Pro Spark Streaming

The Zen of Real-time Analytics using Apache Spark

Zubair Nabi

Apress®

Pro Spark Streaming

The Zen of Real-Time Analytics Using Apache Spark

Zubair Nabi

Apress®

Pro Spark Streaming: The Zen of Real-Time Analytics Using Apache Spark

Zubair Nabi
Lahore, Pakistan

ISBN-13 (pbk): 978-1-4842-1480-0 ISBN-13 (electronic): 978-1-4842-1479-4
DOI 10.1007/978-1-4842-1479-4

Library of Congress Control Number: 2016941350

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `www.springer.com`. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail `rights@apress.com`, or visit `www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `www.apress.com/bulk-sales`.

Any source code or other supplementary materials referenced by the author in this text is available to readers at `www.apress.com`. For detailed information about how to locate your book's source code, go to `www.apress.com/source-code/`.

Printed on acid-free paper

To my father, who introduced me to the sanctity of the written word, who taught me that erudition transcends mortality, and who shaped me into the person I am today. Thank you, Baba.

Contents at a Glance

Contents

About the Author

Zubair Nabi is one of the very few computer scientists who have solved Big Data problems in all three domains: academia, research, and industry. He currently works at Qubit, a London-based start up backed by Goldman Sachs, Accel Partners, Salesforce Ventures, and Balderton Capital, which helps retailers understand their customers and provide personalized customer experience, and which has a rapidly growing client base that includes Staples, Emirates, Thomas Cook, and Topshop. Prior to Qubit, he was a researcher at IBM Research, where he worked at the intersection of Big Data systems and analytics to solve real-world problems in the telecommunication, electricity, and urban dynamics space.

Zubair's work has been featured in *MIT Technology Review*, *SciDev*, *CNET*, and *Asian Scientist*, and on Swedish National Radio, among others. He has authored more than 20 research papers, published by some of the top publication venues in computer science including USENIX Middleware, ECML PKDD, and IEEE BigData; and he also has a number of patents to his credit.

Zubair has an MPhil in computer science with distinction from Cambridge.

About the Technical Reviewer

Lan Jiang is a senior solutions consultant from Cloudera. He is an enterprise architect with more than 15 years of consulting experience, and he has a strong track record of delivering IT architecture solutions for Fortune 500 customers. He is passionate about new technology such as Big Data and cloud computing. Lan worked as a consultant for Oracle, was CTO for Infoble, was a managing partner for PARSE Consulting, and was a managing partner for InSemble Inc. prior to joining Cloudera. He earned his MBA from Northern Illinois University, his master's in computer science from University of Illinois at Chicago, and his bachelor's degree in biochemistry from Fudan University.

Acknowledgments

This book would not have been possible without the constant support, encouragement, and input of a number of people. First and foremost, Ammi and Sumaira deserve my neverending gratitude for being the bedrocks of my existence and for their immeasurable love and support, which helped me thrive under a mountain of stress.

Writing a book is definitely a labor of love, and my friends Devyani, Faizan, Natasha, Omer, and Qasim are the reason I was able to conquer this labor without flinching.

I cannot thank Lan Jiang enough for his meticulous attention to detail and for the technical rigour and depth that he brought to this book. Mobin Javed deserves a special mention for reviewing initial drafts of the first few chapters and for general discussions regarding open and public data.

Last but by no means least, hats off to the wonderful team at Apress, especially Celestin, Matthew, and Rita. You guys are the best.

Introduction

One million Uber rides are booked every day, 10 billion hours of Netflix videos are watched every month, and $1 trillion are spent on e-commerce web sites every year. The success of these services is underpinned by Big Data and increasingly, real-time analytics. Real-time analytics enable practitioners to put their fingers on the pulse of consumers and incorporate their wants into critical business decisions. We have only touched the tip of the iceberg so far. Fifty billion devices will be connected to the Internet within the next decade, from smartphones, desktops, and cars to jet engines, refrigerators, and even your kitchen sink. The future is data, and it is becoming increasingly real-time. Now is the right time to ride that wave, and this book will turn you into a pro.

The low-latency stipulation of streaming applications, along with requirements they share with general Big Data systems—scalability, fault-tolerance, and reliability—have led to a new breed of real-time computation. At the vanguard of this movement is Spark Streaming, which treats stream processing as discrete microbatch processing. This enables low-latency computation while retaining the scalability and fault-tolerance properties of Spark along with its simple programming model. In addition, this gives streaming applications access to the wider ecosystem of Spark libraries including Spark SQL, MLlib, SparkR, and GraphX. Moreover, programmers can blend stream processing with batch processing to create applications that use data at rest as well as data in motion. Finally, these applications can use out-of-the-box integrations with other systems such as Kafka, Flume, HBase, and Cassandra. All of these features have turned Spark Streaming into the Swiss Army Knife of real-time Big Data processing. Throughout this book, you will exercise this knife to carve up problems from a number of domains and industries.

This book takes a use-case-first approach: each chapter is dedicated to a particular industry vertical. Real-time Big Data problems from that field are used to drive the discussion and illustrate concepts from Spark Streaming and stream processing in general. Going a step further, a publicly available dataset from that field is used to implement real-world applications in each chapter. In addition, all snippets of code are ready to be executed. To simplify this process, the code is available online, both on GitHub[1] and on the publisher's web site. Everything in this book is real: real examples, real applications, real data, and real code. The best way to follow the flow of the book is to set up an environment, download the data, and run the applications as you go along. This will give you a taste for these real-world problems and their solutions.

These are exciting times for Spark Streaming and Spark in general. Spark has become the largest open source Big Data processing project in the world, with more than 750 contributors who represent more than 200 organizations. The Spark codebase is rapidly evolving, with almost daily performance improvements and feature additions. For instance, Project Tungsten (first cut in Spark 1.4) has improved the performance of the underlying engine by many orders of magnitude. When I first started writing the book, the latest version of Spark was 1.4. Since then, there have been two more major releases of Spark (1.5 and 1.6). The changes in these releases have included native memory management, more algorithms in MLlib, support for deep learning via TensorFlow, the Dataset API, and session management. On the Spark Streaming front, two major features have been added: `mapWithState` to maintain state across batches and using back pressure to throttle the input rate in case of queue buildup.[2] In addition, managed Spark cloud offerings from the likes of Google, Databricks, and IBM have lowered the barrier to entry for developing and running Spark applications.

Now get ready to add some "Spark" to your skillset!

[1] https://github.com/ZubairNabi/prosparkstreaming.

[2] All of these topics and more will hopefully be covered in the second edition of the book.

CHAPTER 1

The Hitchhiker's Guide to Big Data

From a little spark may burst a flame.

—Dante

By the time you get to the end of this paragraph, you will have processed 1,700 bytes of data. This number will grow to 500,000 bytes by the end of this book. Taking that as the average size of a book and multiplying it by the total number of books in the world (according to a Google estimate, there were 130 million books in the world in 2010[1]) gives 65 TB. That is a staggering amount of data that would require 130 standard, off-the-shelf 500 GB hard drives to store.

Now imagine you are a book publisher and you want to translate all of these books into multiple languages (for simplicity, let's assume all these books are in English). You would like to translate each line as soon as it is written by the author—that is, you want to perform the translation in real time using a stream of lines rather than waiting for the book to be finished. The average number of characters or bytes per line is 80 (this also includes spaces). Let's assume the author of each book can churn out 4 lines per minute (320 bytes per minute), and all the authors are writing concurrently and nonstop. Across the entire 130 million-book corpus, the figure is 41,600,000,000 bytes, or 41.6 GB per minute. This is well beyond the processing capabilities of a single machine and requires a multi-node cluster. Atop this cluster, you also need a real-time data-processing framework to run your translation application. Enter Spark Streaming. Appropriately, this book will teach you to architect and implement applications that can process data at scale and at line-rate.

Before discussing Spark Streaming, it is important to first trace the origin and evolution of Big Data systems in general and Spark in particular. This chapter does just that.

Before Spark

Two major trends were the precursors to today's Big Data processing systems, such as Hadoop and Spark: Web 2.0 applications, for instance, social networks and blogs; and real-time sources, such as sensor networks, financial trading, and bidding. Let's discuss each in turn.

Electronic supplementary material The online version of this chapter (doi:10.1007/978-1-4842-1479-4_1) contains supplementary material, which is available to authorized users.

[1]Leonid Taycher, "Books of the world, stand up and be counted! All 129,864,880 of you," *Google Books Search*, 2010, `http://booksearch.blogspot.com/2010/08/books-of-world-stand-up-and-be-counted.html`.

The Era of Web 2.0

The new millennium saw the rise of Web 2.0 applications, which revolved around user-generated content. The Internet went from hosting static content to content that was dynamic, with the end user in the driving seat. In a matter of months, social networks, photo sharing, media streaming, blogs, wikis, and their ilk became ubiquitous. This resulted in an explosion in the amount of data on the Internet. To even store this data, let alone process it, an entirely different new of computing, dubbed *warehouse-scale computing*,[2, 3] was needed.

In this architecture, data centers made up of commodity off-the-shelf servers and network switches act as a large distributed system. To exploit economies of scale, these data centers host tens of thousands of machines under the same roof, using a common power and cooling mechanism. Due to the use of commodity hardware, failure is the norm rather than the exception. As a consequence, both the hardware topology and the software stack are designed with this as a first principle. Similarly, computation and data are load-balanced across many machines for processing and storage parallelism. For instance, Google search queries are sharded across many machines in a tree-like, divide-and-conquer fashion to ensure low latency by exploiting parallelism.[4] This data needs to be stored somewhere before any processing can take place—a role fulfilled by the relational model for more than four decades.

From SQL to NoSQL

The size, speed, and heterogeneity of this data, coupled with application requirements, forced the industry to reconsider the hitherto role of relational database-management systems as the de facto standard. The relational model, with its Atomicity, Consistency, Isolation, Durability (ACID) properties could not cater to the application requirements and the scale of the data; nor were some of its guarantees required any longer. This led to the design and wide adoption of the Basically Available, Soft state, Eventual consistency (BASE) model. The BASE model relaxed some of the ACID guarantees to prioritize availability over consistency: if multiple readers/writers access the same shared resource, their queries always go through, but the result may be inconsistent in some cases.

This trade-off was formalized by the Consistency, Availability, Partitioning (CAP) theorem.[5, 6] According to this theorem, only two of the three CAP properties can be achieved at the same time.[7] For instance, if you want availability, you must forego either consistency or tolerance to network partitioning. As discussed earlier, hardware/software failure is a given in data centers due to the use of commodity off-the-shelf hardware. For that reason, network partitioning is a common occurrence, which means storage systems must trade off either availability or consistency. Now imagine you are designing the next Facebook, and you have to make that choice. Ensuring consistency means some of your users will have to wait a few milliseconds or even seconds before they are served any content. On the other hand, if you opt for availability, these users will always be served content—but some of it may be stale. For example, a user's Facebook newsfeed might contain posts that have been deleted. Remember, in the Web 2.0 world, the user is the main target (more users mean more revenue for your company), and the user's attention span (and in turn patience span) is very short.[8] Based on this fact, the choice is obvious: availability over consistency.

[2]IEEE Computer Society, "Web Search for a Planet: The Google Cluster Architecture," 2003, http://static.googleusercontent.com/media/research.google.com/en//archive/googlecluster-ieee.pdf.
[3]Luiz André Barroso and Urs Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines* (Morgan& Claypool, 2009), www.morganclaypool.com/doi/abs/10.2200/S00193ED1V01Y200905CAC006.
[4]Jeffrey Dean and Luiz André Barroso, "The Tail at Scale," *Commun. ACM* 56, no 2 (February 2013), 74-80.
[5]First described by Eric Brewer, the Chief Scientist of Inktomi, one of the earliest web giants in the 1990s.
[6]Werner Vogels, "Eventually Consistent – Revisited," *All Things Distributed*, 2008, www.allthingsdistributed.com/2008/12/eventually_consistent.html.
[7]ACID and BASE are not binary choices, though. There is a continuum between the two, with many design points.
[8]This attention span is getting shorter because most users now consume these services on the go on mobile devices.

A nice side property of eventual consistency is that applications can read/write at a much higher throughput and can also shard as well as replicate data across many machines. This is the model adopted by almost all contemporary NoSQL (as opposed to traditional SQL) data stores. In addition to higher scalability and performance, most NoSQL stores also have simpler query semantics in contrast to the somewhat restrictive SQL interface. In fact, most NoSQL stores only expose simple key/value semantics. For instance, one of the earliest NoSQL stores, Amazon's Dynamo, was designed with Amazon's platform requirements in mind. Under this model, only primary-key access to data, such as customer information and bestseller lists, is required; thus the relational model and SQL are overkill. Examples of popular NoSQL stores include key-value stores, such as Amazon's DynamoDB and Redis; column-family stores, such as Google's BigTable (and its open source version HBase) and Facebook's Cassandra; and document stores, such as MongoDB.

MapReduce: The Swiss Army Knife of Distributed Data Processing

As early as the late 1990s, engineers at Google realized that most of the computations they performed internally had three key properties:

- Logically simple, but complexity was added by control code.
- Processed data that was distributed across many machines.
- Had divide-and-conquer semantics.

Borrowing concepts from functional programming, Google used this information to design a library for large-scale distributed computation, called MapReduce. In the MapReduce model, the user only has to provide `map` and `reduce` functions; the underlying system does all the heavy lifting in terms of scheduling, data transfer, synchronization, and fault tolerance.

In the MapReduce paradigm, the `map` function is invoked for each input record to produce key-value pairs. A subsequent internal `groupBy` and shuffle (transparent to the user) group different keys together and invoke the `reduce` function for each key. The `reduce` function simply aggregates records by key. Keys are hash-partitioned by default across `reduce` tasks. MapReduce uses a distributed file system, called the Google File System (GFS), for data storage. Input is typically read from GFS by the `map` tasks and written back to GFS at the end of the `reduce` phase. Based on this, GFS is designed for large, sequential, bulk reads and writes.

GFS is deployed on the same nodes as MapReduce, with one node acting as the master to keep metadata information while the rest of the nodes perform data storage on the local file system. To exploit data locality, `map` tasks are ideally executed on the same nodes as their input: MapReduce ships out computation closer to the data than vice versa to minimize network I/O. GFS divvies up files into chunks/blocks where each chunk is replicated *n* times (three by default). These chunks are then distributed across a cluster by exploiting its typical three-tier architecture. The first replica is placed on the same node if the writer is on a data node; otherwise a random data node is selected. The second and third replicas are shipped out to two distinct nodes on a different rack. Typically, the number of `map` tasks is equivalent to the number of chunks in the input dataset, but it can differ if the input split size is changed. The number of `reduce` tasks, on the other hand, is a configurable value that largely depends on the capabilities of each node.

Similar to GFS, MapReduce also has a centralized master node, which is in charge of cluster-wide orchestration and worker nodes that execute processing tasks. The execution flow is barrier controlled: `reduce` tasks only start processing once a certain number of `map` tasks have completed. This model also simplifies fault-tolerance via re-execution: every time a task fails, it is simply re-executed. For instance, if the output of a `map` task is lost, it can readily be re-executed because its input resides on GFS. If a `reduce` task fails, then if its inputs are still available on the local file system of the `map` tasks (`map` tasks write their intermediate state to the local file system, not GFS) that processed keys from the partition assigned to that reduce task, the input is shuffled again; otherwise, the `map` tasks need to be selectively or entirely re-executed. Tasks (`map` or `reduce`) whose progress rate is slower than the job average, known as *stragglers*, are speculatively executed on free nodes. Whichever copy finishes first—the original or the speculative one—registers its output; the other is killed. This optimization helps to negate hardware heterogeneity. For reduce functions, which are associative and commutative, an optional combiner can also be provided; it is applied locally to the output of each map task. In most cases, this combine function is a local version of the reduce function and helps to minimize the amount of data that needs to be shipped across the network during the shuffle phase.

Word Count a la MapReduce

To illustrate the flow of a typical MapReduce job, let's use the canonical word-count example. The purpose of the application is to count the occurrences of each word in the input dataset. For the sake of simplicity, let's assume that an input dataset—say, a Wikipedia dump—already resides on GFS. The following map and reduce functions (in pseudo code) achieve the logic of this application:

```
map(k, v):
    for word in v.split(" "):
        emit((word, 1))

reduce(k, v):
    sum = 0
    for count in v.iterator():
        sum += count
    emit(k, sum)
```

Here's the high-level flow of this execution:

1. Based on the specified format of the input file (in this case, text) the MapReduce subsystem uses an input reader to read the input file from GFS line by line. For each line, it invokes the map function.
2. The first argument of the map function is the line number, and the second is the line itself in the form of a text object (say, a string). The map function splits the line at word boundaries using space characters. Then, for each word, it emits (to a component, let's call it the *collector*) the word itself and the value 1.
3. The collector maintains an in-memory buffer that it periodically spills to disk. If an optional combiner has been turned on, it invokes that on each key (word) before writing it to a file (called a *spill file*). The partitioner is invoked at this point as well, to slice up the data into per-reducer partitions. In addition, the keys are sorted. Recall that if the reduce function is employed as a combiner, it needs to be associative and commutative. Addition is both, that's why the word-count reduce can also be used as a combiner.
4. Once a configurable number of maps have completed execution, reduce tasks are scheduled. They first pull their input from map tasks (the sorted spill files created by the collector) and perform an *n*-way merge. After this, the user-provided reduce function is invoked for each key and its list of values.
5. The reduce function counts the occurrences of each word and then emits the word and its sum to another collector. In contrast to the map collector, this reduce collector spills its output to GFS instead of the local file system.

Google internally used MapReduce for many years for a large number of applications including Inverted Index and PageRank calculations. Some of these applications were subsequently retired and reimplemented in newer frameworks, such as Pregel[9] and Dremel.[10] The engineers who worked on MapReduce and GFS shared their creations with the rest of the world by documenting their work in the form of research papers.[11, 12] These seminal publications gave the rest of the world insight into the inner wirings of the Google engine.

Hadoop: An Elephant with Big Dreams

In 2004, Doug Cutting and Mike Cafarella, both engineers at Yahoo! who were working on the Nutch search engine, decided to employ MapReduce and GFS as the crawl and index and, storage layers for Nutch, respectively. Based on the original research papers, they reimplemented MapReduce and GFS in Java and christened the project Hadoop (Doug Cutting named it after his son's toy elephant). Since then, Hadoop has evolved to become a top-level Apache project with thousands of industry users. In essence, Hadoop has become synonymous with Big Data processing, with a global market worth multiple billions of dollars. In addition, it has spawned an entire ecosystem of projects, including high-level languages such as Pig and FlumeJava (open source variant Crunch); structured data storage, such as Hive and HBase; and data-ingestion solutions, such as Sqoop and Flume; to name a few. Furthermore, libraries such as Mahout and Giraph use Hadoop to extend its reach to domains as diverse as machine learning and graph processing.

Although the MapReduce programming model at the heart of Hadoop lends itself to a large number of applications and paradigms, it does not naturally apply to others:

- *Two-stage programming model:* A particular class of applications cannot be implemented using a single MapReduce job. For example, a top-k calculation requires two MapReduce jobs: the first to work out the frequency of each word, and the second to perform the actual top-k ranking. Similarly, one instance of a PageRank algorithm also requires two MapReduce jobs: one to calculate the new page rank and one to link ranks to pages. In addition to the somewhat awkward programming model, these applications also suffer from performance degradation, because each job requires data materialization. External solutions, such as Cascading and Crunch, can be used to overcome some of these shortcomings.

- *Low-level programming API:* Hadoop enforces a low interface in which users have to write `map` and `reduce` functions in a general-purpose programming language such as Java, which is not the weapon of choice for most data scientists (the core users of systems like Hadoop). In addition, most data-manipulation tasks are repetitive and require the invocation of the same function multiple times across applications. For instance, filtering out a field from CSV data is a common task. Finally, stitching together a directed acyclic graph of computation for data science tasks requires writing custom code to deal with scheduling, graph construction, and end-to-end fault tolerance. To remedy this, a number of high-level languages that expose a SQL-like interface have been implemented atop Hadoop and MapReduce, including Pig, JAQL, and HiveQL.

---

[9]Grzegorz Malewicz et al., "Pregel: A System for Large-Scale Graph Processing," *Proceedings of SIGMOD '10* (ACM, 2010), 135-146.
[10]Sergey Melnik et al., "Dremel: Interactive Analysis of Web-Scale Datasets, *Proc. VLDB Endow 3*, no. 1-2 (September 2010), 330-339.
[11]Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Proceedings of OSDI 04* 6 (USENIX Association, 2004), 10.
[12]Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, "The Google File System," *Proceedings of SOSP '03* (ACM, 2003), 29-43.

- *Iterative applications:* Iterative applications that perform the same computation multiple times are also a bad fit for Hadoop. Many machine-learning algorithms belong to this class of applications. For example, k-means clustering in Mahout refines its centroid location in every iteration. This process continues until a threshold of iterations or a convergence criterion is reached. It runs a driver program on the user's local machine, which performs the convergence test and schedules iterations. This has two main limitations: the output of each iteration needs to be materialized to HDFS, and the driver program resides in the user's local machine at an I/O cost and with weaker fault tolerance.
- *Interactive analysis:* Results of MapReduce jobs are available only at the end of execution. This is not viable for large datasets where the user may want to run interactive queries to first understand their semantics and distribution before running full-fledged analytics. In addition, most of the time, many queries need to be run against the same dataset. In the MapReduce world, each query runs as a separate job, and the same dataset needs to be read and loaded into memory each time.
- *Strictly batch processing:* Hadoop is a batch-processing system, which means its jobs expect all the input to have been materialized before processing can commence. This model is in tension with real-time data analysis, where a (potentially continuous) stream of data needs to be analyzed on the fly. Although a few attempts[13] have been made to retrofit Hadoop to cater to streaming applications, none of them have been able to gain wide traction. Instead, systems tailor-made for real-time and streaming analytics, including Storm, S4, Samza, and Spark Streaming, have been designed and employed over the last few years.
- *Conflation between control and computation:* Hadoop v1 by default assumes full control over a cluster and its resources. This resource hoarding prevents other systems from sharing these resources. To cater to disparate application needs and data sources, and to consolidate existing data center investments, organizations have been looking to deploy multiple systems, such as Hadoop, Storm, Hama, and so on, on the same cluster and pass data between them. Apache YARN, which separates the MapReduce computation layer from the cluster-management and -control layer in Hadoop, is one step in that direction. Apache Mesos is another similar framework that enables platform heterogeneity in the same namespace.

Sensors, Sensors Everywhere

In tandem with Web 2.0 applications, the early 2000s also witnessed the advent and widespread deployment of sensor networks. During this sensor data boom, they were used to monitor entities as diverse as gas-distribution pipes, home automation systems, environmental conditions, and transportation systems. In this ecosystem, humans also acted as sensors by generating contextual data via smart phones and wearable devices, especially medical devices.[14] These data sources were augmented by data from telecommunication, including call data records, financial feeds from stock markets, and network traffic. The requirements to analyze and store these data sources included low-latency processing, blending data in motion with data at rest, high availability, and scalability. Some initial systems from academia to cater to these needs, dubbed *stream-processing* or *complex event-processing* (CEP) systems, were Aurora,[15] Borealis,[16] Medusa,[17] and

[13]Tyson Condie et al., "MapReduce Online," *Proceedings of NSDI '10* (USENIX, 2010), 21.
[14]This is now known as the Internet of Things (IoT).
[15]`http://cs.brown.edu/research/aurora/`.
[16]`http://cs.brown.edu/research/borealis/`.
[17]`http://nms.csail.mit.edu/projects/medusa/`.

TelegraphCQ.[18] Of these, Aurora was subsequently commercialized by StreamBase Systems[19] (later acquired by TIBCO) as StreamBase CEP, with a high-level description language called StreamSQL, running atop a low-latency dataflow engine. Other examples of commercial systems included IBM InfoSphere Streams[20] and Software AG Apama streaming analytics.[21]

Widespread deployment of IoT devices and real-time Web 2.0 analytics at the end of the 2000s breathed fresh life into stream-processing systems. This time, the effort was spearheaded by the industry. The first system to emerge out of this resurgence was S4[22, 23] from Yahoo!. S4 combines the Actors model with the simplified MapReduce programming model to achieve general-purpose stream processing. Other features include completely decentralized execution (thanks to ZooKeeper, a distributed cluster-coordination system) and lossy failure. The programming model consists of *processing elements* connected via *streams* to form a graph of computation. Processing elements are executed on nodes, and communication takes place via partitioned *events*.

Another streaming system from the same era is Apache Samza[24] (originally developed at LinkedIn), which uses YARN for resource management and Kafka (a pub/sub-based log-queuing system)[25] for messaging. As a result, its message-ordering and -delivery guarantees depend on Kafka. Samza messages have *at-least-once* semantics, and ordering is guaranteed in a Kafka partition. Unlike S4, which is completely decentralized, Samza relies on an application master for management tasks, which interfaces with the YARN resource manager. Stitching together tasks using Kafka messages creates Samza jobs, which are then executed in containers obtained from YARN.

The most popular and widely used system in the Web 2.0 streaming world is Storm.[26] Originally developed at the startup BackType to support its social media analytics, it was open sourced once Twitter acquired the company. It became a top-level Apache project in late 2014 with hundreds of industry deployments. Storm applications constitute a directed acyclic graph (called a *topology*) where data flows from sources (called *spouts*) to output channels (standard output, external storage, and so on). Both intermediate transformations and external output are implemented via *bolts*. Tuple communication channels are established between tasks via named *streams*. A central *Nimbus* process handles job and resource orchestration while each worker node runs a *Supervisor* daemon, which executes tasks (spouts and bolts) in worker processes. Storm enables three delivery modes: at-most-once, at-least-once, and exactly-once. At-most-once is the default mode in which messages that cannot be processed are dropped. At-least-once semantics are provided by the guaranteed tuple-processing mode in which downstream operators need to acknowledge each tuple. Tuples that are not acknowledged within a configurable time duration are replayed. Finally, exactly-once semantics are ensured by Trident, which is a batch-oriented, high-level transactional abstraction atop Storm. Trident is very similar in spirit to Spark Streaming.

In the last few years, a number of cloud-hosted, fully managed streaming systems have also emerged, including Amazon's Kinesis,[27] Microsoft's Azure Event Hubs,[28] and Google's Cloud Dataflow.[29] Let's consider a brief description of Cloud Dataflow as a representative system. Cloud Dataflow under the hood employs

[18]http://telegraph.cs.berkeley.edu/.
[19]www.streambase.com/.
[20]http://www-03.ibm.com/software/products/en/infosphere-streams.
[21]www.softwareag.com/corporate/products/apama_webmethods/analytics/overview/default.asp.
[22]Leonardo Neumeyer et al., "S4: Distributed Stream Computing Platform," *Proceedings of* ICDMW '10 (IEEE, 2010), 170-177.
[23]http://incubator.apache.org/s4/.
[24]http://samza.apache.org/.
[25]Chapter 4 describes Kafka in detail when we analyze the various external sources from which to ingest data.
[26]https://storm.apache.org/.
[27]http://aws.amazon.com/kinesis/.
[28]http://azure.microsoft.com/en-us/services/event-hubs/.
[29]https://cloud.google.com/dataflow/.

MillWheel[30] and MapReduce as the processing engines and FlumeJava[31] as the programming API. MillWheel is a stream-processing system developed in house at Google, with one delineating feature: low watermark. The low watermark is a monotonically increasing timestamp signifying that all events until that timestamp have been processed. This removes the need for strict ordering of events. In addition, the underlying system ensures exactly-once semantics (in contrast to Storm, which uses a custom XOR scheme for deduplication, MillWheel uses Bloom filters). Atop this engine, FlumeJava provides a Java Collections–centric API, wherein data is abstracted in a *`PCollection`* object, which is materialized only when a transform is applied to it. Out of the box it interfaces with BigQuery, PubSub, Cloud BigTable, and many others.

Spark Streaming: At the Intersection of MapReduce and CEP

Before jumping into a detailed description of Spark, let's wrap up this brief sweep of the Big Data landscape with the observation that Spark Streaming is an amalgamation of ideas from MapReduce-like systems and complex event-processing systems. MapReduce inspires the API, fault-tolerance properties, and wider integration of Spark with other systems. On the other hand, low-latency transforms and blending data at rest with data in motion is derived from traditional stream-processing systems.

We hope this will become clearer over the course of this book. Let's get to it.

[30]Tyler Akidau et al., "MillWheel: Fault-Tolerant Stream Processing at Internet Scale," *Proc. VLDB Endow.* 6, no. 11 (August 2013), 1033-1044.
[31]Craig Chambers et al., "FlumeJava: Easy, Efficient Data-Parallel Pipelines, *SIGPLAN Not.* 45, no. 6 (June 2010), 363-375.

CHAPTER 2

Introduction to Spark

There are two major products that came out of Berkeley: LSD and UNIX. We don't believe this to be a coincidence.

—Jeremy S. Anderson

Like LSD and Unix, Spark was originally conceived in 2009[1] at Berkeley,[2] in the same Algorithms, Machines, and People (AMP) Lab that gave the world RAID, Mesos, RISC, and several Hadoop enhancements. It was initially pitched to the academic community as a distributed framework built from the ground up atop the Mesos cross-platform scheduler (then called Nexus). Spark can be thought of as an in-memory variant of Hadoop, with the following key differences:

- *Directed acyclic graph*: Spark applications form a directed acyclic graph of computation, unlike MapReduce, which is strictly two-stage.
- *In-memory analytics:* At the very heart of Spark lies the concept of resilient distributed datasets (RDDs)—datasets that can be cached in memory. For fault-tolerance, each RDD also stores its lineage graph that consists of transformations that need to be partially or fully executed to regenerate it. RDDs accelerate the performance of iterative and interactive applications.

    a. *Iterative applications:* A cached RDD can be reused across iterations without having to read it from disk every time.

    b. *Interactive applications:* Multiple queries can be run against the same RDD.

    RDDs can also be persisted as files on HDFS and other stores. Therefore, Spark relies on a Hadoop distribution to make use of HDFS.
- *Data first:* The RDD abstraction cements data as a first-class citizen in the Spark API. Computation is performed by manipulating an RDD to generate another one, and so on. This is in contrast to MapReduce, where you reason about the dataset only in terms of key-value pairs, and the focus is on the computation.

[1]Matei Zaharia et al., "Spark: Cluster Computing with Working Sets, *Proceedings of HotCloud '10* (USENIX Association, 2010).
[2]*Insert "speed" joke here.*

- *Concise API:* Spark is implemented using Scala, which also underpins its default API. The functional abstracts of Scala naturally fit RDD transforms, such as `map`, `groupBy`, `filter`, and so on. In addition, the use of anonymous functions (or lambda abstractions) simplifies standard data-manipulation tasks.
- *REPL analysis:* The use of Scala enables Spark to use the Scala interpreter as an interactive data-analytics tool. You can, for instance, use the Scala shell to learn the distribution and characteristics of a dataset before a full-fledged analysis.

The first version of Spark was open sourced in 2010, and it went into Apache incubation in 2013. By early 2014, it was promoted to a top-level Apache project. Its current popularity can be gauged by the fact that it has the most contributors (in excess of 700) across all Apache open source projects. Over the last few years, Spark has also spawned a number of related projects:

- *Spark SQL* (and its predecessor, Shark[3]) enables SQL queries to be executed atop Spark. This coupled with the DataFrame[4] abstraction makes Spark a powerful tool for data analysis.
- *MLlib* (similar to Mahout atop Hadoop) is a suite of popular machine-learning and data-mining algorithms. In addition, it contains abstractions to assist in feature extraction.
- *GraphX* (similar to Giraph atop Hadoop) is a graph-processing framework that uses Spark under the hood. In addition to graph manipulation, it also contains a library of standard algorithms, such as PageRank and Connected Components.
- *Spark Streaming* turns Spark into a real-time, stream-processing system by treating input streams as micro-batches while retaining the familiar syntax of Spark.

Installation

The best way to learn Spark (or anything, for that matter) is to start getting your hands dirty right from the onset. The first order of the day is to set up Spark on your machine/cluster. Spark can be built either from source or by using prebuilt versions for various Hadoop distributions. You can find prebuilt distributions and source code on the official Spark web site: `https://spark.apache.org/downloads.html`. Alternatively, the latest source code can also be accessed from Git at `https://github.com/apache/spark`.

At the time of writing, the latest version of Spark is 1.4.0; that is the version used in this book, along with Scala 2.10.5. The Java and Python APIs for Spark are very similar to the Scala API so it should be very straightforward to port all the Scala applications in this book to those languages if required. Note that some of the syntax and components may have changed in subsequent releases.

Spark can also be run in a YARN installation. Therefore, make sure you either use a prebuilt version of Spark with YARN support or specify the correct version of Hadoop while building Spark from source, if you plan on going this route.

[3]Reynold Xin, "Shark, Spark SQL, Hive on Spark, and the Future of SQL on Spark," *Databricks*, July 1, 2014, `https://databricks.com/blog/2014/07/01/shark-spark-sql-hive-on-spark-and-the-future-of-sql-on-spark.html`.
[4]Reynold Xin, Michael Armbrust, and Davies Liu, "Introducing DataFrames in Spark for Large Scale Data Science," *Databricks*, February 17, 2015, `https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html`.

Installing Spark is just a matter of copying the compiled distribution to an appropriate location in the local file system of each machine in the cluster. In addition, set `SPARK_HOME` to that location, and add `$SPARK_HOME/bin` to the `PATH` variable. Recall that Spark relies on HDFS for optional RDD persistence. If your application either reads data from or writes data to HDFS or persists RDDs to it, make you sure you have a properly configured HDFS installation available in your environment.

Also note that Spark can be executed in a local, noncluster mode, in which your entire application executes in a single (potentially multithreaded) process.

Execution

Spark can be executed as a standalone framework, in a cross-platform scheduler (such as Mesos/YARN), or on the cloud. This section introduces you to launching Spark on a standalone cluster and in YARN.

Standalone Cluster

In the standalone-cluster mode, Spark processes need to be manually started. This applies to both the master and the worker processes:

Master

To run the master, you need to execute the following script: `$SPARK_HOME/sbin/start-master.sh`.

Workers

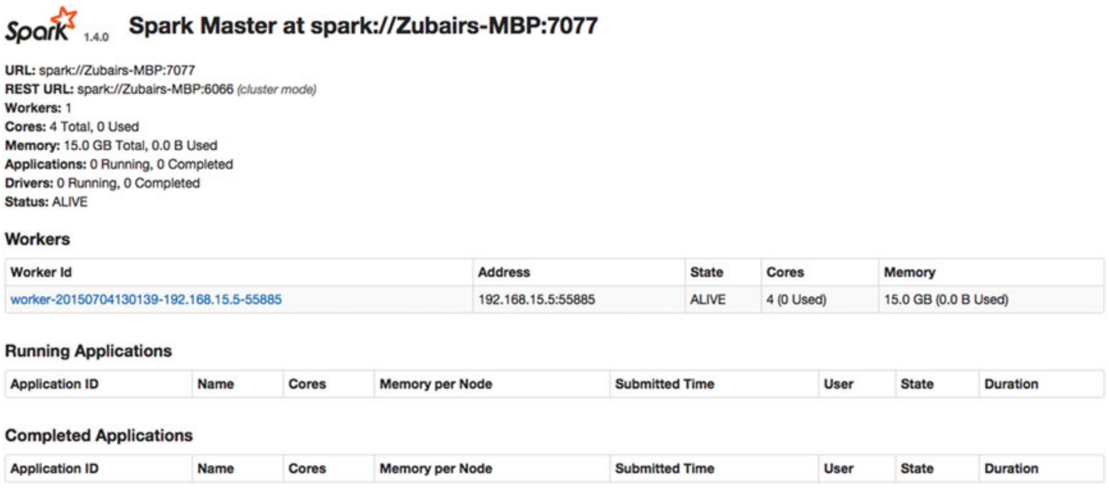
Workers can be executed either manually on each machine or by using a helper script. To execute a worker on each machine, execute the script `$SPARK_HOME/sbin/start-slave.sh <master_url>`, where `master_url` is of the form `spark://hostname:port`. You can obtain this from the log of the master.

