution mode* for Apache Spark.

In this chapter, we take a tour of the main features that make Spark stand out as a stream-processing engine.

The Tale of Two APIs

As we mentioned in [“Introducing Apache Spark”](#), Spark offers two different stream-processing APIs, Spark Streaming and Structured Streaming:

Spark Streaming

This is an API and a set of connectors, in which a Spark program is being served small batches of data collected from a stream in the form of microbatches spaced at fixed time intervals, performs

a given computation, and eventually returns a result at every interval.

Structured Streaming

This is an API and a set of connectors, built on the substrate of a SQL query optimizer, Catalyst. It offers an API based on `DataFrames` and the notion of continuous queries over an unbounded table that is constantly updated with fresh records from the stream.

The interface that Spark offers on these fronts is particularly rich, to the point where this book devotes large parts explaining those two ways of processing streaming datasets. One important point to realize is that both APIs rely on the core capabilities of Spark and share many of the low-level features in terms of distributed computation, in-memory caching, and cluster interactions.

As a leap forward from its MapReduce predecessor, Spark offers a rich set of operators that allows the programmer to express complex processing, including machine learning or event-time manipulations. We examine more specifically the basic properties that allow Spark to perform this feat in a moment.

We would just like to outline that these interfaces are by design as simple as their batch counterparts—operating on a `DStream` feels like operating on an `RDD`, and operating on a streaming `Dataframe` looks eerily like operating on a batch one.

Apache Spark presents itself as a unified engine, offering developers a consistent environment whenever they want to develop a batch or a

streaming application. In both cases, developers have all the power and speed of a distributed framework at hand.

This versatility empowers development agility. Before deploying a full-fledged stream-processing application, programmers and analysts first try to discover insights in interactive environments with a fast feedback loop. Spark offers a built-in shell, based on the Scala *REPL* (short for Read-Eval-Print-Loop) that can be used as prototyping grounds. There are several notebook implementations available, like Zeppelin, Jupyter, or the Spark Notebook, that take this interactive experience to a user-friendly web interface. This prototyping phase is essential in the early phases of development, and so is its velocity.

If you refer back to the diagram in [Figure 3-1](#), you will notice that what we called *results* in the chart are actionable insights—which often means revenue or cost-savings—are generated every time a loop (starting and ending at the business or scientific problem) is traveled fully. In sum, this loop is a crude representation of the experimental method, going through observation, hypothesis, experiment, measure, interpretation, and conclusion.

Apache Spark, in its streaming modules, has always made the choice to carefully manage the cognitive load of switching to a streaming application. It also has other major design choices that have a bearing on its stream-processing capabilities, starting with its in-memory storage.

Spark's Memory Usage

Spark offers in-memory storage of slices of a dataset, which must be initially loaded from a data source. The data source can be a distributed filesystem or another storage medium. Spark's form of in-memory storage is analogous to the operation of caching data.

Hence, a *value* in Spark's in-memory storage has a *base*, which is its initial data source, and layers of successive operations applied to it.

Failure Recovery

What happens in case of a failure? Because Spark knows exactly which data source was used to ingest the data in the first place, and because it also knows all the operations that were performed on it thus far, it can reconstitute the segment of lost data that was on a crashed executor, from scratch. Obviously, this goes faster if that reconstitution (*recovery*, in Spark's parlance), does not need to be totally *from scratch*. So, Spark offers a replication mechanism, quite in a similar way to distributed filesystems.

However, because memory is such a valuable yet limited commodity, Spark makes (by default) the cache short lived.

Lazy Evaluation

As you will see in greater detail in later chapters, a good part of the operations that can be defined on values in Spark's storage have a lazy execution, and it is the execution of a final, eager output operation that will trigger the actual execution of computation in a Spark cluster. It's worth noting that if a program consists of a series of linear operations, with the previous one feeding into the next, the

intermediate results *disappear* right after said next step has consumed its input.

Cache Hints

On the other hand, what happens if we have several operations to do on a single intermediate result? Should we have to compute it several times? Thankfully, Spark lets users specify that an intermediate value is important and how its contents should be safeguarded for later.

Figure 4-1 presents the data flow of such an operation.

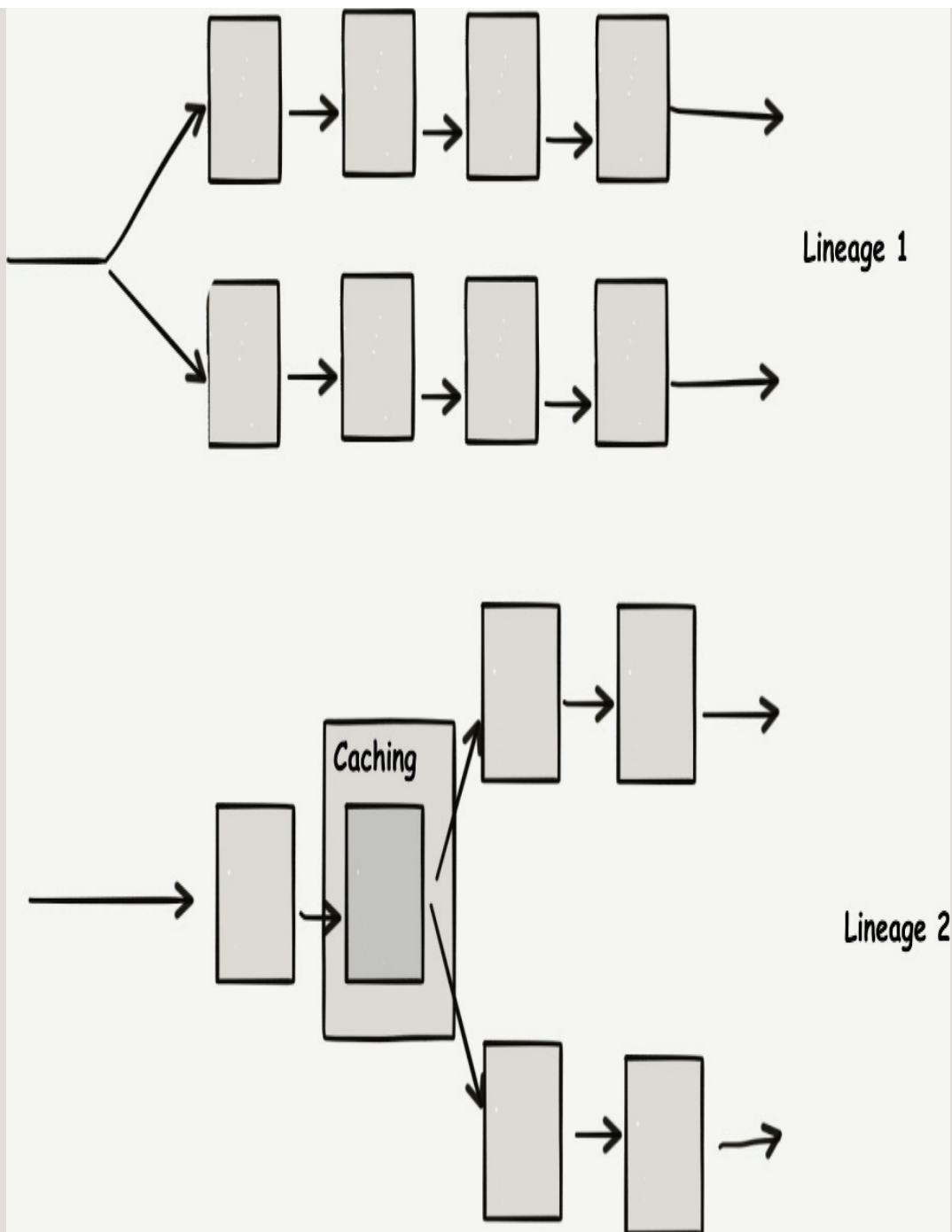


Figure 4-1. Operations on cached values

Finally, Spark offers the opportunity to spill the cache to secondary storage in case it runs out of memory on the cluster, extending the in-memory operation to secondary—and significantly slower—storage

to preserve the functional aspects of a data process when faced with temporary peak loads.

Now that we have an idea of the main characteristics of Apache Spark, let's spend some time focusing on one design choice internal to Spark, namely, the latency versus throughput trade-off.

Understanding Latency

Spark Streaming, as we mentioned, makes the choice of microbatching. It generates a chunk of elements on a fixed interval, and when that interval “tick” elapses, it begins processing the data collected over the last interval. Structured Streaming takes a slightly different approach in that it will make the interval in question as small as possible (the processing time of the last microbatch)—and proposing, in some cases, a continuous processing mode, as well. Yet, nowadays, microbatching is still the dominating internal execution mode of stream processing in Apache Spark.

A consequence of microbatching is that any microbatch delays the processing of any particular element of a batch by at least the time of the batch interval.

