Stream Processing with Apache Spark

Mastering Structured Streaming and Spark Streaming

Gerard Maas and François Garillot

Beijing . Boston . Farnham . Sebastopol . Tokyo

O'REILLY®

SPD®

SHROFF PUBLISHERS & DISTRIBUTORS PVT. LTD.

Mumbai Bangalore Kolkata New Delhi

Stream Processing with Apache Spark

by Gerard Maas and François Garillot

 ISBN: 978-1-491-94424-0
Originally printed in the United States of America.

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

Acquisitions Editor: Rachel Roumeliotis
Developmental Editor: Jeff Bleiel
Production Editor: Nan Barber
Copyeditor: Octal Publishing Services, LLC
Proofreader: Kim Cofer

Indexer: Judith McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

Printing History:
June 2019: First Edition

Revision History for the First Edition: 2019-06-12: First Release

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

First Indian Reprint: July 2019

ISBN: 978-93-5213-858-6

Published by **Shroff Publishers and Distributors Pvt. Ltd.** B-103, Railway Commercial Complex, Sector 3, Sanpada (E), Navi Mumbai 400705 • TEL: (91 22) 4158 4158 • FAX: (91 22) 4158 4141
E-mail:spdorders@shroffpublishers.com • Web:www.shroffpublishers.com Printed at Sap Print Solution Pvt Ltd., Mumbai.

Table of Contents

Foreword

Welcome to *Stream Processing with Apache Spark*!

It's very exciting to see how much both the Apache Spark project, as well as stream processing with Apache Spark have come along since it was first started by Matei Zaharia at University of California Berkeley in 2009. Apache Spark started off as the first unified engine for big data processing and has grown into the de-facto standard for all things big data.

Stream Processing with Apache Spark is an excellent introduction to the concepts, tools, and capabilities of Apache Spark as a stream processing engine. This book will first introduce you to the core Spark concepts necessary to understand modern distributed processing. Then it will explore different stream processing architectures and the fundamental architectural trade-offs between then. Finally, it will illustrate how Structured Streaming in Apache Spark makes it easy to implement distributed streaming applications. In addition, it will also cover the older Spark Streaming (aka, DStream) APIs for building streaming applications with legacy connectors.

In all, this book covers everything you'll need to know to master building and operating streaming applications using Apache Spark! We look forward to hearing about what you'll build!

— Tathagata Das
Cocreator of Spark Streaming and
Structured Streaming

— Michael Armbrust
Cocreator of Spark SQL and Structured Streaming

— Bill Chambers
Coauthor of Spark: The Definitive Guide
May 2019

Preface

Who Should Read This Book?

We created this book for software professionals who have an affinity for data and who want to improve their knowledge and skills in the area of stream processing, and who are already familiar with or want to use Apache Spark for their streaming applications.

We have included a comprehensive introduction to the concepts behind stream processing. These concepts form the foundations to understand the two streaming APIs offered by Apache Spark: Structured Streaming and Spark Streaming.

We offer an in-depth exploration of these APIs and provide insights into their features, application, and practical advice derived from our experience.

Beyond the coverage of the APIs and their practical applications, we also discuss several advanced techniques that belong in the toolbox of every stream-processing practitioner.

Readers of all levels will benefit from the introductory parts of the book, whereas more experienced professionals will draw new insights from the advanced techniques covered and will receive guidance on how to learn more.

We have made no assumptions about your required knowledge of Spark, but readers who are not familiar with Spark's data-processing capabilities should be aware that in this book, we focus on its streaming capabilities and APIs. For a more general view of the Spark capabilities and ecosystem, we recommend *Spark: The Definitive Guide* by Bill Chambers and Matei Zaharia (O'Reilly).

The programming language used across the book is Scala. Although Spark provides bindings in Scala, Java, Python, and R, we think that Scala is the language of choice for streaming applications. Even though many of the code samples could be translated into other languages, some areas, such as complex stateful computations, are best approached using the Scala programming language.

Installing Spark

Spark is an Apache open source project hosted officially by the Apache Foundation, but which mostly uses GitHub for its development. You can also download it as a binary, pre-compiled package at the following address: *https://spark.apache.org/downloads.html*.

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

For the purposes of this book, we use examples and code compatible with Spark 2.4.0, and except for minor output and formatting details, those examples should stay compatible with future Spark versions.

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

To install a Java Development Kit (JDK), we recommend OpenJDK, which is packaged on many systems and architectures, as well.

You can also install the Oracle JDK.

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

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

Learning Scala

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

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

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

Hence, we hope the examples will stay clear enough for any reader to pick up their meanings. However, for the readers who might want a primer on the language and

who are more comfortable learning using a book, we advise *Atomic Scala* [Eckel2013]. For users looking for a reference book to touch up on their knowledge, we recommend *Programming in Scala* [Odersky2016].

The Way Ahead

This book is organized in five parts:

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

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

Part II, Structured Streaming, and Part III, Spark Streaming, follow a consistent structure. You can choose to cover one or the other first, to match your interest and most immediate priorities:

- Maybe you are starting a new project and want to know Structured Streaming? Check! Start in Part II.
- Or you might be jumping into an existing code base that uses Spark Streaming and you want to understand it better? Start in Part III.

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

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

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

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

Bibliography

- [Eckel2013] Eckel, Bruce and Dianne Marsh, *Atomic Scala* (Mindview LLC, 2013).
- [Odersky2016] Odersky, Martin, Lex Spoon, and Bill Venners, *Programming in Scala*, 3rd ed. (Artima Press, 2016).