It is clearly tedious to start a worker process manually on each machine in the cluster, especially in clusters with thousands of machines. To remedy this, create a file named `slaves` under `$SPARK_HOME/conf` and fill it with the hostnames of all the machines in your cluster, one per line. Subsequently executing `$SPARK_HOME/sbin/start-slaves.sh` will seamlessly start worker processes on these machines.[5]

UI

Spark in standalone mode also includes a handy UI that is executed on the same node as the master on port 8080 (see Figure 2-1). The UI is useful for viewing cluster-wide resource and application state. A detailed discussion of the UI is deferred to Chapter 7, when the book discusses optimization techniques for Spark Streaming applications.

[5]This requires passwordless key-based authentication between the master and all worker nodes. Alternatively, you can set `SPARK_SSH_FOREGROUND` and provide a password for each worker machine.



Spark Master at spark://Zubairs-MBP:7077

URL: spark://Zubairs-MBP:7077
REST URL: spark://Zubairs-MBP:6066 *(cluster mode)*
Workers: 1
Cores: 4 Total, 0 Used
Memory: 15.0 GB Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20150704130139-192.168.15.5-55885	192.168.15.5:55885	ALIVE	4 (0 Used)	15.0 GB (0.0 B Used)

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Completed Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Figure 2-1. *Spark UI*

YARN

With YARN, the Spark application master and workers are launched for each job in YARN containers on the fly. The application master starts first and coordinates with the YARN resource manager to grab containers for executors. Therefore, other than having a running YARN deployment and submitting a job, you don't need to launch anything else.

First Application

Get ready for your very first Spark application. In this section, you will implement the batch version of the translation application mentioned in the first paragraph of this book. Listing 2-1 contains the code of the driver program. The driver is the gateway to every Spark application because it runs the `main()` function. The job of this driver is to act as the interface between user code and a Spark deployment. It runs either on your local machine or on a worker node if it's a cluster deployment or running under YARN/Mesos. Table 2-1 explains the different Spark processes and their roles.

Table 2-1. *Spark Control Processes*

Daemon	Description
Driver	Application entry point that contains the `SparkContext` instance
Master	In charge of scheduling and resource orchestration
Worker	Responsible for node state and running executors
Executor	Allocated per job and in charge of executing tasks from that job

■ **Note** It is highly recommended that you run the driver program on a machine on the cluster, such as the master node, especially if the driver pulls in data from the workers, to reduce network traffic.

Listing 2-1. Translation Application

```
1.   package org.apress.prospark
2.
3.   import scala.io.Source
4.
5.   import org.apache.spark.SparkConf
6.   import org.apache.spark.SparkContext
7.
8.   object TranslateApp {
9.     def main(args: Array[String]) {
10.      if (args.length != 4) {
11.        System.err.println(
12.          "Usage: TranslateApp <appname> <book_path> <output_path> <language>")
13.        System.exit(1)
14.      }
15.      val Seq(appName, bookPath, outputPath, lang) = args.toSeq
16.
17.      val dict = getDictionary(lang)
18.
19.      val conf = new SparkConf()
20.        .setAppName(appName)
21.        .setJars(SparkContext.jarOfClass(this.getClass).toSeq)
22.      val sc = new SparkContext(conf)
23.      val book = sc.textFile(bookPath)
24.      val translated = book.map(line => line.split("\\s+").map(word => dict.
         getOrElse(word, word)).mkString(" "))
25.      translated.saveAsTextFile(outputPath)
26.    }
27.
28.    def getDictionary(lang: String): Map[String, String] = {
29.      if (!Set("German", "French", "Italian", "Spanish").contains(lang)) {
30.        System.err.println(
31.          "Unsupported language: %s".format(lang))
32.        System.exit(1)
33.      }
34.      val url = "http://www.june29.com/IDP/files/%s.txt".format(lang)
35.      println("Grabbing dictionary from: %s".format(url))
36.      Source.fromURL(url, "ISO-8859-1").mkString
37.        .split("\\r?\\n")
38.        .filter(line => !line.startsWith("#"))
39.        .map(line => line.split("\\t"))
40.        .map(tkns => (tkns(0).trim, tkns(1).trim)).toMap
41.    }
42.  }
```

Every Spark application needs an accompanying configuration object of type SparkConf. For instance, the application name and the JARs for cluster deployment are provided via this configuration. Typically the location of the master is picked up from the environment, but it can be explicitly provided by setting setMaster() on SparkConf. Chapter 4 discusses more configuration parameters. In this example, a SparkConf object is defined on line 19.

The connection with the Spark cluster is maintained through a `SparkContext` object, which takes as input the `SparkConf` object (line 22). We can make the definition of the driver program more concrete by saying that it is in charge of creating the `SparkContext`. `SparkContext` is also used to create input RDDs. For instance, you create an RDD, with alias `book`, out of a single book on line 23. Note that `textFile(..)` internally uses Hadoop's `TextInputFormat`, which tokenizes each file into lines. `bookPath` can be both a local file system location or an HDFS path.

Each RDD object has a set of transformation functions that return new RDDs after applying an operation. Think of it as invoking a function on a Scala collection. For instance, the `map` transformation on line 24 tokenizes each line into words and then reassembles it into a sentence after single-word translation. This translation is enabled by a dictionary (line 17), which you generate from an online source. Without going into the details of its creation via the `getDictionary()` method (line 28), suffice to say that it provides a mapping between English words and the target language. This `map` transformation is executed on the workers.

Spark applications consist of transformations and actions. Actions are generally output operations that trigger execution—Spark jobs are only submitted for execution when an output action is performed. Put differently, transformations in Spark are lazy and require an action to fire. In the example, on line 25, `saveAsTextFile(..)` is an output action that saves the RDD as a text file. Each action results in the execution of a Spark job. Thus each Spark application is a concert between different entities. Table 2-2 explains the difference between them.[6]

Table 2-2. *Spark Execution Hierarchy*

Entity	Description
Application	One instance of a `SparkContext`
Job	Set of stages executed as a result of an action
Stage	Set of transformations at a shuffle boundary
Task set	Set of tasks from the same stage
Task	Unit of execution in a stage

Let's now see how you can build and execute this application. Save the code from Listing 2-1 in a file with a `.scala` extension, with the following folder structure: `./src/main/scala/FirstApp.scala`.

Build

Similar to Java, a Scala application also needs to be compiled into a JAR for deployment and execution. Staying true to pure Scala, this book uses sbt[7] (Simple Build Tool) as the build and dependency manager for all applications. sbt relies on Ivy for dependencies. You can also use the build manager of your choice, including Maven, if you wish.

Create an `.sbt` file with the following content at the root of the project directory:

```
name := "FirstApp"

version := "1.0"

scalaVersion := "2.10.5"

libraryDependencies += "org.apache.spark" %% "spark-core" % "1.4.0"
```

[6]A task may or may not correspond to a single transformation. This depends on the dependencies in a stage. Refer to Chapter 4 for details on dependencies.
[7]www.scala-sbt.org/.

sbt by default expects your Scala code to reside in `src/main/scala` and your test code to reside in `src/test/scala`. We recommend creating a fat JAR to house all dependencies using the sbt-assembly plugin.[8] To set up sbt-assembly, create a file at `./project/assembly.sbt` and add `addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.11.2")` to it. Creating a fat JAR typically leads to conflicts between binaries and configuration files that share the same relative path. To negate this behavior, sbt-assembly enables you to specify a merge strategy to resolve such conflicts. A reasonable approach is to use the first entry in case of a duplicate, which is what you do here. Add the following to the start of your build definition (`.sbt`) file:

```
import AssemblyKeys._

assemblySettings

mergeStrategy in assembly <<= (mergeStrategy in assembly) { mergeStrategy => {
 case entry => {
   val strategy = mergeStrategy(entry)
   if (strategy == MergeStrategy.deduplicate) MergeStrategy.first
   else strategy
 }
}}
```

To build the application, execute `sbt assembly` at the command line, with your working directory being the directory where the `.sbt` file is located. This generates a JAR at `./target/scala-2.10/FirstApp-assembly-1.0.jar`.

Execution

Executing a Spark application is very similar to executing a standard Scala or Java program. Spark supports a number of execution modes.

Local Execution

In this mode, the entire application is executed in a single process with potentially multiple threads of execution. Use the following command to execute the application in local mode

```
$SPARK_HOME/bin/spark-submit --class org.apress.prospark.TranslateApp --master local[n]
./target/scala-2.10/FirstApp-assembly-1.0.jar <app_name> <book_path> <output_path> <language>
```

where `n` is the number of threads and should be greater than zero.

Standalone Cluster

In standalone cluster mode, the driver program can be executed on the submission machine (as shown in Figure 2-2):

```
$SPARK_HOME/bin/spark-submit --class org.apress.prospark.TranslateApp --master <master_
url> ./target/scala-2.10/FirstApp-assembly-1.0.jar <app_name> <book_path> <output_path>
<language>
```

[8]https://github.com/sbt/sbt-assembly.

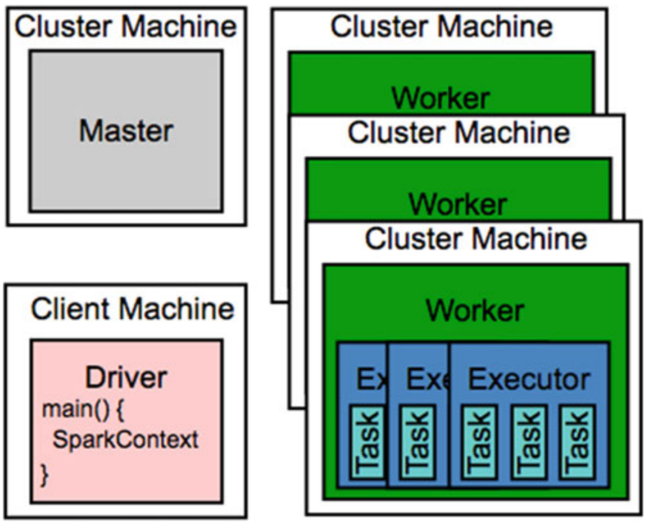


Figure 2-2. *Standalone cluster deployment with the driver running on the client machine*

Alternatively, the driver can be executed on the cluster (see Figure 2-3):

```
$SPARK_HOME/bin/spark-submit -class org.apress.prospark.TranslateApp --master <master_url>
--deploy-mode cluster ./target/scala-2.10/FirstApp-assembly-1.0.jar <app_name> <book_path>
<output_path> <language>
```

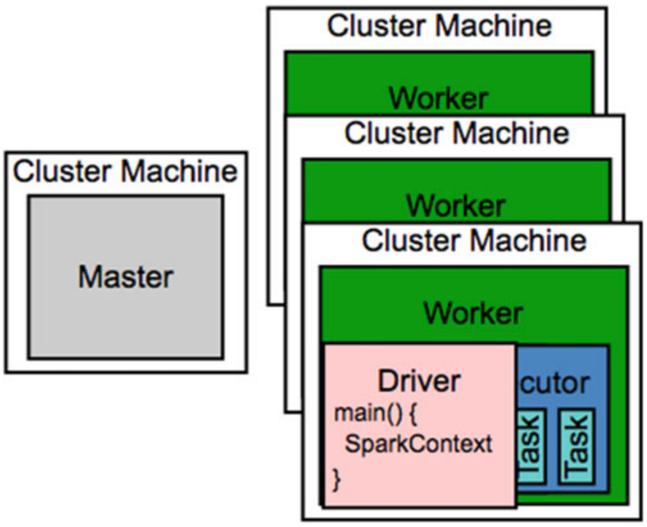


Figure 2-3. *Standalone cluster deployment with the driver running on a cluster*

Note that in both cases, the jobs execute on the worker nodes, and only the execution location of the driver program varies.

YARN

Similar to standalone cluster mode, under YARN,[9] the driver program can be executed either on the client

```
$SPARK_HOME/bin/spark-submit --class org.apress.prospark.TranslateApp --master yarn-client
./target/scala-2.10/FirstApp-assembly-1.0.jar <app_name> <book_path> <output_path> <language>
```

or on the YARN cluster:

```
$SPARK_HOME/bin/spark-submit --class org.apress.prospark.TranslateApp --master yarn-
cluster ./target/scala-2.10/FirstApp-assembly-1.0.jar <app_name> <book_path> <output_path>
<language>
```

Similar to the distributed cluster mode, the jobs execute on the YARN NodeManager nodes in both cases, and only the execution location of the driver program varies.

The rest of the chapter walks through some of the artifacts introduced in the example application to take a deep dive into the inner workings of Spark.

SparkContext

As mentioned before, SparkContext is the main application entry point. It serves many purposes, some of which are described in this section.

Creation of RDDs

SparkContext has utility functions to create RDDs for many data sources out of the box. Table 2-3 lists some of the common ones. Note that these functions also accept an argument specifying an optional number of slices/number of partitions.

Table 2-3. *RDD Creation Methods Exposed by SparkContext*

Signature	Description
`parallelize[T](seq: Seq[T]): RDD[T]`	Converts a Scala collection into an RDD.
`range(start: Long, stop: Long, step: Long): RDD[Long]`	Creates an RDD of Longs from start to stop (exclusive) with an increment of step.
`hadoopFile[K, V](path: String, inputFormatClass: Class[_ <: InputFormat[K, V]], keyClass: Class[K], valueClass: Class[V]): RDD[(K, V)]`	Returns an RDD for a Hadoop file located at path, parameterized by K,V, and using inputFormatClass for reading.
`textFile(path: String): RDD[String]`	Returns an RDD for a Hadoop text file located at path. Under the hood, it invokes hadoopFile() by using TextInputFormat as the inputFormatClass, LongWriteable as the keyClass, and Text as the valueClass. It is important to highlight that the key in this case is the position in the file, whereas the value is a line.

(*continued*)

[9]Spark uses HADOOP_CONF_DIR and YARN_CONF_DIR to access HDFS and talk to the YARN resource manager.

Table 2-3. *(continued)*

Signature	Description
`sequenceFile[K, V](path: String, keyClass: Class[K], valueClass: Class[V]): RDD[(K, V)]`	Returns an RDD for a Hadoop `SequenceFile` located at `path`. Internally, invokes `hadoopFile()` by passing `SequenceFileInputFormat` as the `inputFormatClass`.
`newAPIHadoopFile[K, V, F <: NewInputFormat[K, V]]( path: String, inputFormatClass: Class[F], keyClass: Class[K], valueClass: Class[V]): RDD[(K, V)`	Returns an RDD for a Hadoop file located at `path`, parameterized by `K`, `V`, and `F`, based on the new Hadoop API introduced in version 0.21.[10]
`wholeTextFiles(path: String): RDD[(String, String)]`	Returns an RDD for whole Hadoop files located at `path`. Under the hood, uses `String` as both the key and the value and `WholeTextFileInputFormat` as the `InputFormatClass`. Note that the key is the file path, whereas the value is the entire content of the file(s). Use this method if the files are small. For larger files, use `textFile()`.
`union[T](rdds: Seq[RDD[T]]): RDD[T]`	Returns an RDD that is the union of all input RDDs of the same type.

Note that each function returns an RDD with an associated object type. For instance, `parallelize()` returns a `ParallelCollectionRDD` (which knows how to serialize and slice up a Scala collection), and `textFile()` returns a `HadoopRDD` (which knows how to read data from HDFS and to create partitions). More on RDDs later.

Handling Dependencies

Due to its distributed nature, Spark tasks are parallelized across many worker nodes. Typically, the data (RDDs) and the code (such as closures) are shipped out by Spark; but in certain cases, the task code may require access to an external file or a Java library. The `SparkContext` instance can also be used to handle these external dependencies (see Table 2-4).

Table 2-4. *Dependency-Handling Features of `SparkContext`*

Signature	Description
`addFile(path: String): Unit`	Downloads the file present at `path` to every node. This file can be accessed via `SparkFiles.get(filename: String)`. In addition to being a local or HDFS location, `path` can also point to a remote HTTP/FTP location.
`addJar(path: String): Unit`	Adds the file at `path` as a JAR dependency for all tasks executed in this `SparkContext`.

[10]Tom White, "What's New in Apache Hadoop 0.21," Cloudera, August 26, 2010, `http://blog.cloudera.com/blog/2010/08/what%E2%80%99s-new-in-apache-hadoop-0-21/`.

Creating Shared Variables

Spark transformations manipulate independent copies of data on worker machines, and thus there is no shared state between them. In certain cases, though, some state may need to be shared across workers and the driver—for instance, if you need to calculate a global value. `SparkContext` provides two types of shared variables:

- *Broadcast variables:* As the name suggests, broadcast variables are read-only copies of data broadcast by the driver program to worker tasks—for instance, to share a copy of large variables. Spark by default ships out the data required by each task in a stage. This data is serialized and deserialized before the execution of each task. On the other hand, the data in a broadcast variable is transported via efficient P2P communication and is cached in deserialized form. Therefore, broadcast variables are useful only when they are required across multiple stages in a job. Table 2-5 outlines the creation and use of broadcast variables.

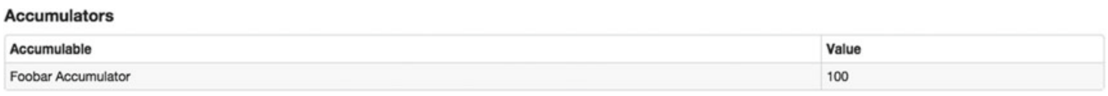
Table 2-5. *Broadcast Variable Creation and Use*

Signature	Description
`broadcast(v: T): Broadcast[T]`	Broadcasts `v` to all nodes, and returns a broadcast variable reference. In tasks, the value of this variable can be accessed through the `value` attribute of the reference object. After creating a broadcast variable, do not use the original variable `v` in the workers.

- *Accumulators:* Accumulators are variables that support associative functions such as `increment`. Their prime property is that tasks running on workers can only write to them, and only the driver program can read their value. Thus they are handy for implementing counters or manipulating objects that allow += and/or `add` operations. Table 2-6 showcases out of the box Accumulators in Spark.

Table 2-6. *Out-of-the-Box Accumulators in Spark*

Signature	Description
`accumulator[T](initialValue: T): Accumulator[T]`	Returns an accumulator of type `T`. Tasks can then directly reference it and perform `add` and += operations. Only the driver can read its value by accessing its `value` attribute.
`accumulator[T](initialValue: T, name: String): Accumulator[T]`	The additional `name` argument enables this accumulator to be viewed in the UI. Note that the accumulator is displayed on the UI page for each stage that updates its value. For instance, Figure 2-4 shows a named accumulator: Foobar Accumulator.
`accumulableCollection [R, T] (initialValue: R): Accumulable[R, T]`	Returns an accumulator for a `Collection` of type `R`. Note that `R` should implement += and ++= operations. Standard choices include `mutable.HashSet`, `mutable.ArrayBuffer`, and `mutable.HashMap`.



Accumulators

Accumulable	Value
Foobar Accumulator	100

Figure 2-4. *Spark UI screenshot of a named accumulator*

Job execution

Transparent to you, `SparkContext` is also in charge of submitting jobs to the scheduler. These jobs are submitted every time an RDD action is invoked. A job is broken down into stages, which are then broken down into tasks. These tasks are subsequently distributed across the various workers. You will learn more about scheduling in Chapter 4.

RDD

Resilient distributed datasets (RDDs) lie at the very core of Spark. Almost all data in a Spark application resides in them. Figure 2-5 gives a high-level view of a typical Spark workflow that revolves around the concept of RDDs. As the figure shows, data from an external source or multiple sources is first ingested and converted into an RDD that is then *transformed* into potentially a series of other RDDs before being written to an external sink or multiple sinks.

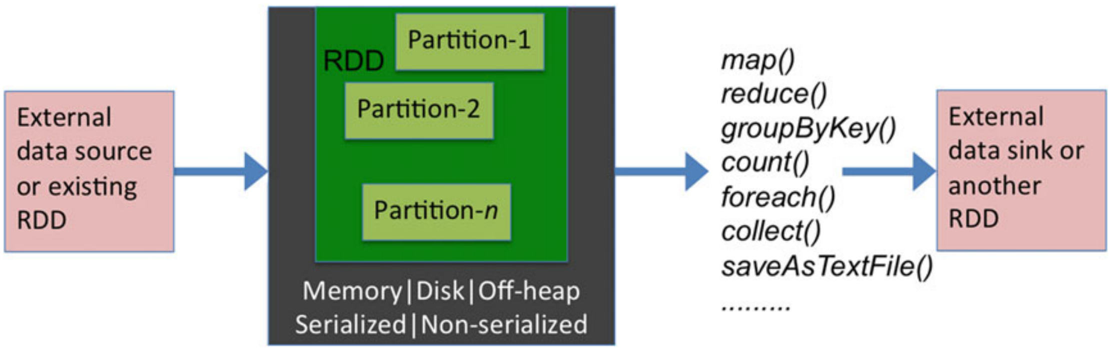


Figure 2-5. *RDDs in a nutshell*

An RDD in essence is an envelope around data partitions. The persistence level of each RDD is configurable; the default behavior is regeneration under failure. Keeping lineage information—parent RDDs that it depends on—enables RDD regeneration. An RDD is a first-class citizen in the Spark order of things: applications make progress by transforming or actioning RDDs. The base RDD class exposes simple transformations (`map`, `filter`, and so on). Derived classes build on this by extending and implementing three key methods: `compute()`, `getPartitions()`, and `getDependencies()`. For instance, `UnionRDD`, which takes the union of multiple RDDs, simply returns the partitions and dependencies of the unionized RDDs in its `rdd.partitions` (public version of `getPartitions()`) and `rdd.dependencies` (sugared version of `getDependencies()`) methods, respectively. In addition, its `compute()` method invokes the respective `compute()` methods of its constituent RDDs.

Similarly, certain transformations that apply to only specific RDD types are enabled with Scala implicit conversions. A good example of such functionality are methods that only apply to key-value pair RDDs, such as `groupByKey()` and `reduceByKey()`, which reside in the `PairRDDFunctions` class. Table 2-7 lists widely used RDDs.

Table 2-7. *Examples of Specialized RDDs*

RDD Type	Description
`CartesianRDD`	Obtained as a result of calculating the Cartesian product of two RDDs
`HadoopRDD`	Represents data stored in any Hadoop-compatible store: local FS, HDFS, S3, and HBase
`JdbcRDD`	Contains results of a SQL query execution via a JDBC connection
`NewHadoopRDD`	Same as `HadoopRDD` but uses the new Hadoop API
`ParallelCollectionRDD`	Contains a Scala `Collections` object
`PipedRDD`	Used to pipe the contents of an RDD to an external command, such as a bash command or a Python script
`UnionRDD`	Wrapper around multiple RDDs to treat them as a single RDD

As mentioned, RDDs can also be transformed via implicit conversions. Table 2-8 lists some of these conversion classes.

Table 2-8. *RDD Conversion Functions*

RDD Conversion Class	Description
`DoubleRDDFunctions`	Functions that can be applied to RDDs of `Doubles`. These functions include `mean()`, `variance()`, `stdev()`, and `histogram()`.
`OrderedRDDFunctions`	Ordering functions that apply to key-value pair RDDs. Functions include `sortByKey()`.
`PairRDDFunctions`	Functions applicable to key-value pair RDDs. Functions include `combineByKey()`, `aggregateByKey()`, `reduceByKey()`, and `groupByKey()`.
`SequenceFileRDDFunctions`	Functions to convert key-value pair RDDs to Hadoop `SequenceFiles`. For example, `saveAsSequenceFile()`.

Persistence

The persistence level of RDDs explores different points in the space between CPU, IO, and memory cost. This value can be set by making a call to the `persist()` method exposed by each RDD. The `cache()` method defaults to a persistence level of `MEMORY_ONLY.` Persistence is handled by `BlockManager`, which internally maintains a memory store and a disk store. In addition, RDDs can also be checkpointed using the `checkpoint()` function. Unlike caching, checkpointing directly saves the RDD to HDFS and does not keep its lineage information.

Table 2-9 explains the features of each persistence level exposed by the `StorageLevel` class.

Table 2-9. *RDD Storage Levels*

Storage Level	Description
`NONE`	Default persistence level. Entails regeneration of the RDD on failure.
`DISK_ONLY`	The RDD is persisted on disk.
`DISK_ONLY_2`	Same as previous, but on two machines.
`MEMORY_ONLY`	The RDD is persisted in memory in unserialized form. Partitions that do not fit in memory fall back on recomputation.
`MEMORY_ONLY_2`	Same as previous, but on two machines.
`MEMORY_ONLY_SER`	The RDD is persisted in memory in serialized form.
`MEMORY_ONLY_SER_2`	Same as previous, but on two machines.
`MEMORY_AND_DISK`	The RDD is first persisted in memory in unserialized format. Partitions that do not fit are spilled to disk.
`MEMORY_AND_DISK_2`	Same as previous, but on two machines.
`MEMORY_AND_DISK_SER`	Similar to `MEMORY_AND_DISK`, but the partitions are kept serialized.
`MEMORY_AND_DISK_SER_2`	Same as previous, but on two machines.
`OFF_HEAP`	RDDs are offloaded to Tachyon (now called Alluxio),[11] which is an in-memory data store. This greatly reduces the memory footprint and GC overhead of the worker JVMs. Note that you need to have a running Tachyon deployment for this to work (discussed in detail in Chapter 7).

Persisted RDDs can also be unpersisted by calling their `unpersist()` method. By default, this is a blocking call, but it can be made asynchronous by passing `false` to the method.

Transformations

As discussed earlier, applications make progress by transforming RDDs. These transforms can be grouped into four broad categories:

- *Mappings* transform the input RDD to an output one by calling a user-defined function. In most cases, the output RDD is smaller or equivalent in size—in terms of number of data points—to the input one (see Table 2-10).
- *Variation* operators either increase or decrease the number of partitions in an RDD or perform fission or fusion of RDDs (see Table 2-11).
- *Key-value* (including aggregation) transformations apply only to RDDs with key-value pairs. Invoking a `map` function that returns a 2-tuple typically creates these key-value pairs (see Table 2-12).
- *Miscellaneous* operators perform various tasks such as taking a sample of elements in an RDD (see Table 2-13).

[11]`http://tachyon-project.org/`.

Table 2-10. *Mapping Transformations*

Transformation (Mapping)	Description
`map[U](function): RDD[U]`	Standard `map` function from functional programming that invokes the user-provided `function` for each data point in the input RDD to create an output RDD of the same length.
`flatMap[U](function): RDD[U]`	Same as `map()`, but flattens the RDD, so the size of the output may be different than that of the input RDD.
`mapPartitions[U](function): RDD[U]`	Invokes the passed `function` for each partition in the RDD, as opposed to a standard `map()` that is invoked for each data point. The provided function should be able to take as input an iterator over the data points in a partition. This is more efficient than `map()` because it reduces the number of function calls. (In Chapter 6, you see how mapPartitions() can make writing to external storage more efficient.)
`mapPartitionsWithIndex[U](function): RDD[U]`	Similar to `mapPartitions()`, but alongside the iterator, the function should also accept a partition index. This is useful if the logic requires keeping track of the index of the partition being processed.
`filter[U](function): RDD[U]`	Filters out values that fit a particular condition. To this end, the provided `function` needs to return a Boolean value corresponding to the given condition.

Table 2-11. *Variation Transformations*

Transformation (Variation)	Description
`coalesce(numPartitions: Int, shuffle: Boolean): RDD[T]`	Decreases the number of partitions in an RDD. Under the hood, it logically merges partitions by treating them as a conglomerate. This behavior is ensured by setting `shuffle` to `false`—the default behavior. Setting `shuffle` to `true` forces a shuffle phase that uses hash partitioning to physically repartition the RDD. This also means the number of partitions can be both decreased and increased via this mechanism.
`repartition(numPartitions: Int): RDD[T]`	Increases or decreases the number of partitions in the RDD by always forcing a shuffle. Under the hood, it invokes `coalesce()` by passing `shuffle` as `true`.
`union(other: RDD[T]): RDD[T]`	Returns another RDD that is the union of this RDD with the passed RDD. The concrete implementation of the returned RDD is provided by `UnionRDD`.
`intersection(other: RDD[T]): RDD[T]`	Takes the intersection of the calling RDD with the passed one.

Table 2-12. *Key-Value Pair Transformations*

Transformation (Key-Value)	Conversion Class	Description
`repartitionAndSortWithinPartitions(partitioner: Partitioner): RDD[(K, V)]`	`OrderedRDDFunctions`	Increases or decreases the number of partitions and the key space of the RDD based on the supplied partitioner, and also sorts the keys in each partition. Note that this transform only applies to RDDs where the key is sortable, and hence it resides in `OrderedRDDFunctions`.
`groupByKey(): RDD[(K, Iterable[V])]`	`PairRDDFunctions`	Groups the values by key. The returned RDD contains a key-value pair for each group, where the value is an iterator over the grouped values.
`sortByKey(): RDD[(K, V)]`	`OrderedRDDFunctions`	Sorts the RDD by key. The key should extend the `Ordering` Scala trait. By default, it sorts the keys in ascending order by keeping the number of partitions intact. Both of these values can be tweaked by passing arguments, `ascending`: `Boolean` = `false` and `numPartitions`: Int = `#Partitions`. Note that if your logic requires changing the number of partitions and also sorting the keys, it is more efficient to use `repartitionAndSortWithinPartitions()`, which pushes the sorting into the shuffle infrastructure.
`reduceByKey(function: (V, V) ⇒ V): RDD[(K, V)]`	`PairRDDFunctions`	Aggregates the values for each key using the provided `function`. The function is invoked for each pair of values (aggregated ones as well) until a final aggregated value is obtained. Note that `reduceByKey()` can only be used for associative and commutative functions. This gives the added benefit of `map`-side combiners a la MapReduce. As a result, it is more efficient to use `reduceByKey()` as opposed to first `groupByKey()` and then `reduce()` (explained in the "Actions" section). Furthermore, `reduceByKey()` uses the first element of the RDD as the initial value for its internal accumulator. To explicitly supply an initial value, use `foldByKey()`. It is also important to highlight that the return type of the aggregation function needs to be the same as the type of the values under aggregation. If your logic requires this type to be different, use `combineByKey()`.
`foldByKey(intialValue: V)(func: (V, V) ⇒ V): RDD[(K, V)]`	`PairRDDFunctions`	Similar to `reduceByKey()`, but uses `initialValue` to initialize the internal accumulator.

(*continued*)

Table 2-12. (*continued*)

Transformation (Key-Value)	Conversion Class	Description
`combineByKey[C](createCombiner: (V) ⇒ C, mergeValue: (C, V) ⇒ C, mergeCombiners: (C, C) ⇒ C): RDD[(K, C)]`	`PairRDDFunctions`	Aggregates the values of the RDD, but allows the return type of the aggregation function to be different than the input type. Requires three functions as arguments: ● `createCombiner` to initialize the accumulator ● `mergeValue` to merge pairs of values ● `mergeCombiners` to merge two accumulators
`aggregateByKey[U](initialValue: U)(seqOp: (U, V) ⇒ U, combOp: (U, U) ⇒ U): RDD[(K, U)]`	`PairRDDFunctions`	Similar to `combineByKey()`, but instead of using a function to initialize the accumulator, accepts an explicit value, `initialValue,` for the initialization.
`join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]`	`PairRDDFunctions`	Performs an inner join on the RDD with another one. Variants for left and right outer joins also exist: `leftOuterJoin()` and `rightOuterJoin()`.
`cogroup[W](other: RDD[(K, W)]): RDD[(K, (Iterable[V], Iterable[W]))]`	`PairRDDFunctions`	Groups this RDD with another one by pairing values that share a key.
`subtractByKey[W](other: RDD[(K, W)]): RDD[(K, V)]`	`PairRDDFunctions`	Returns an RDD containing keys that exist in the calling RDD but not the passed one.
`mapValues[U](function): RDD[(K, U)]`	`PairRDDFunctions`	Invokes `function` for each value without touching the keys or the partitioning.
`flatMapValues[U](function): RDD[(K, U)]`	`PairRDDFunctions`	Similar to `mapValues()`, but values are flattened.

Note that `groupByKey()`, `reduceByKey()`, `foldByKey()`, `combineByKey()`, `aggregateByKey()`, `join()`, `cogroup()`, and `subtractByKey()` by default use the default parallelism level and the default partitioner. You can invoke their overloaded variants to change both.

Table 2-13. *Miscellaneous Transformations*

Transformation (Misc.)	Description
`cartesian[U](other: RDD[U]): RDD[(T, U)]`	Takes the Cartesian product of two RDDs and returns the result in the form of a `CartesianRDD`.
`distinct(): RDD[T]`	Returns an RDD containing distinct elements from the calling RDD.
`pipe(command: String): RDD[String]`	Executes an external command, and pipes the elements of the RDD through it. Returns a `PipedRDD` with the results. An overloaded version also accepts an environment key-value map.
`sample(withReplacement: Boolean, fraction: Double): RDD[T]`	Samples a random subset of the elements of the RDD. The `withReplacement` argument decides whether elements can be sampled multiple times, and `fraction` decides the expected total size of the resulting RDD as a fraction of this RDD's size.

Refer to Chapter 3 for examples of using these transforms in the context of DStreams.

Actions

Actions kick off the execution of Spark jobs. Each action is either a data-egress point to an external data store or an ingress point into the driver program. Under the hood, each action invokes the `SparkContext` instance to schedule its execution. Similar to transformations, these actions vary depending on the RDD type. Table 2-14 lists actions that write to an external data source, and Table 2-15 contains those that cause data to be materialized into the driver program.

Table 2-14. *Actions that Trigger Data Materialization to External Storage*

Action	Description
`foreach(function): Unit`	A generic function for invoking operations with side effects. For each element in the RDD, it invokes the passed `function`. This is generally used for manipulating accumulators or writing to external stores.
`foreachPartition(function): Unit`	Similar to `foreach()`, but instead of invoking `function` for each element, it calls it for each partition. The function should be able to accept an iterator. This is more efficient than `foreach()` because it reduces the number of function calls (just like `mapPartitions()`).
`saveAsHadoopFile[K, V](path: String): Unit`	Saves the key-value RDD to a Hadoop-compatible file system at location `path`.
`saveAsNewAPIHadoopFile[K, V](path: String): Unit`	Similar to `saveAsHadoopFile()`, but uses the new Hadoop API.
`saveAsTextFile(path: String): Unit`	Saves the RDD as a text file on a Hadoop-compatible file system at location `path`. Invokes `saveAsHadoopFile()` under the hood by using `NullWritable` as the key.
`saveAsSequenceFile(path: String): Unit`	Saves the RDD as a Hadoop sequence file at location `path`. Makes a call to `saveAsHadoopFile()` under the hood.
`saveAsObjectFile(path: String): Unit`	Saves the elements in the RDD as serializable objects at location `path`. Invokes `saveAsSequenceFile()` under the hood by employing `NullWritable` as the key and `BytesWritable` as the value.
`saveAsHadoopDataset(conf: JobConf): Unit`	Saves the output to a Hadoop-compatible data store such as HBase. Uses `conf` to decide the output format and other settings.