Firstly, microbatches create a baseline latency. The jury is still out on how small it is possible to make this latency, though approximately one second is a relatively common number for the lower bound. For many applications, a latency in the space of a few minutes is sufficient; for example:

- Having a dashboard that refreshes you on key performance indicators of your website over the last few minutes
- Extracting the most recent trending topics in a social network
- Computing the energy consumption trends of a group of households
- Introducing new media in a recommendation system

Whereas Spark is an equal-opportunity processor and delays all data elements for (at most) one batch before acting on them, some other streaming engines exist that can fast-track some elements that have priority, ensuring a faster responsiveness for them. If your response time is essential for these specific elements, alternative stream processors like Apache Flink or Apache Storm might be a better fit. But if you're just interested in fast processing *on average*, such as when monitoring a system, Spark makes an interesting proposition.

Throughput-Oriented Processing

All in all, where Spark truly excels at stream processing is with throughput-oriented data analytics.

We can compare the microbatch approach to a train: it arrives at the station, waits for passengers for a given period of time and then transports all passengers that boarded to their destination. Although taking a car or a taxi for the same trajectory might allow one passenger to travel faster door to door, the batch of passengers in the train ensures that far more travelers arrive at their destination. The

train offers higher throughput for the same trajectory, at the expense that some passengers must wait until the train departs.

The Spark core engine is optimized for distributed batch processing. Its application in a streaming context ensures that large amounts of data can be processed per unit of time. Spark amortizes the overhead of distributed task scheduling by having many elements to process at once and, as we saw earlier in this chapter, it utilizes in-memory techniques, query optimizations, caching, and even code generation to speed up the transformational process of a dataset.

When using Spark in an end-to-end application, an important constraint is that downstream systems receiving the processed data must also be able to accept the full output provided by the streaming process. Otherwise, we risk creating application bottlenecks that might cause cascading failures when faced with sudden load peaks.

Spark's Polyglot API

We have now outlined the main design foundations of Apache Spark as they affect stream processing, namely a rich API and an in-memory processing model, defined within the model of an execution engine. We have explored the specific streaming modes of Apache Spark, and still at a high level, we have determined that the predominance of microbatching makes us think of Spark as more adapted to throughput-oriented tasks, for which more data yields more quality. We now want to bring our attention to one additional aspect where Spark shines: its programming ecosystem.

Spark was first coded as a Scala-only project. As its interest and adoption widened, so did the need to support different user profiles, with different backgrounds and programming language skills. In the world of scientific data analysis, Python and R are arguably the predominant languages of choice, whereas in the enterprise environment, Java has a dominant position.

Spark, far from being just a library for distributing computation, has become a polyglot framework that the user can interface with using Scala, Java, Python, or the R language. The development language is still Scala, and this is where the main innovations come in.

CAUTION

The coverage of the Java API has for a long time been fairly synchronized with Scala, owing to the excellent Java compatibility offered by the Scala language. And although in Spark 1.3 and earlier versions Python was lagging behind in terms of functionalities, it is now mostly caught up. The newest addition is R, for which feature-completeness is an enthusiastic work in progress.

This versatile interface has let programmers of various levels and backgrounds flock to Spark for implementing their own data analytics needs. The amazing and growing richness of the contributions to the Spark open source project are a testimony to the strength of Spark as a federating framework.

Nevertheless, Spark's approach to best catering to its users' computing goes beyond letting them use their favorite programming language.

Fast Implementation of Data Analysis

Spark's advantages in developing a streaming data analytics pipeline go beyond offering a concise, high-level API in Scala and compatible APIs in Java and Python. It also offers the simple model of Spark as a practical shortcut throughout the development process.

Component reuse with Spark is a valuable asset, as is access to the Java ecosystem of libraries for machine learning and many other fields. As an example, Spark lets users benefit from, for instance, the Stanford CoreNLP library with ease, letting you avoid the painful task of writing a tokenizer. All in all, this lets you quickly prototype your streaming data pipeline solution, getting first results quickly enough to choose the right components at every step of the pipeline development.

Finally, stream processing with Spark lets you benefit from its model of fault tolerance, leaving you with the confidence that faulty machines are not going to bring the streaming application to its knees. If you have enjoyed the automatic restart of failed Spark jobs, you will doubly appreciate that resiliency when running a 24/7 streaming operation.

In conclusion, Spark is a framework that, while making trade-offs in latency, optimizes for building a data analytics pipeline with agility: fast prototyping in a rich environment and stable runtime performance under adverse conditions are problems it recognizes and tackles head-on, offering users significant advantages.

To Learn More About Spark

This book is focused on streaming. As such, we move quickly through the Spark-centric concepts, in particular about batch processing. The most detailed references are [Karau2015] and [Chambers2018].

On a more low-level approach, the official documentation in the [Spark Programming guide](#) is another accessible must-read.

Summary

In this chapter, you learned about Spark and where it came from.

- You saw how Spark extends that model with key performance improvements, notably in-memory computing, as well as how it expands on the API with new higher-order functions.
- We also considered how Spark integrates into the modern ecosystem of big data solutions, including the smaller footprint it focuses on, when compared to its older brother, Hadoop.
- We focused on the streaming APIs and, in particular, on the meaning of their microbatching approach, what uses they are appropriate for, as well as the applications they would not serve well.
- Finally, we considered stream processing in the context of Spark, and how building a pipeline with agility, along with a reliable, fault-tolerant deployment is its best use case.

Chapter 5. Spark's Distributed Processing Model

As a distributed processing system, Spark relies on the availability and addressability of computing resources to execute any arbitrary workload.

Although it's possible to deploy Spark as a standalone distributed system to solve a punctual problem, organizations evolving in their data maturity level are often required to deploy a complete data architecture, as we discussed in [Chapter 3](#).

In this chapter, we want to discuss the interaction of Spark with its computational environment and how, in turn, it needs to adapt to the features and constraints of the environment of choice.

First, we survey the current choices for a cluster manager: YARN, Mesos, and Kubernetes. The scope of a cluster manager goes beyond running data analytics, and therefore, there are plenty of resources available to get in-depth knowledge on any of them. For our purposes, we are going to provide additional details on the cluster manager provider by Spark as a reference.

After you have an understanding of the role of the cluster manager and the way Spark interacts with it, we look into the aspects of fault tolerance in a distributed environment and how the execution model of Spark functions in that context.

With this background, you will be prepared to understand the data reliability guarantees that Spark offers and how they apply to the streaming execution model.

Running Apache Spark with a Cluster Manager

We are first going to look at the discipline of distributing stream processing 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 *multitenancy*, 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 pieces of software that receive utilization requests from a number of users, match them to some resources, reserve the resources on behalf of the users for a given duration, and place user applications onto a number of resources for them to use. The challenges of the cluster manager's role include nontrivial tasks such as figuring out the best placements of user 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 can shine or break include fragmentation of tasks, optimal placement, availability, preemption, and prioritization. Cluster management is, therefore, a discipline in and of 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

Some examples of popular cluster managers include the following:

- 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 for the existence of Apache Spark
- 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

Where Spark can 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 look at the following:

- Spark's own cluster managers and how their *special purpose* means that they take on less responsibility in the domain of

fault tolerance or multitenancy than production cluster managers like Mesos, YARN, or Kubernetes.

- How there is a standard level of *delivery guarantees* expected out of a distributed streaming application, how they differ from one another, and how Spark meets those guarantees.
- How microbatching, a distinctive factor of Spark's approach to stream processing, comes from the decade-old model of *bulk-synchronous processing* (BSP), and paves the evolution path from Spark Streaming to Structured Streaming.

Spark's Own Cluster Manager

Spark has two internal cluster managers:

The *local* cluster manager

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

The *standalone* cluster manager

A relatively simple, Spark-only cluster manager that is rather limited in its availability to slice and dice resource allocation. The standalone cluster manager holds and makes available the entire worker node on which a Spark executor is deployed and started. It also expects the executor to have been predeployed 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 bare-bones resource management solution that allows you to focus on improving Spark in an environment without any bells and whistles. The standalone cluster manager is not recommended for production deployments.

As a summary, Apache Spark is a *task scheduler* in that what it schedules are *tasks*, units of distribution of computation that have been extracted from the user program. Spark also communicates and is deployed through cluster managers including Apache Mesos, YARN, and Kubernetes, or allowing for some cases its own standalone cluster manager. The purpose of that communication is to reserve a number of *executors*, which are the units 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 the following:

- Limited processes (e.g., in some basic use cases of YARN), in which processes have their 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), in which containers are a relatively lightweight resource reservation technology that is born out of the cgroups and namespaces 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* (VMs), themselves coming with specific cores and memory reservation.