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

The online repository for this book contains supplemental material to enhance the learning experience with interactive notebooks, working code samples, and a few projects that let you experiment and gain practical insights on the subjects and techniques covered. It can be found at *https://github.com/stream-processing-with-spark*.

The notebooks included run on the Spark Notebook, an open source, web-based, interactive coding environment developed with a specific focus on working with Apache Spark using Scala. Its live widgets are ideal to work with streaming applications as we can visualize the data as it happens to pass through the system.

The Spark Notebook can be found at *https://github.com/spark-notebook/spark-notebook*, and pre-built versions can be downloaded directly from their distribution site at *http://spark-notebook.io*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Stream Processing with Apache Spark*

by Gerard Maas and François Garillot (O'Reilly). Copyright 2019 François Garillot and Gerard Maas Images, 978-1-491-94424-0."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For almost 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/stream-proc-apache-spark*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

This book drastically evolved from its original inception as a learning manual for Spark Streaming to become a comprehensive resource on the streaming capabilities of Apache Spark. We would like to thank our reviewers for their invaluable feedback that helped steer this book into its current form. We are especially grateful to Russell Spitzer from Datastax, Serhat Yilmaz from Facebook, and Giselle Van Dongen from Klarrio.

We would like to extend our gratitude to Holden Karau for her help and advice in the early stages of the draft and to Bill Chambers for his continued support as we added coverage of Structured Streaming.

Our editor at O'Reilly, Jeff Bleiel, has been a stronghold of patience, feedback, and advice as we progressed from early ideas and versions of the draft until the completion of the content you have on your hands. We also would like to thank Shannon Cutt, our first editor at O'Reilly for all of her help in getting this project started. Other people at O'Reilly were there to assist us at many stages and help us move forward.

We thank Tathagata Das for the many interactions, in particular during the early days of Spark Streaming, when we were pushing the limits of what the framework could deliver.

From Gerard

I would like to thank my colleagues at Lightbend for their support and understanding while I juggled between book writing and work responsibilities. A very special thank you to Ray Roestenburg for his pep talks in difficult moments; to Dean Wampler for always being supportive of my efforts in this book; and to Ruth Stento for her excellent advice on writing style.

A special mention to Kurt Jonckheer, Patrick Goemaere, and Lieven Gesquière who created the opportunity and gave me the space to deepen my knowledge of Spark; and to Andy Petrella for creating the Spark Notebook, but more importantly, for his contagious passion and enthusiasm that influenced me to keep exploring the intersection of programming and data.

Most of all, I would like to express my infinite gratitude to my wife, Ingrid, my daughters Layla and Juliana, and my mother, Carmen. Without their love, care, and understanding, I wouldn't have been able to get through this project.

From François

I'm very grateful to my colleagues at Swisscom and Facebook for their support during the writing of this book; to Chris Fregly, Paco Nathan, and Ben Lorica for their advice and support; and to my wife AJung for absolutely everything.

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<m-dash>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, of failure.

Dealing with uncertainty is an important aspect of stream processing.

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

Why would we want to deal with the additional trouble? In the next section, we 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 turnaround 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.

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).

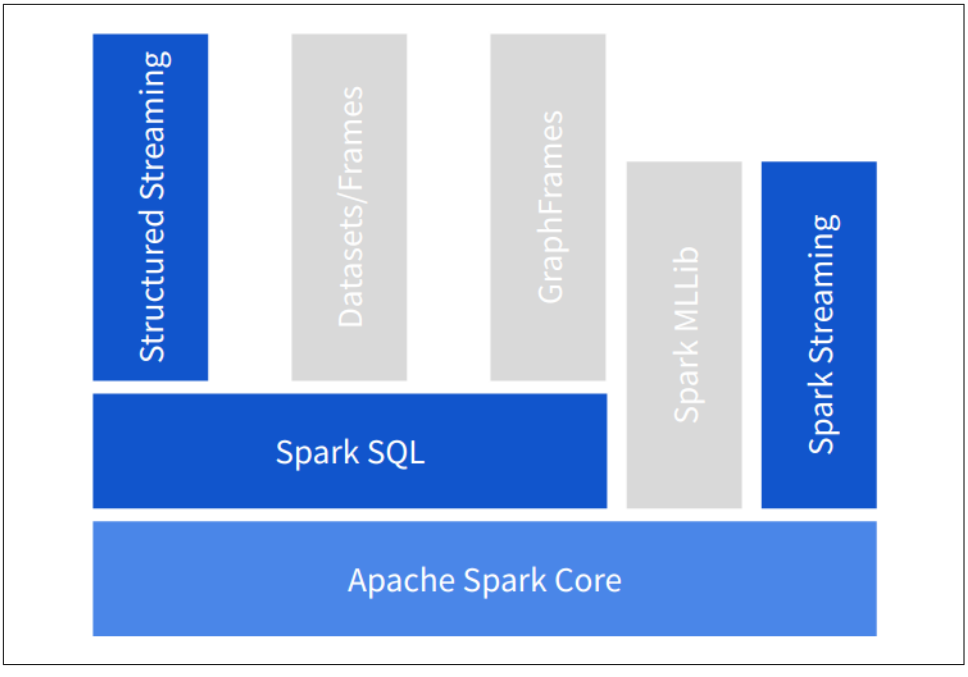

Figure 1-1. Abstraction layers (horizontal) and libraries (vertical) offered by Spark

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.