Table 2-15. *Actions that Materialize Data into the Driver Program*

Action	Description
`reduce(function: (T, T) ⇒ T): T`	Invokes the user-defined `function` for every pair of values (and aggregated values) in the RDD. Returns the results to the driver program. Note that `reduce()` is an action, whereas `reduceByKey()` is not, because the latter may entail a larger number of partitions and a large key space—enough to overwhelm the driver.
`collect(): Array[T]`	Returns all the contents of the RDD to the driver program in the form of an array. An overloaded variant also accepts a function to filter the contents of the fetched array.
`collectAsMap(): Map[K, V]`	Collects the data in `PairRDD` as a `Map`.
`count(): Long`	Counts the number of elements in the calling RDD.
`countByKey(): Map[K, Long]`	`PairRDD` variant of `count()`.
`countApproxDistinct(relativeSD: Double): Long`	Counts the number of distinct elements in the RDD by using the HyperLogLog algorithm. `relativeSD` decides the relative accuracy, with larger values requiring less space.
`countApproxDistinctByKey(relative SD: Double = 0.05): RDD[(K, Long)]`	The key-value variant of `countApproxDistinct()`.
`take(n: Int): Array[T]`	Returns the first `n` elements from this RDD in the form of an array.
`first(): T`	Returns the first element of the RDD. Simply invokes `take(1)` behind the scenes.
`takeOrdered(k: Int): Array[T]`	Returns the first `k` elements from this RDD while preserving the natural ordering of the elements. Conceptually, a bottom-`k` query.
`top(k: Int): Array[T]`	Returns the top `k` elements from this RDD.
`takeSample(withReplacement: Boolean, num: Int): Array[T]`	Action version of `sample()`.
`lookup(key: K): Seq[V]`	Returns a sequence of values associated with key `k`.

It should be abundantly clear by now that Spark is a functional Big Data processing framework wherein RDDs are immutable collections of data that are transformed to achieve application logic. Applications are side-effect-free for the most part: the number of actions is typically smaller than the number of transformations. Transformations are lazy and only triggered when a downstream action is invoked.

Summary

RDDs constitute the cornerstone of Spark. They are the unit of fault-tolerance, data ingestion and storage, and transformation. These data envelopes in concert with transformations, actions, and a small number of other primitives can be used to stitch together almost any embarrassingly parallel application. Spark maximizes productivity by enabling you to focus on analytics while the underlying engine takes care of all the distributed magic. Starting from the basics of application development, this chapter has explored the various power tools that Spark puts at your disposal.

You now have enough background to jump into Spark Streaming and transition from a largely static batch-processing world to one where everything is in motion and time is of the essence.

CHAPTER 3

DStreams: Real-Time RDDs

Always in motion is the future.

—Grand Jedi Master Yoda, *Star Wars Episode V: The Empire Strikes Back*

According to IBM, 60% of all sensory information loses value in a few milliseconds if it is not acted on.[1] Bearing in mind that the Big Data and analytics market has reached $125 billion and a large chunk of this will be attributed to IoT in the future,[2] the inability to tap real-time information will result in a loss of billions of dollars. Examples of some of these applications include a telco working out how many of its users have used Whatsapp in the last 30 minutes, a retailer keeping track of the number of people who have said positive things about its products today on social media, or a law enforcement agency looking for a suspect using data from traffic CCTV. This is the primary reason stream-processing systems like Spark Streaming will define the future of real-time analytics. There is also a growing need to analyze both data at rest and data in motion to drive applications, which makes systems like Spark—which can do both—all the more attractive and powerful. It's a system for all Big Data seasons.

In this chapter, you see how Spark uses micro-batching to achieve real-time computation. In the process, you learn how Spark Streaming not only keeps the familiar Spark API intact but also, under the hood, uses RDDs for storage as well as fault-tolerance. This enables Spark practitioners to jump into the streaming world from the outset. With that in mind, let's get right to it.

From Continuous to Discretized Streams

Traditional stream processing consists of routing records from data sources to sinks with intermediate processing operators that enforce user logic. A good real-world example of stream processing is the kitchen of a typical restaurant, which consists of waiters, dishwashers, prep cooks, line cooks, sous chefs, and a main chef who collectively implement culinary applications. Waiters take orders and bring them to the main chef, who instructs the prep cooks on the initial preparation. This may entail chopping ingredients, making dough, preparing icing, and so on. Their product is passed on to a line cook who is dedicated to a particular kitchen station: the fryer, the grill, and so on. Once the line cook is done, the dish goes to either the sous chef or the main chef for the finishing touches, which may be as simple as garnishing it with herbs. The waiter then takes the fully prepared dish to the customer. After the customer has finished their meal,

[1]Pietro Lee, "Data Has a Gravity and Is Attracting Decisions," May 26, 2015, www.slideshare.net/pieroleo/data-has-a-gravity-and-is-attracting-decisions.
[2]Gil Press, "6 Predictions for the $125 Billion Big Data Analytics Market in 2015," *Forbes*, December 11, 2014, www.forbes.com/sites/gilpress/2014/12/11/6-predictions-for-the-125-billion-big-data-analytics-market-in-2015/.

the physical plate/bowl ends up in front of the dishwasher, who does the final cleaning. Note that at any instance in time, a number of dishes may be in various stages of the cooking pipeline, because customers arrive and order at different times. In addition, to ensure this parallelism, a kitchen employs many prep cooks, line cooks, and so on.

In stream-processing speak, the waiter who brings in the order is the data source. The order itself is the record to be processed. The various cooks and chefs constitute processing operators, whereas the customer is the application driver. There are multiple orders, which follow parallel processing streams. Such pipelines have to deal with two major issues: random faults and stragglers. In very large kitchens with scores of cooks and orders, the probability of an accident is very high. Furthermore, one or two cooks may be slower than everyone else, leading to the buildup of back pressure. Conventional systems deal with faults by either maintaining a redundant replica of each processing stream or buffering records upstream and replaying failed ones. Both approaches double the amount of data storage across the system. In addition, neither method has an efficient mechanism to deal with stragglers. Imagine restaurants having to cook two dishes instead of one for each order.

An alternative way to deal with these issues is to batch different orders together and have everyone in the kitchen collectively perform one function in tandem. For instance, everyone would first perform the work of the prep cooks, followed by line cooks, and so on. Because the entire staff would be focusing on one task at a time, their throughput would be much higher at the expense of latency. In case of an accident, such as a cook burning a key ingredient, the other cooks would help to bring that cook up to speed by reperforming all the steps from the beginning in addition to their current work. Note that this can only be done for ingredients and intermediate steps that apply to a single batch of dishes. Moreover, the same mechanism is used to tackle stragglers: if one or more cooks are holding up the batch, the rest of the staff can pitch in to bring them up to speed.

This is exactly how Spark Streaming operates: by treating stream processing as discrete-time micro-batch processing, called *discretized streams* or `DStreams`. Every time there is a failure or a straggler, other nodes use the lineage of the lost partitions to regenerate them in addition to their standard processing tasks. Certain operations need to maintain state across batches; their state is checkpointed every now and then to negate infinite state regeneration.

First Streaming Application

Similar to the previous chapter, let's jump in to the deep end of the pool by starting with a running book-translation example. If you recall, you were able to process books that had already been materialized in the previous chapter. You can update that application using the Spark Streaming API to make it real-time; see Listing 3-1. Notice that the changes to the code are mostly cosmetic. In fact, there is almost a one-to-one mapping between regular Spark primitives and those exposed by Spark Streaming. This is primarily because Spark Streaming uses RDD primitives behind the scenes. A 30,000-foot view of the application is that it can be used to translate books from English to any other language as soon as they are written by the authors.

Listing 3-1. Streaming Version of the Translation Application

```
1.   package org.apress.prospark
2.
3.   import scala.io.Source
4.   import org.apache.spark.SparkConf
5.   import org.apache.spark.SparkContext
6.   import org.apache.spark.streaming.Seconds
7.   import org.apache.spark.streaming.StreamingContext
8.   import org.apache.hadoop.io.LongWritable
9.   import org.apache.hadoop.fs.Path
10.  import org.apache.hadoop.mapreduce.lib.input.TextInputFormat
```

```
11.  import org.apache.hadoop.io.Text
12.
13.  object StreamingTranslateApp {
14.    def main(args: Array[String]) {
15.      if (args.length != 4) {
16.        System.err.println(
17.          "Usage: StreamingTranslateApp <appname> <book_path> <output_path> <language>")

18.        System.exit(1)
19.      }
20.      val Seq(appName, bookPath, outputPath, lang) = args.toSeq
21.
22.      val dict = getDictionary(lang)
23.
24.      val conf = new SparkConf()
25.        .setAppName(appName)
26.        .setJars(SparkContext.jarOfClass(this.getClass).toSeq)
27.      val ssc = new StreamingContext(conf, Seconds(1))
28.
29.      val book = ssc.textFileStream(bookPath)
30.      val translated = book.map(line => line.split("\\s+").map(word =>
         dict.getOrElse(word, word)).mkString(" "))
31.      translated.saveAsTextFiles(outputPath)
32.
33.      ssc.start()
34.      ssc.awaitTermination()
35.    }
36.
37.    def getDictionary(lang: String): Map[String, String] = {
38.      if (!Set("German", "French", "Italian", "Spanish").contains(lang)) {
39.        System.err.println(
40.          "Unsupported language: %s".format(lang))
41.        System.exit(1)
42.      }
43.      val url = "http://www.june29.com/IDP/files/%s.txt".format(lang)
44.      println("Grabbing dictionary from: %s".format(url))
45.      Source.fromURL(url, "ISO-8859-1").mkString
46.        .split("\\r?\\n")
47.        .filter(line => !line.startsWith("#"))
48.        .map(line => line.split("\\t"))
49.        .map(tkns => (tkns(0).trim, tkns(1).trim)).toMap
50.    }
51.
52.  }
```

The SparkConf object (line 24) remains unchanged. Your friend SparkContext, on the other hand, has been replaced with StreamingContext, which, as the name suggests, enables streaming applications. Along with SparkConf, it takes a batch size, which dictates the time interval at which the application is invoked for each micro-batch of the input data. In this example, the batch size is 1 second (line 27).

`StreamingContext` is created by the driver program and maintains the connection with the Spark subsystem. It is also used to create `DStreams` from various input sources. `textFileStream()`[3] (line 29) uses `StreamingContext` to create a `DStream` for a book in a directory (`bookPath`) on a file system. `textFileStream` uses `TextInputStream` (analogous to `textFile()`) under the hood to tokenize the book into lines. You then translate each line into the required language using the same translation logic as the batch example from the previous chapter. Post-translation, the translated book is written back to the file system (line 31).

Unlike batch applications, which have a clear life cycle—they finish execution as soon as they have processed their input files—streaming execution has no marked start and stop. Both need to be explicitly sign-posted. Table 3-1 lists methods that orchestrate the execution of Spark Streaming applications. Line 33 (`start()`) in the sample application kicks off the execution, and line 34 (`awaitTermination()`) waits for completion.

Table 3-1. *Methods That Dictate the Lifecycle of a Spark Streaming Application*

Method	Description
`start()`	Starts the execution of the application.
`awaitTermination()`	Waits for the termination of the application. A variant, `awaitTerminationOrTimeout(long)`, accepts a timeout value in milliseconds.
`stop()`	Forces the application to stop execution. An overloaded version of `stop()` can also be used to wait for all data to be processed before stopping. In addition, `stop()` takes a Boolean as input to control whether `SparkContext` should be stopped. This is set to `true` by default.

Once the application has fully processed the file, you have the translated version. From a result perspective, the outcome is identical to that of the example in Chapter 2. The advantage of this approach becomes apparent when you begin thinking of incremental input and output. In strictly batch mode, the entire book needs to be present before it can be processed. In contrast, in this micro-batch processing approach, books can be incrementally added to a data source (the file system, in this case), and they are processed almost instantaneously. Let's build and execute the application now.

Build and Execution

The Spark Streaming codebase is modular, which means to use a certain project, you only have to add it as a separate dependency rather than include a bloated JAR. For instance, you only need to add the following to the `.sbt` configuration file from the previous chapter to pull in Spark Streaming as a dependency:

```
libraryDependencies += "org.apache.spark" %% "spark-streaming" % "1.4.0"
```

The rest of the build and execution process remains the same.

Time to dissect `StreamingContext`.

StreamingContext

Similar to `SparkContext`, `StreamingContext` is the main entry point for real-time applications. It also serves as the umbilical cord of the driver program with the Spark engine. `SparkContext` and `StreamingContext` are not mutually exclusive. In fact, `StreamingContext` internally has a reference to `SparkContext` (which can be

[3]Note that `textFileStream` only reads files that have been added to the directory within the last batch interval and are not older than `spark.streaming.minRememberDuration` (60 seconds by default).

accessed as StreamingContext#sparkContext). Streaming applications can also use SparkContext during their execution. For instance, dependencies can be handled by using addFile() and addJar(). Another use case is the creation of shared variables, such as broadcast variables and accumulators.

The features of StreamingContext can be grouped on the basis of their functionality. That is what we do next.

Creating DStreams

The unit of application development from the user perspective is a DStream. It is also the data-ingestion point. StreamingContext is used to read data from various real-time sources and convert it to DStreams. Files, sockets, Akka actors, or others—StreamingContext covers them all. Some of these are presented in Table 3-2.

Table 3-2. *Creation of DStreams Enabled by StreamingContext*

Signature	Description
fileStream[K, V, F <: NewInputFormat[K, V]] (dirPath: String, filterFunc: Path => Boolean, newFilesOnly: Boolean)	Creates a DStream to process files located at dirPath parametrized by K,V, and uses F as the input format for reading. filterFunc is used to filter out files: say, those with a certain extension. newFilesOnly dictates whether only new files that have been added to dirPath in the last interval should be considered. In a typical setting, an external process appends files to the directory, which are then ingested by this DStream every batch interval.
fileStream[K, V, F <: NewInputFormat[K, V]] (dirPath: String)	Creates a DStream to process files located at dirPath, parametrized by K,V and using F as the input format for reading. By default, it considers all new files (except those that start with ".") at dirPath in every batch.
textFileStream(dirPath: String)	Returns a DStream for text files added at dirPath. Under the hood, it invokes fileStream() by using LongWritable as the key class, Text as the value class, and TextInputFormat as the input format. Note that this method ignores all old files and only considers files that have been created in the last spark.streaming.minRememberDuration seconds. To change this behavior, you need to use fileStream() and set newFilesOnly to false while parametrizing it with LongWritable, Text, and TextInputFormat.
socketStream[T](hostname: String, port: Int, converter: (InputStream) => Iterator[T], storageLevel: StorageLevel)	Wraps around a socket connected to hostname on port in a DStream. converter is a function to convert a byte stream into an object type expected by the application. storageLevel decides the persistence level of the DStream in the block store.
socketTextStream(hostname: String, port: Int)	Connects to hostname on port, and interprets the data as UTF-8 encoded, newline-delimited text data. The data is stored with level StorageLevel.MEMORY_AND_DISK_SER_2.
rawSocketStream[T](hostname: String, port: Int)	In cases where the data has already been serialized in a format that Spark understands, this method is used to bypass the cost of deserialization. It reads the data from hostname:port and writes it directly to the block store.
actorStream[T] (akkaProperties: Props, name: String)	Creates a DStream for an Akka actor where name is the name of the actor and akkaProperties contains its definition.
queueStream[T](queue: Queue[RDD[T]])	Converts a queue of RDDs to a DStream. By default, the behavior is to dequeue one RDD in each batch interval.

These are not the only sources from which Spark Streaming can ingest data. The Spark Streaming codebase contains connectors for popular real-time sources, such as Kafka, Flume, and Twitter. These are discussed in detail in Chapter 5.

DStream Consolidation

In certain applications, it is useful to merge similar streams into one: for instance, when reading from multiple but similar data sources. `StreamingContext` contains a `union()` method that provides this functionality as shown in Table 3-3.

Table 3-3. *Consolidation Method Provided by* `StreamingContext`

Signature	Description
`union[T](streams: Seq[DStream[T]])`	Takes as input a sequence of `DStreams`, and returns their union

Job Execution

`StreamingContext` also handles scheduling, transparent to the user. Every time an action is invoked on a `DStream`, corresponding jobs are submitted to the Spark scheduler back end via `StreamingContext`.

DStreams

`DStreams` are to Spark Streaming what RDDs are to Spark. Applications make progress by ingesting, transforming, and exporting `DStreams`. Conceptually, a `DStream` converts a potentially endless flow of data into discrete batches of RDDs. Every batch interval, newly generated RDDs are consumed by pushing them through user-defined logic. You have the option of customizing the persistence of each `DStream`. On failure, all nodes take part in regeneration of the lost RDDs. Generally, no micro-batch invariant state is maintained—each batch is stateless.

This model works well for a large class of applications. For instance, the translation application does not need to maintain state across batches because it only converts RDDs of books from one form to another. Similarly, a solution that analyzes a stream of temperature data and generates a flag if it exceeds a threshold only needs to make a decision based on data in the current batch.

Another class of applications needs to maintain state across logic invocations. A minor enhancement of the previous two examples can help to illustrate this point. In addition to translating books, you may also want to keep track of the top-ten words (in terms of word frequency) that have been translated so far. In a similar vein, maintaining a histogram of temperature change over the course of an entire application requires state across all batches. Spark exposes primitives that allow you to achieve such behavior. Stateful `DStreams` have the same fault-tolerance properties as regular ones, with a single catch: they need to be checkpointed regularly to negate endless regeneration.

Figure 3-1 shows a high-level view of Spark Streaming execution. At every batch interval, the source pulls in data and converts it to regular RDDs using the block store. Like any standard RDD, RDDs that constitute a `DStream` are partitioned and have configurable persistence levels. The only (somewhat obvious) catch is that this persistence level is set at the `DStream` level and applies to all the RDDs that make up that `DStream`. From the figure, it is evident that each batch is a self-contained, end-to-end execution step. Applications make progress by repeatedly executing this batch model every time interval. The base class `DStream` has methods and functionality that apply to all variants. Each of its derived classes needs to implement three methods: `slideDuration()`, `dependencies()`, and `compute()`. The first method returns the time interval after which this `DStream` generates an RDD, and the second returns a list of its dependencies. `compute()` is the method that is invoked every batch interval to generate an RDD.

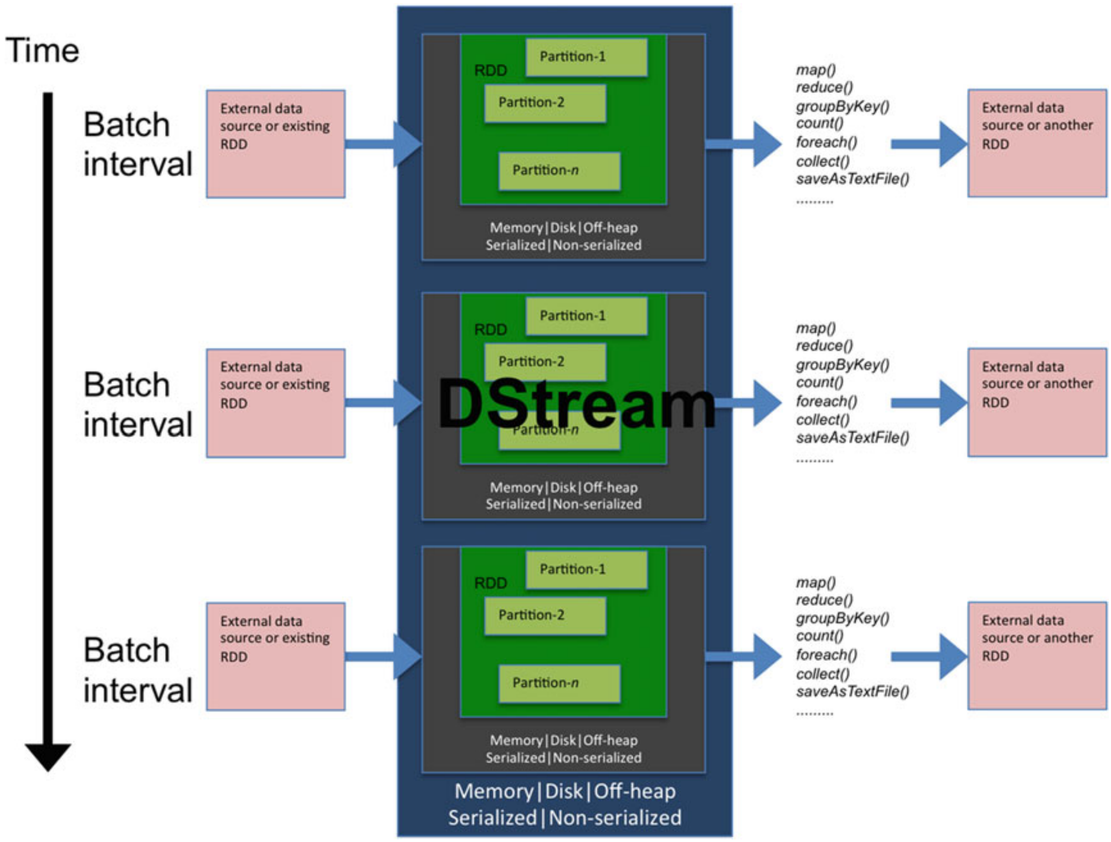

Time
Batch interval
External data source or existing RDD
RDD
Partition-1
Partition-2
Partition-n
Memory|Disk|Off-heap Serialized|Non-serialized
map() reduce() groupByKey() count() foreach() collect() saveAsTextFile() ....
External data source or another RDD
Batch interval
Batch interval
DStream
Memory|Disk|Off-heap Serialized|Non-serialized

***Figure 3-1.** `DStreams`: Standing on the shoulders of RDDs. Time is on the y-axis, and per-batch execution is on the x-axis.*

The operations and transformations applied to `DStreams` are similar if not identical to the ones for RDDs, such as `map()`, `reduce()`, `count()`, and so on. As you see shortly, a few of them are unique to streaming analytics. But first, let's walk through the execution of an application to get a feel for all the events that take place behind the scenes.

The Anatomy of a Spark Streaming Application

Let's dissect the example translation application:

1. Depending on the submission process, the driver program begins executing in either the user's local machine or a cluster node. On execution, the driver program creates a new `StreamingContext` and connects to the Spark subsystem. Note that a `StreamingContext` can also be created from a checkpointed file or an existing `SparkContext`. *Checkpointing* is the process of saving the state of the `StreamingContext` or a `DStream` to secondary storage. In case of `DStreams`, checkpointing ensures that the root of the lineage graph remains within bounds. That is why checkpointing is a mandatory step for stateful `DStream` operations, as you see shortly.

   Internally, `StreamingContext` creates a `SparkContext` object and a `JobScheduler` object. The latter is used to ship streaming jobs to Spark.

2. The rest of the lines before `start()` set up the computation but do not actually perform the execution. Let's wait on those lines and first examine `start()`. Behind the scenes, `start()` kicks off the execution of `JobScheduler`, which in turn starts `JobGenerator.` `JobGenerator` is in charge of creating jobs from `DStreams`, RDD checkpointing, and metadata checkpointing every batch interval. `awaitTermination()` uses a condition variable-like mechanism to block until either the application explicitly invokes `stop()` or the application is terminated.

3. The first actual processing line to be executed is the creation of an input `DStream` using `textFileStream()`, which in turn invokes `fileStream().` `fileStream()` returns a `FileInputDStream` object.

4. `FileInputDStream` internally monitors the specified directory on the file system, and every batch interval picks up new files that have become visible. Each one of these files is turned into an RDD. `FileInputDStream` in its `compute()` methods returns a `UnionRDD` of all these files.

5. Invoking `map()` on the `FileInputDStream` results in the creation of a `MappedDStream`, which simply invokes the `map` function on the underlying RDD.

6. Once the `map` function has finished execution, `saveAsTextFiles()` results in the invocation of `saveAsTextFile()` for each RDD in the `DStream`.

7. Steps 4 to 6 are repeated (potentially) forever.

Figure 3-2 shows the execution of a single batch. Each blue box represents a single `DStream` with a single RDD (because you only analyzed a single book). There are four `DStreams`. The first contains a `UnionRDD` for the input data. `textFileStream()` internally performs a map transformation to cull the key from the key-value pair of line number and line content. That accounts for the first `MapPartitionsRDD`. It is worth highlighting that behind the scenes, an operation on a `DStream` invokes the same operation on its underlying RDDs. For example, a `map` on a `DStream` applies the `map` to each RDD in it. The second `map` operation is from line 30 of Listing 3-1. The last `MapPartitionsRDD` is for the output operation on line 31 (internally, `saveAsTextFiles` makes a call to `saveAsTextFile` on each RDD, which in turn calls a `save` function for each partition in that RDD using a `map`).

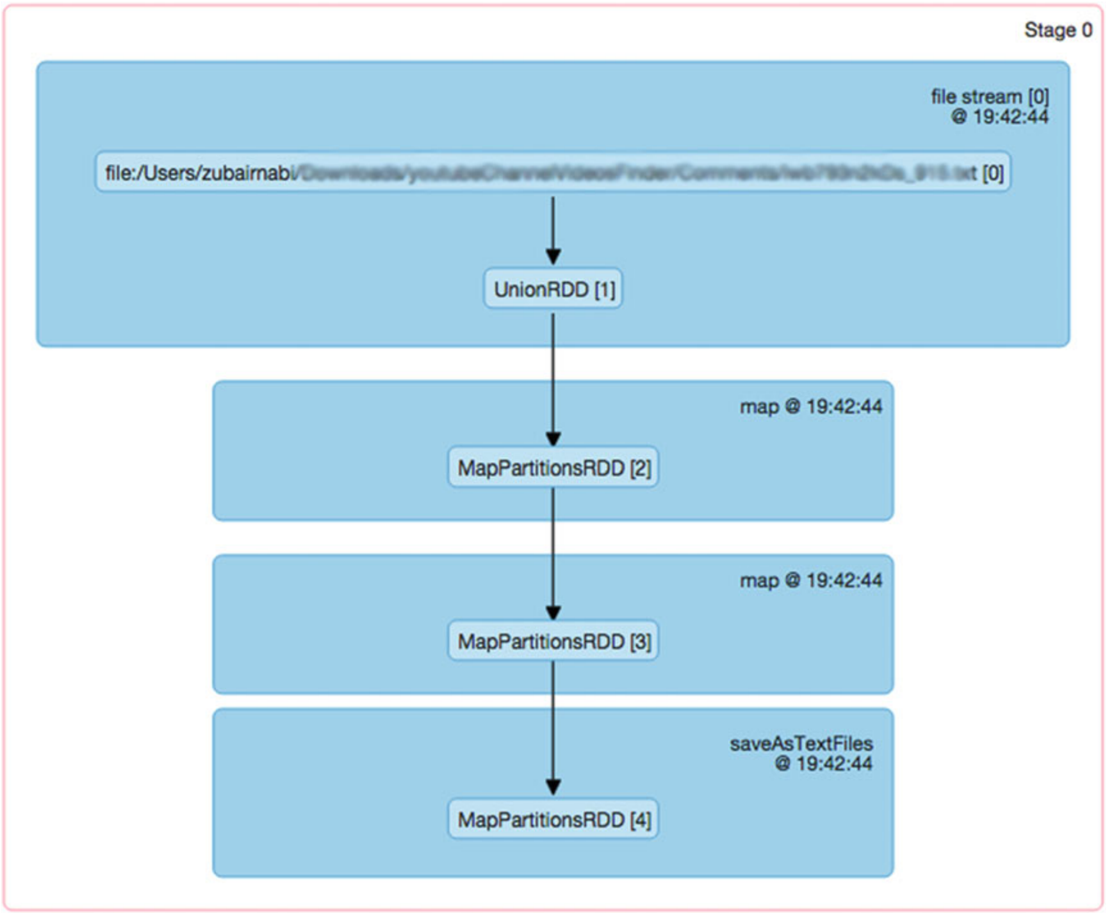
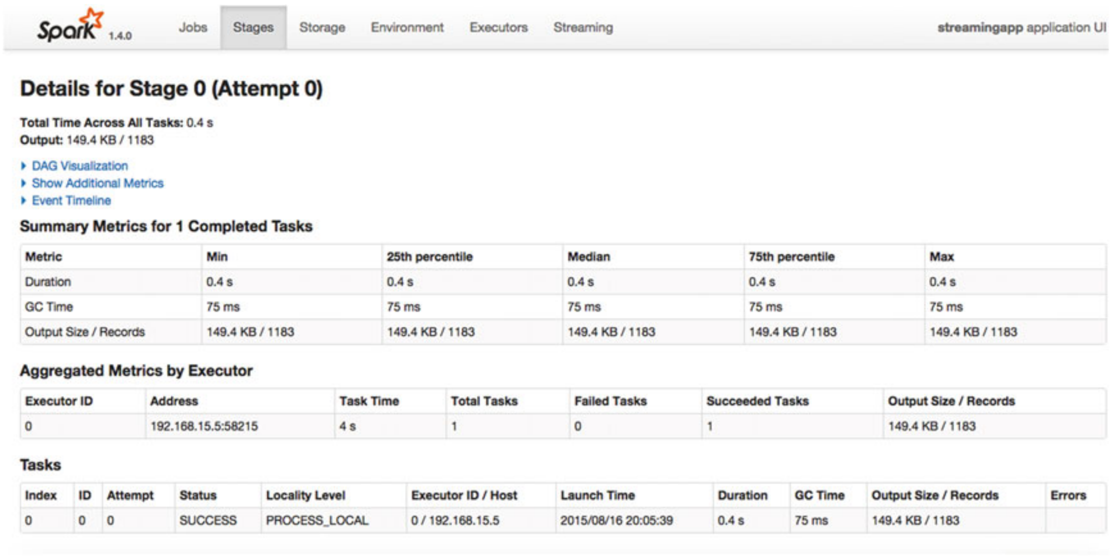


Figure 3-2. *Visual depiction of the execution graph for a single batch*

■ **Note** Figure 3-2 was obtained via the Spark UI. To view it, from the main UI page, click the application ID in the Running Applications section. Then jump to Application Detail UI, which takes you to the Spark Jobs page. Clicking any completed job enables you to view completed stages for that job. Clicking any stage lands you on the Details page (Figure 3-3) for that particular stage. The DAG Visualization option on this page generates the graph in Figure 3-2. Chapter 7 digs into the specifics of the Spark UI.



Details for Stage 0 (Attempt 0)

Total Time Across All Tasks: 0.4 s
Output: 149.4 KB / 1183

Summary Metrics for 1 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.4 s	0.4 s	0.4 s	0.4 s	0.4 s
GC Time	75 ms	75 ms	75 ms	75 ms	75 ms
Output Size / Records	149.4 KB / 1183	149.4 KB / 1183	149.4 KB / 1183	149.4 KB / 1183	149.4 KB / 1183

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Output Size / Records
0	192.168.15.5:58215	4 s	1	0	1	149.4 KB / 1183

Tasks

Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Output Size / Records	Errors
0	0	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.15.5	2015/08/16 20:05:39	0.4 s	75 ms	149.4 KB / 1183	

Figure 3-3. *Details page for a stage*

One thing should be very evident from this walkthrough: a `DStream` is just a time-driven, logical wrapper around RDDs. In most situations, all operations are pushed down into the RDD layer, which allows Spark to support both batch and streaming models in the same system.

Similar to RDDs, many specialized variants of `DStreams` exist. Table 3-4 lists some of the more popular ones.

Table 3-4. *Specialized `DStreams`*

DStream Type	Description
`ConstantInputDStream`	`DStream` that returns the same RDD in every batch interval. This is useful for testing purposes.
`FileInputDStream`	Monitors a directory on the file system, and returns a `UnionRDD` for all new files in each batch interval.
`ForEachDStream`	Represents a `DStream` whose RDDs can have side effects.
`FilteredDStream`	Obtained as a result of applying a `filter` operation to a `DStream`.
`InputDStream`	Base class for all `DStreams` that fetch input from external sources by running a service in the driver program.
`MappedDStream`	Obtained as a result of applying a `map` operation to a `DStream`.
`SocketInputDStream`	Represents RDDs created from reading data from a socket. Runs the connection on a worker node.
`StateDStream`	RDDs for this `DStream` are obtained by applying a stateful operation to the RDDs for the same `DStream` from the previous batch.
`UnionDStream`	Logical consolidation of many `DStreams`. Created by invoking the `union()` method exposed by `StreamingContext`.
`WindowedDStream`	A `DStream` that returns all RDDs in a running-time window.

DStreams that operate on top of key-value pairs can be obtained via an implicit conversion. Importing PairDStreamFunctions enables this.

Let's illustrate this via a concrete example using monthly comment dumps from Reddit,[4] which is akin to an online bulletin board. This dataset contains comments on the web site alongside metadata. Each line consists of a single JSON object with data including author, body, subreddit_id, and so on; Table 3-5 lists some of the fields.[5] Assume that an external process copies this dump to the target folder periodically.

Table 3-5. *Important Fields from the Reddit Dataset*

Field Name	Type	Description
author	String	The name of the comment poster
created_utc	Long	The comment timestamp in UTC epoch-second format
body	String	The body of the comment
subreddit	String	The subreddit (Reddit speak for *topic*) the comment belongs to

Download a few of the files from the dump, and copy them to HDFS at inputPath. Listing 3-2 contains code to read this dataset.

Listing 3-2. Reading the Reddit Hourly Dumps

```
val comments = ssc.fileStream[LongWritable, Text, TextInputFormat](inputPath, (f: Path) =>
true, newFilesOnly=false).map(pair => pair._2.toString)
```

You use fileStream() explicitly instead of textFileStream() because the latter only reads files that have been recently added to the input folder. In this case, because you copied the files offline, you need to read existing files. You use fileStream(), set newFilesOnly to false, and use LongWritable as the key and Text as the value. TextInputFormat also decompresses the input files, which is why you can directly feed it .gz files. Setting newFilesOnly to false ensures that you also read existing files. There is a catch, though: you only read files that you added in the last 60 seconds. This is decided by the configuration parameter spark.streaming.minRememberDuration. You can either increase its value beyond 60 or, alternatively, touch the files to update their timestamps to fall in the 60-second window.

To illustrate the creation of a key-value DStream, let's index each record by using the author ID as key. As shown in Listing 3-3, you first use json4s to parse each record and get the author ID in the map function. You output a 2-tuple where the author ID is the key and the record itself is the value; creating a key-value DStream is just a matter of returning a 2-tuple from a transformation.

Listing 3-3. Indexing Each Record by Author ID

```
val keyedByAuthor = comments.map(rec => ((parse(rec) \ "author").values, rec))
```

It you're really curious and astute, you will have noticed that each RDD in this example contains only a single partition; but if you uncompress the files manually and rerun the application, the number of partitions goes up. This is because for uncompressed files, each HDFS split is turned into a partition. Compressed files,[6] on the other hand, are unsplittable, so each file results in a single partition. This obviously results in suboptimal parallelism. The solution is to repartition the DStream and its underlying RDDs. You will see an example of that shortly.

[4]http://files.pushshift.io/reddit/comments/monthly.
[5]The complete list of fields is available at https://github.com/reddit/reddit/wiki/JSON.
[6]gzip files cannot be split, but bzip2 and LZO are splittable.

■ **Note** Compressed files, such as gzipped ones, are not splittable and hence result in a single RDD partition. A good practice is to repartition such a `DStream` manually after creating it.