CLUSTER OPERATIONS

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

Note that in an enterprise-level production cluster management domain, we also encounter notions such as job queues, priorities, multitenancy options, and preemptions 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 you to have a firm grasp of the specifics of your 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 a 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 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 ideally operated near peak capacity over their lifetime.

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

This sort of resiliency is especially relevant in stream processing given that the applications we're scheduling are 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 into the system against the actions and clicks and navigation of users visiting the site, we potentially have a data source that will be available for the entire 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 streaming computation makes the resiliency and fault-tolerance characteristics 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, stream processing 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. Thus, a very primitive form of fault tolerance is detecting the failure of a particular node of our deployment, stopping the computation, and restarting from scratch. That process can take more than twice the original duration that we had budgeted for that computation, but if we are not in a hurry, this is still acceptable.

For stream processing, *we need to keep receiving data* and thus potentially storing it, if the recovering cluster is not ready to assume any processing yet. This can pose a problem at a high throughput: if we try restarting from scratch, we will need not only to reprocess all of the data that we have observed since the beginning of the application—which in and of itself can be a challenge—but during that reprocessing of historical data, we will 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 intractable 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 nonfunctional.

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 understand 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 possesses 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 readiness and liveness state. If these cluster managers 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 then 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. Its responsibility is limited to the container that runs the user's code.

That responsibility boundary is where the Spark resilience features start. To recover from a failed node, Spark needs to do the following:

- Determine whether that node contains some state that should be reproduced in the form of checkpointed files
- Understand at which stage of the job a node should rejoin the computation

The goal here is for us to explore that if a node is being replaced by the cluster manager, Spark has capabilities that allow it to take advantage of this new node and to distribute computation onto it.

Within this context, our focus is 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 hardware failure or because its work was simply preempted by a higher-priority job. Apache Spark is blissfully unaware of the *why*, and focuses on the *how*.

Data Delivery Semantics

As you 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*, in which some parts of our cluster are not able to communicate with other parts of that cluster, as illustrated in Figure 5-1.

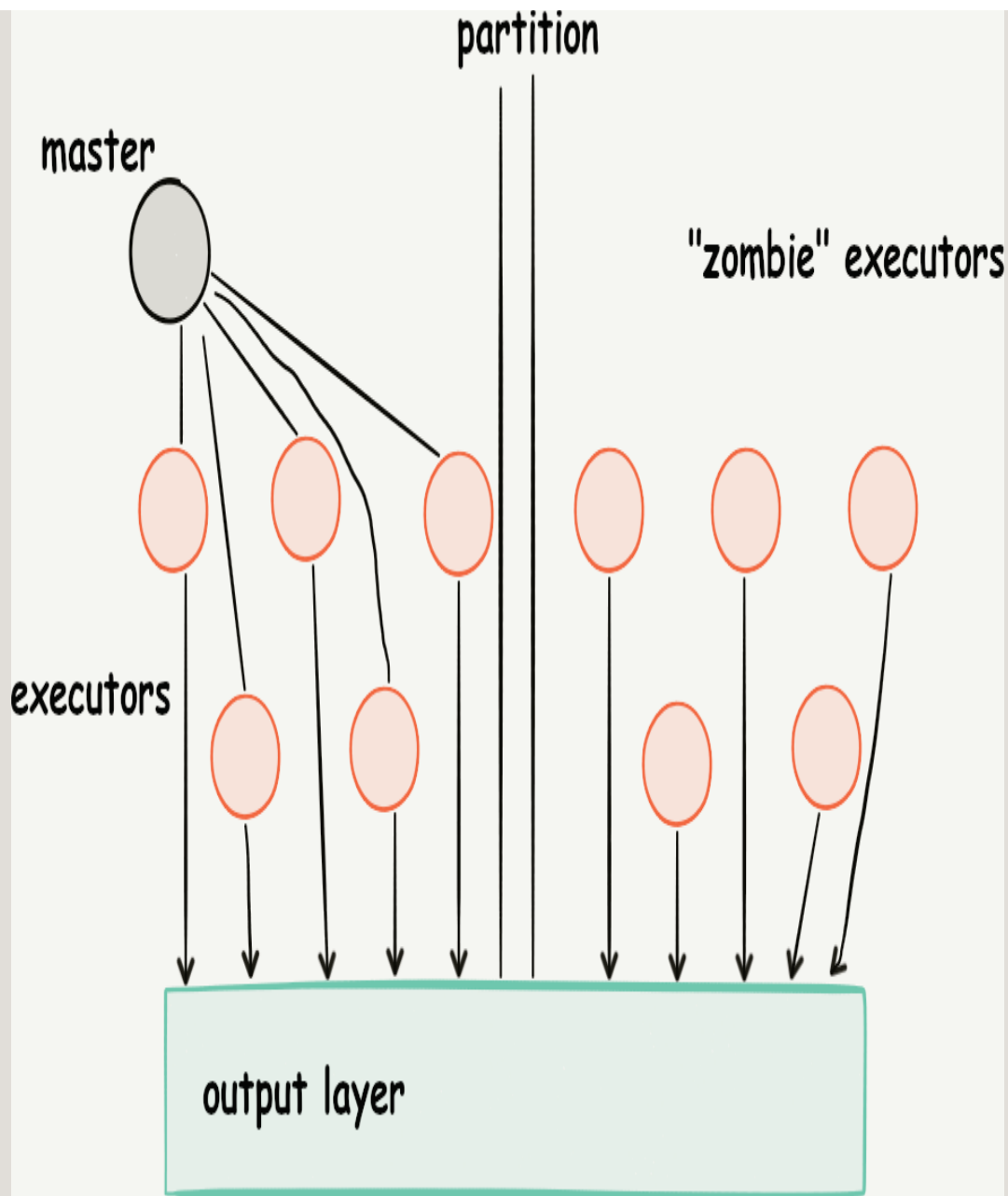


Figure 5-1. A network partition

Spark has been designed using a *driver/executor* architecture. A specific machine, the *driver*, is tasked with keeping track of the *job progression* along with 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 *driver*

might be able to keep track of only the part of the executors that form the initial cluster. In the other section of our partition, we will find nodes that are entirely able to function, but will simply be unable to account for the proceedings of their computation to the *driver*.

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

Because the *driver*, 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 partitioning 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.

It is not only the aforementioned problem that causes different processing semantics. Another important reason is that when output from a stream-processing application and state checkpointing cannot be completed in one atomic operation, it will cause data corruption if failure happens between checkpointing and outputting.

These challenges have therefore led to a distinction between *at least once* processing and *at most once* processing:

At least once

This processing ensures that every element of a stream has been processed once or more.

At most once

This processing ensures that every element of the stream is processed once or less.

Exactly once

This is the combination of “at least once” and “at most once.”

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 a streaming process suffers a partial failure in which some nodes need to be replaced or some data needs to be recomputed, we need to reprocess 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, there is a chance for you, under certain conditions, to lose 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 bookkeeping system at the output of our streaming process.

Microbatching and One-Element-at-a-Time

In this section, we want to address 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.

Microbatching: An Application of Bulk-Synchronous Processing

Spark Streaming, the more mature model of stream processing 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

- 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 its 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 synchronization point at the “driver” that allows the work to progress to the next step. Between those scheduled steps, all of the executors on the cluster are doing the same thing.

Note that what is being passed around in this scheduling process are the functions that describe the processing that the user wants to execute on the data. 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 frequency at which further rounds of data processing are scheduled is dictated by a time interval. This time interval is an arbitrary duration that is measured in batch processing time; that is, what you would expect to see as a “wall clock” time observation in your cluster.

For stream processing, we choose to implement barriers at small, fixed intervals that better approximate the real-time notion of data processing.

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:

- A number of components, each performing processing and/or memory functions
- A router that delivers messages point to point between components
- 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 when 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 use it as a way to speak of the timing of synchronization of parallel computation in Spark's `DStreams`.

One-Record-at-a-Time Processing

By contrast, one-record-at-a-time processing functions by *pipelining*: it analyzes 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

latter case, each step of the computation is materialized at some place in the cluster at any given point.

Systems that function mostly according to this paradigm include Apache Flink, Naiad, Storm, and IBM Streams. (You can read more on these in [Chapter 29](#).) This does not necessarily mean that those systems are incapable of microbatching, 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 microbatching system is therefore the time needed to complete the reception of the current microbatch (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.

Microbatching Versus One-at-a-Time: The Trade-Offs

Despite their higher latency, microbatching systems offer significant advantages:

- 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

synchronization can 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.

- Our BSP 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.
- Having data available *as a set* that we can probe or inspect at the beginning of the microbatch allows us to perform efficient optimizations that can provide ideas on the way to compute on the data. Exploiting that on *each* microbatch, we can consider the specific case rather than the general processing, which is used for all possible input. For example, we could take a sample or compute a statistical measure before deciding to process or drop each microbatch.