Transformations

Like RDD transformations, `DStream` transformations can be clumped together on the basis of their functionality. The categories are mapping, variation, aggregation, key-value, windowing, and miscellaneous. Let's walk through each category in turn by using the Reddit dataset. Note that to view the output, this section assumes you are using `saveAsTextFiles()` on the resulting `DStream`. You will see other `DStream` actions in the next section.

Mapping

map[U](function): DStream[U]

Applies the user-provided `function` to each element in the `DStream` to construct a new `DStream`.

Listing 3-4 lets you group the comments by day. It uses a `map` function to convert the `created_utc` field—which is a Unix timestamp—into a date and returns that as the key. A subsequent transform can perform the actual grouping.

Listing 3-4. Using a Map to Convert `created_utc` to `Date`

```
val sdf = new SimpleDateFormat("yyyy-MM-dd")
val tsKey = "created_utc"
val secs = 1000L
val keyedByDay = comments.map(rec => {
    val ts = (parse(rec) \ tsKey).values
        (sdf.format(new Date(ts.toString.toLong * secs)), rec)
})
```

mapPartitions[U](function): DStream[U]

Applies the user-provided `function` to each partition in the RDDs that constitute this `DStream` (see Listing 3-5).

Listing 3-5. Using mapPartitions to convert "created_utc" per partition to Date

```
val keyedByDay = comments.mapPartitions(iter => {
    var ret = List[(String, String)]()
            while (iter.hasNext) {
                val rec = iter.next
                        val ts = (parse(rec) \ tsKey).values
                        ret .::= (sdf.format(new Date(ts.toString.toLong * secs)), rec)
            }
    ret.iterator
})
```

This code is more efficient than Listing 3-4 for two reasons:

- The number of function invocations is lower.
- The `SimpleDateFormat (sdf)`, `String` (`tsKey`), and `Long` (`secs`) objects are shipped out to workers for each partition instead of each element.

flatMap[U](function): DStream[U]

Applies the user-provided `function` to each element in the `DStream` and then flattens out the content.

To highlight the difference between a regular `map` and a `flatMap`, it is useful to look at an example. Suppose you want to spit out the body of the comments as newline-separated words. Using a regular `map` (Listing 3-6) does not achieve this, because the content of the `DStream` is an array of strings for each record.

Listing 3-6. Attempt at Mapping the Record Body into Words

```
val wordTokens = comments.map(rec => {
      ((parse(rec) \ "body")).values.toString.split(" ")
})
```

Alternatively, a `flatMap` (Listing 3-7) flattens out the array of strings so a subsequent write to file will result in one word per line.

Listing 3-7. `flatMap`, Enabling the Contents of the Resulting `DStream` to Be Flattened

```
val wordTokens = comments.flatMap(rec => {
      ((parse(rec) \ "body")).values.toString.split(" ")
})
```

filter(function): DStream[T]

Uses the user-provided predicate `function` to filter out elements that do not match the condition. Listing 3-8 showcases a `filter` operation that only keeps comments from the AskReddit subreddit.

Listing 3-8. Filtering Out AskReddit Subreddits

```
val filterSubreddit = comments.filter(rec =>
  (parse(rec) \ "subreddit").values.toString.equals("AskReddit"))
```

transform[U](function): DStream[U]

Allows arbitrary operations to be applied to each RDD in this `DStream`. This is most useful when you need to use operations directly exposed by RDDs. For instance, `DStreams` currently do not allow you to sort their elements, but RDDs do. Listing 3-9 uses that functionality to sort the records by author.

Listing 3-9. Sorting RDDs by Author

```
val sortedByAuthor = comments.transform(rdd =>
(rdd.sortBy(rec => (parse(rec) \ "author").values.toString)))
```

Variation

union(that: DStream[T]): DStream[T]

Returns the union of this `DStream` with the passed one. The types of both `DStreams` need to be the same. For instance, this can be used if you have to ingest the same dataset from multiple locations. Taking the union enables you to treat all those datasets as a single enveloped `DStream`. Listing 3-10 shows an example.

Listing 3-10. Taking the Union of Two `DStreams`

```
val merged = comments.union(commentsFromNetwork)
```

repartition(numPartitions: Int): DStream[T]

Increases or decreases the number of partitions of each RDD in this `DStream`. Listing 3-11 increases the number of partitions of the compressed dataset from Listing 3-2.

Listing 3-11. Increasing the Number of Partitions in a `DStream`

```
val repartitionedComments = comments.repartition(4)
```

glom(): DStream[Array[T]]

Treats each partition in the underlying RDDs as an array. Although this increases the memory footprint of the operation, it decreases the amount of data that needs to be shuffled. It is largely useful to break out problems that first require determining the local result and then shuffling to perform an aggregation. A good example is trying to figure out the minimum `created_utc` timestamp in all the records (the earliest posted comment). The snippet of code in Listing 3-12 lets you calculate the minimum timestamp in each partition, which a subsequent aggregation can consume to give you a global minimum.

Listing 3-12. Calculating the Minimum Timestamp in Each Partition

```
val rddMin = comments.glom().map(arr =>
  arr.minBy(rec => ((parse(rec) \ "created_utc").values.toString.toInt)))
```

Aggregation

count(): DStream[Long]

Counts the number of elements in the RDDs of this `DStream`. Under the hood, this uses a combination of `map`, `transform`, and `reduce` to achieve this calculation (see Listing 3-13).

Listing 3-13. Counting the Number of Elements in Each RDD of This `DStream` per Batch

```
val recCount = comments.count()
```

countByValue(): DStream[(T, Long)]

Counts the frequency of each distinct element in the RDDs of this `DStream`. Internally, this invokes a `map()` followed by a `reduceByKey()` (see Listing 3-14).

Listing 3-14. Determines the Frequency of Each Distinct Record

```
val recCount = comments.countByValue()
```

The canonical MapReduce word-count application can be implemented by a `flatMap()` followed by `countByValue()`.

reduce(reduceFunc: (T, T) ⇒ T): DStream[T]

Aggregates the values in each RDD into a single value by invoking `reduceFunc` on each pair of values. Listing 3-15 counts the total number of words in the body of all messages per batch using a `reduce`.

Listing 3-15. Counting the Total Number of Words in the Body

```
val totalWords = comments.map(rec => ((parse(rec) \ "body").values.toString))
.flatMap(body => body.split(" "))
.map(word => 1)
.reduce(_ + _)
```

The number of partitions and the partitioner for the reduction can also be tweaked by using overloaded variants of the same function.

Key-value

groupByKey(): DStream[(K, Iterable[V])]

Groups the values of each key in this `DStream`. Listing 3-16 uses `groupBy` to count the number of comments published by each user and then sorts them by their frequency.

Listing 3-16. Sorting Authors by the Number of Comments Posted

```
val topAuthors = comments.map(rec => ((parse(rec) \ "author").values.toString, 1))
.groupByKey()
.map(r => (r._2.sum, r._1))
.transform(rdd => rdd.sortByKey(ascending = false))
```

reduceByKey(reduceFunc: (V, V) ⇒ V): DStream[(K, V)]

Reduces the values for each key. Logically, this can be thought of as a `groupByKey()` followed by an aggregation. In practice, it is more efficient, especially in case of large datasets, because it minimizes the amount of data that needs to be transferred across the wire via the use of combiners. Listing 3-17 reimplements the code from Listing 3-16 using `reduceByKey()`.

Listing 3-17. Sorting Authors by the Number of Comments Using `reduceByKey()`

```
val topAuthors = comments.map(rec => ((parse(rec) \ "author").values.toString, 1))
 .reduceByKey(_ + _)
 .map(r => (r._2, r._1))
 .transform(rdd => rdd.sortByKey(ascending = false))
```

combineByKey[C](createCombiner: (V) ⇒ C, mergeValue: (C, V) ⇒ C, mergeCombiner: (C, C) ⇒ C, partitioner: Partitioner): DStream[(K, C)]

Combines the values by key by invoking user-defined accumulation and merge functions.
Arguments:

1. `createCombiner`: Function to create the combiner
2. `mergeValue`: Function to accumulate the values of each partition
3. `mergeCombiner`: Function to merge two accumulators across partitions
4. `partitioner`: The partitioner to use

Every time a new key is encountered, the `createCombiner` method is called to spawn a new combiner instance for it. On subsequent encounters, `mergeValue` is used to accumulate its values. Once all keys have been exhausted, `mergeCombiner` is invoked to merge accumulators across partitions.

Unlike with `reduceByKey()` and its variants, the input and output types of the values can be different. Note that `reduce()` under the hood invokes `reduceByKey()` with the key as `null`. In turn, `reduceByKey()` calls `combineByKey()` by

- Using the same types for input and output
- Using the same `reduceFunc` for both `mergeValue` and `mergeCombiner`

`combineByKey()` is useful to implement functionality that requires custom accumulation, such as calculating averages. Listing 3-18 ranks the users in the dataset based on the average length of their comments.

Listing 3-18. Ranking Authors on the Basis of the Average Length of Their Comments

```
val topAuthorsByAvgContent = comments.map(rec
        => ((parse(rec) \ "author").values.toString, (parse(rec) \ "body").values.
        toString.split(" ").length))
.combineByKey(
        (v) => (v, 1),
        (accValue: (Int, Int), v) => (accValue._1 + v, accValue._2 + 1),
        (accCombine1: (Int, Int), accCombine2: (Int, Int)) => (accCombine1._1 +
        accCombine2._1, accCombine1._2 + accCombine2._2),
        new HashPartitioner(ssc.sparkContext.defaultParallelism)
        )
.map({case (k, v) => (k, v._1 / v._2.toFloat)})
.map(r => (r._2, r._1))
.transform(rdd => rdd.sortByKey(ascending = false))
```

join[W](other: DStream[(K, W)]): DStream[(K, (V, W))]

Performs a join between this `DStream` and the `other` one. To illustrate this function, let's use another dataset, which contains the 750 most popular subreddits as of 2014.[7] Each record contains the industry, subreddit URL, number of subscribers, and submission type. Your goal is to use this dataset to add an industry to each record in the original Reddit comments dataset. Listing 3-19 contains the code for this query. Note that `popular` is another `DStream` created via `fileStream()`.

[7]Scott Tousley, "750 Popular Subreddits, Categorized by Industry and Submission Type," *Siege Media*, updated April 11, 2016, www.siegemedia.com/popular-subreddits-by-industry.

Listing 3-19. Join Between Two DStreams

```
val keyedBySubreddit = comments.map(rec => (((parse(rec)) \ "subreddit").values.
toString, rec))
val keyedBySubreddit2 = popular.map(rec => ({
  val t = rec.split(",")
  (t(1).split("/")(4), t(0))
  }))
val commentsWithIndustry = keyedBySubreddit.join(keyedBySubreddit2)
```

Methods also exist for performing `fullOuterJoin`, `leftOuterJoin`, and `rightOuterJoin`.

cogroup[W](other: DStream[(K, W)]): DStream[(K, (Iterable[V], Iterable[W]))]

Groups elements that share the same key across `DStreams` (and their RDDs). It can be thought of as a combination of a `group by` and `join`. Listing 3-20 has the same logic as Listing 3-19 but uses a cogroup instead of `join`.

Listing 3-20. Cogrouping Two Datasets

```
val keyedBySubreddit = comments.map(rec => (((parse(rec)) \ "subreddit").values.
toString, rec))
val keyedBySubreddit2 = popular.map(rec => ({
  val t = rec.split(",")
  (t(1).split("/")(4), t(0))
  }))
val commentsWithIndustry = keyedBySubreddit.cogroup(keyedBySubreddit2)
```

updateStateByKey[S](updateFunc: (Seq[V], Option[S]) ⇒ Option[S]): DStream[(K, S)]

Updates the state of each key in the `DStream` using `updateFunc`. Can be used to maintain arbitrary state for each key across batches. For example, if you wanted to maintain a list of the number of comments per subreddit topic, you would implement something along the lines of the code in Listing 3-21. Because `updateStateByKey()` is a stateful operation,[8] checkpointing needs to be enabled (line 2).

Listing 3-21. `updateStateByKey()` as a Means to Maintain State Across Batches

```
1.   val checkpointPath = ....
2.   ssc.checkpoint(checkpointPath)
3.   val updateFunc = (values: Seq[Int], state: Option[Int]) => {
4.     val currentCount = values.sum
5.     val previousCount = state.getOrElse(0)
6.     Some(currentCount + previousCount)
7.   }
8.   val keyedBySubreddit = comments.map(rec => (((parse(rec)) \ "subreddit").values.
     toString, 1))
9.   val globalCount = keyedBySubreddit.updateStateByKey(updateFunc)
10.  .map(r => (r._2, r._1))
11.  .transform(rdd => rdd.sortByKey(ascending = false))
```

[8]Checkpointing is required for all stateful operations.

The return type of the update method is Option because in the first batch, state is None. The updateFunction takes as input the current state of each key in the form of Seq[V]. This first argument is a sequence because there may be many values for the same key in a batch. The second argument is an Option with the previous state of the key. The new state is simply the sum of the previous and the current states.

A variant of upstateStateByKey() also takes in an initial RDD that is used to seed the first batch. This can be used to use precomputed state. Keys can also be added and removed in every batch. Chapter 6 analyzes updateStateByKey() in detail where you look at the different ways to maintain batch-invariant state.

■ **Note** The number partitions or the default partitioner for each key-value operation can be provided by using overloaded methods.

Windowing

Real-time analytics also require applying queries to temporal and spatial windows of streaming data. Examples of such queries include spitting out the number of users in a particular cellular base station in the last 15 minutes, determining the number of tweets sent for a particular hash tag in a 30-minute time window, and raising a flag every time 1 million distinct orders are placed on an e-commerce web site.

window(windowDuration: Duration, slideDuration: Duration): DStream[T]

Accumulates values in a window for windowDuration time and slides the window forward every slideDuration. Both windowDuration and slideDuration need to be multiples of the batch interval. If you want to figure out the frequency of comments in all subreddits in a 5-second window, you can use something similar to Listing 3-22. The windowing logic in simple English can be read as "flush 5 seconds of data every 5 seconds."

Listing 3-22. Counting the Number of Distinct Times a Subreddit Was Commented On in a Window

```
val distinctSubreddits = comments.map(rec =>((parse(rec)) \ "subreddit").values.
toString)
val windowedRecs = distinctSubreddits.window(Seconds(5), Seconds(5))
val windowedCounts = windowedRecs.countByValue()
```

If slideDuration is not provided, the value of the batch interval is used instead. Figure 3-4 illustrates the windowing operation from Listing 3-22.

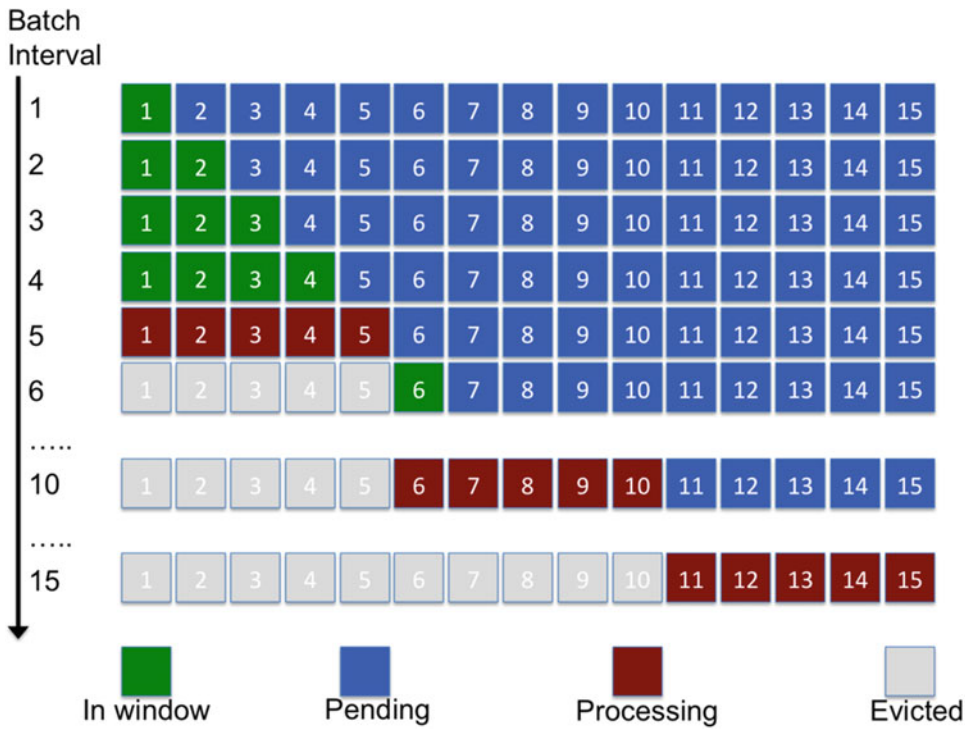


Figure 3-4. *Window with a window duration and slide duration of 5 seconds*

Each individual square on the x-axis represents an RDD, and the entire array represents a `DStream`. Time is on the y-axis with a batch interval of 1 second. Green RDDs are present in the current window, whereas blue ones are pending. Red RDDs are currently in processing, and grey ones have been evicted. With a slide duration of 5, the first time the RDDs in the window are processed is in batch interval 5. After that, all the in-window RDDs are evicted. The window is refilled again until batch interval 10, when it is processed again, and so on. Figure 3-5 illustrates the processing with a slide interval of 2 seconds.

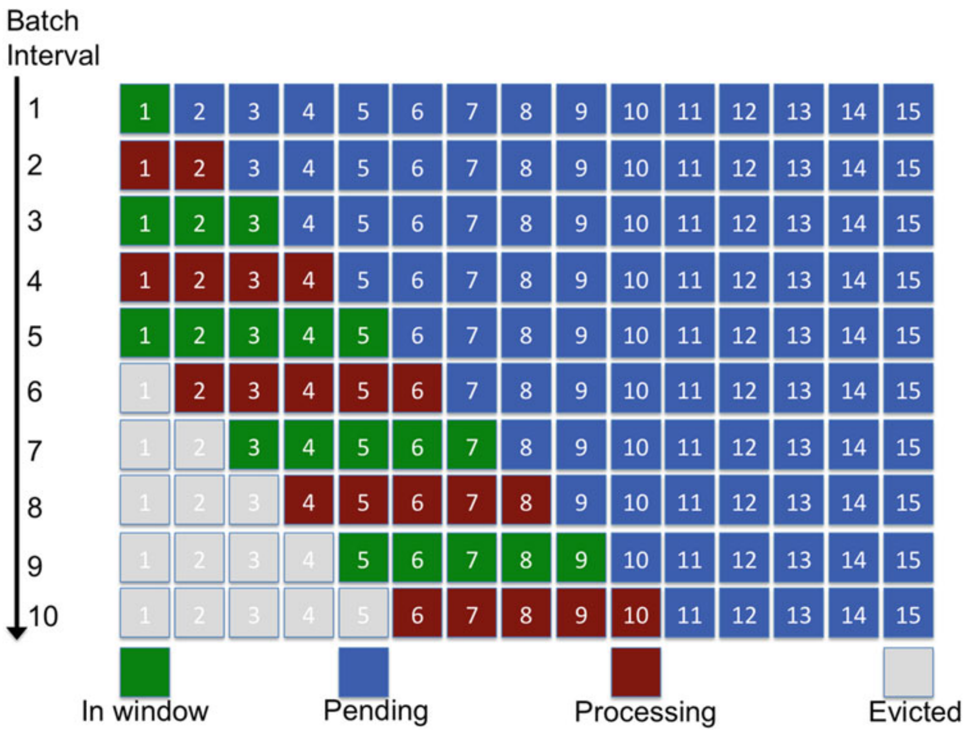


***Figure 3-5.** Window with window duration of 5 seconds and slide interval of 2 seconds*

Changing the slide interval to 2 enables the window to be processed every 2 seconds, meaning it slides forward every 2 seconds. Note that until batch interval 6, the number of RDDs in the window is fewer than 5, but from that point onward—once the window has filled up—5 RDDs are processed every 2 seconds. Also note that these windows are overlapping.

There are also a number of convenience methods that clump the specific logic with a windowing operation. The names are self-explanatory:

```
reduceByWindow(reduceFunc: (T, T) ⇒ T, windowDuration: Duration, slideDuration: Duration):
DStream[T]
countByWindow(windowDuration: Duration, slideDuration: Duration): DStream[Long]
countByValueAndWindow(windowDuration: Duration, slideDuration: Duration): DStream[(T, Long)]
groupByKeyAndWindow(windowDuration: Duration, slideDuration: Duration): DStream[(K,
Iterable[V])]
reduceByKeyAndWindow(reduceFunc: (V, V) ⇒ V, windowDuration: Duration, slideDuration:
Duration): DStream[(K, V)]
```

Actions

Actions trigger the execution of Spark Streaming jobs and, in pure functional speak, result in side-effect generation that may be external I/O or data ingestion into the driver. Each action internally creates a new `Streaming` job instance and passes it to `StreamingContext`, which in turn passes it the scheduler back end.

print(num: Int): Unit

Prints the first `num` number of elements to standard output of all RDDs in this `DStream` every batch interval.

print(): Unit

Prints the first ten elements of each RDD to standard output (see Listing 3-23). Internally invokes `print(10)`.

saveAsObjectFiles(prefix: String): Unit

Saves each RDD as a serialized object (see Listing 3-23). The convention for the output files and location is `prefix-TIME_IN_MS` in every batch. An optional suffix argument changes the convention to `prefix-TIME_IN_MS.suffix`.

saveAsTextFiles(prefix: String): Unit

Saves each RDD as a text file (see Listing 3-23). The naming convention is the same as for `saveAsObjectFiles()`.

Listing 3-23. Output Operations Applied to Listing 3-22

```
windowedCounts.print(10)
windowedCounts.saveAsObjectFiles("subreddit", "obj")
windowedCounts.saveAsTextFiles("subreddit", "txt")
```

saveAsHadoopFiles[F <: OutputFormat[K, V]](prefix: String, suffix: String): Unit

Saves each RDD as a Hadoop file parameterized with `K, V` (see Listing 3-24). Use this function if custom types for `key` and `value` are used by the `DStream`.

saveAsNewAPIHadoopFiles[F <: OutputFormat[K, V]](prefix: String, suffix: String): Unit

Same as `saveAsHadoopFiles()`, but uses the new Hadoop API introduced in version 0.21 (see Listing 3-24).

Listing 3-24. Output Operations Applied to Listing 3-21

```
globalCount.saveAsHadoopFiles("subreddit", "hadoop",
    classOf[IntWritable], classOf[Text], classOf[TextOutputFormat[IntWritable, Text]])
globalCount.saveAsNewAPIHadoopFiles("subreddit", "newhadoop",
    classOf[IntWritable], classOf[Text], classOf[NewTextOutputFormat[IntWritable, Text]])
```

foreachRDD(foreachFunc: (RDD[T]) ⇒ Unit): Unit

Applies an operation to each RDD in the `DStream`. Used for enabling side effects such as writing to external data sinks or to maintain a global count. All actions listed previously use `foreachRDD` in some form. In contrast to `transform()`, which applies an operation to each RDD but returns a `DStream`, `foreachRDD` only emits side effects. Let's assume that you want to emit the number of elements in each RDD to a log for auditing and monitoring. Listing 3-25 shows how you can achieve this by using a custom Logger and a `foreachRDD`.

Listing 3-25. Using a `foreachRDD` to Emit Data to a Logger

```
val LOG = LogManager.getLogger(this.getClass)
comments.foreachRDD(rdd => {
  LOG.info("RDD: %s, Count: %d".format(rdd.id, rdd.count()))
})
```

`foreachRDD` is a powerful external primitive. Chapter 6 analyzes more of its design patterns while looking at side effects.

Summary

`DStreams` constitute the basic building blocks of stream processing in Spark Streaming. They reside on top of RDDs and the underlying processing engine to enable streaming applications via the micro-batch processing model. Applications directly transform and manipulate `DStreams` while the subsystem seamlessly applies these operations to RDDs under the hood. Data can be ingested from and written back to a large variety of solutions, including HDFS, sockets, and Akka actors. The simple transformations and abstractions provided by the API can be mixed and matched to create rich real-time processing pipelines. These pipelines achieve processing on the terabyte scale while driving the business processes of well-known companies including Netflix, Uber, Shazam, and Pinterest.[9] Now that you have a basic handle on Spark Streaming as a whole, the next step is to learn its best practices, which is the goal of the next chapter.

[9]Tathagata Das, "Spark Streaming: What Is It and Who's Using It?" *Datanami*, November 30, 2015, `www.datanami.com/2015/11/30/spark-streaming-what-is-it-and-whos-using-it/`.

CHAPTER 4

High-Velocity Streams: Parallelism and Other Stories

If you wish to make an apple pie from scratch, you must first invent the universe.

—Carl Sagan, *Cosmos*

To analyze data in motion, you must first invent the Big Data ecosystem. In this ecosystem, designing and implementing a streaming application is easy. Making it ready for prime time, on the other hand, is much harder due to the scale of the data, the complexity of the application, and the richness of the setup: every application has a plethora of moving parts. It is extremely nontrivial to optimize each application because of the plurality of knobs and their interplay. To engender applications that scale with the input and are production ready, you must have the right tools and the knowledge to use those tools, as well as insight into the inner wirings of the target system. Having gotten a taste for real-time data processing the Spark Streaming way, you are now ready to take a deep dive into its internals.

The previous chapter used hourly data dumps from Reddit to drive the discussion. Following the same mantra, this chapter has a scientific theme: analyzing streaming data from the Voyager 1 space probe. The chapter kicks off with the most important topic in distributed systems: parallelism. The various configuration parameters exposed by both Spark and its streaming layer are also on the menu. Most importantly, common design patterns and best practices come under discussion along the way.

One Giant Leap for Streaming Data

On August 25, 2012, Voyager 1 left the heliosphere[1] and became the first manmade object to cross the Rubicon of interstellar space. Like every NASA spacecraft, Voyager 1 is instrumented with hundreds of sensors that generate a large amount of data. These sensors include imaging systems, spectrometers, and plasma wave sensors. This data is relayed over 18 billion kilometers with a latency of close to a day. The size of the data directly generated by the sensors[2] and also post analysis is on the terabyte scale.

[1]The spherical like region that defines the reach of plasma or "solar wind" from the Sun. Anything beyond that is interstellar space.
[2]One of the most iconic images of Earth, *"Pale Blue Dot"*, was taken by Voyager 1: https://en.wikipedia.org/wiki/Pale_Blue_Dot.

Z. Nabi, *Pro Spark Streaming*, DOI 10.1007/978-1-4842-1479-4_4

In addition to entering interstellar space, other milestones for the probe include entering an asteroid belt and observation of Jupiter and Saturn. So how did NASA scientists figure out that Voyager 1 had exited the heliosphere? By analyzing the data relayed by its sensors. In the spirit of that momentous event in human history—and to pay homage to Carl Sagan, a personal hero of mine—this chapter uses data from Voyager 1 as a running data source.

The dataset spans roughly 37 years (1977–2014) and is available as open data from the NASA CDAWeb web site.[3] Each hourly record in the dataset contains 29 attributes including distance from the Sun, proton density, and azimuth angle (presented in Table 4-1). The data used in this chapter was downloaded from February 14 1990—the day Voyager 1 snapped the *Pale Blue Dot* picture—until December 31, 2014, in the form of a TSV. Although it only spans a few hundred megabytes, it is a good illustrative example of the sort of data and applications that the scientific community has to deal with on an almost daily basis. To appreciate the scale of some of these problems, consider the Square Kilometer Array project, which is slated to produce 1 exabyte of data every day starting in 2020.

***Table 4-1.** Fields in the Voyager 1 Dataset*

#	Description	Unit
1	Year	Year
2	Seconds of the year	Seconds
3	Heliocentric distance	AU[4]
4	Heliographic inertial (HGI) latitude of the spacecraft position at the start of the data interval	Degrees
5	HGI longitude of the spacecraft position at the start of the data interval	Degrees
6	B field magnitude	nT[5]
7	Magnitude of the average field	nT
8	BR (radial magnetic field) in the radial-tangential-normal (RTN) coordinate system (without uncertainty)	nT
9	BR with uncertainty	nT
10	BT (tangential magnetic field) in the RTN coordinate system (without uncertainty)	nT
11	BT with uncertainty	nT
12	BN (normal magnetic field) in the RTN coordinate system (without uncertainty)	nT
13	BN with uncertainty	nT
14	Bulk flow speed	Km/s
15	THETA—elevation angle of the velocity vector (RTN)	Degrees
16	PHI—azimuth angle of the velocity vector (RTN)	Degrees
17	Proton density	Cc/n
18	Proton temperature	Kelvin

(*continued*)

[3] http://cdaweb.gsfc.nasa.gov/.
[4] Astronomical units: the mean distance between the Sun and Earth.
[5] Nano Tesla: strength of the magnetic field.

Table 4-1. *(continued)*

#	Description	Unit
19	Proton flux 0.57—1.78 energy bins, MeV,[6] LECP[7]	pfu[8]
20	Proton flux 3.40—17.6 energy bins, MeV, LECP	pfu
21	Proton flux 22.0—31.0 energy bins, MeV, LECP	pfu
22	Proton flux 1.894—2.605 energy bins, MeV, CRS[9] (6-hr[10])	pfu
23	Proton flux 4.200—6.240 energy bins, MeV, CRS (6-hr)	pfu
24	Proton flux 3.256—8.132 energy bins, MeV, CRS (6-hr)	pfu
25	Proton flux 3.276—8.097 energy bins, MeV (6-hr)	pfu
26	Proton flux 6.343—42.03 energy bins, MeV (6-hr)	pfu
27	Proton flux 17.88—26.81 energy bins, MeV (6-hr)	pfu
28	Proton flux 30.29—69.47 energy bins, MeV (6-hr)	pfu
29	Proton flux 132.8—242.0 energy bins, CRS (6-hr)	MeV

Parallelism

Recall from Chapter 2 that a Spark application consists of jobs that in turn are divided into stages. The actual entity that is executed on worker nodes is a task, which is a pipeline of multiple transforms in the same shuffle region. The natural question to ask is, “How do you control the number of instances of each entity?” And, more important, “How do you achieve an optimal value for each?” Let’s try to find the answer.

Worker

A *worker* is a control process that runs on a cluster node. Note that workers are only launched directly in standalone cluster mode. When using a cluster manager, such as YARN or Mesos, this functionality is delegated to its entities. For instance, under YARN, the `NodeManager` performs the role of the worker. Therefore, the following worker parallelism discussion only applies to standalone cluster mode.

By default, Spark executes one worker process per node. For most setups, this number suffices, but this is suboptimal in cases where workers have a large amount of memory. This is the case pre-Spark 1.4,[11] where it leads to two shortcomings:[12]

- Frequent JVM garbage collection hurts performance.

[6]Million electron volts: the energy of particles.
[7]Low energy charged particle: galactic cosmic radiation at low energy.
[8]Particle flux unit: the rate of transfer of particles through a unit area.
[9]Cosmic ray subsystem: high-energy particles in plasma.
[10]6-hour resolution.
[11]Pre-Spark 1.4, a worker in standalone mode could execute only a single executor (`https://issues.apache.org/jira/browse/SPARK-1706`). Users of older versions also had to execute multiple workers per node to run multiple executors.
[12]“Multiple Spark Worker Instances on a Single Node. Why More of Less Is More Than Less,” Sonra, June 3, 2015, `http://sonra.io/multiple-spark-worker-instances-on-a-single-node-why-more-of-less-is-more-than-less/`.

- Compressed oops[13] cannot be used for heap size greater than 32 GB. In this situation, it is useful to run more than one worker JVM to amortize the cost of garbage collection across them and to enable compressed oops. The environment variable `SPARK_WORKER_INSTANCES` dictates the number of worker processes per node. This can be set directly either in the terminal environment or in `$SPARK_HOME/conf/spark-env.sh`.

If you choose to set `SPARK_WORKER_INSTANCES` to greater than 1, you also need to adjust the CPU assignment for each worker via `SPARK_WORKER_CORES`. A good rule of thumb is to keep `SPARK_WORKER_CORES` × `SPARK_WORKER_INSTANCES` equal to the total number of cores on each node; otherwise, workers may interfere with each other, because as by default each worker greedily uses all the available cores. Similarly, the memory allocation of each worker (decided by `SPARK_WORKER_MEMORY`) should be adjusted accordingly, because each worker is by default configured to use all the memory minus 1 GB.[14]

Note that this is not a problem in Spark 1.4 and beyond, because multiple executors are executed in the same worker process.

Executor

An *executor* is a JVM process that executes on each worker. An executor is exclusive to an application, and a typical application runs many executors. In turn, a worker can execute multiple executor processes at a time. Table 4-2 enumerates the various configuration parameters that affect the number of executors and their resource allocation.

Table 4-2. *Setting Resource Allocation for Applications and Executors*

Configuration Parameter	Description	Default Value (YARN)	Default Value (Standalone)
`spark.executor.cores`	Number of cores per executor	1	All cores
`spark.executor.memory`	Amount of memory per executor	512 MB	512 MB
`spark.cores.max`	Number of cores per application across the cluster	Not used	`spark.deploy.defaultCores`
`spark.deploy.defaultCores`	Number of cores per application across the cluster in standalone mode	Not used	All cores
`spark.executor.instances`	Number of executors per application	2	Not used
`spark.dynamicAllocation.enabled`	Scales the number of executors for an application up and down based on workload heuristics	false	Not used

[13]Compressed oops is a technique used by modern JVMs to compress ordinary object pointers, leading to efficient memory packing.
[14]This memory is reserved for the OS and other services.

In standalone mode, the number of executors is determined by `spark.cores.max` and `spark.executor.cores`. For instance, with a cluster of two machines where each machine has four cores and `spark.executor.cores` is 2 and `spark.cores.max` is 4, each application gets two executors.

Choosing the Number of Executors

The following rules of thumb should be followed when carving the cluster into executors:[15]

- Leave some resources for the OS and other services on each node: at least 1 CPU and 1 GB.
- If running under YARN, also account for the application master.
- If using pre-1.4 Spark, do not create large executors, because they lead to frequent JVM garbage-collection pauses and other JVM issues. Similarly, try not to create very small executors, because broadcast variables are always executor local, leading to unnecessary data copying.
- Be mindful of the data requirements of each application. For instance, an application that processes 1 billion records per second should obviously have more resources than one that processes a few thousand. The data-ingestion rate should also be used as a proxy to work out the number of executors.

Dynamic Executor Allocation

In most cases, you do not know the resource requirements of applications up front. Therefore, it is nontrivial and also suboptimal to divvy up resources between executors. Overallocating resources leads to underutilization, whereas underallocation leads to data loss—especially in streaming applications, which are dynamic and have strict latency and quality-of-service requirements. For instance, a streaming application might experience traffic spikes that would require more resources to process. In such situations, elasticity is key. That is where the dynamic resource allocation service introduced in Spark 1.2 comes in. It enables Spark to dynamically scale the number of executors up and down in reaction to application semantics. Currently, this feature is only supported for YARN, but it is slated for standalone execution as well in a future release.

Dynamic executor allocation can be enabled by setting the configuration parameter `spark.dynamicAllocation.enabled` to `true`. To explain the algorithm behind this allocation and its associated parameters, let's look at an example. Imagine your streaming application is analyzing data from Voyager 1 and it encounters a previously undiscovered celestial body. Also imagine that Voyager 1 is configured to turn on extra sensors when such an event takes place. Any increase in the data rate as well as data volume will start to stress your existing executors which will cause build up of tasks in the queue.

When tasks begin pending for `spark.dynamicAllocation.schedulerBacklogTimeout`, a request for extra executors will be sent out. If there are still pending tasks after the first set of extra executors have been assigned, requests for executors will be fired off regularly at `spark.dynamicAllocation.sustainedSchedulerBacklogTimeout`. The metric for the number of requested executors is very similar to TCP slow start: the number of requested executors increases exponentially (1, 2, 4, 8, and so on).

[15]Dedicate resources on one machine for the Spark master in case of standalone deployment and the resource manager in case of YARN.

Once the spike in data from the unusual encounter has subsided, Voyager 1 will return to sending data at its normal rate. Executors that are idle for `spark.dynamicAllocation.executorIdleTimeout` time will simply be reclaimed by YARN. It is important to highlight that under normal circumstances, the shuffle operation is enabled by an executor. This is in tension with dynamic allocation/deallocation wherein executors are spawned and reaped on demand, which may lead to the output of `map` tasks being lost. Therefore, with dynamic executor allocation, an external shuffle service needs to be used.[16]

Task

Tasks in Spark are the unit of execution. Tasks execute as threads instead of processes in the executor, to enable in-memory RDDs. By default, there is a one-to-one mapping between a core and a task. This number can be increased via the configuration parameter `spark.task.cpus`. Use this setting for heavy-duty, CPU-bound tasks. Examples of such operations include compression and matrix multiplication. Matrix multiplication has a number of applications, such as solving a system of linear equations. For instance, if your application needs to calculate the orbit of Voyager 1 based on its data, you need to use Lambert's method of orbit determination using position and velocity (information available in the dataset). This boils down to a matrix multiplication problem. In such a case, you need to increase `spark.task.cpus` beyond 1.

On the flip side, to interleave the execution of multiple tasks atop the same CPU, you need to overcommit the CPU. This scenario is applicable to tasks that are I/O or memory bound. For instance, post-analysis, if you want to archive Voyager 1 data in, say, Amazon's S3, the final `foreachRDD` tasks will be I/O bound, especially if your streaming application resides in a non-Amazon cluster due to network latency. If you want a CPU overcommit factor of 2, you set `SPARK_WORKER_CORES` × `SPARK_WORKER_INSTANCES` to twice the number of cores.

Parallelism, Partitions, and Tasks

Task parallelism is the single most important factor when it comes to performance in Spark Streaming. Finding the optimum value is nontrivial: setting it too high leads to contention, whereas setting it too low results in underutilization of resources.

You know that each `DStream` is made up of RDDs that in turn are made up of partitions. Each partition is a self-contained piece of data, which is operated on by a task. Therefore, there is an almost one-to-one mapping between the number of partitions in an RDD and the number of tasks. In a typical setting, the number of partitions in a stage (and, in turn, its parallelism) remains the same. This is known as a *narrow dependency*: a partition gets its data from a single partition in the preceding transformation, leading to pipelined execution. This applies to `map()`, `flatMap()`, `filter()`, `union()`, and so on. `coalesce()` with shuffle set to `false` also results in a narrow dependency even though it takes in multiple partitions.

The number of tasks across partitions can vary, which results in a *wide dependency*: a partition reads in records from multiple partitions in the preceding transformation. This applies to all `*ByKey` operations, such as `groupByKey()`and `reduceByKey()`; joining operations, such as `join()`, `cogroup()`, and so on; and `repartition()`. Let's go through an example to drive home the point.

The code in Listing 4-1 enables you to gauge the presence of a solar particle event[17] based on the data from Voyager 1. Solar particle events are caused by accelerated particles, such as protons emitted by the Sun. A solar storm can disrupt communication and can become a radiation hazard to spaceships. Attributes 19 to 29 in Table 4-1 represent proton flux readings binned by energy. Your goal is to find the distribution of solar storms across years. Let's use an overly sensitive threshold of 1.0 pfu for any one of the energy bins for storm detection. To minimize the amount of data per record up front, you perform a projection to keep only year and energy bins (lines 3–4). The next step is to filter out records that do not reach the threshold of 1.0 (line 5). Following this, records are grouped by year, aggregated, and sorted (lines 5–7). You finally write the output to a directory (line 7).

[16]More details about using an external shuffle service are available at `https://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation`.
[17]"Solar Particle Event," *Wikipedia*, `https://en.wikipedia.org/wiki/Solar_particle_event`.

Listing 4-1. Yearly Histogram of Proton Flux Events (pfu > 1.0)

```
val voyager1 = ssc.textFileStream(inputPath)
voyager1.map(rec => {
  val attrs = rec.split("\\s+")
  ((attrs(0).toInt), attrs.slice(18, 28).map(_.toDouble))
}).filter(pflux => pflux._2.exists(_ > 1.0)).map(rec => (rec._1, 1))
  .reduceByKey(_ + _)
  .transform(rec => rec.sortByKey(ascending = false, numPartitions = 1)).
  saveAsTextFiles(outputPath)
```

Figure 4-1 presents the dependencies for the code in Listing 4-1. Transforms with the same color reside in the same narrow dependency zone, and a color transition represents a wide dependency. For instance, partitions that constitute RDDs, which in turn reside in the `MappedDStream` emitted by the first `map`, are filtered, as is by the `filter` operator. In contrast, each task of the `reduceByKey` operator needs all instances of the keys in its partition space, which could potentially be present in each partition of its preceding `MappedDStream`. Note that this example ignores the internal transformations that are carried out by the input source and the output action.

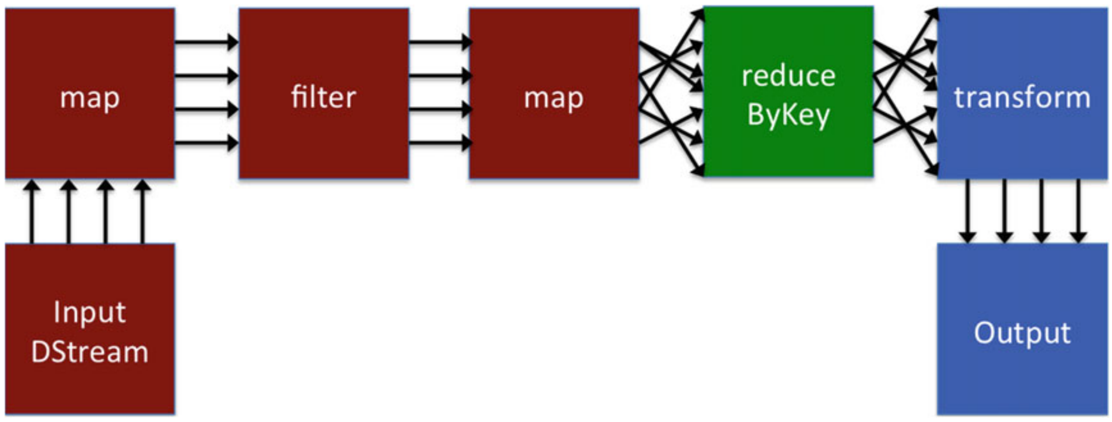


Figure 4-1. *Dependency graph of the solar event distribution application.* `DStreams` *with the same color reside in the same dependency zone. Additionally, straight arrows represent a narrow dependency, and crossed arrows represent a wide dependency (or a shuffle operation).*

Task Parallelism

The number of partitions dictates the number of tasks assigned to each `DStream`. Put differently, `DStream` parallelism is a function of the number of partitions. This number depends on the dependency type: the number of partitions across the `DStreams` that fall in a narrow dependency zone remains the same, but it changes across a shuffle zone due to a wide dependency. This is illustrated in Figure 4-2 for the same sample code. The input `DStream` has eight partitions, each corresponding to an HDFS split. The same number of partitions propagates until `reduceByKey`, which necessitates a shuffle operation. The number of partitions for shuffle operations is dictated by either `spark.default.parallelism` (details in Table 4-3) or the number of maximum partitions in the RDDs that constitute the parent `DStream` if `spark.default.parallelism` is not set. Additionally, this number can be tweaked by explicitly passing a parallelism value to the transformation. This is useful when you know that the amount of data after the transformation will expand or shrink. For instance, `reduceByKey` in Listing 4-1 shrinks the number of data records to a number per year, and keeping four partitions for a handful of data records is a waste. Therefore, you explicitly pass a value of 1 to the `sortByKey` transform.

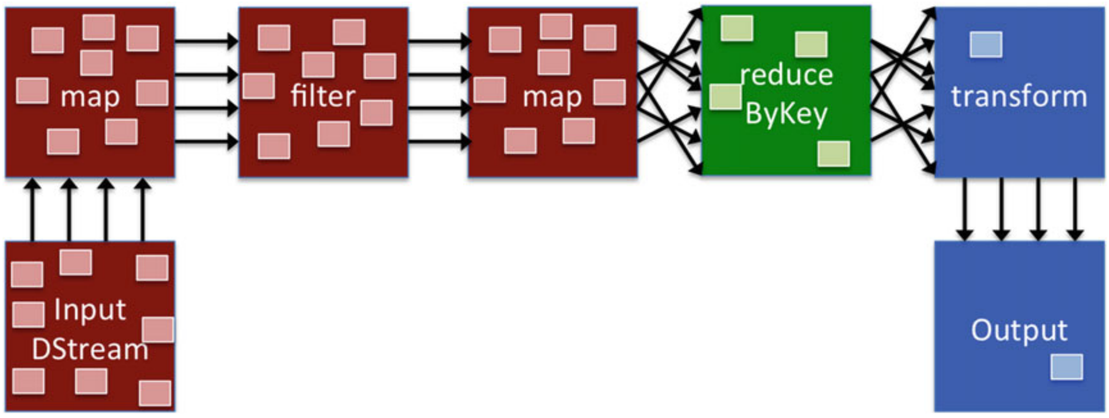


Figure 4-2. *`DStream` task parallelism across the entire solar event application. Each small box in the `DStream` represents a partition and, in turn, the number of tasks. Note how the number of tasks in a dependency zone remains the same.*

Table 4-3. *Deciding the Parallelism of Transformations That Induce a Shuffle Operation*

Configuration Parameter	Description	Default Value
`spark.default.parallelism`	Parallelism of operations that cause a shuffle, such as all `*ByKey` operations and `join`, `cogroup`, and so on	The maximum number of partitions in the parent RDD

In some cases, it is also useful to increase the number of partitions, primarily because fewer partitions means more frequent garbage collection and regular spills to disk, leading to suboptimal performance.[18] Increasing the number of partitions can be achieved either by passing an explicit parallelism value to the transformation or by invoking the `repartition()` transformation downstream. Alternatively, you can change the chunk size if you are reading from HDFS.

The next natural question to ask is, "How do you achieve the optimal parallelism for each transformation?" Unfortunately, there is no magic formula. The best way to achieve this is via trial and error by increasing this number in a controlled fashion, using the same method that is employed for regular Spark. This method, which is recommended by Cloudera,[19] specifies multiplying the number of partitions by 1.5 in each trial until performance stops improving. This can be gauged via the Spark UI, discussed briefly in the next section and in detail in Chapter 7.

[18]Sandy Ryza, "How-to: Tune Your Apache Spark Jobs (Part 2)," *Cloudera*, March 30, 2015, `http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/`
[19]Ibid.

Batch Intervals

The batch interval is another magical knob that directly affects performance. It determines the recurring period at which the transformation pipeline is invoked on streaming data. Put differently, it dictates the amount of data that is pushed through the graph of transformations in each micro-batch. For large-scale streaming applications, the goal is invariably to process data at *line rate*: the rate at which data is received on the wire. A corollary is that the batch interval should be set to enable line-rate processing. Setting the batch interval too high means high latency in addition to extra memory footprint per batch. Setting it too low, on the other hand, means high scheduling overhead and potential underutilization of resources. Finding the sweet spot between the two is nontrivial.

For instance, if Voyager 1 sends data every 500 milliseconds, then the batch interval should be comparable. This statement is based on the assumption that the processing pipeline also has a latency of 500 milliseconds; otherwise it will lead to backpressure and performance collapse. A mismatch will result in a situation similar to the one shown in Figure 4-3, where the batch latency (total delay) is greater than the input data rate, leading to back pressure.

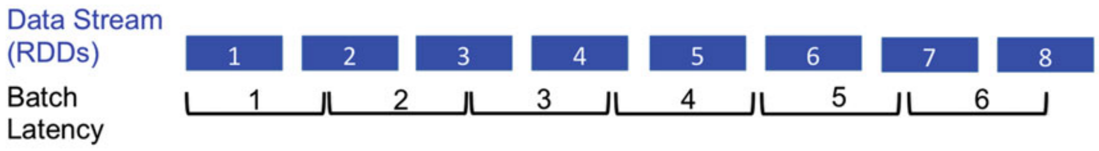


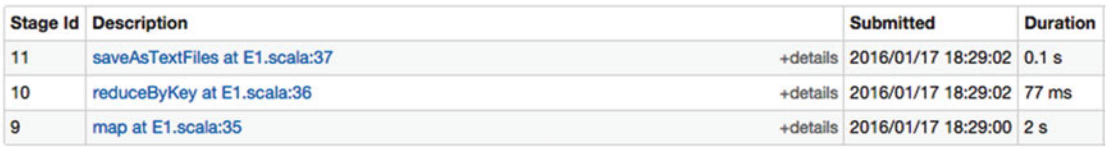
Figure 4-3. *Mismatch between the input data rate and batch latency (or the set batch interval)*

Spark Streaming internally represents the batch interval as a `Duration` object with units of milliseconds, seconds, and minutes, which is passed to `StreamingContext`. The smallest value for the batch interval is 1 millisecond, which means Spark Streaming does not enable latency on the microsecond scale. Fortunately, not many applications have that sort of latency requirements. Similar to choosing parallelism, the optimum value for the batch interval also needs to be calculated by trial and error. A good strategy is to start with a high value and then reduce it in steps (say, 1.5 times) each trial. The Spark log can be used to work out the stability of the system—that is, whether the batch interval is sufficient to keep up with the data rate. In the log, look for "Total delay." If this value remains close to the batch interval, then the system is stable. Otherwise, try increasing the parallelism as described in the previous section or reducing the processing latency of the pipeline by using the optimizations that follow this section.

Consider the application from Listing 4-1 again. You start with a generous batch interval of 10 seconds. An inspection of the logs shows the following line:

```
JobScheduler: Total delay: 2.634 s for time 1453037340000 ms (execution: 2.545 s)
```

This means the total delay for this particular batch was ~3 seconds, which is smaller than the batch interval. Assuming there might be data spikes at some point, 10 seconds seems to be a good value. You can also get a breakdown of this delay by stage by going to the Spark UI; see Figure 4-4.



Stage Id	Description		Submitted	Duration
11	saveAsTextFiles at E1.scala:37	+details	2016/01/17 18:29:02	0.1 s
10	reduceByKey at E1.scala:36	+details	2016/01/17 18:29:02	77 ms
9	map at E1.scala:35	+details	2016/01/17 18:29:00	2 s

Figure 4-4. *Breakdown of the processing delay of each stage per job in a batch. The last column contains the processing delay.*

There is a single action in the code (`saveAsTextFiles`) and hence one job. From Figure 4-2, you know that this job constitutes three stages. The first stage, which processes more data, naturally takes longer; the second and third, which only need to reduce and sort a small amount of data, take just ~0.1 second collectively. The Streaming tab in the UI can be used to gain more insight into your application. It contains information including the input rate, scheduling delay, processing time, and total delay, plus a timeline and histogram for each of these quantities. For stable applications, each should be less than the batch interval (you can figure this out by inspecting the histogram). Refer to Chapter 7 for more details.

The block interval is a sister configuration parameter that has implications for performance. It is introduced in Chapter 5 during the discussion of ingesting data using secondary solutions, such as Kafka.

■ **Note** The batch interval should be comparable to the total delay of the processing pipeline.

Scheduling

Because cluster resources are generally limited, and to use economies of scale, processing frameworks use schedulers to arbitrate resources between applications. Spark Streaming requires scheduling on three different levels among the following:

- Applications
- Batches
- Jobs

The first and the third are taken care of by the underlying Spark core, and Spark Streaming manages batches.

Inter-application Scheduling

A typical deployment of Spark contains multiple applications. The cluster manager, such as standalone or YARN, for the deployment arbitrates resource allocation between them. The details of their schedulers are enumerated in Table 4-4.

Table 4-4. *Inter-application Schedulers in Spark*

Scheduler	Policy	Resource Usage
Standalone	FIFO: first in, first out	Each application tries to grab all available resources. Use `spark.cores.max`, `spark.executor.cores` and `spark.executor.memory` to control this behavior.
YARN	FIFO, Capacity, or Fair[20]	Handled by YARN containers.

[20]http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

Batch Scheduling

This is handled by Streaming `JobScheduler`, which generates jobs from `DStreamGraph` for every batch and publishes them to Spark.

Inter-job Scheduling

A Spark Streaming application typically consists of more than one job. These jobs share the resources allocated to the executors for the application. By default, Spark uses a FIFO policy to allocate resources to jobs: the first job grabs all resources, then the second one, and so on. A subsequent job can start scheduling tasks as the previous job begins releasing its resources. The downside of this approach is that if jobs at the head of the queue take a very long time, downstream jobs have to wait. This is undesirable for streaming applications where latency is key.

To remedy this, the default scheduler can be replaced with the Hadoop Fair scheduler, which tries to equally distribute resources across jobs. Therefore, actions in a streaming job can execute in parallel, improving application latency. This can be enabled by setting `spark.scheduler.mode` to `FAIR`.

One Action, One Job

Each Spark Streaming action results in the execution of a separate job, even if the lineage graph of each has a common ancestor. This means jobs may end up repeating certain transformations. Consider the example in Figure 4-5.

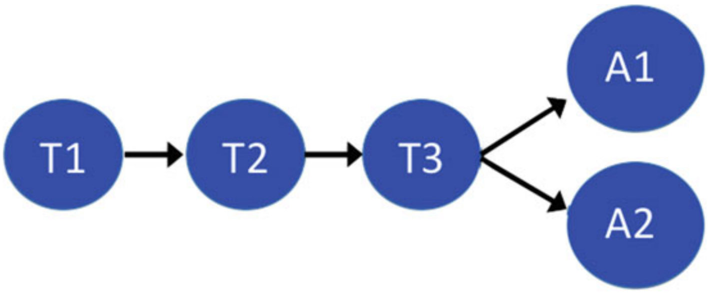


Figure 4-5. *Streaming application with two actions: T (transform) and A (action)*

This results in the following schedule, as shown in Figure 4-6:

- `Job0`: `T1`, `T2`, `T3`, `A1`
- `Job1`: `T1`, `T2`, `T3`, `A2`

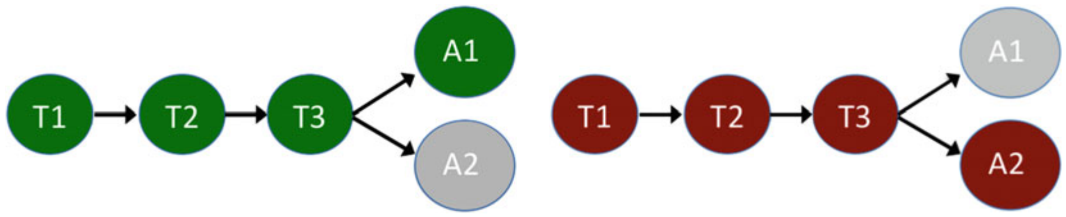


Figure 4-6. *Execution schedule for two jobs in the same application*

To understand this behavior, you need to understand the RDD-centric, action-triggered scheduling in Spark. When action `A1` in a batch for the first job is triggered, the scheduler walks up its lineage graph until it finds an RDD that is materialized, which in this case is the input of `T1`. Due to microbatching behavior, all of these RDDs are reaped by garbage collection before the second job is spawned. When the second job executes, the same pattern takes place, because the RDD generated as output of `T3` is no longer available. To enable job 2 to have access to that RDD, you need to explicitly cache it by invoking `cache()` (or `persist()` with a configurable persistence level) on it, as presented in Listing 4-2.

Listing 4-2. Caching the Output of One Job to Negate Redundant Execution

```
val t3 = t2.map(x => func(x)).cache()
t3.println(10) //A1
t3.println(20) //A2
```

This enables job 2 to avoid redundant work, but it still executes only after job 1 has completed. This is because by default, only one job is executed by the streaming scheduler. This is controlled by the configuration parameter `spark.streaming.concurrentJobs`, which is set to 1. Set this number to the number of actions across your jobs to achieve inter-streaming job parallelism. This is still an experimental parameter, so use it with caution. In addition, it can increase end-to-end latency due to resource contention between the parallel jobs.

This is also useful for other scenarios:

- *Batch parallelism:* Batches are pipelined, which means the batch at interval `T2` is scheduled only after the batch at `T1` has completed. But if batch 1 takes longer than the batch interval, the schedule of batch 2 will not be "batch interval" away from it. Increasing the number of concurrent jobs will result in interleaving of execution across batches.
- *Parallel streams:* Some applications may ingest data from multiple sources with parallel processing branches and multiple actions. For instance, an application may read financial data as well as weather data and analyze both in tandem after performing a join of the two datasets. Parallel processing for these can also be triggered via the same configuration parameter.

Memory

The previous section looked at how explicitly caching RDDs can negate redundant processing—but there is no such thing as a free lunch. In this instance, the cost is memory or disk storage. Naively caching everything in memory can stress the heap and garbage collector. In cases where your application keeps running out of heap space, you can increase it by setting `spark.executor.memory`. The format is the standard one used for the JVM. For instance, to set the heap size to 4 GB, just pass it `4096m`.[21]

In certain cases, Spark also uses off-heap data structures, such as byte buffers used by Java NIO.[22] Under YARN, this additional memory allocation is handled by `spark.yarn.executor.memoryOverhead` with a default value of `max(executorMemory * 0.10, 384)`. If your application uses copious amounts of off-heap memory, you should increase this factor. In general, increasing the heap size or off-heap memory should be the last straw. Your foremost goal should be to reduce the memory footprint of the application. Let's look at three options to achieve this.

Serialization

Keeping RDDs in memory in serialized form can reduce memory use and improve garbage collection. The latter is due to the fact that individual records are stored in a single serialized buffer. In addition, you can spit out the RDDs to disk at the expense of performance. Another option is to offload RDD management to Tachyon, as discussed in Chapter 7.

Be default, Spark uses Java serialization, which is not the most efficient of implementations. This is exacerbated in the case of streaming applications, where tens of thousands of objects need to be marshaled and distributed across the cluster in a latency-sensitive fashion. Fortunately, Spark supports Kryo, which is almost an order of magnitude more efficient and performant than standard Java serialization.[23] Kryo can be turned on by setting `spark.serializer` to `org.apache.spark.serializer.KryoSerializer`. I recommend that you always use Kryo for your streaming applications. Kryo contains serializers for most Java primitives out of the box. For custom classes, you need to register a custom serializer with Spark. Let's tweak our running example to illustrate the use of Kryo.

Listing 4-3 presents the code for a custom `ProtonFlux` class. It basically gives structure to the proton flux fields in the dataset and also enables you to check for the presence of a solar storm. To use Kryo, you simply need to implement the `KryoSerializable` interface and override its `read` and `write` methods.

Listing 4-3. Custom Serialization Class for `ProtonFlux` Objects

```
import com.esotericsoftware.kryo.{KryoSerializable,Kryo}
import com.esotericsoftware.kryo.io.{Output, Input}

class ProtonFlux(
    var year: Int,
    var bin0_57to1_78: Double,
    var bin3_40to17_6: Double,
    var bin22_0to31_0: Double,
    var bin1_894to2_605: Double,
    var bin4_200to6_240: Double,
```

[21]In local standalone mode, this can be achieved by directly using JVM options: `-Xmx 4906m`.
[22]This is done by Spark, for instance, to improve shuffle performance.
[23]`https://github.com/eishay/jvm-serializers/wiki`.

```
    var bin3_256to8_132: Double,
    var bin3_276to8_097: Double,
    var bin6_343to42_03: Double,
    var bin17_88to26_81: Double,
    var bin30_29to69_47: Double,
    var bin132_8to242_0: Double
  ) extends KryoSerializable {

  def this(year: String, bin0_57to1_78: String, bin3_40to17_6: String,
      bin22_0to31_0: String, bin1_894to2_605: String, bin4_200to6_240: String,
      bin3_256to8_132: String, bin3_276to8_097: String, bin6_343to42_03: String,
      bin17_88to26_81: String, bin30_29to69_47: String, bin132_8to242_0: String) {
    this(year.toInt, bin0_57to1_78.toDouble, bin3_40to17_6.toDouble,
        bin22_0to31_0.toDouble, bin1_894to2_605.toDouble, bin4_200to6_240.toDouble,
        bin3_256to8_132.toDouble, bin3_276to8_097.toDouble, bin6_343to42_03.toDouble,
        bin17_88to26_81.toDouble, bin30_29to69_47.toDouble, bin132_8to242_0.toDouble)
  }

  def isSolarStorm = (bin0_57to1_78 > 1.0 || bin3_40to17_6 > 1.0
    || bin22_0to31_0 > 1.0 || bin1_894to2_605 > 1.0 || bin4_200to6_240 > 1.0
    || bin3_256to8_132 > 1.0 || bin3_276to8_097 > 1.0 || bin6_343to42_03 > 1.0
    || bin17_88to26_81 > 1.0 || bin30_29to69_47 > 1.0 || bin132_8to242_0 > 1.0)

  override def write(kryo: Kryo, output: Output) {
    output.writeInt(year)
    output.writeDouble(bin0_57to1_78)
    output.writeDouble(bin3_40to17_6)
    output.writeDouble(bin22_0to31_0)
    output.writeDouble(bin1_894to2_605)
    output.writeDouble(bin4_200to6_240)
    output.writeDouble(bin3_256to8_132)
    output.writeDouble(bin3_276to8_097)
    output.writeDouble(bin6_343to42_03)
    output.writeDouble(bin17_88to26_81)
    output.writeDouble(bin30_29to69_47)
    output.writeDouble(bin132_8to242_0)
  }

  override def read(kryo: Kryo, input: Input) {
    year = input.readInt()
    bin0_57to1_78 = input.readDouble()
    bin3_40to17_6 = input.readDouble()
    bin22_0to31_0 = input.readDouble()
    bin1_894to2_605 = input.readDouble()
    bin4_200to6_240 = input.readDouble()
    bin3_256to8_132 = input.readDouble()
    bin3_276to8_097 = input.readDouble()
    bin6_343to42_03 = input.readDouble()
    bin17_88to26_81 = input.readDouble()
    bin30_29to69_47 = input.readDouble()
    bin132_8to242_0 = input.readDouble()
  }

}
```

You can use this class in your code by registering it with Spark. Listing 4-4 contains the code to achieve that. You first tell Spark to switch to Kryo serialization (line 4) and then register your custom serialization class (line 5). The rest of the code uses this class to simplify the application. This example highlights how simple it is to use Kryo while achieving rich dividends in terms of performance.

Listing 4-4. Registering a Custom Kryo Serialization Class with Spark

```
1.  val conf = new SparkConf()
2.      .setAppName(appName)
3.      .setJars(SparkContext.jarOfClass(this.getClass).toSeq)
4.      .set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
5.      .registerKryoClasses(Array(classOf[ProtonFlux]))
6.
7.  val ssc = new StreamingContext(conf, Seconds(10))
8.
9.  val voyager1 = ssc.textFileStream(inputPath)
10. val projected = voyager1.map(rec => {
11.         val attrs = rec.split("\\s+")
12.                 new ProtonFlux(attrs(0), attrs(18), attrs(19), attrs(20), attrs(21),
13.               attrs(22), attrs(23), attrs(24), attrs(25), attrs(26), attrs(27),
14.               attrs(28))
15.     })
16. val filtered = projected.filter(pflux => pflux.isSolarStorm)
17. val yearlyBreakdown = filtered.map(rec => (rec.year, 1))
18.     .reduceByKey(_ + _)
19.     .transform(rec => rec.sortByKey(ascending = false))
20.     yearlyBreakdown.saveAsTextFiles(outputPath)
```

Compression

In addition to serializing RDDs, you can also compress them by setting `spark.rdd.compress` to `true`. Doing so trades CPU cycles for memory.

Garbage Collection

As mentioned earlier, stream processing can stress the standard JVM garbage collection due to the sheer number of objects. This can increase the latency of real-time applications by causing frequent pauses. To negate this behavior, the use of the concurrent mark sweep (CMS) garbage collector is recommended for both the driver and the executors, which reduces pause time by running garbage collection concurrently with the application. Enable it for the driver program by passing `--driver-java-options -XX:+UseConcMarkSweepGC` to `spark-submit`. For executors, CMS garbage collection is turned on by setting the parameter `spark.executor.extraJavaOptions` to `XX:+UseConcMarkSweepGC`.

Every Day I'm Shuffling

Every time you trigger a shuffle operation, you copy data back and forth across the cluster. Therefore, shuffling has very high disk and network I/O costs. There are a number of rules of thumb you should follow while designing a streaming application.

Early Projection and Filtering

Similar to standard database techniques, it is always a good idea to project and filter early to reduce the amount of data that is processed by downstream operators. For instance, in Listing 4-1, line 4, you project the fields early on to confine yourself to fields that are actually required for the business logic.

Always Use a Combiner

Combiners are map-side aggregations that greatly reduce the amount of data that needs to be transferred across the wire during a shuffle. The streaming API contains a number of transformations that have built-in combiners. Let's take the example of `reduceByKey()`. Line 6 in Listing 4-1 could also have been implemented using `groupByKey().mapValues(_.sum)`; you use `reduceByKey(_ + _)` because it enables local aggregation at the partition level before the shuffle.

Generous Parallelism

Shuffle operations internally employ a hash map to separate the partition space of grouping operations. This can cause tasks to run out of heap space. You can avoid this by increasing the parallelism of `*ByKey()` tasks to reduce their working set (the amount of state they need to maintain in memory).

File Consolidation

In situations with a large number of `reduce` tasks, it is useful to consolidate intermediate files to improve disk seeks. Setting `spark.shuffle.consolidateFiles` to `true` turns on this consolidation. This invariably improves performance in Ext4 and XFS file systems. For Ext3, in certain cases it may actually degrade performance, especially on machines with more than eight cores.[24]

More Memory

The executor Java heap is shared between RDDs, shuffle, and application objects. By default, RDDs use 60% (`spark.storage.memoryFraction`) of the memory, and shuffle has 20% at its disposal (`spark.shuffle.memoryFraction`). Excessive use spills the contents of the aggregation phase of the shuffle to disk. If your application contains many shuffle steps, you should consider increasing the share of shuffle memory to reduce the number of spills to disk. This obviously trades RDD storage memory for shuffle memory.

[24]Aaron Davidson and Andrew Or, "Optimizing Shuffle Performance in Spark," www.cs.berkeley.edu/~kubitron/courses/cs262a-F13/projects/reports/project16_report.pdf.

Summary

The performance of a Spark Streaming application is a function of parallelism, the batch interval, scheduling, memory, and shuffle. To get the best out of the underlying system for your application, each one of these entities needs to be optimized. This is easier said than done, though, because each needs to be experimented with via trial and error. As a result, application development and maintenance is an evolving process that requires a lot of patience. This task can be simplified to a large extent by using some of the techniques outlined in this chapter.

So far, all the example applications have required reading data from file. In most real-world applications, however, data is received from external sources and processed directly. This source might be the API of a social network such as Twitter or a stream of log files emitted by machines in a large datacenter sent via Kafka. In the next chapter, you wean yourself away from file input and focus on more dynamic sources.

CHAPTER 5

Real-Time Route 66: Linking External Data Sources

If you want to go somewhere, goto is the best way to get there.

—Ken Thompson

Most data in the wild is dynamic and has a firm lifecycle: it is created, ingested, analyzed, and then culled or put in cold storage. This lifecycle generally has a strict time budget outside of which it is useless. The time budget for streaming data can be on a millisecond scale. Regardless of latency requirements, the first step is invariably transporting the data to a processing platform while perhaps traversing the entire Internet. Any pipelined architecture can only be as fast as its slowest link. For this reason, even before the data has landed in the data center, the choice of the transport solution—even though technically it is not part of your application—can substantially affect performance. With this in mind, this chapter is dedicated to ingesting data from solutions such as Kafka, Flume, and MQTT. In the process, you also write your own connector for HTTP to learn the ropes of connecting to external data sources.

Extending Ken Thompson's `goto` philosophy, one of the best ways to go somewhere is to ride a bike. Toward this end, this chapter uses bike-sharing data from New York City to underscore the key features of different data-ingestion solutions and their connectors in Spark Streaming. Your initial stop is the most basic data connector possible: a simple socket. Once you have whetted your appetite for external connectors, the chapter moves on to more complex solutions including Kafka and Flume and also taps Twitter. To wrap up the chapter, you write a custom connector to get a handle on the status of bike stations in New York City in real time.

Smarter Cities, Smarter Planet, Smarter Everything

Earth's resources are limited, but human needs have become virtually unlimited. This coupled with the drive toward urbanization is stressing the planet's infrastructure, from transportation to electricity and utilities. Cognizant of this, urban planners and technologists have been pushing a "smart" agenda in the last few years. This entails instrumenting every aspect of the urban environment to achieve optimal resource utilization, sharing, capacity planning, and monitoring. Doing so involves using information from traffic signals, loop inductors, CCTV cameras, and onboard systems to predict congestion or using data from smart meters to tightly couple supply and demand in electricity generation. "Smarter everything" requires an entire ecosystem of sensors, data collectors, analyzers, and actioners.

Z. Nabi, *Pro Spark Streaming*, DOI 10.1007/978-1-4842-1479-4_5

One of the key ingredients of smarter everything is a sharing economy wherein a plurality of players interact, collaborate, and share a particular resource. Examples include ride-sharing apps such as Uber, accommodation sharing via Airbnb, and P2P e-commerce enabled by the likes of Etsy. Some of these services are in the public sector and are subsidized by the government. One of the major success stories in this domain is bike sharing. Close to 1,000 cities around the world have bike-sharing programs,[1] which collectively have roughly 1 million bikes.[2] Just in New York City, which has the largest program in the United States, riders have accrued 100 million miles in 22 million trips[3] since the start of the program.

New York City also makes all bike-sharing data open. Of this data, this chapter uses two major datasets: trip histories[4] and station feed.[5] The former is released every month in a batch and contains the complete details of each trip undertaken during that period. Table 5-1 enumerates every field in this dataset and its units. To get a feel for real-time computation, you will use a custom program to replay this data. The station feed dataset is a live JSON stream of the status of each bike station in New York City. This status includes information about whether the station is functional and the number of free bikes. These fields are presented in Table 5-2.

Table 5-1. *Fields in the NYC Bike Trip History Dataset*

#	Field	Units
1	Trip duration	Seconds
2	Start time	MM/DD/YYYY H:mm
3	Stop time	MM/DD/YYYY H:mm
4	Start station ID	
5	Start station name	
6	Start station latitude	Degrees
7	Start station longitude	Degrees
8	End station ID	
9	End station name	
10	End station latitude	Degrees
11	End station longitude	Degrees
12	Bike ID	
13	User type	Customer/Subscriber[6]
14	Birth year	YYYY
15	Gender	0: Unknown, 1: Male, 2: Female

[1]Felix Richter, "Bike-Sharing Is Taking Off Around the World," *Statista*, March 19, 2015, `www.statista.com/chart/3325/bike-sharing-systems-worldwide/`.
[2]"The Bike Sharing World - 2014 -Year End Data," January 6, 2015, *The Bike-sharing Blog*, `http://bike-sharing.blogspot.com/2015/01/the-bike-sharing-world-2014-year-end.html`.
[3]Citi Bike Data 2015 Q3, *Citi Bike*, `http://datawrapper.dwcdn.net/rNb8Y/30/`.
[4]`https://www.citibikenyc.com/system-data`.
[5]`https://www.citibikenyc.com/stations/json`.
[6]*Customer*: holder of 24-hour or 7-day pass. *Subscriber*: annual member.