More importantly, the simple presence of the microbatch 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 microbatch, even for mere instants, *looks* like data at rest.

Bringing Microbatch and One-Record-at-a-Time Closer Together

The marriage between microbatching and one-record-at-a-time processing as is implemented in systems like Apache Flink or Naiad

is still a subject of research.¹]

Although it does not solve every issue, Structured Streaming, which is backed by a main implementation that relies on microbatching, does not expose that choice at the API level, allowing for an evolution that is independent of a fixed-batch interval. In fact, the default internal execution model of Structured Streaming is that of microbatching with a dynamic batch interval. Structured Streaming is also implementing continuous processing for some operators, which is something we touch upon in [Chapter 15](#).

Dynamic Batch Interval

What is this notion of *dynamic batch interval*? The dynamic batch interval is the notion that the recomputation of data in a streaming `DataFrame` or `Dataset` consists of 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 is 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 microbatches 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 is produced, a new query will be started and scheduled by the Spark driver.

Structured Streaming Processing Model

The main steps in Structured Streaming processing are as follows:

1. 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.
2. This is followed by logical planning, the construction of successive steps to be executed on data, followed by query planning (intrastep optimization).
3. And then the launch and scheduling of the actual computation by adding a new batch of data 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.

The Disappearance of the Batch Interval

We now briefly explain what Structured Streaming batches mean and their impact with respect to operations.

In Structured Streaming, the batch interval that we are using is no longer a computation budget. With Spark Streaming, the idea was

that if we produce data every two minutes and flow data into Spark's memory every two minutes, we should produce the results of computation on that batch of data in at least two minutes, to clear the memory from our cluster for the next microbatch. Ideally, as much data flows out as flows in, and the usage of the collective memory of our cluster remains stable.

With Structured Streaming, without this fixed time synchronization, 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 a higher granularity than we used to in the time of a conservative batch interval.

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 of longer-lived pipeline that persists across several batches, for the purpose of creating near-continuous queries. See [Venkataraman2016]

Chapter 6. Spark's Resilience Model

In most cases, a streaming job is a long-running job. By definition, streams of data observed and processed over time lead to jobs that run continuously. As they process data, they might accumulate intermediary results that are difficult to reproduce after the data has left the processing system. Therefore, the cost of failure is considerable and, in some cases, complete recovery is intractable.

In distributed systems, especially those relying on commodity hardware, failure is a function of size: the larger the system, the higher the probability that some component fails at any time. Distributed stream processors need to factor this chance of failure in their operational model.

In this chapter, we look at the resilience that the Apache Spark platform provides us: how it's able to recover partial failure and what kinds of guarantees we are given for the data passing through the system when a failure occurs. We begin by getting an overview of the different internal components of Spark and their relation to the core data structure. With this knowledge, you can proceed to understand the impact of failure at the different levels and the measures that Spark offers to recover from such failure.

Resilient Distributed Datasets in Spark

Spark builds its data representations on *Resilient Distributed Datasets* (RDDs). Introduced in 2011 by the paper “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing” [Zaharia2011], RDDs are the foundational data structure in Spark. It is at this ground level that the strong fault tolerance guarantees of Spark start.

RDDs are composed of partitions, which are segments of data stored on individual nodes and tracked by the Spark driver that is presented as a location-transparent data structure to the user.

We illustrate these components in Figure 6-1 in which the classic *word count* application is broken down into the different elements that comprise an RDD.

```
.textFile(...) .flatMap(l => l.split(" ")) .map(w => (w,1)) .reduceByKey(_ + _)
```

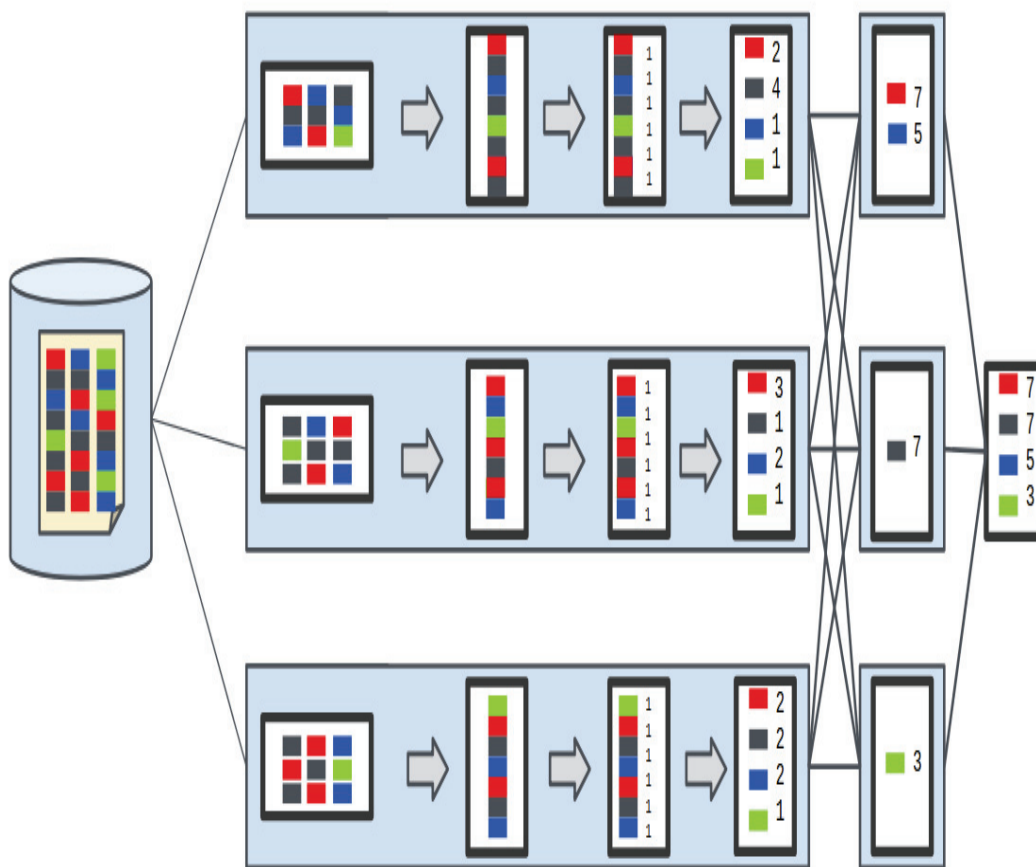


Figure 6-1. An RDD operation represented in a distributed system

The colored blocks are data elements, originally stored in a distributed filesystem, represented on the far left of the figure. The data is stored as partitions, illustrated as columns of colored blocks inside the file. Each partition is read into an executor, which we see as the horizontal blocks. The actual data processing happens within the executor. There, the data is transformed following the transformations described at the RDD level:

- `.flatMap(l => l.split(" "))` separates sentences into words separated by space.
- `.map(w => (w,1))` transforms each word into a tuple of the form `(<word>, 1)` in this way preparing the words for

counting.

- `.reduceByKey(_ + _)` computes the count, using the `<word>` as a key and applying a sum operation to the attached number.
- The final result is attained by bringing the partial results together using the same `reduce` operation.

RDDs constitute the programmatic core of Spark. All other abstractions, batch and streaming alike, including `DataFrames`, `DataSets`, and `DStreams` are built using the facilities created by RDDs, and, more important, they inherit the same fault tolerance capabilities. We provide a brief introduction of the RDDs programming model in [“RDDs as the Underlying Abstraction for DStreams”](#).

Another important characteristic of RDDs is that Spark will try to keep their data preferably in-memory for as long as it is required and provided enough capacity in the system. This behavior is configurable through storage levels and can be explicitly controlled by calling caching operations.

We mention those structures here to present the idea that Spark tracks the 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.