***Table 5-2.** Fields in the NYC Bike Station Feed*

#	Field	Units
1	Station ID	
2	Station name	
3	Number of available docks	
4	Total number of docks	
5	Latitude	Degrees
6	Longitude	Degrees
7	Status	
8	Status key	1: In Service, 2: Planned, 3: Not In Service, and 4: De-registered
9	Number of available bikes	
10	Street address	
11	Street address 2	
12	City	
13	Postal Code	
14	Location	
15	Altitude	
16	Test station	True/False
17	Last communication time	YYYY-MM-DD HH:mm:ss AM/PM
18	Land mark	

ReceiverInputDStream

A typical Spark Streaming batch starts with the creation of an `InputDStream` and stops with a side effect. As shown in Figure 5-1, currently `InputDStreams` can be of five main types:

`ConstantInputDStream`: Repeats the same RDD in every batch. Only used for testing.

`FileInputDStream` : Generates RDDs from files present on a file system. Examples include `textFileStream`.

`ReceiverInputDStream`: Places a receiver on each node, and generates RDDs from incoming data.

`QueueInputDStream`: Converts data from a Scala queue to a `DStream`.

`DirectKafkaInputDStream`: Represents a stream of `KafkaRDD` wherein each RDD corresponds to a Kafka partition.

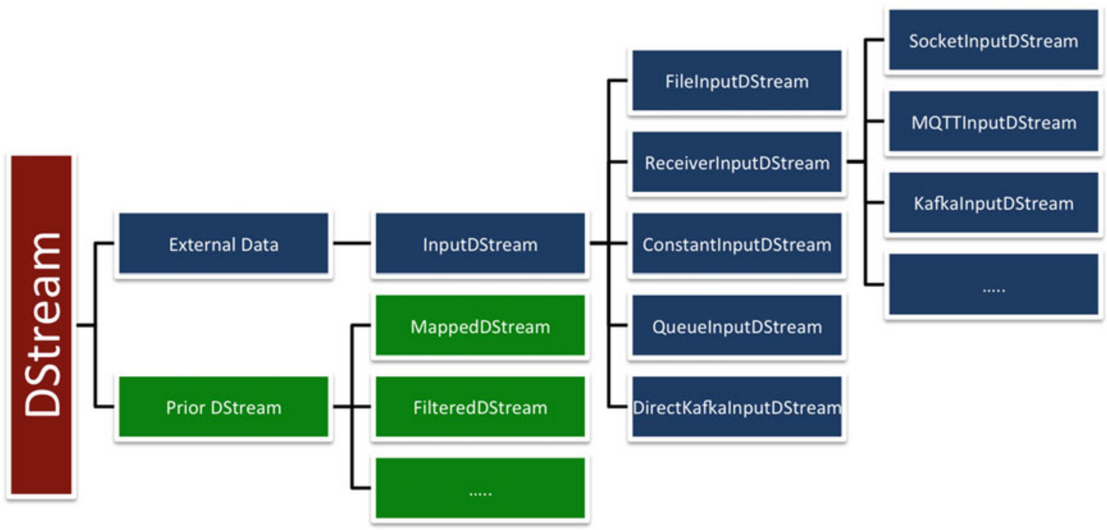


Figure 5-1. *Hierarchical view of different `DStreams`*

Unlike a standard `InputDStream`, which is executed on a single node, `ReceiverInputDStreams` are generated by placing receivers on worker nodes. This is necessary for high-data-volume solutions such as Kafka, which can easily overwhelm a single node. With `ReceiverInputDStreams`, a single node does not become the bottleneck, because network reception is distributed across many workers. Spark 1.3 also added support for a receiver-less "direct" `DStream` for Kafka to ensure better end-to-end reliability guarantees.

A standard `InputDStream` exposes `start()` and `stop()` methods and inherits a `compute()` method from `DStream`. These need to be extended by any concrete implementation of a connector. As the names suggest, `start()` and `stop()` sign-post the beginning and the end of an `InputDStream`. The `compute()` method is in charge of returning new RDDs in every batch. For instance, `FileInputDStream` returns newly added files from a directory in every batch in its `compute()` method. The service/thread for this method is executed on the driver node.

`ReceiverInputDStream` extends this interface with a `getReceiver()` method, which is expected to return a `Receiver` object that is executed on worker nodes. This receiver connects to input sources and fetches data to store in the Spark block store. Receivers can be both reliable and unreliable. You drill down into the details of `Receivers` toward the end of this chapter when you implement your own `ReceiverInputDStream` for HTTP. For now, you have enough insight to jump to some `ReceiverInputDStream` implementations, starting with `SocketInputDStream`.

Sockets

TCP sockets are the simplest type of network data sources, because flow control and congestion control are directly delegated to the underlying TCP protocol layer. `StreamingContext` out of the box exposes two functions to create a `SocketInputDStream`. The first of these is `socketStream()`, which creates a raw TCP socket. Its signature is presented in Listing 5-1.

Listing 5-1. Function to Create a Raw `SocketInputDStream`

```
socketStream[T: ClassTag](hostname: String, port: Int, converter: (InputStream) =>
Iterator[T], storageLevel: StorageLevel)
```

hostname and port represent the hostname:port pair of the TCP source. You need to explicitly parameterize the generic SocketInputDStream with the type of the input (T), similar to any other generic class in Scala, such as Collections. An iterator of the same type needs to be returned by the convertor function passed as the third argument. This function receives a Java InputStream as input and is expected to read data from the input stream and convert it to an Iterator with objects of type T. The last argument, storageLevel, dictates the persistence level of the received data objects.

A concrete example is the second socket function exposed by StreamingContext: socketTextStream(hostname: String, port: Int), which converts a raw socket into a SocketInputDStream of newline (\n) delimited UTF-8 Strings. Under the hood, it invokes socketStream() as shown in Listing 5-2.

Listing 5-2. Inner Wiring of socketTextStream

```
storageLevel = StorageLevel.MEMORY_AND_DISK_SER_2
socketStream[String](hostname, port, SocketReceiver.bytesToLines, storageLevel)
```

By default, it uses a persistence level of MEMORY_AND_DISK_SER_2, which you may recall persists the input data objects in memory and on disk in serialized form on two nodes. This can be overridden by passing a third StorageLevel object to socketTextStream(). The function is parameterized with the String type. The convertor function for SocketReceiver is presented in Listing 5-3. As expected, it returns an Iterator that converts an input stream to UTF-8 Strings.

Listing 5-3. Function to Convert a Raw Input Stream to a String Iterator

```
def bytesToLines(inputStream: InputStream): Iterator[String] = {
  val dataInputStream = new BufferedReader(new InputStreamReader(inputStream, "UTF-8"))
  new NextIterator[String] {
    protected override def getNext() = {
      val nextValue = dataInputStream.readLine()
      if (nextValue == null) {
        finished = true
      }
      nextValue
    }

    protected override def close() {
      dataInputStream.close()
    }
  }
}
```

Behind the scenes, SocketInputDStream uses a SocketReceiver object, which extends the Receiver interface and simply runs a thread to store objects read from the socket to the block store in the worker's memory heap.

Let's move on to a concrete example, which uses SocketInputDStream and the New York City trip history dataset. Your goal is to determine the breakdown of journey lengths by year of birth: that is, whether people from a certain age group travel more via shared bikes in New York City. You first employ a custom driver program in Java that reads zipped trip history files from HDFS and feeds them to a socket. Your Spark Streaming application then acts as a socket client and performs the actual calculation. You need a driver program for three data sources covered in this chapter: socket, MQTT, and Kafka. Therefore, you first structure it in the form of an abstract class, dubbed AbstractDriver, which is extended by all three of the target systems. The Java implementation is presented in Listing 5-4.

Listing 5-4. Abstract Driver Program That Feeds a Data-Transport System

```
import java.io.BufferedReader;
import java.io.File;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Enumeration;
import java.util.zip.ZipEntry;
import java.util.zip.ZipFile;

import org.apache.log4j.LogManager;
import org.apache.log4j.Logger;

public abstract class AbstractDriver {

    private static final Logger LOG = LogManager.getLogger(AbstractDriver.class);

    private String path;

    public AbstractDriver(String path) {
        this.path = path;
    }

    public abstract void init() throws Exception;

    public abstract void close() throws Exception;

    public abstract void sendRecord(String record) throws Exception;

    public void execute() throws Exception {

        try {
            init();
            File dirPath = new File(path);
            if (dirPath.isDirectory()) {
                File[] files = new File(path).listFiles();
                for (File f : files) {
                    LOG.info(String.format("Feeding zipped file %s", f.getName()));
                    ZipFile zFile = null;
                    try {
                        zFile = new ZipFile(f);
                        Enumeration<? extends ZipEntry> zEntries = zFile.entries();

                        while (zEntries.hasMoreElements()) {
                            ZipEntry zEntry = zEntries.nextElement();
                            LOG.info(String.format("Feeding file %s", zEntry.
                            getName()));
                            try (BufferedReader br = new BufferedReader(
                                    new InputStreamReader(zFile.
                                    getInputStream(zEntry)))) {
```

```
47.                                  // skip header
48.                                  br.readLine();
49.                                  String line;
50.                                  while ((line = br.readLine()) != null) {
51.                                      sendRecord(line);
52.                                  }
53.                              }
54.                          }
55.                      } catch (IOException e) {
56.                          LOG.error(e.getMessage());
57.                      } finally {
58.                          if (zFile != null) {
59.                              try {
60.                                  zFile.close();
61.                              } catch (IOException e) {
62.                                  LOG.error(e.getMessage());
63.                              }
64.                          }
65.                      }
66.                  }
67.              } else {
68.                  LOG.error(String.format("Path %s is not a directory", path));
69.              }
70.          } finally {
71.              close();
72.          }
73.      }
74.  }
```

The AbstractDriver program takes as input a path to the directory containing all the zipped files from the dataset. This is then used to read each line (line 50) from each file (line 42) in the zip and feed it to the data-transfer mechanism (line 51) of a concrete data-transport system in the form of a string. Each implementation also needs to provide initialization (`init()`) and shutdown (`close()`) code.

The driver implementation for a socket is shown in Listing 5-5. Internally, it employs a custom `SocketStream` class (line 49) to feed the socket. In turn, `SocketStream` uses `ServerSocketChannel` (line 53) from NIO to launch a socket server in a thread. It also exposes a `sendMsg()` function (line 89) that sends a UTF-8 encoded, new-line delimited string through the socket (line 91-92). This enables you to directly use `socketTextStream()` to ingest data into a Spark Streaming application at the other end.

Listing 5-5. Socket Driver Program

```
1.   import java.io.IOException;
2.   import java.net.InetSocketAddress;
3.   import java.nio.ByteBuffer;
4.   import java.nio.channels.ServerSocketChannel;
5.   import java.nio.channels.SocketChannel;
6.   import java.nio.charset.StandardCharsets;
7.   import java.util.concurrent.ExecutionException;
8.
9.   import org.apache.log4j.LogManager;
10.  import org.apache.log4j.Logger;
11.
```

```
public class SocketDriver extends AbstractDriver {

    private static final Logger LOG = LogManager.getLogger(SocketDriver.class);

    private String hostname;
    private int port;
    private SocketStream socketStream;

    public SocketDriver(String path, String hostname, int port) {
        super(path);
        this.hostname = hostname;
        this.port = port;
    }

    @Override
    public void init() throws Exception {
        socketStream = new SocketStream(hostname, port);
        LOG.info(String.format("Waiting for client to connect on port %d", port));
        SocketChannel socketChan = socketStream.init();
        LOG.info(String.format("Client %s connected on port %d", socketChan.
        getRemoteAddress(), port));
        socketStream.kickOff(socketChan);
        socketStream.start();
    }

    @Override
    public void close() throws IOException {
        socketStream.done();
        if (socketStream != null) {
            socketStream.close();
        }
    }

    @Override
    public void sendRecord(String record) throws Exception {
        socketStream.sendMsg(record + "\n");
    }

    static class SocketStream extends Thread {

        private String hostname;
        private int port;
        private ServerSocketChannel server;
        private volatile boolean isDone = false;
        private SocketChannel socket = null;
        private long totalBytes;
        private long totalLines;

        public SocketStream(String hostname, int port) {
            this.hostname = hostname;
            this.port = port;
            totalBytes = 0;
            totalLines = 0;
        }

        public SocketChannel init() throws IOException {
            server = ServerSocketChannel.open();
            server.bind(new InetSocketAddress(hostname, port));
            LOG.info(String.format("Listening on %s", server.getLocalAddress()));
            return server.accept();
        }

        public void kickOff(SocketChannel socket) {
            LOG.info("Kicking off data transfer");
            this.socket = socket;
        }

        @Override
        public void run() {
            try {
                while (!isDone)  {
                    Thread.sleep(1000);
                }
            } catch (Exception e) {
                LOG.error(e);
            }
        }

        public void sendMsg(String msg) throws IOException, InterruptedException,
        ExecutionException {
            if (socket != null) {
                ByteBuffer buffer = ByteBuffer.wrap(msg.getBytes(StandardCharsets.
                UTF_8));
                int bytesWritten = socket.write(buffer);
                totalBytes += bytesWritten;
            } else {
                throw new IOException("Client hasn't connected yet!");
            }
            totalLines++;
        }

        public void done() {
            isDone = true;
        }

        public void close() throws IOException {
            if (socket != null) {
                socket.close();
                socket = null;
            }
            LOG.info(String.format("SocketStream is closing after writing %d bytes and
            %d lines", totalBytes,
                        totalLines));
        }
    }

    public static void main(String[] args) throws Exception {

        if (args.length != 3) {
            System.err.println("Usage: SocketDriver <path_to_input_folder> <hostname>
            <port>");
            System.exit(-1);
        }

        String path = args[0];
        String hostname = args[1];
        int port = Integer.parseInt(args[2]);

        SocketDriver driver = new SocketDriver(path, hostname, port);
        try {
            driver.execute();
        } finally {
            driver.close();
        }
    }
}
```

At the other end of the socket, the Spark Streaming application (Listing 5-6) reads from the socket (line 26) and tokenizes each line into attribute fields (line 27). You only need the year of birth (field 13) and journey length (field 0), so you project those fields (line 28) into a key-value pair with year of birth as the key. You then aggregate journey lengths by year of birth (line 29). Line 30 performs two tasks:

- Year of birth is converted to age (by subtracting it from the current year using the function normalizeYear).
- Key-values are swapped to enable sorting by journey length, which takes place on the next line.

You finally save the output to an HDFS file (line 32); it shows that people in their late 20s and early 30s account for the longest trips.

This application sorts journey time by age at every batch interval. As you can see, using a SocketInputDStream is only a matter of passing a hostname and port to helper functions while the underlying Spark Streaming subsystem takes care of the rest.

Listing 5-6. Spark Streaming Application Using SocketInputDStream to Break Down Journey Length in the NYC Bike Trip History Dataset by Birth Year

```
1.  import org.apache.spark.SparkContext
2.  import org.apache.spark.SparkConf
3.
4.  import org.apache.spark.streaming.{ Seconds, StreamingContext }
5.  import org.apache.spark.streaming.dstream.PairDStreamFunctions
6.
7.  import java.util.Calendar
8.
```

```
object TripByYearApp {
   def main(args: Array[String]) {
     if (args.length < 4) {
       System.err.println(
         "Usage: TripByYearApp <master> <appname> <hostname> <port>" +
           " In local mode, <master> should be 'local[n]' with n > 1")
       System.exit(1)
     }
     val Seq(master, appName, hostname, port) = args.toSeq

     val conf = new SparkConf()
       .setAppName(appName)
       .setMaster(master)
       .setJars(SparkContext.jarOfClass(this.getClass).toSeq)

     val ssc = new StreamingContext(conf, Seconds(10))

     ssc.socketTextStream(hostname, port.toInt)
       .map(rec => rec.split(","))
       .map(rec => (rec(13), rec(0).toInt))
       .reduceByKey(_ + _)
       .map(pair => (pair._2, normalizeYear(pair._1)))
       .transform(rec => rec.sortByKey(ascending = false))
       .saveAsTextFiles("TripByYear")

     ssc.start()
     ssc.awaitTermination()
   }

   def normalizeYear(s: String): String = {
     try {
       (Calendar.getInstance().get(Calendar.YEAR) - s.toInt).toString
     } catch {
       case e: Exception => s
     }
   }
 }
```

In certain scenarios where the ingest rate is much slower in comparison to batch execution, it is recommended that you create multiple `SocketInputDStreams`.[7] This load-balances receivers across multiple workers. Your data source also needs to push data to multiple sockets, resulting in one `SocketInputDStream` per socket at the Spark Streaming end.

To achieve this, consider the snippet of code in Listing 5-7. `nSockets` is the total number of `SocketInputDStreams` that need to be created. The code assumes that the ports for the sockets are sequential starting at `basePort`. These `SocketInputDStreams` are then merged into a single one (line 2) to enable a common chain of transformations. Recall that `union` is only a logical operation, not a physical one, which keeps the parallelism of the merged streams intact.

[7]In fact, this concept is applicable to all `ReceiverInputDStreams`.

Listing 5-7. Creating Multiple SocketInputDStreams to Load-Balance Receivers Across Workers

```
val streams = (0 to nSockets.toInt - 1).map(i => ssc.socketTextStream(hostname,
basePort.toInt + i))
val uniStream = ssc.union(streams)
```

Although sockets are a convenient and simple abstraction to transport data, they are too low level: they have no notion of structure, fault tolerance, semantics, or partitioning. These features are generally provided by application-level frameworks layered on top of storage and the network, including log aggregators, message queues, and pipes. One such system is MQTT.

MQTT

Message Queue Telemetry Transport (MQTT) is an example of a traditional pub/sub-based message queue protocol wherein producers publish messages to a topic while consumers read from it.[8] A broker server handles storage, relay, and arbitration. Targeted toward machine-to-machine communication, MQTT has been designed for IoT, which makes it lightweight, low latency, and delay tolerant. It supports the entire spectrum of message-guarantee options: at-most-once, at-least-once, and exactly once. A wide range of message brokers including ActiveMQ, IBM Websphere MQ, RabbitMQ, and Mosquitto support MQTT.

The MQTT connector in Spark Streaming is implemented in a separate library. It uses the Eclipse Paho MQTT client.[9] As a result, the following two dependencies need to be added to your sbt file:

```
library Dependencies += "org.apache.spark" %% "spark-streaming-mqtt" % "1.4.0"
library Dependencies += "org.eclipse.paho" % "org.eclipse.paho.client.mqttv3" % "1.0.1"
```

Creating an MQTTInputDStream is a matter of invoking MQTTUtils.createStream(ssc: StreamingContext, brokerUrl: String, topic: String, storageLevel: StorageLevel), where ssc is a StreamingContext instance, brokerUrl is the URL of the MQTT broker in the format tcp://hostname:port, topic is the name of the pub/sub topic, and storageLevel is the persistence level for the blocks of the InputDStream.

To illustrate the use of MQTTInputDStream, let's consider another example: working out the yearly distribution of bike trips per day. Similar to sockets, you extend the AbstractDriver, which replays the bike trip dataset via MQTT. The code for the driver is given in Listing 5-8.

Listing 5-8. Custom Driver Program to Publish Messages from a Folder with Zipped Files to MQTT

```
import java.nio.charset.StandardCharsets;

import org.apache.log4j.LogManager;
import org.apache.log4j.Logger;
import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttException;
import org.eclipse.paho.client.mqttv3.MqttMessage;
import org.eclipse.paho.client.mqttv3.MqttTopic;
import org.eclipse.paho.client.mqttv3.persist.MemoryPersistence;

```

[8]Valerie Lampkin et al. "Building Smarter Planet Solutions with MQTT and IBM WebSphere MQ Telemetry," IBM Redbooks.
[9]www.eclipse.org/paho/.

```java
public class MqttDriver extends AbstractDriver {

    private static final Logger LOG = LogManager.getLogger(MqttDriver.class);

    private final String brokerUrl;
    private final String topic;
    private MqttClient client;
    private MqttTopic mqttTopic;

    public MqttDriver(String path, String brokerUrl, String topic) {
        super(path);
        this.brokerUrl = brokerUrl;
        this.topic = topic;
    }

    @Override
    public void init() throws Exception {
        client = new MqttClient(brokerUrl, MqttClient.generateClientId(), new MemoryPersistence());
        LOG.info(String.format("Attempting to connect to broker %s", brokerUrl));
        client.connect();
        mqttTopic = client.getTopic(topic);
        LOG.info(String.format("Connected to broker %s", brokerUrl));
    }

    @Override
    public void close() throws Exception {
        if (client != null) {
            client.disconnect();
        }
    }

    @Override
    public void sendRecord(String record) throws Exception {
        try {
            mqttTopic.publish(new MqttMessage(record.getBytes(StandardCharsets.UTF_8)));
        } catch (MqttException e) {
            if (e.getReasonCode() == MqttException.REASON_CODE_MAX_INFLIGHT) {
                Thread.sleep(10);
            }
        }
    }

    public static void main(String[] args) throws Exception {

        if (args.length != 3) {
            System.err.println("Usage:MqttDriver <path_to_input_folder> <broker_url> <topic>");
            System.exit(-1);
        }

60.         String path = args[0];
61.         String brokerUrl = args[1];
62.         String topic = args[2];
63.
64.         MqttDriver driver = new MqttDriver(path, brokerUrl, topic);
65.         try {
66.             driver.execute();
67.         } finally {
68.             driver.close();
69.         }
70.     }
71.
72. }
```

Notice that this code is very similar to the socket driver code. The main difference is that the custom SocketStream thread is replaced with MqttClient, which takes as input the URL of the MQTT broker in the format tcp://hostname:port for its initialization (line 28). In addition, it requires an MqttClientPersistence object, which is used to store in-flight incoming and outgoing messages depending on the quality of service. The client library implements two—MqttDefaultFilePersistence and MemoryPersistence—which as the names suggest, persist in-flight messages either on disk or in memory. You then create a topic object for your user-defined topic (line 31).

After creating the topic, you are ready to publish messages, which is exactly what takes place on line 45. Each message first needs to be converted to an MqttMessage object. MQTT supports three QoS options, which are listed in Table 5-3. This value can be configured for each MqttMessage object via its setQoS method.

Table 5-3. *QoS Options for MQTT*

#	QoS	Description
0	At-most-once	Similar to UDP in essence. Messages are sent over the wire and may be dropped if there is a failure along the way.
1	At-least-once	Messages need to be acknowledged by the client. In case of a failure, messages may be repeated, which means they need to be stored at the sender end. This is the default QoS.
2	Exactly once	Messages are sent only once due to an internal deduplication mechanism. Similar to at-least-once semantics, messages need to be cached at the sender end while the client has not acknowledged them.

MQTT brokers also enforce a limit on the number of in-flight messages. Therefore, publishers need to be mindful of this limit and back off every time it is hit. You achieve this on lines 46–48, where the driver goes to sleep briefly, allowing the consumer to drain the topic.

The Spark Streaming consumer application (Listing 5-9) is very similar to TripByYearApp. The only major difference is that now you want to keep a running count throughout the lifetime of the application, to maintain batch-invariant state. Toward this end, you use updateStateByKey() (line 29) instead of reduceByKey() and pass it a stateful count function (line 38).

Listing 5-9. Spark Streaming Application That Ingests Data from MQTT and Determines the Daily Trip Distribution

```scala
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.rdd.RDD.rddToOrderedRDDFunctions
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.Seconds
import org.apache.spark.streaming.StreamingContext
import org.apache.spark.streaming.dstream.DStream.toPairDStreamFunctions
import org.apache.spark.streaming.mqtt.MQTTUtils

object YearlyDistributionApp {
  def main(args: Array[String]) {
    if (args.length != 4) {
      System.err.println(
        "Usage: YearlyDistributionApp <appname> <brokerUrl> <topic> <checkpointDir>")
      System.exit(1)
    }
    val Seq(appName, brokerUrl, topic, checkpointDir) = args.toSeq

    val conf = new SparkConf()
      .setAppName(appName)
      .setJars(SparkContext.jarOfClass(this.getClass).toSeq)

    val ssc = new StreamingContext(conf, Seconds(10))
    ssc.checkpoint(checkpointDir)

    MQTTUtils.createStream(ssc, brokerUrl, topic, StorageLevel.MEMORY_ONLY_SER_2)
      .map(rec => rec.split(","))
      .map(rec => (rec(1).split(" ")(0), 1))
      .updateStateByKey(statefulCount)
      .map(pair => (pair._2, pair._1))
      .transform(rec => rec.sortByKey(ascending = false))
      .saveAsTextFiles("YearlyDistribution")

    ssc.start()
    ssc.awaitTermination()
  }

  val statefulCount = (values: Seq[Int], state: Option[Int]) => Some(values.sum +
  state.getOrElse(0))

}
```

MQTT is really useful for use cases that entail diversity and plurality in consumers and topics. A good example is IoT data that needs to be, say, partitioned into topics based on sensor type and needs to be consumed by a number of downstream systems, such as a real-time processing app and a data warehouse for long-term storage.

A different class of applications requires shipping large swathes of log data from one location to another. This data generally is not broken down semantically into topics. Consequently, using a message-queue system like MQTT is overkill. Such applications are better served by log aggregation systems, such as Flume, which is the next target system.

Flume

Flume[10] is a distributed log-aggregation system with a dataflow architecture. Data, in the form of *events,* flows from *sources* to *sinks* via *channels*. Channels dictate the reliability semantics of the connection. Out of the box, Flume provides memory, file, JDBC, and Kafka-based channels. The routing policy of channels, including setting up backup routes, is also configurable. All of these entities execute asynchronously in *agents* (JVM processes) on worker nodes. Events flow through channels in the form of hop-by-hop transactions: a channel forgets an event only after it has been acknowledged by the next channel in the pipeline or the sink endpoint. The serialization of events can be customized to follow application requirements, with default support for a number of formats including Avro and Thrift. Flume flows are set up through a Java-style key-value configuration file.

Listing 5-10 shows a sample Flume configuration file. It consists of a single flow: source ➤ channel ➤ sink. The library source `spooldir` monitors a local file system directory and turns each line from each input file into an event. The configuration employs an `Avro` sink that encodes each event into an Avro record and makes it available on the specified hostname and port. These two components are connected via a memory channel. `capacity` is the size of the memory channel event buffer, and `transactionCapacity` specifies the number of events that are copied in one transaction from the source to the sink.

Listing 5-10. Sample Flume Configuration with a Single Source, Channel, and Sink

```
# components on this agent
a1.sources = src-1
a1.sinks = snk-1
a1.channels = ch-1

# source
a1.sources.src-1.type = spooldir
a1.sources.src-1.channels = ch-1
a1.sources.src-1.spoolDir = <path_dir>

# sink
a1.sinks.snk-1.type = avro
a1.sinks.snk-1.hostname = localhost
a1.sinks.snk-1.port = 44444

# channel
a1.channels.ch-1.type = memory
a1.channels.ch-1.capacity = 10000
a1.channels.ch-1.transactionCapacity = 1000

# bind source, sink, and channel
a1.sources.src-1.channels = ch-1
a1.sinks.snk-1.channel = ch-1
```

Connecting to Flume from a Spark Streaming application requires the following dependency:

```
libraryDependencies += "org.apache.spark" %% "spark-streaming-flume" % "1.4.0"
```

[10]https://flume.apache.org/.

Similar to MQTT, using a `FlumeInputDStream` requires only a single line

```
FlumeUtils.createStream(ssc: StreamingContext, sinkHostname: String, sinkPort: Int,
storageLevel: StorageLevel)
```

where `ssc` is a `StreamingContext` instance, `sinkHostname` is the hostname of the Flume sink, `sinkPort` is its port, and `storageLevel` is the persistence level for the blocks of the `InputDStream`.

Spark Streaming supports two modes for ingesting data from Flume: push and pull. In the former, a receiver acts as a Flume agent to connect to an Avro sink, which pushes data to it. In the pull-based model, a custom Flume sink needs to be registered in your Flume configuration to buffer events. This data is periodically ingested by a reliable Flume receiver, which acknowledges these events. The latter provides better reliability and fault-tolerance guarantees because a Flume transaction is marked successful only after its data has been stored and replicated by Spark.

Let's look at using the push-based approach first. In this setup, the Spark Streaming application needs to be running before the Flume agent is executed.

Push-Based Flume Ingestion

In your Spark Streaming application, you want to determine the daily distribution of each user type: how many trips customers and subscribers undertake in a day. Listing 5-11 contains the main snippet of code for this application. On line 1, you provide the hostname and port of the Flume Avro sink. You project only the date (position 1) and user type (position 12) on line 2 along with a unit value to allow a subsequent aggregation. You may not receive all the data for a date in a batch, so you need to maintain counts across batches. Therefore, you use `updateStateByKey` instead of `reduceByKey`. You use the same stateful count function as the one in Listing 5-9. To ensure that user-type counts from the same date end up in the same partition, you repartition to 1 (line 5). The number of records in each partition is small (only two per day), so this does not degrade concurrent performance. Sorting the data by key, which is a `date-userType` pair, ensures that records are temporally monotonic (line 6). Note that for brevity, boilerplate code is omitted.

Listing 5-11. Push-Based Flume Application to Segment the Bike Trips Dataset by User Type Every Day.

```
1.   FlumeUtils.createStream(ssc, hostname, port.toInt, StorageLevel.MEMORY_ONLY_SER_2)
2.         .map(rec => new String(rec.event.getBody().array()).split(","))
3.         .map(rec => ((rec(1).split(" ")(0), rec(12)), 1))
4.         .updateStateByKey(statefulCount)
5.         .repartition(1)
6.         .transform(rdd => rdd.sortByKey(ascending = false))
7.         .saveAsTextFiles(outputPath)
```

Once the Spark Streaming application is up and running, you need to set up a Flume application with the configuration (saved in, say, `flumePush.conf`) from Listing 5-10. Note that `a1.sources.src-1.spoolDir` needs to be set to the folder containing the New York City bike trips data. The last step is to execute the Flume agent:

```
$FLUME_HOME/bin/flume-ng agent -n a1 -c flumeConf -f flumeConf/flumePush.conf
```

Pull-Based Flume Ingestion

In this model, the roles are reversed: a custom Spark Flume agent acts as the server, and the Spark Streaming application connects to it. You need to make two changes to the setup from the previous section: use a Spark Avro sink, and use a polling Flume connector in the Spark Streaming application.

In the Flume configuration in Listing 5-10, replace `avro` for `a1.sinks.snk-1.type` with `org.apache.spark.streaming.flume.sink.SparkSink` (saving the file as `flumePull.conf`). In addition, you need to have the Spark Flume connector,[11] Scala lib,[12] and Apache Language Commons[13] on the classpath of your Flume agent. To do this, create the directory path `$FLUME_HOME/plugins.d/snk-1/lib` and copy the three dependency JARs to it. Execute Flume as before:

```
$FLUME_HOME/bin/flume-ng agent -n a1 -c flumeConf -f flumeConf/flumePull.conf
```

In the Spark Streaming code (Listing 5-11), replace line 1 with

```
FlumeUtils.createPollingStream(ssc, hostname, port.toInt, StorageLevel.MEMORY_ONLY_SER_2)
```

You are now running the same application with stronger reliability and fault-tolerance guarantees. Under these guarantees, for instance, RDDs can be recovered from the block store in case of a worker failure.

Kafka

Kafka is a cross between a log aggregator and a pub/sub messaging system.[14] It couples the scalability and recovery properties of the former with the interface and reliability properties of the latter. As a result, Kafka looks and smells like a messaging system on the surface but is actually a log aggregator under the hood. It uses partitioned write-ahead commit logs to maintain all data. Data is semantically grouped under a topic, which is just a logical tag for a queue to which producers write and from which consumers read.

Each topic is sharded into partitions of fixed size and stored on broker machines, where each partition is a self-contained, append-only commit log. Each partition is also replicated across brokers with one broker acting as the replica leader. Instead of an explicit timestamp or ID, the relative position of a message in the log determines its location and temporal order. Consumers are clubbed into a consumer group and jointly consume a topic. Each consumer in a consumer group exclusively reads from a partition.[15] Another interesting feature of Kafka is that the system itself does not maintain any state information about consumers. It is the job of the consumer to update and checkpoint its offset. This means in case of a failure, the consumer may replay messages that it has already consumed. Therefore, Kafka provides at-least-once semantics by default. At-most-once delivery is achieved by disabling retries at the producer end and committing the offset of a batch of messages prior to consuming it. Ensuring exactly once semantics is a more involved process because it requires deduplication of messages at both the producer and the consumer ends. Kafka simplifies this by providing the offset.

[11]`http://search.maven.org/remotecontent?filepath=org/apache/spark/spark-streaming-flume-sink_2.10/1.4.0/spark-streaming-flume-sink_2.10-1.4.0.jar`.
[12]`http://search.maven.org/remotecontent?filepath=org/scala-lang/scala-library/2.10.4/scala-library-2.10.4.jar`.
[13]`http://search.maven.org/remotecontent?filepath=org/apache/commons/commons-lang3/3.3.2/commons-lang3-3.3.2.jar`.
[14]J. Kreps, N. Narkhede, and J. Rao, "Kafka: A Distributed Messaging System for Log Processing," *Proceedings of NetDB*, 2011.
[15]No such coordination exists across consumer groups.

Another consequence of obliviousness to consumers is that Kafka does not know whether all consumers of a topic have completed processing. As a result, it cannot use a consumer reference counter to garbage-collect messages. Instead, it uses a temporal timeout to remove messages. A nice side effect of this approach is that consumers can rewind and replay messages explicitly in case of failure. For instance, if messages are being consumed by a Hadoop mapper, then in case of a failure, a respawned mapper can replay its input from the beginning. State information, such as consumer group mappings and active brokers, is maintained in ZooKeeper.

Spark Streaming has two flavors of connectors for Kafka. One uses the high-level Kafka consumer API in concert with a receiver, write-ahead log, and ZooKeeper to provide at-least-once semantics, and the second (direct version) uses the simple Kafka consumer API[16] in tandem with checkpoint-based offset tracking to ensure exactly once semantics. Both require adding the following to your `sbt` configuration.

```
libraryDependencies += "org.apache.spark" %% "spark-streaming-kafka" % "1.4.0"
```

The application you will use to dissect Kafka, tallies the number of journeys undertaken between every pair of stations in each interval. To highlight hotspots in real time, these pairs are sorted by popularity. Listing 5-12 shows the code for the Kafka driver. It uses a `Producer` object to set up a connection (line 16) to a Kafka broker. It then sends lines from the dataset to the broker with the specified topic (line 30). Configuration information for the producer is provided via a configuration file. For this particular example, you set only two properties:

- *`metadata.broker.list`* contains comma-separated pairs of broker locations (hostname1:port1, hostname2:port2, ...).
- *`serializer.class`* decides the serialization for the message and the topic (if `key.serializer` is not set), for which you use `StringEncoder`: the type of the message body.

Other relevant parameters are listed in Table 5-4.