In [Figure 6-2](#), we show the same *word count* program, now in the form of the user-provided code (left) and the resulting internal RDD chain of operations. This dependency chain forms a particular kind of graph, a

Directed Acyclic Graph (DAG). The DAG informs the scheduler, appropriately called `DAGScheduler`, on how to distribute the computation and is also the foundation of the failure-recovery functionality, because it represents the internal data and their dependencies.

```
val file = spark.textFile("hdfs://...")
val wordsRDD = file.flatMap(line =>
  line.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)
val scoreRdd = words.map{case (k,v) => (v,k)}
```

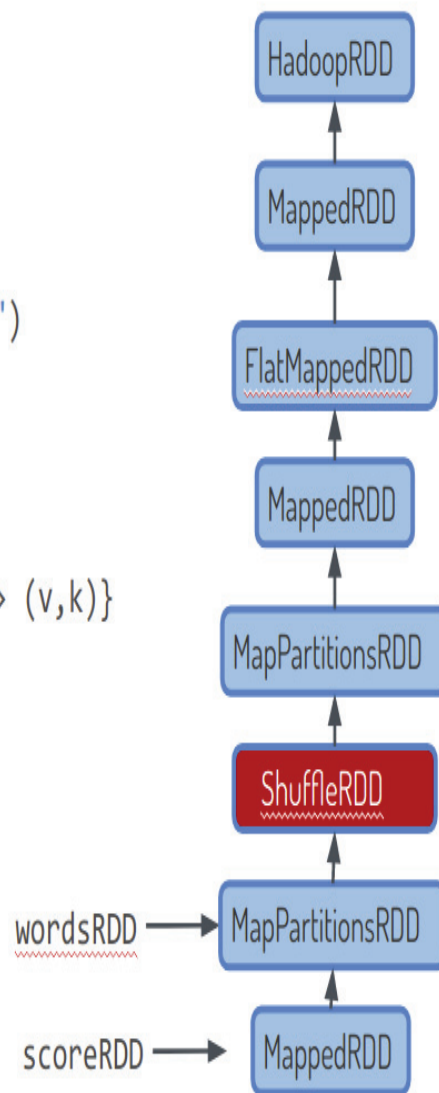


Figure 6-2. RDD lineage

As the system tracks the ordered creation of these distributed data collections, it tracks the work done, and what's left to accomplish.

Spark Components

To understand at what level fault tolerance operates in Spark, it's useful to go through an overview of the nomenclature of some core concepts. We begin by assuming that the user provides a program that ends up being divided into chunks and executed on various machines, as we saw in the previous section, and as depicted in Figure 6-3.

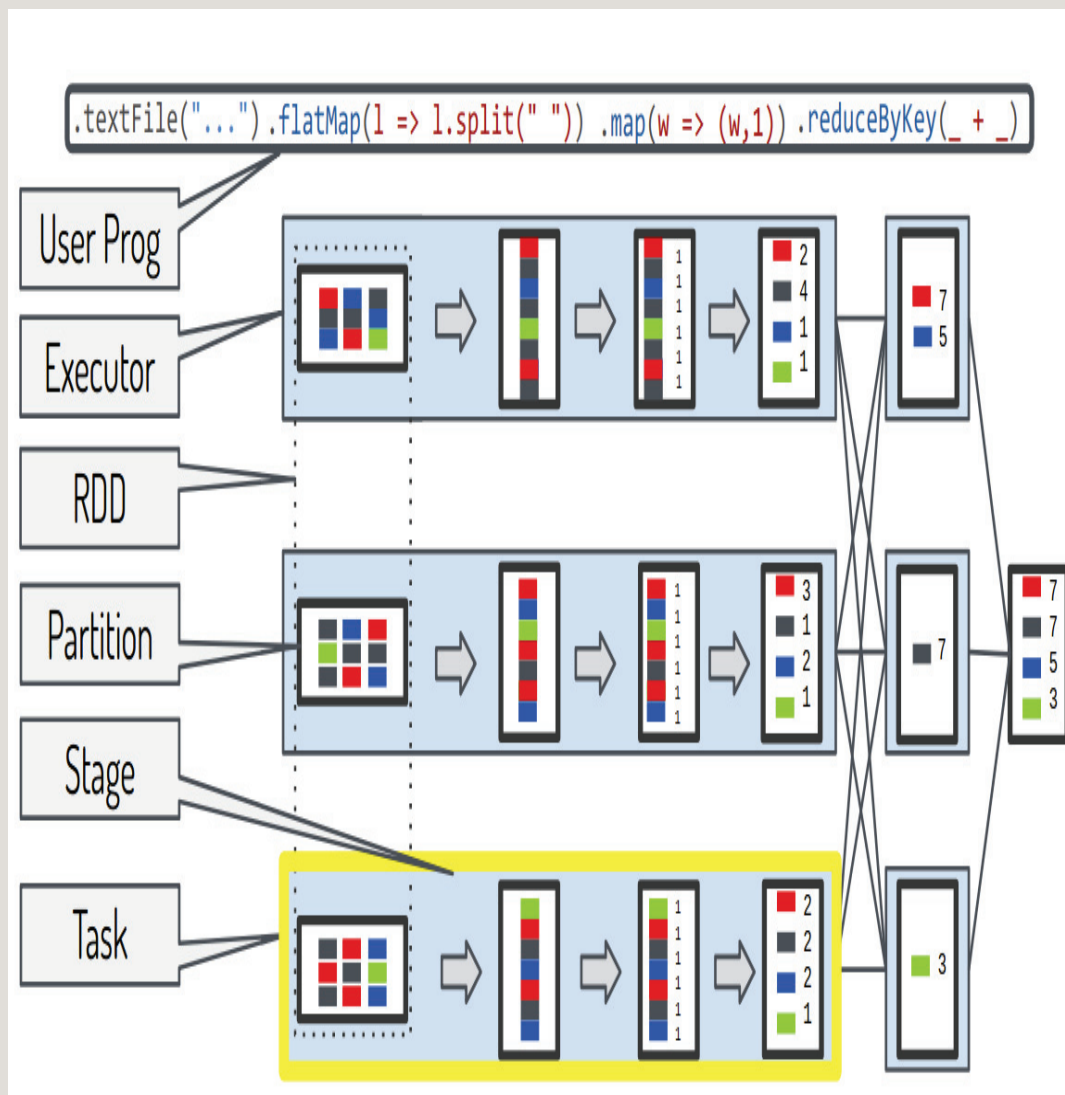


Figure 6-3. Spark nomenclature

Let's run down those steps, illustrated in Figure 6-3, 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, and so on), 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 Query plan is a similar but more advanced concept in Spark SQL.

RDD

A logical representation of a distributed, resilient, dataset. In the illustration, we see that the initial RDD comprises three parts, called partitions.

Partition

A partition is a physical segment of a dataset that can be loaded independently.

Stages

The user's operations are then grouped into *stages*, whose boundary separates user operations into steps that must 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 of any interdependent stages can be running at any given time.

Jobs

After 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. (Although it's not represented explicitly, in [Figure 6-3](#), the job is the complete set of transformations.)

Tasks

Depending on where their source data is on the cluster, jobs can then 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 aforementioned notions: restart and checkpointing operations that occur at the task, job, stage, or program level.

Spark's Fault-Tolerance Guarantees

Now that we have seen the “pieces” that constitute the internal machinery in Spark, we are ready to understand that failure can happen at many different levels. In this section, we see Spark fault-tolerance guarantees organized by “increasing blast radius,” from the more modest to the larger failure. We are going to investigate the following:

- How Spark mitigates Task failure through restarts
- How Spark mitigates Stage failure through the shuffle service
- How Spark mitigates the disappearance of the *orchestrator* of the user program, through driver restarts

When you've completed this section, you will have a clear mental picture of the guarantees Spark affords us at runtime, letting you understand the failure scenarios that a well-configured Spark job can deal with.

Task Failure Recovery

Tasks can fail when the infrastructure on which they are running has a failure or logical conditions in the program lead to an sporadic job, like `OutOfMemory`, network, storage errors, or problems bound to the quality of the data being processed (e.g., a parsing error, a `NumberFormatException`, or a `NullPointerException` to name a few common exceptions).

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 for a storage level whose setting ends in `_2`, such as `MEMORY_ONLY_SER_2`), the task does not need to have its input recomputed, because a copy of it exists in complete form on another machine of the cluster. We can then use this input to restart the task. [Table 6-1](#) summarizes the different storage levels configurable in Spark and their characteristics in terms of memory usage and replication factor.

Table 6-1. Spark storage levels

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
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, the Spark driver will need to consult the DAG that stores the user-specified computation to determine 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 recomputation of several others, up to a stage boundary.

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 lifesaver if you have a dying downstream executor, able to rely on 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?

Stage Failure Recovery

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 event to mitigate. Recurrent task failures will lead to the failure of the stage that contains that task. This brings us to the second facility that allows Spark to resist arbitrary stage failures: the *shuffle service*.

When this failure occurs, it always means some rollback 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, since Spark 1.3 we have the shuffle service, which lets you work on map data that is saved and distributed through the cluster with a good locality, but, more important, through a server that is not a Spark task. It's an external file exchange service written in Java that has no dependency on Spark and is made to be a much longer-running service than 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 backend, 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 Recovery

Having seen how Spark recovers from the failure of a particular task and stage, 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 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 might 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 then 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, because the driver process will be relaunched by the cluster manager. But this only allows recovery from scratch, given that the temporary state of the computation—previously stored in the driver machine—might have been lost.

CHECKPOINTING

To avoid losing intermediate state in case of a driver crash, Spark offers the option of checkpointing; 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 (e.g., Hadoop Distributed File System [HDFS]) because having the driver try to reconstruct the state of intermediate RDDs from its local filesystem 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 come back to the subject of checkpointing in detail much later, in [Chapter 24](#). In the meantime, there is still one component of any Spark cluster whose potential failure we have not yet addressed: the master node.

Summary

This tour of Spark-core's fault tolerance and high-availability modes should have given you 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 or Structured Streaming, but that all these lessons apply to the streaming APIs in that they are required to deliver long-running, fault-tolerant and yet performant applications.

Note also that these facilities reflect different concerns in the frequency of faults for a particular cluster. These facilities reflect different concerns for the frequency of faults in a particular cluster:

- Features such as setting up a failover master node kept up-to-date through Zookeeper are really about avoiding a single point of failure in the design of a Spark application.
- The Spark Shuffle Service is here to avoid any problems with a shuffle step at the end of a long list of computation steps making the whole fragile through a faulty executor.

The later is 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.

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

Appendix A. References for Part I

- **[Armbrust2018]** Armbrust, M., T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. “Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark,” May 27, 2018. <https://stanford.io/2Jia3iY>.
- **[Bhartia2016]** Bhartia, R. “Optimize Spark-Streaming to Efficiently Process Amazon Kinesis Streams,” AWS Big Data blog, February 26, 2016. <https://amzn.to/2E7I69h>.
- **[Chambers2018]** Chambers, B., and Zaharia, M., *Spark: The Definitive Guide*. O’Reilly, 2018.
- **[Chintapalli2015]** Chintapalli, S., D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Musbaum, K. Patil, B. Peng, and P. Poulosky. “Benchmarking Streaming Computation Engines at Yahoo!” Yahoo! Engineering, December 18, 2015. <http://bit.ly/2bhgMJd>.
- **[Das2013]** Das, Tathagata. “Deep Dive With Spark Streaming,” Spark meetup, June 17, 2013. <http://bit.ly/2Q8Xzem>.
- **[Das2014]** Das, Tathagata, and Yuan Zhong. “Adaptive Stream Processing Using Dynamic Batch Sizing,” 2014 ACM Symposium on Cloud Computing. <http://bit.ly/2WTOuby>.
- **[Das2015]** Das, Tathagata. “Improved Fault Tolerance and Zero Data Loss in Spark Streaming,” Databricks

Engineering blog. January 15, 2015. <http://bit.ly/2HqH614>.

- [Dean2004] Dean, Jeff, and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters,” OSDI San Francisco, December, 2004. <http://bit.ly/15LeQej>.
- [Doley1987] Doley, D., C. Dwork, and L. Stockmeyer. “On the Minimal Synchronism Needed for Distributed Consensus,” *Journal of the ACM* 34(1) (1987): 77-97. <http://bit.ly/2LHRy9K>.
- [Dosa2007] Dósa, György. “The Tight Bound of First fit Decreasing Bin-Packing Algorithm Is $\text{FFD}(I) \leq (11/9)\text{OPT}(I) + 6/9$.” In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*. Springer-Verlag, 2007.
- [Dunner2016] Dünner, C., T. Parnell, K. Atasu, M. Sifalakis, and H. Pozidis. “High-Performance Distributed Machine Learning Using Apache Spark,” December 2016. <http://bit.ly/2JoSgH4>.
- [Fischer1985] Fischer, M. J., N. A. Lynch, and M. S. Paterson. “Impossibility of distributed consensus with one faulty process,” *Journal of the ACM* 32(2) (1985): 374–382. <http://bit.ly/2Ee9tPb>.
- [Gibbons2004] Gibbons, J. “An unbounded spigot algorithm for the digits of π ,” *American Mathematical Monthly* 113(4) (2006): 318-328. <http://bit.ly/2VwwvH2>.
- [Greenberg2015] Greenberg, David. *Building Applications on Mesos*. O’Reilly, 2015.
- [Halevy2009] Halevy, Alon, Peter Norvig, and Fernando Pereira. “The Unreasonable Effectiveness of Data,” *IEEE*

Intelligent Systems (March/April 2009).

<http://bit.ly/2VCveD3>.

- [Karau2015] Karau, Holden, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark*. O'Reilly, 2015.
- [Kestelyn2015] Kestelyn, J. "Exactly-once Spark Streaming from Apache Kafka," Cloudera Engineering blog, March 16, 2015. <http://bit.ly/2EniQfJ>.
- [Kleppmann2015] Kleppmann, Martin. "A Critique of the CAP Theorem," arXiv.org:1509.05393, September 2015. <http://bit.ly/30jxsG4>.
- [Koeninger2015] Koeninger, Cody, Davies Liu, and Tathagata Das. "Improvements to Kafka Integration of Spark Streaming," Databricks Engineering blog, March 30, 2015. <http://bit.ly/2Hn7dat>.
- [Kreps2014] Kreps, Jay. "Questioning the Lambda Architecture," O'Reilly Radar, July 2, 2014. <https://oreil.ly/2LSEdqz>.
- [Lamport1998] Lamport, Leslie. "The Part-Time Parliament," *ACM Transactions on Computer Systems* 16(2): 133–169. <http://bit.ly/2W3zr1R>.
- [Lin2010] Lin, Jimmy, and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan & ClayPool, 2010. <http://bit.ly/2YD9wMr>.
- [Lyon2013] Lyon, Brad F. "Musings on the Motivations for Map Reduce," Nowhere Near Ithaca blog, June, 2013, <http://bit.ly/2Q3OHXe>.
- [Maas2014] Maas, Gérard. "Tuning Spark Streaming for Throughput," Virdata Engineering blog, December 22, 2014.

<http://www.virdata.com/tuning-spark/>.

- [Marz2011] Marz, Nathan. “How to beat the CAP theorem,” Thoughts from the Red Planet blog, October 13, 2011. <http://bit.ly/2KpKDQq>.
- [Marz2015] Marz, Nathan, and James Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Manning, 2015.
- [Miller2016] Miller, H., P. Haller, N. Müller, and J. Boullier “Function Passing: A Model for Typed, Distributed Functional Programming,” *ACM SIGPLAN Conference on Systems, Programming, Languages and Applications: Software for Humanity, Onward!* November 2016: (82-97). <http://bit.ly/2EQASaf>.
- [Nasir2016] Nasir, M.A.U. “Fault Tolerance for Stream Processing Engines,” arXiv.org:1605.00928, May 2016. <http://bit.ly/2Mpz66f>.
- [Shapira2014] Shapira, Gwen. “Building The Lambda Architecture with Spark Streaming,” Cloudera Engineering blog, August 29, 2014. <http://bit.ly/2XoyHBS>.
- [Sharp2007] Sharp, Alexa Megan. “Incremental algorithms: solving problems in a changing world,” PhD diss., Cornell University, 2007. <http://bit.ly/2Ie8MGX>.
- [Valiant1990] Valiant, L.G. “Bulk-synchronous parallel computers,” *Communications of the ACM* 33:8 (August 1990). <http://bit.ly/2IgX3ar>.
- [Vavilapalli2013] Vavilapalli, et al. “Apache Hadoop YARN: Yet Another Resource Negotiator,” ACM Symposium on Cloud Computing, 2013. <http://bit.ly/2Xn3tuZ>.

- [Venkat2015] Venkat, B., P. Padmanabhan, A. Arokiasamy, and R. Uppalapati. “Can Spark Streaming survive Chaos Monkey?” The Netflix Tech Blog, March 11, 2015.
<http://bit.ly/2WkDJmr>.
- [Venkataraman2016] Venkataraman, S., P. Aurojit, K. Ousterhout, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. “Drizzle: Fast and Adaptable Stream Processing at Scale,” Tech Report, UC Berkeley, 2016.
<http://bit.ly/2HW08Ot>.
- [White2010] White, Tom. *Hadoop: The Definitive Guide*, 4th ed. O’Reilly, 2015.
- [Zaharia2011] Zaharia, Matei, Mosharaf Chowdhury, et al. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing,” UCB/EECS-2011-82.
<http://bit.ly/2IfZE4q>.
- [Zaharia2012] Zaharia, Matei, Tathagata Das, et al. “Discretized Streams: A Fault-Tolerant Model for Scalable Stream Processing,” UCB/EECS-2012-259.
<http://bit.ly/2MpuY6c>.

Part II. Structured Streaming

In this part, we examine Structured Streaming.

We begin our journey by exploring a practical example that should help you build your intuition for the model. From there, we examine the API and get into the details of the following aspects of stream processing:

- Consuming data using sources
- Building data-processing logic using the rich Streaming Dataframe/Dataset API
- Understanding and working 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 explore the current developments in this exciting new streaming API and provide insights into experimental areas like machine learning applications and near-real-time data processing with continuous streaming.