Listing 5-12. Kafka Driver Program

```
1.   import java.util.Properties;
2.
3.   import kafka.javaapi.producer.Producer;
4.   import kafka.producer.KeyedMessage;
5.   import kafka.producer.ProducerConfig;
6.
7.   public class KafkaDriver extends AbstractDriver {
8.
9.       private final String topic;
10.      private Producer<String, String> producer;
11.
12.      public KafkaDriver(String path, String topic, Properties props) {
13.          super(path);
14.          this.topic = topic;
```

[16]The high-level API has additional features such as offset tracking, load-balancing across consumer groups, and partition-subscription tracking over the simple API.

```
            ProducerConfig config = new ProducerConfig(props);
            producer = new Producer<String, String>(config);
        }

        @Override
        public void init() throws Exception {
        }

        @Override
        public void close() throws Exception {
            producer.close();
        }

        @Override
        public void sendRecord(String record) throws Exception {
            producer.send(new KeyedMessage<String, String>(topic, record));
        }

        public static void main(String[] args) throws Exception {

            if (args.length != 3) {
                System.err.println("Usage: KafkaDriver <path_to_input_folder> <brokerUrl>
                <topic>");
                System.exit(-1);
            }

            String path = args[0];
            String brokerUrl = args[1];
            String topic = args[2];

            Properties props = new Properties();
            props.put("metadata.broker.list", brokerUrl);
            props.put("serializer.class", "kafka.serializer.StringEncoder");

            KafkaDriver driver = new KafkaDriver(path, topic, props);
            try {
                driver.execute();
            } finally {
                driver.close();
            }
        }
    }
```

***Table 5-4.** Major Kafka Producer Configuration Parameters*

Parameter	Default	Description
`request.required.acks`	0	Controls the acknowledgement behavior of the producer. 0: No wait for acknowledgements (latency: low, durability: low). 1: Only wait for acknowledgement from the leader (latency: medium, durability: medium). -1: Wait for acknowledgement from all in-sync replicas (latency: high, durability: high).
`request.timeout.ms`	10000	The timeout associated with `request.required.acks`: how long the broker should wait to satisfy the specified `required.required.acks`.
`producer.type`	sync	Whether messages should be sent synchronously or asynchronously (with batching).
`compression.codec`	none	The compression codec to use. Possible values: none, gzip, and snappy.
`key.serializer.class`	`serializer.class`	The serializer to use for the key.

Receiver-Based Kafka Consumer

A `KafkaInputDStream` can be created as follows:

```
KafkaUtils.createStream(ssc: StreamingContext, zkQuorum: String, consumerGroupId: String,
topics: Map[String, Int], storageLevel: StorageLevel)
```

`zkQuorum` is the URL of ZooKeeper, which is used by the receiver to store message offsets and locate the broker. `consumerGroupId` is the consumer group to which this application belongs. `topics` is a hash map of `topicId` ➤ `num_of_consumer_threads`. Note that `num_of_consumer_threads` per `topicId` only decides how many threads are used by the receiver to consume data from Kafka, not Spark parallelism—this value has no bearing on the number of partitions. At the same time, `num_of_consumer_threads` should be less than or equal to the number of partitions in the topic; otherwise, some threads will remain idle.

This use is illustrated in Listing 5-13. By default, under this setup, data can be lost if the driver program restarts, because the executors are killed as well and their in-memory buffers are lost. This can be remedied by using write-ahead logs, wherein the transaction is written to a durable log before it is applied. This ensures zero data loss in the face of failure: the pending data is flushed from the log on a restart, and any lost buffered data is replayed from the source.[17] To enable this feature, you need to set `spark.streaming.receiver.writeAheadLog.enable` to `true` in the Spark configuration and also provide a checkpoint directory (`ssc.checkpoint()`) for the write-ahead log.

[17]Tathagata Das, "Improved Fault-tolerance and Zero Data Loss in Spark Streaming," *Databricks*, January 5, 2015, `https://databricks.com/blog/2015/01/15/improved-driver-fault-tolerance-and-zero-data-loss-in-spark-streaming.html`.

Listing 5-13. Receiver-Based Spark Streaming Kafka Consumer

```
1.   val topics = Map[String, Int](
2.     topic -> 1)
3.   KafkaUtils.createStream(ssc, zkQuorum, consumerGroupId, topics, StorageLevel.MEMORY_
     ONLY_SER).map(_._2)
4.     .map(rec => rec.split(","))
5.     .map(rec => ((rec(3), rec(7)), 1))
6.     .reduceByKey(_ + _)
7.     .repartition(1)
8.     .map(rec => (rec._2, rec._1))
9.     .transform(rdd => rdd.sortByKey(ascending = false))
10.    .saveAsTextFiles(outputPath)
```

The first thing to note in Listing 5-13 is that the persistence level is set to MEMORY_ONLY_SER (line 3) instead of MEMORY_ONLY_SER_2. This is to ensure that data is not replicated twice. The write-ahead log on HDFS is already replicated, so setting MEMORY_ONLY_SER_2 will replicate the data twice: once in the log and once in the block store.

The application counts the frequency of start and end station pairs (line 4–6) and orders them by that frequency (lines 8–9). The results show that some of the most frequent journeys are ones where the same station is the start and end point.

Under the hood, the address of the ZooKeeper quorum (zookeeper.connect) and consumer group ID (group.id) are passed as properties to the Kafka consumer API. The API internally also hardcodes a value of 10,000 for zookeeper.connection.timeout.ms.

Outside of these three configuration parameters, you may wish to customize other parts of the system. A variant of KafkaUtils.createStream() also accepts a map of configuration parameters in this format:

```
KafkaUtils.createStream[K,V,KeyDecoder,ValueDecoder](ssc: StreamingContext, kafkaParameters:
Map[String, String], topics: Map[String, Int], storageLevel: StorageLevel)
```

For instance, if you explicitly wanted to provide a consumer ID—by default, one is generated automatically—you would need to do something similar to Listing 5-14. Table 5-5 lists other relevant configuration parameters.

Listing 5-14. Kafka Consumer with Custom Configuration

```
1.   val params = Map[String, String](
2.     "zookeeper.connect" -> zkQuorum,
3.     "group.id" -> consumerGroupId,
4.     "consumer.id" -> consumerId)
5.   KafkaUtils.createStream[String, String, StringDecoder, StringDecoder](ssc, params,
     topics, StorageLevel.MEMORY_ONLY_SER).map(_._2)
```

Table 5-5. *Major Kafka Consumer Configuration Parameters*

Parameter	Default	Description
`fetch.message.max.bytes`	1048576	The maximum number of bytes read by the consumer in one fetch request for each partition for a topic. It must be at least as large as the largest message in your data stream. Play with this value if your Kafka driver generates `kafka.common.FailedToSendMessageException` exceptions.
`auto.commit.interval.ms`	60000	How often to commit consumer offsets to ZooKeeper in ms. A lower value is useful when, for enhanced durability, you would like to commit to ZooKeeper more often.
`auto.offset.reset`	`largest`	Which offset to jump to if there is no initial offset for this consumer in ZooKeeper. Possible values: `largest`: Only fetch the most recent data. `smallest`: Fetch data from the smallest offset onward.
`partition.assignment.strategy`	`range`	The partition-assignment strategy for consumers. Possible values: `range` and `roundrobin`.
`zookeeper.session.timeout.ms`	6000	The expected heartbeat interval for the consumer.

There are two major ways to control parallelism while reading from Kafka:

- Increase `num_of_consumer_threads` to the number of topic partitions.
- Create multiple topics, and load-balance messages across them. These multiple `KafkaInputDStreams` can then be merged in a `union()` operation (along the same lines as Listing 5-7).

Under certain circumstances, this approach does not ensure exactly once semantics and falls back on at-least-once: for instance, if the receiver has received messages and crashes before the offsets have been updated in ZooKeeper. In addition, maintaining a write-ahead log affects performance because messages need to be written to HDFS before they can be consumed. Let's move on to the direct Kafka consumer, which rectifies these problems.

Direct Kafka Consumer

As the name suggests, the direct Kafka consumer skips receivers and ZooKeeper and uses the simple API directly to consume messages. This means it needs to track offsets internally. To do so, at the beginning of each batch, the connector reads the partition offsets for each topic from Kafka and uses them to ingest data.[18] To ensure exactly once semantics, it tracks offset information in Spark Streaming checkpoints. It also has two further advantages over the receiver-based approach:

- *Performance*: Unlike the receiver approach, in which data is replicated twice (once by Kafka and then by the write-ahead log), the data-replication buck in the direct model is passed to Kafka. The Spark Streaming application only maintains offsets.

[18]Cody Koeninger, Davies Liu, and Tathagata Das, "Improvements to Kafka Integration of Spark Streaming," *Databricks*, March 30, 2015, `https://databricks.com/blog/2015/03/30/improvements-to-kafka-integration-of-spark-streaming.html`.

- *Parallelism:* There is a one-to-one mapping between Kafka partitions and RDD partitions. This means you do not need to load-balance data across topics and consumers or create multiple KafkaInputDStreams to take advantage of parallelism.

At the Spark Streaming application's end, only the stream-creation method from KafkaUtils needs to be replaced; the rest of the code remains the same. Listing 5-15 showcases the changes you need to make to the code from Listing 5-13. topics (line 1) is now a Set instead of a map, because the direct consumer does not need to use additional threads. Additionally, because you are skipping ZooKeeper, you need to directly provide the consumer with a comma-separated list of the URLs of Kafka brokers for bootstrapping (line 5). Note that because these brokers are used only for bootstrapping, not all brokers need be referenced in the list. Finally, you do not need to provide a persistence level because rereading data directly from Kafka ensures fault tolerance.

Listing 5-15. Using a Direct KafkaInputDStream

```
1.   val topics = Set(topic)
2.   val params = Map[String, String](
3.     "zookeeper.connect" -> zkQuorum,
4.     "group.id" -> consumerGroupId,
5.     "bootstrap.servers" -> brokerUrl)
6.   KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder]
     (ssc, params, topics).map(_._2)
```

Having explored solutions that are typically deployed in-house by organizations—MQTT, Flume, and Kafka—let's now use a fully managed data source, Twitter, to deliver data to Spark Streaming.

Twitter

Since its inception, Twitter has become one of the largest social media web sites in the world, with over 300 million monthly active users who send 500 million tweets per day.[19] These tweets cover a wide range of topics, from celebrity gossip and stock market prospects to social media activism and soggy fries.[20] As a result, Twitter has also become a huge repository of information of business interest. For instance, an organization can in real time work out the public perception of its products via sentiment analysis of tweets. To facilitate this, Twitter has a public API that enables users to gain access to the Twitter firehose. This API has clients in a number of languages.[21] In the case of Java, Twitter4J is arguably the most popular choice.

Twitter4J is also used by the Twitter connector for Spark Streaming. This connector has the following signature:[22]

```
TwitterUtils.createStream(ssc: StreamingContext, twitterAuth: Option[Authorization],
filters: Seq[String], storageLevel: StorageLevel)
```

Here, twitterAuth is a Twitter4J Authorization object with security credentials and filters is a sequence of strings to match against the Twitter firehose. The target application searches for New York City bike-related tweets and makes them actionable by printing their count to stdout and their content to file. Listing 5-16 contains the code for this application.

[19]https://about.twitter.com/company.
[20]Stuart Dredge, "Twitter: Why #SoggyFries Make for a Tasty Future in Big-Data Revenue," *The Guardian*, November 5, 2014, www.theguardian.com/technology/2014/nov/05/twitter-soggyfries-big-data-advertising.
[21]https://dev.twitter.com/overview/api/twitter-libraries.
[22]Sbt dependency: libraryDependencies += "org.apache.spark" %% "spark-streaming-twitter" % "1.4.0".

Listing 5-16. Using the Twitter API Connector to Read Tweets That Mention New York City Bikes

```
1.   val cb = new ConfigurationBuilder()
2.   cb.setOAuthConsumerKey(consumerKey)
3.   cb.setOAuthConsumerSecret(consumerSecret)
4.   cb.setOAuthAccessToken(accessToken)
5.   cb.setOAuthAccessTokenSecret(accessTokenSecret)
6.
7.   val twitterAuth = new TwitterFactory(cb.build()).getInstance().getAuthorization()
8.
9.   val tweetStream = TwitterUtils.createStream(ssc, Some(twitterAuth), Array("nyc citi
     bike", "nyc bike share"))
10.  tweetStream.count().print()
11.  tweetStream.saveAsTextFiles(outputPath)
```

To access the Twitter API, you first need to provide it with your Twitter credentials (lines 1–7). These credentials can be obtained by registering the application as a Twitter app.[23] The credentials are passed to the `createStream` method along with an array of keywords that serve as a filter to match tweets against. If this filter is not provided, the API falls back on randomly sampling a subset of the firehose. The output of this application prints the count of the tweets per interval (line 10) and saves their content to secondary storage (line 11).

Block Interval

Recall that the number of partitions in the input stream decides the parallelism of first-level tasks. For an input stream generated by reading data from HDFS, each partition corresponds to an HDFS block. What about input streams generated from receivers? Enter the block interval. It dictates the duration of data coalescence into a block before it is registered in the block store. `spark.streaming.blockInterval` has a default value of 200 ms. Block interval and batch interval go hand in glove: the number of blocks per interval (and hence map-like transform parallelism) is decided by `batchInterval/blockInterval`. For instance, using the default block interval for a batch interval of 1 s (1000 ms/200 ms), the number of blocks is five.

For good resource utilization, the number of tasks should at least match the number of cores per node. Therefore, the value of `blockInterval` should be set accordingly. It is generally recommended that you keep the block interval set to at least 50 ms to amortize the cost of task launch.

Custom Receiver

Any `InputDStream` that connects to an external data source needs to implement the `Receiver` interface. This receiver resides on worker nodes and is in charge of ingesting external data and writing it to the block store. Table 5-6 lists methods from this interface.

[23] https://apps.twitter.com/app/.

Table 5-6. *Methods from the `Receiver` Interface*

Method	Description
`onStart()`	Invoked when the receiver starts. Generally used to initialize state and spawn threads. This should be kept lightweight and should never be made blocking.
`onStop()`	Called when the receiver needs to exit. Operation is generally the reverse of `onStart`: used to stop threads and deallocate state. The same nonblocking stipulation applies.
`store(dataItem: T)`	Saves `dataItem` to Spark's memory. Used to implement unreliable receivers.
`store(dataBuffer: ArrayBuffer[T])`	Saves an `ArrayBuffer` of items to Spark's memory in a blocking call. Used to implement reliable receivers.
`restart(message: String, throwable: Throwable)`	Restarts the receiver: in the background, `onStop()` and `onStart()` are called. The restart delay is dictated by `spark.streaming.receiverRestartDelay`.
`stop(message: String, throwable: Throwable)`	Stops the receiver.
`reportError(message: String, throwable: Throwable)`	Reports errors to the driver. Does not change the state of the receiver.
`preferredLocation : Option[String]`	Specifies a preferred node for this receiver. Should return a hostname.
`isStarted()`	Can be used to check whether the receiver has started.
`isStopped()`	Same as the previous, but for stop state.

The reliability of a receiver is dictated by the flavor of the `store()` method it uses. Unreliable receivers store one item at a time whose state can be lost in case of a failure. Reliable receivers, on the other hand, store an `ArrayBuffer` of items via a blocking call that returns only after all items have been replicated by Spark. This ensures better fault tolerance. The receiver also needs to send acknowledgements to the data sender, which makes the design of a reliable receiver more involved.

HttpInputDStream

Spark Streaming out of the box has no connector for HTTP. In light of that, Listing 5-17 implements one to help you get the hang of the receiver API.

Listing 5-17. Custom `HttpInputDStream` and `HttpReceiver` to Ingest Data from HTTP Sources

```
import java.util.Timer
import java.util.TimerTask

import scala.reflect.ClassTag

import org.apache.http.client.methods.HttpGet
import org.apache.http.impl.client.CloseableHttpClient
import org.apache.http.impl.client.HttpClients
import org.apache.http.util.EntityUtils
import org.apache.spark.Logging
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.StreamingContext
import org.apache.spark.streaming.api.java.JavaDStream
import org.apache.spark.streaming.api.java.JavaDStream.fromDStream
import org.apache.spark.streaming.api.java.JavaStreamingContext
import org.apache.spark.streaming.dstream.DStream
import org.apache.spark.streaming.dstream.ReceiverInputDStream
import org.apache.spark.streaming.receiver.Receiver

class HttpInputDStream(
    @transient ssc_ : StreamingContext,
    storageLevel: StorageLevel,
    url: String,
    interval: Long) extends ReceiverInputDStream[String](ssc_) with Logging {

  def getReceiver(): Receiver[String] = {
    new HttpReceiver(storageLevel, url, interval)
  }
}

class HttpReceiver(
    storageLevel: StorageLevel,
    url: String,
    interval: Long) extends Receiver[String](storageLevel) with Logging {

  var httpClient: CloseableHttpClient = _
  var trigger: Timer = _

  def onStop() {
    httpClient.close()
    logInfo("Disconnected from Http Server")
  }

  def onStart() {
    httpClient = HttpClients.createDefault()
    trigger = new Timer()
    trigger.scheduleAtFixedRate(new TimerTask {
      def run() = doGet()
    }, 0, interval * 1000)

    logInfo("Http Receiver initiated")
  }

  def doGet() {
    logInfo("Fetching data from Http source")
    val response = httpClient.execute(new HttpGet(url))
    try {
      val content = EntityUtils.toString(response.getEntity())
      store(content)
    } catch {
```

```
61.          case e: Exception => restart("Error! Problems while connecting", e)
62.        } finally {
63.          response.close()
64.        }
65.
66.      }
67.
68.  }
69.
70.  object HttpUtils {
71.    def createStream(
72.      ssc: StreamingContext,
73.      storageLevel: StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2,
74.      url: String,
75.      interval: Long): DStream[String] = {
76.      new HttpInputDStream(ssc, storageLevel, url, interval)
77.    }
78.
79.    def createStream(
80.      jssc: JavaStreamingContext,
81.      storageLevel: StorageLevel,
82.      url: String,
83.      interval: Long): JavaDStream[String] = {
84.      implicitly[ClassTag[AnyRef]].asInstanceOf[ClassTag[String]]
85.      createStream(jssc.ssc, storageLevel, url, interval)
86.    }
87.  }
```

You first implement an `HttpInputDStream` that takes as input the URL of the source in its constructor (line 20). The `ReceiverInputDStream` abstract class requires that you implement a custom receiver, which in this case is an `HttpReceiver` (line 31). `HttpReceiver` is implemented following the interface listed in Table 5-6 and uses the Apache `HttpClient` library for Java[24] (line 36). In a nutshell, the library exposes methods to initiate blocking HTTP requests. Because HTTP calls are stateless,[25] recurring calls need to be made to the target endpoint in each interval. To enable this, a `Timer` (line 46) is used to schedule the request at a regular interval (line 47). This interval is assumed to be the same as the batch interval in Spark Streaming.

The actual request is initiated in a custom function (line 54) that makes a blocking call and stores the content of the response in the block store (line 59). In case of an exception, the receiver is restarted (line 61). Note that this is an unreliable receiver, because HTTP servers in general have no concept of acknowledgement. Finally, you implement a couple of helper functions to simplify the creation of an `HttpInputDStream` from Scala (line 71) and Java (line 79).

The Spark Streaming consumer application to test the `InputDStream` you just created consists of ingesting the New York City bike station status feed (`www.citibikenyc.com/stations/json`) over HTTP and spitting out the details of out-of-order stations to file. This is shown in Listing 5-18.

[24]`https://hc.apache.org/httpcomponents-client-4.5.x/`.

[25]This is not necessarily true in every case. Starting with HTTP 1.1, data chunks of unknown length can be transferred in the same session (see `http://tools.ietf.org/html/rfc7230#section-4.1`). This means HTTP servers can asynchronously stream data to clients. A Spark Streaming receiver can therefore initiate a connection and then receive data asynchronously. An example of such a receiver is at `https://github.com/actions/meetup-stream/blob/master/src/main/scala/receiver/MeetupReceiver.scala`.

Listing 5-18. Ingesting the NYC Bike Station Status Feed via the Custom HTTP Connector and Logging Nonfunctional Stations

```
1.   HttpUtils.createStream(ssc, url = "https://www.citibikenyc.com/stations/json", interval
     = batchInterval)
2.   .flatMap(rec => (parse(rec) \ "stationBeanList").children)
3.     .filter(rec => {
4.       implicit val formats = DefaultFormats
5.       (rec \ "statusKey").extract[Integer] != 1
6.     })
7.     .map(rec => rec.filterField {
8.       case JField("id", _) => true
9.       case JField("stationName", _) => true
10.      case JField("statusValue", _) => true
11.      case _ => false
12.    })
13.    .map(rec => {
14.      implicit val formats = DefaultFormats
15.      (rec(0)._2.extract[Integer], rec(1)._2.extract[String], rec(2)._2.extract[String])
16.    })
17.    .saveAsTextFiles(outputPath)
```

The application connects to the HTTP source using the custom `HttpInputDStream`. The data at the source is in the form of a large JSON object with nested JSON objects for each bike station. This is parsed into individual objects in line 2. The application then filters out stations that are functional (line 3). Subsequently, you keep only three fields: `id`, `stationName`, and `statusValue` (line 7). Before writing them to an HDFS directory on line 17, you unJSONify their types (line 13). City administrators and bike users can now use this application to avoid bike stations that aren't functional.

Summary

Pretty much everything is a potential source of data. This data may hold the cure for cancer, the signs of global warming, or the next billion-dollar idea. But before it can be analyzed and made actionable, it needs to be transported to a data center or a private cluster. A number of solutions explore different points in the design space that spans fault tolerance, latency, throughput, simplicity, and API semantics. This chapter was dedicated to exploring these options. Your journey down Route 66 of Spark Streaming data ingestion has taken you from using Kafka to ingest New York City shared bike data to using Twitter to ingest tweets in real time. En route, you learned how to write your own custom receiver to consume data from HTTP sources. Your journey is by no means over: the next chapter, among other topics, looks at emitting data from Spark Streaming to external sinks.

CHAPTER 6

The Art of Side Effects

He who performs not practical work nor makes experiments will never attain to the least degree of mastery.

—Jabir ibn Hayyan (Geber)

Spark Streaming applications by design are stateless and side-effect free: running the same application an infinite number of times results in the same behavior and output. Similar to functional programming, this simplifies debugging and reasoning about the state of a program, because input and output paths are deterministic. Although side-effect-free applications have many advantages, in distributed systems side effects cannot be completely avoided, especially when interfacing with external systems. For this reason, Spark Streaming provides a primitive called `foreachRDD`, which is the Swiss Army Knife of side effects for micro-batch processing.

This chapter introduces design patterns for enabling side effects in Spark Streaming applications. Along the way, you look at emitting output to external solutions such as HBase, Cassandra, and Redis and at maintaining state across batches. You use real-time financial feeds as a dataset. One of the main goals is to minimize the overhead of making frequent use of `foreachRDD`. In addition, you gain hands-on knowledge of using industry-standard solutions, such as HBase, as a storage solution. To attain mastery of `foreachRDD`, this practical work is necessary.

Taking Stock of the Stock Market

The global stock market experienced little innovation since its inception in the 12th century until the middle of the 20th century. Then the information age happened, fueled initially by the computer and then by the Internet. The timescale of buying and selling actions on the market went from hours to milliseconds, engendering stock market 2.0. In addition, human intervention began to diminish due to algorithmic trading, wherein machines supported by mathematical models and AI make decisions. In a similar vein, both real-time data and historical data from all major international markets can be accessed through APIs. Overall, these trends have lowered the barrier to entry for individuals to jump in to the online trading domain. Stocks can now be traded and fortunes made and lost with the click of a button.

Z. Nabi, *Pro Spark Streaming*, DOI 10.1007/978-1-4842-1479-4_6

Stock market 2.0 is driven by access to real-time stock market tickers and feeds. A number of licensed and free APIs expose these feeds: free ones include Yahoo Finance[1] and MSN Money.[2] These feeds are used by propriety solutions, such as Bloomberg Terminal,[3] Thomson Reuters Eikon,[4] and custom tools.[5] In a similar vein, this chapter pairs real-time data from Yahoo Finance with Spark Streaming to create stock-market-centric applications.

One of the world's foremost services in this domain, Yahoo Finance, exposes both a web service and an advanced querying service enabled by Yahoo Query Language (YQL).[6] YQL provides a SQL-like interface to query data from Yahoo APIs. It can be used to query any data on the Internet that follows the Open Data Table format.[7] Let's try to grab the current stock value for IBM using the web service in JSON format. Pasting the following URL in a browser results in the response shown in Listing 6-1:

```
http://finance.yahoo.com/webservice/v1/symbols/IBM/quote?format=json&view=detail
```

Listing 6-1. Response of the Current Stock Value Web Request for IBM via Yahoo Finance

```
{
  "list": {
    "meta": {
      "type": "resource-list",
      "start": 0,
      "count": 1
    },
    "resources": [
      {
        "resource": {
          "classname": "Quote",
          "fields": {
            "change": "-1.520004",
            "chg_percent": "-1.005693",
            "day_high": "150.779999",
            "day_low": "149.179993",
            "issuer_name": "International Business Machines Corporation",
            "issuer_name_lang": "International Business Machines Corporation",
            "name": "International Business Machines",
            "price": "149.619995",
            "symbol": "IBM",
            "ts": "1444766537",
            "type": "equity",
            "utctime": "2015-10-13T20:02:17+0000",
            "volume": "3915707",
            "year_high": "183.790000",
            "year_low": "140.560000"
          }
```

[1]http://finance.yahoo.com/.
[2]www.msn.com/en-us/money.
[3]www.bloomberg.com/professional/.
[4]http://financial.thomsonreuters.com/en/products/tools-applications/trading-investment-tools/eikon-trading-software.html.
[5]https://github.com/brymck/finansu.
[6]https://developer.yahoo.com/yql/.
[7]www.datatables.org/.

```
        }
      }
    ]
  }
}
```

A similar request using YQL (`select * from yahoo.finance.quotes where symbol in ("IBM")`) is as follows. Its response is omitted for brevity, because it has substantially more fields (close to 80).[8] This list of columns can be accessed online, though.[9] This is a far richer dataset, so you use it in the applications throughout this chapter:

`https://query.yahooapis.com/v1/public/yql?q=select%20*%20from%20yahoo.finance.quotes%20where%20symbol%20in%20(%22IBM%22)%0A%09%09&format=json&diagnostics=true&env=http%3A%2F%2Fdatatables.org%2Falltables.env`

foreachRDD

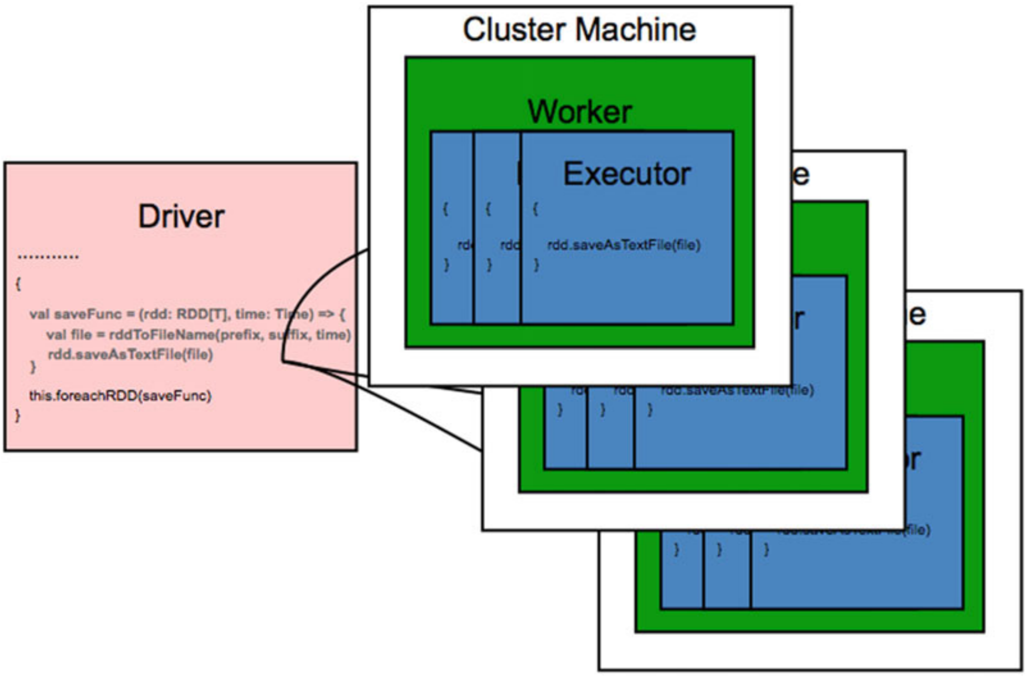
`foreachRDD`, as the name suggests, is applied as an output action to each RDD in a `DStream`. All operations in the function or closure passed to it are invoked in the driver program. Typically, this function invokes an action on an RDD, which is scheduled on worker nodes. To understand this behavior, consider the standard `saveAsTextFiles` action from the Spark codebase (Listing 6-2), which can be applied to any `DStream` to write its partitions to an HDFS-like file system.

Listing 6-2. Anatomy of a Typical `foreachRDD`-Enabled Action

```
1.   def saveAsTextFiles(prefix: String, suffix: String = ""): Unit = ssc.withScope {
2.     val saveFunc = (rdd: RDD[T], time: Time) => {
3.       val file = rddToFileName(prefix, suffix, time)
4.       rdd.saveAsTextFile(file)
5.     }
6.     this.foreachRDD(saveFunc)
7.   }
```

This creates a function to call `saveAsTestFile` on each RDD and provides it with a formatted filename. This means line 4 is actually scheduled on worker nodes, whereas the rest of the code is executed on the driver stack, as shown in Figure 6-1. This bifurcation in the execution path has implications for the design, performance, and utility of such actions. To drive home the point, let's look at a concrete example.

[8]YQL queries can also be executed via its console: `https://developer.yahoo.com/yql/console/`.
[9]`www.datatables.org/yahoo/finance/yahoo.finance.quotes.xml`.



```
............
{
  val saveFunc = (rdd: RDD[T], time: Time) => {
    val file = rddToFileName(prefix, suffix, time)
    rdd.saveAsTextFile(file)
  }
  this.foreachRDD(saveFunc)
}
```

```
{
  rdd.saveAsTextFile(file)
}
```

Figure 6-1. *Scheduling a foreachRDD action*

The example application publishes stock information to MQTT. Specifically, the goal is to grab the stock values of ten technology companies from Yahoo Finance, transform them, and then publish them to an MQTT topic (Listing 6-3).

Listing 6-3. Naïve Implementation of Publishing Yahoo Finance Stock Data to an MQTT Sink

```
HttpUtils.createStream(ssc, url = "https://query.yahooapis.com/v1/public/
yql?q=select%20*%20from%20yahoo.finance.quotes%20where%20symbol%20in%20(%22IBM,GOOG,MSF
T,AAPL,FB,ORCL,YHOO,TWTR,LNKD,INTC%22)%0A%09%09&format=json&diagnostics=true&env=http%3
A%2F%2Fdatatables.org%2Falltables.env", interval = batchInterval)
.flatMap(rec => {
  val query = parse(rec) \ "query"
  ((query \ "results" \ "quote").children).map(rec => JObject(JField("Timestamp", query
  \ "created")).merge(rec))
})
.map(rec => {
  implicit val formats = DefaultFormats
  rec.children.map(f => f.extract[String]) mkString ","
})
.foreachRDD { rdd =>
  val client = new MqttClient(outputBrokerUrl, MqttClient.generateClientId(), new
  MemoryPersistence())
  client.connect()
  rdd.foreach(rec => client.publish(topic, new MqttMessage(rec.
  getBytes(StandardCharsets.UTF_8))))
14.     client.disconnect()
15.     client.close()
16.   }
```

The application uses the HTTP receiver from the last chapter (line 1) to fetch the current stock value of IBM, Google, Microsoft, Apple, Facebook, Oracle, Yahoo, Twitter, LinkedIn, and Intel. The lookup is enabled by a YQL query wrapped in an HTTP GET call. The data is in the form of nested JSON, so you first flatten it into a per-company record and add a timestamp (lines 2–5). This is then converted to a CSV record for simplified consumer-side processing (lines 6–9). Once the data has been fetched and massaged, the next step is to emit it to MQTT. Using `foreachRDD`, you design the sink so that the MQTT client connection is created (lines 11–12) and reaped (lines 14–15) in the driver JVM while it is used to publish messages in the worker (line 13).

On the surface, this code looks fine—but the devil is in the details. Running this application raises an `org.apache.spark.SparkException: Task not serializable` exception caused by `java.io.NotSerializableException: org.eclipse.paho.client.mqttv3.MqttClient`. This is because the MQTT client connection object, which needs to be shipped out to worker nodes, is not serializable. In fact, this non-serializable property applies to most connection objects, whether database connections or simple sockets—anything that needs to be shipped to worker nodes needs to be serializable. Bearing this in mind, let's improve the code.

Per-Record Connection

The second version of the `foreachRDD` sink (Listing 6-4) mitigates the serialization problem by using a per-record connection object at the worker: an MQTT client connection is created (lines 4–5), used (line 6), and reaped (line 7–8) for each output record.

Listing 6-4. Per-Record Stateful Connection Creation in `foreachRDD`

```
1.    .foreachRDD { rdd =>
2.      rdd.foreach { rec =>
3.        {
4.          val client = new MqttClient(outputBrokerUrl, MqttClient.generateClientId(), new
            MemoryPersistence())
5.          client.connect()
6.          client.publish(topic, new MqttMessage(rec.getBytes(StandardCharsets.UTF_8)))
7.          client.disconnect()
8.          client.close()
9.        }
10.     }
11.   }
```

This code works fine in theory, but in production it is very inefficient, because it is overkill to establish a per-record connection. It would be more efficient to amortize this cost over many records. This mantra drives the next iteration of the sink.

Per-Partition Connection

The key improvement in Listing 6-5 is to tease out the creation of the connection from the per-record `foreach` and move it to a `foreachPartition` (line 2) construct. This enables a per-partition connection model. This connection is re-created for each partition, which can be further improved on by maintaining a partition-invariant static connection object. This static connection is reused as long as the executor JVM is up.

Listing 6-5. Per-Partition Stateful Connection Creation in `foreachRDD`

```
.foreachRDD { rdd =>
  rdd.foreachPartition { par =>
    val client = new MqttClient(outputBrokerUrl, MqttClient.generateClientId(), new MemoryPersistence())
    client.connect()
    par.foreach(rec => client.publish(topic, new MqttMessage(rec.getBytes(StandardCharsets.UTF_8))))
    client.disconnect()
    client.close()
  }
}
```

The use of a static connection also has the implication that it is used concurrently by tasks in the same executor. Therefore, the connection object should be thread-safe. This design is more involved, because it entails instance objects and automatic decommission; it is at the center of the next iteration of the sink.

Static Connection

The first order of the day is to create a static object for the sink using the built-in Scala `apply`[10] method to return a concrete singleton implementation (Listing 6-6). The MQTT client singleton is created and initiated first (lines 3–4). Now that the client object is a singleton, you have no control over its lifecycle. In other words, you want to keep reusing the same static connection, perhaps forever. The only time the connection needs to be closed is when the executor is exiting. So, you register a JVM shutdown hook to release the connection state (lines 5–7).

Listing 6-6. Singleton Object Instance for the MQTT Client Connection

```
1.   object MqttSink {
2.     val brokerUrl = "tcp://localhost:1883"
3.     val client = new MqttClient(brokerUrl, MqttClient.generateClientId(), new MemoryPersistence())
4.     client.connect()
5.     sys.addShutdownHook {
6.       client.disconnect()
7.       client.close()
8.     }
9.
10.    def apply(): MqttClient = {
11.      client
12.    }
13.  }
```

This sink can then directly be invoked in `foreachPartition`, nested in a `foreachRDD` as shown in Listing 6-7. This serves the purpose of keeping static connections on the worker nodes.