Chapter 7. Introducing Structured Streaming

In data-intensive enterprises, we find many large datasets: log files from internet-facing servers, tables of shopping behavior, and NoSQL databases with sensor data, just to name a few examples. All of these datasets share the same fundamental life cycle: They started out 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 than a *data stream* being materialized onto secondary 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 analyzing data at rest. It is particularly useful for data that is *structured* in nature; that is, it follows a defined schema. The `Dataset` API in Spark combines the expressivity of a SQL-like API with type-safe collection operations that are reminiscent of the Scala collections and the Resilient Distributed Dataset (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 who are used to developing in a functional paradigm. This higher level of abstraction is intended to support modern data engineering 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 while it is 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 continuously 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 new concepts to the `Dataset` and `DataFrame` APIs, such as support for event time, *watermarking*, and different output modes that determine for how long past data is actually stored.

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

First Steps with Structured Streaming

In the previous section, we learned about the high-level concepts that constitute Structured Streaming, such as sources, sinks, and queries. We are now going to explore Structured Streaming from a practical perspective, using a simplified web log analytics use case as an example.

Before we begin 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 exercise 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, 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 analytic logic 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.

ONLINE RESOURCES

For this example, we 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 this exercise, the original log file has been split into daily files and each log line has been formatted as JSON. The compressed NASA-weblogs file can be downloaded from <https://github.com/stream-processing-with-spark>.

Download this dataset and place it in a folder on your computer.

Batch Analytics

Given that we are working with archive log files, we have access to all of the data at once. Before we begin building our streaming

application, let's take a brief *intermezzo* to have a look at what a classical batch analytics job would look like.

ONLINE RESOURCES

For this example, we will use the `batch_weblogs` notebook in the online resources for the book, located at <https://github.com/stream-processing-with-spark>][<https://github.com/stream-processing-with-spark>].

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

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

Next, we declare the schema of the data as a `case class` to use the typed `Dataset` API. Following the formal description of the dataset (at [NASA-HTTP](#)), 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.*

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 because it's internally supported by Spark and does not require any additional cast that other options might 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 begin 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 dayOfMonth column and the request URL and we count over this aggregate. We finally order using descending order to get the top URLs first:

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

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 the most traffic. To find those, we first filter on `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 earlier:

```
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](#). It's closely followed by `countdown/`, the days prior to the launch.

Streaming Analytics

In the previous section, we explored historical NASA web log records. We found trending events in those records, but much later than when the actual events happened.

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

We can use the lessons that we have learned 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 that 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 analyzing.

ONLINE RESOURCES

For this example, we will use the notebooks `weblog_TCP_server` and `streaming_weblogs` found in the online resources for the book, located at <https://github.com/stream-processing-with-spark>.

Connecting to a Stream

If you recall from the introduction of this chapter, 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 you 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 `DataFrame` and `Dataset` API for static data but with some modifications and limitations that you 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 [“The Socket Source”](#) 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 proceed to create a Dataset, following these steps:

```
import java.sql.Timestamp
case class WebLog(host:String,
                  timestamp: Timestamp,
                  request: String,
                  http_reply:Int,
                  bytes: Long
                )
val webLogSchema = Encoders.product[WebLog].schema ❶
val jsonStream = stream.select(from_json($"value",
webLogSchema) as "record") ❷
val webLogStream: Dataset[WebLog] =
jsonStream.select("record.*").as[WebLog] ❸
```

- ❶ Obtain a schema from the `case class` definition
- ❷ Transform the text value to JSON using the JSON support built into Spark SQL
- ❸ Use the Dataset API to transform the JSON records to WebLog objects

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

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 easily answer when we have access to all of the data. However, how do we count records that are constantly arriving? The answer is that some operations that 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 in the following code snippet 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 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.

Creating a Query

What are popular URLs? In what time frame? Now that we have immediate analytic access to the stream of web logs, 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 in 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 five minutes of event data. Given that our timeline is simulated, the five minutes might happen much faster or slower than the clock time. In this way, we can clearly appreciate how Structured Streaming uses the timestamp information in the events to keep track of the event timeline.

As we learned from the batch analytics, we should extract the URLs and select only content pages, like *.html*, *.htm*, or directories. Let's apply that acquired knowledge first before proceeding to define our windowed 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 contain only the visited URL and filtered out all noncontent pages. Now, we 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 of the steps that 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; for example, to a file in a filesystem, 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 visit all these concepts in detail later on. For now, let's use them empirically and observe the results.

For our query, shown in [Example 7-1](#), we 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.

Example 7-1. Writing a stream to a sink

```
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. We can observe this by querying the tables registered on Spark SQL:

```
scala> spark.sql("show tables").show()
+-----+-----+-----+
|database|tableName|isTemporary|
+-----+-----+-----+
|        | urlranks |      true |
+-----+-----+-----+
```

In the expression in [Example 7-1](#), `query` is of type `StreamingQuery` and it's a handler to control the query life cycle.

Exploring the Data

Given that we are accelerating the log timeline on the producer side, after a few seconds, we can execute the next command to see the result of the first windows, as illustrated in [Figure 7-1](#).

Note how the processing time (a few seconds) is decoupled from the event time (hundreds of minutes of logs):

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

request	window	count
/shuttle/missions/sts-70/mission-sts-70.html	{"start":"2018-02-02T18:18:00.000+01:00","end":"2018-02-02T18:23:00.000+01:00"}	8
/shuttle/countdown/	{"start":"2018-02-02T18:17:00.000+01:00","end":"2018-02-02T18:22:00.000+01:00"}	8
/shuttle/countdown/	{"start":"2018-02-02T18:18:00.000+01:00","end":"2018-02-02T18:23:00.000+01:00"}	8
/shuttle/countdown/	{"start":"2018-02-02T18:20:00.000+01:00","end":"2018-02-02T18:25:00.000+01:00"}	7
/shuttle/countdown/	{"start":"2018-02-02T18:21:00.000+01:00","end":"2018-02-02T18:26:00.000+01:00"}	7
/shuttle/countdown/liftoff.html	{"start":"2018-02-02T18:22:00.000+01:00","end":"2018-02-02T18:27:00.000+01:00"}	7
/shuttle/missions/sts-70/mission-sts-70.html	{"start":"2018-02-02T18:17:00.000+01:00","end":"2018-02-02T18:22:00.000+01:00"}	7
/shuttle/countdown/liftoff.html	{"start":"2018-02-02T18:20:00.000+01:00","end":"2018-02-02T18:25:00.000+01:00"}	6
/shuttle/countdown/	{"start":"2018-02-02T18:16:00.000+01:00","end":"2018-02-02T18:21:00.000+01:00"}	6
/ksc.html	{"start":"2018-02-02T18:17:00.000+01:00","end":"2018-02-02T18:22:00.000+01:00"}	6

Figure 7-1. URL ranking: query results by window

We explore event time in detail in [Chapter 12](#).

Summary

In these first steps into Structured Streaming, you have seen the process behind the development of a streaming application. By

starting with a batch version of the process, you gained intuition about the data, and using those insights, we created a streaming version of the job. In the process, you 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. You're now 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 the Spark SQL `DataFrames` and `Datasets` APIs of Spark SQL. By extending these APIs to support streaming workloads, Structured Streaming inherits the traits of the high-level language introduced by Spark SQL as well as the underlying optimizations, including the use of the Catalyst query optimizer and the low overhead memory management and code generation delivered by Project Tungsten. 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 currently available only in Scala. Thanks to the intermediate query representation used in Spark SQL, the performance of the programs is identical regardless of the *language binding* used.

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*. You learned these concepts in “[The Effect of Time](#)”.

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

In this chapter, we examine 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` as demonstrated in [Example 8-1](#).

Example 8-1. Creating a local Spark Session

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

val spark = SparkSession
  .builder()
  .appName("StreamProcessing")
```

```
.master("local[*]")  
.getOrCreate()
```

USING THE SPARK SHELL

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

Sources: Acquiring Streaming Data

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

For example, the code in [Example 8-2](#) creates a `File` streaming source. We specify the type of source using the `format` method. The method `schema` lets us provide a schema for the data stream, which is mandatory for certain source types, such as this `File` source.

Example 8-2. File streaming source

```
val fileStream = spark.readStream  
  .format("json")  
  .schema(schema)  
  .option("mode", "DROPMALFORMED")  
  .load("/tmp/data/src")  
  
>fileStream:  
org.apache.spark.sql.DataFrame = [id: string, timestamp:  
timestamp ... ]
```

Each source implementation has different options, and some have tunable parameters. In [Example 8-2](#), we are setting the option `mode` to `DROPMALFORMED`. This option instructs the JSON stream processor to drop any line that neither complies with the JSON format nor matches the provided schema.

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`.

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 the execution mode used in a Spark program:

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

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

Available Sources

As of Spark v2.4.0, the following streaming sources are supported:

`json, orc, parquet, csv, text, textFile`

These are all file-based streaming sources. The base functionality is 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.

`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.

We look at sources in detail in [Chapter 10](#).

Transforming Streaming Data

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

WARNING

Remember that `DataFrame` is an alias for `Dataset[Row]`. Although this might seem like a small technical distinction, when used from a typed language such as Scala, the `Dataset` API presents a typed interface, whereas 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.

There's also a performance impact when using operations on a typed `Dataset`. Although the SQL expressions used by the `DataFrame` API can be understood and further optimized by the query planner, closures provided in `Dataset` operations are opaque to the query planner and therefore might run slower than the exact same `DataFrame` counterpart.

Assuming that we are using data from a sensor network, in [Example 8-3](#) we are selecting the fields `deviceId`, `timestamp`, `sensorType`, and `value` from a `sensorStream` and filtering to only those records where the sensor is of type `temperature` and its value is higher than the given `threshold`.

Example 8-3. Filter and projection

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

Likewise, we can aggregate our data and apply operations to the groups over time. [Example 8-4](#) shows that we can use `timestamp` information from the event itself to define a time window of five minutes that will slide every minute. We cover event time in detail in [Chapter 12](#).

What is important to grasp here is that the Structured Streaming API is practically the same as the Dataset API for batch analytics, with some additional provisions specific to stream processing.

Example 8-4. Average by sensor type over time

```
val avgBySensorTypeOverTime = sensorStream
  .select($"timestamp", $"sensorType", $"value")
  .groupBy(window($"timestamp", "5 minutes", "1 minute"),
$"sensorType")
  .agg(avg($"value"))
```

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* (O'Reilly, 2018) by Bill Chambers and Matei Zaharia as a comprehensive reference.

Streaming API Restrictions on the DataFrame API

As we hinted in the previous chapter, some operations that are offered by the standard `DataFrame` and `Dataset` API do not make sense

on a streaming context.

We gave the example of `stream.count`, which does not make sense to use on a stream. In general, operations that require immediate materialization of the underlying dataset are not allowed.

These are the API operations not directly supported on streams:

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

Next to these operations, stream-stream and static-stream `joins` are partially supported.

UNDERSTANDING THE LIMITATIONS

Although some operations, like `count` or `limit`, do not make sense on a stream, some other stream operations are computationally difficult. For example, `distinct` is one of them. To filter duplicates in an arbitrary stream, it would require that you remember all of 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. Although we can't count the stream, we could count messages received per minute or count the number of devices of a certain type.

In [Example 8-5](#), we define a count of events per `sensorType` per minute.

Example 8-5. 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`. We examine about output modes in greater detail in [“outputMode”](#).

STREAM DEDUPLICATION

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

```
stream.dropDuplicates("<key-column>") ...
```

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`

Although `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 from where to consume the data 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`, as shown in [Example 8-6](#).

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

Example 8-6. File streaming sink

```
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 = ...
```

We cover sinks in detail in [Chapter 11](#).

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 v2.4.0, 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 writes the results to a filesystem. The format is specified by providing the format name: `csv`, `hive`, `json`, `orc`, `parquet`, `avro`, or `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, one element at the time.

`foreachBatch sink`

`foreachBatch` is a programmatic sink interface that provides access to the complete `DataFrame` that corresponds to each underlying microbatch of the Structured Streaming execution.

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.

`update`

Adds new and updated records since the last trigger to the output stream. `update` is meaningful only 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 the complete internal representation of the stream. This mode also relates to aggregations, because for nonaggregated streams, we would need to remember all records seen so far, which is unrealistic. From a practical perspective, `complete` mode is recommended only when you are aggregating values over low-cardinality criteria, like *count of visitors by country*, for which we know that the number of countries is bounded.

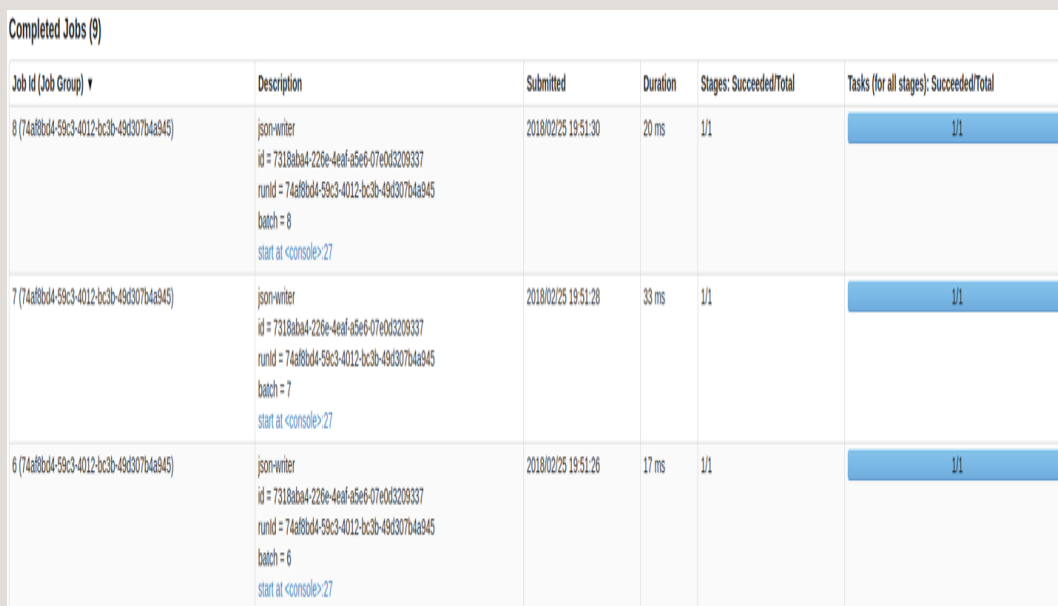
UNDERSTANDING THE APPEND SEMANTIC

When the streaming query contains aggregations, the definition of *final* becomes nontrivial. 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 the append output mode in combination with aggregate queries is restricted to queries for which the aggregation is expressed using event-time and it defines a watermark. In that case, append will output an event as soon as the watermark has expired and hence it's considered that no new records 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.

queryName

With `queryName`, we can provide a name for the query that is used by some sinks and also presented in the job description in the Spark Console, as depicted in Figure 8-1.



Job id (Job Group) ▼	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
8 (74a88bd4-59c3-4012-bc3b-49d307b4a945)	json-writer id = 7318aba4-226e-4eef-a5e6-07e03209337 runid = 74a88bd4-59c3-4012-bc3b-49d307b4a945 batch = 8 start at <console> 27	2018/02/25 19:51:30	20 ms	1/1	1/1
7 (74a88bd4-59c3-4012-bc3b-49d307b4a945)	json-writer id = 7318aba4-226e-4eef-a5e6-07e03209337 runid = 74a88bd4-59c3-4012-bc3b-49d307b4a945 batch = 7 start at <console> 27	2018/02/25 19:51:28	33 ms	1/1	1/1
6 (74a88bd4-59c3-4012-bc3b-49d307b4a945)	json-writer id = 7318aba4-226e-4eef-a5e6-07e03209337 runid = 74a88bd4-59c3-4012-bc3b-49d307b4a945 batch = 6 start at <console> 27	2018/02/25 19:51:26	17 ms	1/1	1/1

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 can have specific configuration we can customize 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 settings 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.

`org.apache.spark.sql.streaming.Trigger` provides the following supported triggers:

```
ProcessingTime(<interval>)
```

Lets us specify a time interval that will dictate the frequency of the query results.

`Once()`

A particular `Trigger` that lets us execute a streaming job once. It is useful for testing and also to apply a defined streaming job as a single-shot batch operation.

`Continuous(<checkpoint-interval>)`

This trigger switches the execution engine to the experimental `continuous` engine for low-latency processing. The `checkpoint-interval` parameter indicates the frequency of the asynchronous checkpointing for data resilience. It should not be confused with the `batch interval` of the `ProcessingTime` trigger. We explore this new execution option in [Chapter 15](#).

start()

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 from the source, processed, and produced to the sink. `start()` returns a `StreamingQuery` object, which is a handle to manage the individual life cycle of each query. This means that we can simultaneously start and stop multiple queries independently of one other within the same `sparkSession`.

Summary

After reading this chapter, you should have a good understanding of the Structured Streaming programming model and API. In this

chapter, you learned the following:

- Each streaming program starts by defining a source and what sources are currently available.
- We can reuse most of the familiar `Dataset` and `DataFrame` APIs for transforming the streaming data.
- Some common operations from the `batch` API do not make sense in streaming mode.
- Sinks are the configurable definition of the stream output.
- The relation between output modes and aggregation operations in the stream.
- All transformations are lazy and we need to `start` our stream to get data flowing through the system.

In the next chapter, you apply your 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 handling, window definitions, the concept of watermarks, and arbitrary state handling.