[10]Typically used in Scala to return a concrete instance of a class.

Listing 6-7. Using a Static Connection in `foreachRDD`

```
.foreachRDD { rdd =>
  rdd.foreachPartition { par =>
    par.foreach(message => MqttSink().publish(topic, new MqttMessage(message.
    getBytes(StandardCharsets.UTF_8))))
  }
}
```

This solution has a minor drawback: the connection to the remote server is established regardless of whether the client is referenced: for instance, if no data is ever received at the executors. Let's try to remedy that by using lazy evaluation to establish the connection only the first time it is referenced.

Lazy Static Connection

The implementation in Listing 6-8 is very similar to the one in Listing 6-6, except that the creation of the connection object is performed in a lazy fashion (line 2) in a custom serializable class. That is, the connection is established only when it is referenced the very first time (in the executor). This enables you to create the singleton in the driver and only pass a creation template to the worker.[11]

Listing 6-8. Creating an MQTT Client Connection Singleton Using Lazy Evaluation

```
1.   class MqttSinkLazy(brokerUrl: String) extends Serializable {
2.     lazy val client = {
3.       val client = new MqttClient(brokerUrl, MqttClient.generateClientId(), new
         MemoryPersistence())
4.       client.connect()
5.       sys.addShutdownHook {
6.         client.disconnect()
7.         client.close()
8.       }
9.       client
10.    }
11.  }
12.
13.  object MqttSinkLazy {
14.    val brokerUrl = "tcp://localhost:1883"
15.    val client = new MqttSinkLazy(brokerUrl)
16.
17.    def apply(brokerUrl: String): MqttSinkLazy = {
18.      client
19.    }
20.  }
```

At the Spark Streaming end (Listing 6-9), to ensure that the object is sent only once to the workers, you send it via a broadcast variable (line 1). In the `foreachRDD`, you simply use that broadcasted variable to publish messages to MQTT (line 4).

[11]Marcin Kuthan, "Spark and Kafka Integration Patterns," *Allegro Tech*, August 6, 2015, `http://allegro.tech/2015/08/spark-kafka-integration.html`.

Listing 6-9. Using a Broadcast Variable to Create a Connection Object in the Driver and Use It in `foreachRDD`

```
val mqttSink = ssc.sparkContext.broadcast(MqttSinkLazy(outputBrokerUrl))
....
.foreachRDD { rdd =>
  rdd.foreachPartition { par => par.foreach(message => mqttSink.value.client.
  publish(topic, new MqttMessage(message.getBytes(StandardCharsets.UTF_8))))

  }
}
```

This design may lead to contention between tasks in an executor because they all rely on a single instance of an object. Let's try to reduce this overhead by maintaining a pool of static connection objects that each partition can use across batches.[12] You use the Apache Commons Pool library[13] to perform the actual pooling.

Static Connection Pool

`BasePooledObjectFactory` is the basic base class for pooled objects under Apache Commons. Listing 6-10 extends it to support `MqttClient` objects (line 15). This involves providing hooks for the lifecycle of the connection object. You then create a static pool for pooled objects (line 4), with the appropriate hook for reclamation (lines 6–8). The maximum number of objects in the pool is also controllable and should be set to the number of cores (or vcores in the case of YARN) per executor (line 5). Recall from Chapter 4 that the maximum number of tasks per executor is dictated by the number of cores assigned to that executor. As a result, the maximum size of the connection pool should reflect this number. The pool instance singleton is accessible via the `apply` method (lines 10–12).

Listing 6-10. Implementing a Static Pool to Amortize the Cost of `MqttClient` Connection Objects

```
1.   object MqttSinkPool {
2.     val poolSize = 8
3.     val brokerUrl = "tcp://localhost:1883"
4.     val mqttPool = new GenericObjectPool[MqttClient](new MqttClientFactory(brokerUrl))
5.     mqttPool.setMaxTotal(poolSize)
6.     sys.addShutdownHook {
7.       mqttPool.close()
8.     }
9.
10.    def apply(): GenericObjectPool[MqttClient] = {
11.      mqttPool
12.    }
13.  }
14.
15.  class MqttClientFactory(brokerUrl: String) extends BasePooledObjectFactory[MqttClient] {
16.    override def create() = {
17.      val client = new MqttClient(brokerUrl, MqttClient.generateClientId(), new
         MemoryPersistence())
```

[12]https://gist.github.com/koen-dejonghe/39c10357607c698c0b04.
[13]https://commons.apache.org/proper/commons-pool/.

```
    client.connect()
    client
  }
  override def wrap(client: MqttClient) = new DefaultPooledObject[MqttClient](client)
  override def validateObject(pObj: PooledObject[MqttClient]) = pObj.getObject.isConnected()
  override def destroyObject(pObj: PooledObject[MqttClient]) = {
    pObj.getObject.disconnect()
    pObj.getObject.close()
  }
  override def passivateObject(pObj: PooledObject[MqttClient]) = {}
}
```

In the foreachRDD construct (Listing 6-11), you borrow a connection object for each partition (line 3) and return it once all the records in a partition have been processed (line 5).

Listing 6-11. Using a Static Pool Object in a foreachRDD

```
1.   .foreachRDD { rdd =>
2.     rdd.foreachPartition { par =>
3.       val mqttSink = MqttSinkPool().borrowObject()
4.       par.foreach(message => mqttSink.publish(topic, new MqttMessage(message.
         getBytes(StandardCharsets.UTF_8))))
5.       MqttSinkPool().returnObject(mqttSink)
6.     }
7.   }
```

The use of the various design patterns outlined in this section depends on the particular requirements of your application (see Table 6-1). One thing should be clear, though: connection objects should be reused as aggressively as possible.

Table 6-1. *Comparison of Design Patterns for Maintaining Connection Objects*

Design Pattern	Advantage	Disadvantage
Per-record connection	Does not need to be serialized and sent over to executors from the driver.	Very inefficient because a connection per record is created and destroyed.
Per-partition connection	Extends the lifecycle of a connection to cover an entire partition instead of a record.	Begins stressing resources when the number of partitions increases.
Static connection	A single connection per JVM is maintained.	The connection is established regardless of whether it is used.
Lazy static connection	A connection is established only the first time the client object is used.	Thread-safety leads to resource contention turning the connection into a bottleneck.
Static connection pool	A pool of static connections is used throughout.	

Now that you have had a deep dive into foreachRDD operations, the next section looks at emitting data to two popular NoSQL stores: HBase and Cassandra.

Scalable Streaming Storage

One of the most common use cases for streaming applications is to drive business-critical dashboards. In the case of stock market analysis, a typical example is to provide a moving-average price of securities. The average is calculated over a specific number of time intervals to smooth out any noise and to observe long-term trends. This trend is then relayed to a dashboard to enable an investor to make a buy/sell decision.

This can easily be implemented using a Spark Streaming application. A simple design consists of calculating the moving average in the application and emitting its result to a socket. At the other end of the socket, a dashboard can read data via a `select` call and display the contents. This design is shown in Figure 6-2.

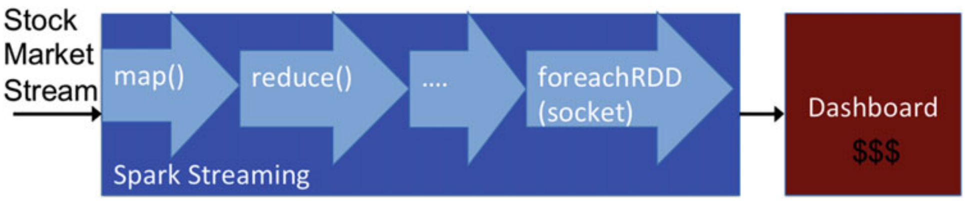


Figure 6-2. *Stock market dashboard powered by Spark Streaming*

This design has two major shortcomings: it lacks fault tolerance, so data will be lost if either the Spark application or the dashboard crashes; and it only provides instantaneous analysis, so historical data cannot be analyzed. The first shortcoming can be resolved by using a message queue solution such as Kafka. The second, on the other hand, requires using a persistent store. You can use HBase to act as a buffer and a store to hold historical data: Spark Streaming writes its output to HBase, and the dashboard reads data from HBase only after the data has been replicated (Figure 6-3).

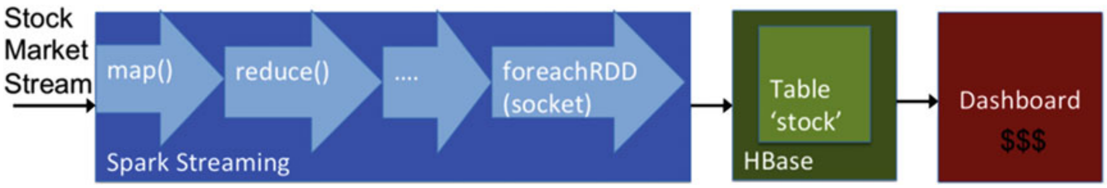


Figure 6-3. *Stock market dashboard powered by Spark Streaming and HBase*

HBase

HBase can be thought of as a distributed, persisted, and sorted multidimensional map. It is the open source variant of Google's Bigtable.[14] In HBase, data is laid out in tables, in the form of cells, which are indexed by row and column keys. Row keys are sorted in alphabetical order. Therefore, the choice of row key can affect performance, as you see shortly. All operations—reads and writes—are atomic at the row-key level. Columns are grouped into column families, which is the unit of access control and compression.

[14]Fay Chang et al., "Bigtable: A Distributed Storage System for Structured Data," *Proceedings of OSDI '06*, 7 (USENIX Association, 2006).

Columns that need to be accessed together are typically stored in the same column family for efficient access. Each cell also has an associated timestamp, which either is generated by the system at insertion time or is user defined. This means a cell may have many temporal versions. Older versions are garbage-collected by the system based on a predefined policy. Any sort of data can be persisted in a cell, ranging from strings and counters to complex types. The HBase subsystem stores all records as byte arrays, so any type that can be converted to a byte array can be persisted. The API consists of simple key-based `get` and `put` operations, although other complex operations, such as `scan` and `append`, are also supported.

Under the hood, HBase slices rows into regions, each of which stores a range of rows. A region is the basic unit of distribution, replication, and fault-tolerance. HBase uses HDFS behind the scenes to store all of its control and user data. As a result, replication and fault-tolerance are delegated to HDFS. Architecturally, HBase consists of a central `HMaster`, which is in charge of region distribution, slave coordination, and HDFS file management. At any given time, only a single `HMaster` is active, whereas a number of them may be running in passive mode to ensure high availability. Inter-`HMaster` coordination is implemented via ZooKeeper. Each region is entrusted to an `HRegionServer`, which arbitrates reads and writes to regions and is co-located with HDFS data nodes. `HRegionServers` also cache data to ensure good performance. A log-structure merge-tree-based representation enables efficient writes.

To understand the data layout design of HBase in concrete terms, let's walk through a real-world example. Suppose you want to store stock information for all the companies on Yahoo Finance. In financial applications, stocks from each industry vertical—technology, finance, e-commerce, and so on—are typically accessed together. Therefore, you exploit this property by prepending an abbreviation of the vertical to the name of the company, delimited by a period: for example, `tech.ibm` and `tech.goog`. This ensures that companies from the same vertical end up in the same region. In addition, you group similar values into column entities. All values related to price—last trade price, price paid, and so on—are stored in the `price` column family, and all volume-centric values are stored in the `volume` column family. This layout is shown in Table 6-2.

Table 6-2. *Schema of Yahoo Finance Data in HBase*

Row key	Column Family: Price		Column Family: Volume	
	LastTrade PriceOnly	**PriceSales**	**Volume**	**AverageDaily Volume**
fin.gs	187.01	2.42	2573727	3216030
fin.jpm	63.90	2.65	13880244	16479900
fin.ms	32.55	1.80	10952312	13008400
…	…	…		
tech.appl	115.28	3.03	66340901	59367800
tech.fb	103.77	19.67	25962155	32084300
tech.goog	712.78	6.73	2716615	2342240
…	…	…		

Stock Market Dashboard

Before your Spark Streaming application can be triggered, you need to set up HBase.[15] The goal is to emit data to a table, so the first task is to create a table named `stock` with column family `avgprice`. This can be done via the HBase shell using this command:

```
create 'stock', 'avgprice'
```

Next, the following dependencies need to be added to the `.sbt` file for the application project for client dependencies:

```
libraryDependencies += "org.apache.hbase" % "hbase-client" % "1.1.2"
libraryDependencies += "org.apache.hbase" % "hbase-server" % "1.1.2"
libraryDependencies += "org.apache.hbase" % "hbase-common" % "1.1.2"
```

You are now ready to begin implementing Spark code to calculate a moving average for the ten technology companies mentioned earlier using your custom HTTP receiver. This code is presented in Listing 6-12.

Listing 6-12. Calculating a Moving Average for Stock Price Values and Emitting Them to HBase

```
HttpUtils.createStream(ssc, url = "https://query.yahooapis.com/v1/public/yql?q=select%20*%20from%20yahoo.finance.quotes%20where%20symbol%20in%20(%22IBM,GOOG,MSFT,AAPL,FB,ORCL,YHOO,TWTR,LNKD,INTC%22)%0A%09%09&format=json&diagnostics=true&env=http%3A%2F%2Fdatatables.org%2Falltables.env",
  interval = batchInterval)
  .flatMap(rec => {
    implicit val formats = DefaultFormats
    val query = parse(rec) \ "query"
    ((query \ "results" \ "quote").children)
      .map(rec => ((rec \ "symbol").extract[String], (rec \ "LastTradePriceOnly").extract[String].toFloat))
  })
  .reduceByKeyAndWindow((x: Float, y: Float) => (x + y), Seconds(windowSize), Seconds(slideInterval))
  .foreachRDD(rdd => {
    val hbaseConf = HBaseConfiguration.create()
    hbaseConf.set(TableOutputFormat.OUTPUT_TABLE, tableName)
    hbaseConf.set("hbase.master", hbaseMaster);
    val jobConf = new Configuration(hbaseConf)
    jobConf.set("mapreduce.job.outputformat.class", classOf[TableOutputFormat[Text]].getName)
    rdd.map(rec => {
      val put = new Put(rec._1.getBytes)
      put.addColumn(columnFamilyName.getBytes, columnName.getBytes, Bytes.toBytes(rec._2 / (windowSize / batchInterval)))
      (rec._1, put)
    }).saveAsNewAPIHadoopDataset(jobConf)
  })
```

[15] Download HBase (ver 1.1.2) from https://hbase.apache.org/ and run it via `$HBASE_HOME/bin/start_hbase.sh`. The shell can be accessed via `$HBASE_HOME/bin/hbase shell`. Note that the default settings constitute a test setup and should not be used in production. For details of a multinode production-grade installation, please consult the HBase documentation.

Of the 80+ fields, only `LastTradePriceOnly` is required to calculate a moving average; therefore you extract and project only this field, along with the stock symbol name (lines 3–8). You then perform a moving-sum calculation using a `reduceByKeyAndWindow` operation. `windowSize` dictates the length of the moving sum, and `slideInterval` its granularity (line 9). This sum is now ready to be pushed to HBase (after dividing it by `windowSize`, of course). This requires the use of your trusted friend `foreachRDD`. The actual flush to HBase is enabled by `saveAsNewAPIHadoopDataset()` for each RDD.

`saveAsNewAPIHadoopDataset()`, if you recall from Chapter 2, can be used to emit data to any Hadoop-compatible store, including HBase. This function takes as input a Hadoop configuration file (line 14), which contains details about the HBase `HMaster` (line 13) and table (line 12) and the output format (line 15). `TableOutputFormat`[16] describes an HBase table, and its parameterization dictates the format of the row key: Text, in this case. The default output format for the value is `ImmutableBytesWritable`, which means you need to convert your output `float` to a byte array. Because you are inserting records into HBase, you use the `Put` mutation (line 17) and provide it with the row key. For each column, you also need to specify the column family name, qualifier, and value (line 18). Note that the sum is converted to an average by dividing it by the window size (the temporal value of the window normalized by the batch interval) just before it is written to the table. This operation needs to be repeated for each record in the RDD (lines 16–20). Once data starts landing in HBase, its presence can be verified by performing a `scan` of the stock table, as shown in Listing 6-13.

Listing 6-13. Performing a Scan of the Stock Table in HBase

```
hbase(main):002:0> scan 'stock'
ROW                               COLUMN+CELL
 AAPL                             column=avgprice:price, timestamp=1446095400305,
                                  value=A\x8F\x1F\xBE
 FB                               column=avgprice:price, timestamp=1446095400305,
                                  value=Az\x14z
 GOOG                             column=avgprice:price, timestamp=1446095400305,
                                  value=B\xD5\xE2\x90
 IBM                              column=avgprice:price, timestamp=1446095400305,
                                  value=A\xA8\xFE\xFA
 INTC                             column=avgprice:price, timestamp=1446095400305,
                                  value=@\xA6\x9B\xA6
 LNKD                             column=avgprice:price, timestamp=1446095400288,
                                  value=A\xFF\xDA\x1D
 MSFT                             column=avgprice:price, timestamp=1446095400305,
                                  value=A\x01\x8DP
 ORCL                             column=avgprice:price, timestamp=1446095400305,
                                  value=@\xBA\x9F\xBE
 TWTR                             column=avgprice:price, timestamp=1446095400288,
                                  value=@\x94-\x0E
 YHOO                             column=avgprice:price, timestamp=1446095400288,
                                  value=@\xA8\xE3U
10 row(s) in 0.0360 seconds
```

[16] `org.apache.hadoop.hbase.mapreduce.TableOutputFormat`.

SparkOnHBase

Using HBase as a Hadoop-compatible file system permits you to read from and write to it in a Spark Streaming application, but this is a loose coupling at best. The API is very verbose, and the RDDs and `DStreams` are HBase agnostic. SparkOnHBase[17] a project from Cloudera, aims to remedy this by providing native support for HBase. SparkOnHBase enables the creation of RDDs from HBase scans; bulk-load, `put`, `get`, and `delete` operations from RDDs and `DStreams`; and per-partition `map` and `foreach` operations with access to an HBase connection, among others. Let's try to simplify the implementation of the previous example using SparkOnHBase (see Listing 6-14).

Listing 6-14. Using SparkOnHBase to Perform a Bulk HBase `put` Operation

```
1.    val hbaseConf = HBaseConfiguration.create()
2.    val hContext = new HBaseContext(ssc.sparkContext, hbaseConf)
3.
4.    val windowed = HttpUtils.createStream(ssc, url = "https://query.yahooapis.com/v1/public/
      yql?q=select%20*%20from%20yahoo.finance.quotes%20where%20symbol%20in%20(%22IBM,GOOG,MSFT
      ,AAPL,FB,ORCL,YHOO,TWTR,LNKD,INTC%22)%0A%09%09&format=json&diagnostics=true&env=http%3A%
      2F%2Fdatatables.org%2Falltables.env",
5.      interval = batchInterval)
6.      .flatMap(rec => {
7.        implicit val formats = DefaultFormats
8.        val query = parse(rec) \ "query"
9.        ((query \ "results" \ "quote").children)
10.         .map(rec => ((rec \ "symbol").extract[String], (rec \ "LastTradePriceOnly").
          extract[String].toFloat))
11.      })
12.      .reduceByKeyAndWindow((x: Float, y: Float) => (x + y), Seconds(windowSize),
         Seconds(slideInterval))
13.
14.    hContext.streamBulkPut[(String, Float)](windowed, TableName.valueOf(tableName), rec => {
15.      val put = new Put(rec._1.getBytes)
16.      put.addColumn(columnFamilyName.getBytes, columnName.getBytes, Bytes.toBytes(rec._2 /
         (windowSize / batchInterval)))
17.      put
18.    })
```

The entry point for SparkOnHBase is an `HBaseContext` (line 2) object that requires a `SparkContext` and an `HBaseConfiguration` for initialization. This can be used to perform most operations included in the library. For instance, instead of relying on copious amounts of boilerplate code in a `foreachRDD` transform, a single `streamBulkPut` (line 14) transform can be used to spit out data to HBase. The third argument to `streamBulkPut` needs to be a function that is given records from the `DStream` as input and is required to return a `Put` object.

SparkOnHBase was merged upstream into the HBase codebase in 2015[18] but is available only in the `2.0.0-SNAPSHOT` version of HBase at the time of writing. Therefore, to use it, add the following to your build definition file:

[17]Ted Malaska, "New in Cloudera Labs: SparkOnHBase," *Cloudera*, December 18, 2014, http://blog.cloudera.com/blog/2014/12/new-in-cloudera-labs-sparkonhbase/.
[18]Ted Malaska, "Apache Spark Comes to Apache HBase with HBase-Spark Module," *Cloudera*, August 13, 2015, http://blog.cloudera.com/blog/2015/08/apache-spark-comes-to-apache-hbase-with-hbase-spark-module/.

```
libraryDependencies += "org.apache.hbase" % "hbase-client" % "2.0.0-SNAPSHOT"
libraryDependencies += "org.apache.hbase" % "hbase-server" % "2.0.0-SNAPSHOT"
libraryDependencies += "org.apache.hbase" % "hbase-common" % "2.0.0-SNAPSHOT"
libraryDependencies += "org.apache.hbase" % "hbase-spark" % "2.0.0-SNAPSHOT"
resolvers += "Apache Snapshot Repository" at "https://repository.apache.org/content/
repositories/snapshots"
```

HBase has strong consistency guarantees. These at times can affect write performance, whereas reads are extremely fast. Depending on the use case and the data semantics, Cassandra, another NoSQL store, can achieve better write performance. This is not always true, though; the best method to choose a data store is to base the decision on benchmarks and application and data requirements.

For completeness, let's emit data to Cassandra instead of HBase.

Cassandra

Similar to HBase, Cassandra is also a distributed, multidimensional sorted map in which values are indexed by key. Unlike HBase, which relies on HDFS, Cassandra implements its own replication strategy for fault tolerance and high availability. As a result, write operations are atomic at the key level for each replica. Tables in Cassandra are called *column families*, which are nested sorted maps: the outer map is indexed by row key, and the inner one is indexed by column key. This makes lookups very efficient. Column families reside in *keyspaces*, which are similar to the concept of a database. Cassandra also supports super-column families, which are just nested columns—a third-level map. The use of multidimensional maps also simplifies the API: typical operations are `set`, `get`, `update`, and `del`.

Cassandra has a completely decentralized model wherein each request is routed to any node in the cluster, which in turn determines the final hop: the node, which holds a replica for the request key. The selection of this replica is dictated by the read/write semantics of the requests. It uses consistent hashing (over a hypothetical ring) to partition data across nodes. The replication factor is configurable per keyspace. Due to its decentralized model, there is no central node orchestration mechanism. Instead, nodes use a gossip-based technique to share cluster-membership information. Data is stored on the local file system at each node, supported by log-structured merge-trees (similar to HBase).

You now have enough background information about Cassandra to get your hands dirty. Let's start by setting up the environment[19] to dump data from the stock market application using Cassandra's Thrift API-based CLI (Listing 6-15).

Listing 6-15. Creating a Column Family in Cassandra

```
create keyspace stock;
use stock;
create column family avgprice with comparator = UTF8Type;
update column family avgprice with column_metadata = [{column_name: price,
validation_class: FloatType}];
assume avgprice keys as utf8;
```

The commands create a keyspace called `stock` with a single column family dubbed `avgprice`. The data type for the column name is a simple UTF string.[20] The next step is to add a `price` column with type `float` to the `avgprice` family. This column family is now ready to receive data.

[19]Cassandra (ver 2.1.11) can be downloaded from `http://cassandra.apache.org/`. To start using the default single-node configuration, use `$CASSANDRA_HOME/bin/cassandra start`. The Cassandra CLI can be started via `$CASSANDRA_HOME/bin/cassandra-cli`.

[20]Data types for column values are called *validators*, and data types for column names are called *comparators*.

Using Cassandra libraries in Spark Streaming requires adding the following line to your `.sbt` file:

```
libraryDependencies += "org.apache.cassandra" % "cassandra-all" % "2.1.11"
```

Listing 6-16 lists a `foreachRDD` transform for Cassandra to replace the HBase one in Listing 6-12.

Listing 6-16. Emitting Data to a Cassandra Column Family

```
1.   .foreachRDD(rdd => {
2.           val jobConf = new Configuration()
3.           ConfigHelper.setOutputRpcPort(jobConf, cassandraPort)
4.           ConfigHelper.setOutputInitialAddress(jobConf, cassandraHost)
5.           ConfigHelper.setOutputColumnFamily(jobConf, keyspace, columnFamilyName)
6.           ConfigHelper.setOutputPartitioner(jobConf, "Murmur3Partitioner")
7.           rdd.map(rec => {
8.             val c = new column()
9.             c.setName(ByteBufferUtil.bytes(columnName))
10.            c.setValue(ByteBufferUtil.bytes(rec._2 / (windowSize / batchInterval)))
11.            c.setTimestamp(System.currentTimeMillis)
12.            val m = new Mutation()
13.            m.setColumn_or_supercolumn(new ColumnOrSuperColumn())
14.            m.column_or_supercolumn.setColumn(c)
15.            (ByteBufferUtil.bytes(rec._1), Arrays.asList(m))
16.          }).saveAsNewAPIHadoopFile(keyspace, classOf[ByteBuffer],
             classOf[List[Mutation]], classOf[ColumnFamilyOutputFormat], jobConf)
17.        })
```

Similar to HBase, the first order of the day is to set up configuration parameters (lines 2–6) that set the location of the Cassandra cluster, the name of the keyspace and column family, and the output partitioner (which decides how to divvy up data across the cluster). You use the `MurmurHash`-based partitioner, which uniformly distributes the data across nodes. Mutations to each column in a row need to be specified explicitly. The column family only has a single column, for which you need to specify a name, value, and timestamp (lines 8–11). It then needs to be wrapped in a mutation object (lines 12–14) and returned as the value in a `PairRDD` `foreachRDD` operation (line 15). You finally invoke `saveAsNewAPIHadoopFile` on each RDD to emit it to Cassandra, where it lands in the avgprice column family. Listing 6-17 shows the output of a `list` operation on this column family via the Cassandra Thrift CLI.

Listing 6-17. Stock Ticker Application Output in Cassandra

```
[default@stock] list avgprice;
Using default limit of 100
Using default cell limit of 100
-------------------
RowKey: TWTR => (name=price, value=29.06, timestamp=1446175330214)
-------------------
RowKey: AAPL => (name=price, value=120.53, timestamp=1446175330215)
-------------------
RowKey: FB => (name=price, value=104.88, timestamp=1446175330215)
-------------------
RowKey: ORCL => (name=price, value=38.86, timestamp=1446175330214)
-------------------
RowKey: INTC => (name=price, value=34.03, timestamp=1446175330213)
-------------------
RowKey: YHOO => (name=price, value=35.05, timestamp=1446175330214)
```

```
-------------------
RowKey: IBM => (name=price, value=140.55, timestamp=1446175330214)
-------------------
RowKey: GOOG => (name=price, value=716.92, timestamp=1446175330215)
-------------------
RowKey: LNKD => (name=price, value=217.0, timestamp=1446175330214)
-------------------
RowKey: MSFT => (name=price, value=53.36, timestamp=1446175330214)
10 Rows Returned.
Elapsed time: 6.76 msec(s).
```

Spark Cassandra Connector

The explicit foreachRDD based approach to spit out data to Cassandra (or any other Hadoop-compatible data store) is very involved: it requires you to reason about the output format, data types, and mutations. Fortunately, in the case of Cassandra, DataStax (the startup that provides commercial support for Cassandra) has a connector for Spark that greatly simplifies the ingress and egress process.[21] To use it, add the following to your build definition file:

```
libraryDependencies += "com.datastax.spark" %% "spark-cassandra-connector" % "1.4.0"
```

You also need to import com.datastax.spark.connector.streaming._ and com.datastax.spark.connector._ into your Spark application. The location of the Cassandra cluster is specified by setting spark.cassandra.connection.host and spark.cassandra.connection.port in SparkConf.

The Thrift API for Cassandra is almost deprecated; the new interface revolves around Cassandra Query Language (CQL). CQL provides a SQL-like interface to query Cassandra and ships with a command-line tool, cqlsh, which is similar in spirit to the previous CLI. Be mindful of the port, though, when setting spark.cassandra.connection.port, because the CQL interface (default port: 9042) has a different port than the Thrift API (default: 9160).

The DataStax connector also enables you to directly invoke CQL. This is extremely useful for tasks such as creating tables.[22] Listing 6-18 contains the updated code, which uses this connector. In lines 1–4, you set up the keyspace and table for your application. conf is the SparkConf object. Lines 6–14 are identical to the previous two examples and are only provided for completeness. Emitting data to Cassandra is just a matter of invoking the saveToCassandra method and giving it the keyspace and table name (line 16).

Listing 6-18. Emitting Data to Cassandra Using the Spark Cassandra Connector from DataStax

```
1.   CassandraConnector(conf).withSessionDo { session =>
2.     session.execute(s"CREATE KEYSPACE IF NOT EXISTS %s WITH REPLICATION = {'class':
       'SimpleStrategy', 'replication_factor': 1 }".format(keyspace))
3.     session.execute(s"CREATE TABLE IF NOT EXISTS %s.%s (key TEXT PRIMARY KEY, %s FLOAT)".
       format(keyspace, tableName, columnName))
4.   }
5.
6.   HttpUtils.createStream(ssc, url = "https://query.yahooapis.com/v1/public/
     yql?q=select%20*%20from%20yahoo.finance.quotes%20where%20symbol%20in%20(%22IBM,GOOG,MSF
     T,AAPL,FB,ORCL,YHOO,TWTR,LNKD,INTC%22)%0A%09%09&format=json&diagnostics=true&env=http%3
     A%2F%2Fdatatables.org%2Falltables.env",
```

[21] https://github.com/datastax/spark-cassandra-connector.

[22] In the CQL world, Cassandra column families are now called *tables*.

```
interval = batchInterval)
.flatMap(rec => {
  implicit val formats = DefaultFormats
  val query = parse(rec) \ "query"
  ((query \ "results" \ "quote").children)
    .map(rec => ((rec \ "symbol").extract[String], (rec \ "LastTradePriceOnly").
    extract[String].toFloat))
})
.reduceByKeyAndWindow((x: Float, y: Float) => (x + y), Seconds(windowSize),
Seconds(slideInterval))
.map(stock => (stock._1, stock._2 / (windowSize / batchInterval)))
.saveToCassandra(keyspace, tableName)
```

Listing 6-19 shows the output table via CQL.

Listing 6-19. Displaying the Stock Ticker Application Output Table in Cassandra via CQL

```
cqlsh:stock> SELECT * FROM avgprice2;

 key  | price
------+--------
 TWTR |  28.46
 AAPL |  119.5
   FB | 101.97
 ORCL |  38.84
 INTC |  33.86
 YHOO |  35.62
  IBM | 140.08
 GOOG | 710.81
 LNKD | 240.87
 MSFT |  52.64

(10 rows)
```

Global State

Spark Streaming batches are by design stateless: all transformations during a batch affect only the RDDs in that batch. The only exception to this rule is `updateStateByKey()`, which maintains state across RDDs. The downside is that only state for data, which occurs in the data stream itself, can be manipulated. This section sketches some recipes that apply stateful operations to batch-invariant RDDs.

Static Variables

The simplest state is in the form of counters. For instance, how can you keep track of the maximum and minimum stock volume across all securities, count the number of times any stock price has hit 500, and print a message when this counter has reached 1,000? You can use static variables in the driver program in tandem with `foreachRDD`. Specifically, you can take advantage of the fact that `foreachRDD` is invoked in the driver program, so if you update the value of any static variables, the state is applicable across RDDs. Listing 6-20 provides code that uses this approach.

Listing 6-20. Using Static Counters and foreachRDD to Maintain Statistics Across RDDs/Batches

```
1.   var globalMax: AtomicLong = new AtomicLong(Long.MinValue)
2.   var globalMin: AtomicLong = new AtomicLong(Long.MaxValue)
3.   var globalCounter500: AtomicLong = new AtomicLong(0)
4.
5.   HttpUtils.createStream(ssc, url = "https://query.yahooapis.com/v1/public/
     yql?q=select%20*%20from%20yahoo.finance.quotes%20where%20symbol%20in%20(%22IBM,GOOG,MSF
     T,AAPL,FB,ORCL,YHOO,TWTR,LNKD,INTC%22)%0A%09%09&format=json&diagnostics=true&env=http%3
     A%2F%2Fdatatables.org%2Falltables.env",
6.     interval = batchInterval)
7.     .flatMap(rec => {
8.       implicit val formats = DefaultFormats
9.       val query = parse(rec) \ "query"
10.      ((query \ "results" \ "quote").children)
11.        .map(rec => ((rec \ "symbol").extract[String], (rec \ "LastTradePriceOnly").
           extract[String].toFloat, (rec \ "Volume").extract[String].toLong))
12.    })
13.    .foreachRDD(rdd => {
14.      val stocks = rdd.take(10)
15.      stocks.foreach(stock => {
16.        val price = stock._2
17.        val volume = stock._3
18.        if (volume > globalMax.get()) {
19.          globalMax.set(volume)
20.        }
21.        if (volume < globalMin.get()) {
22.          globalMin.set(volume)
23.        }
24.        if (price > 500) {
25.          globalCounter500.incrementAndGet()
26.        }
27.      })
28.      if (globalCounter500.get() > 1000L) {
29.        println("Global counter has reached 1000")
30.        println("Max ----> " + globalMax.get)
31.        println("Min ----> " + globalMin.get)
32.        globalCounter500.set(0)
33.      }
34.    })
```

This code has two noteworthy aspects:

- It uses atomic variables to ensure that calculations remain atomic even in the face of concurrent access and modification.
- Projected data is ingested into the driver process (line 14). This works because you know the number of stock records and the amount of data is small enough to not overwhelm the driver node heap. Note that because all the data has been materialized in the driver program, the inner foreach (line 15) is executed in the driver JVM, not on the worker executors. That is why you can maintain global numbers.

This approach works well if you know the number of counters upfront and the amount of global state is small enough to fit in the driver JVM memory. What happens when one or both of these statements is not true? For instance, what if you wanted to keep the maximum and minimum for each stock value in the data stream but did not know the stock symbols of interest beforehand? Let's look at some alternatives.

updateStateByKey()

The most obvious choice is `updateStateByKey()`, which allows you to track the state of keys across RDDs. Using a custom `update` function, you can re-perform the `max`, `min`, and `count` calculation in each invocation (see Listing 6-21).[23] In contrast to the previous example, all state manipulation takes place in the `update` function (lines 12–22). The batch-invariant variable for each key is a tuple of the form `(min, max, count)`. The current value of this tuple is passed to the `update` function as the second argument. This can be enhanced further to perform any arbitrary computation and hold any collection, such as a map.

Listing 6-21. Using `updateStateByKey()` for Per-Key Statistics Across RDDs

```
1.   HttpUtils.createStream(ssc, url = "https://query.yahooapis.com/v1/public/
     yql?q=select%20*%20from%20yahoo.finance.quotes%20where%20symbol%20in%20(%22IBM,GOOG,MSF
     T,AAPL,FB,ORCL,YHOO,TWTR,LNKD,INTC%22)%0A%09%09&format=json&diagnostics=true&env=http%3
     A%2F%2Fdatatables.org%2Falltables.env",
2.     interval = batchInterval)
3.     .flatMap(rec => {
4.       implicit val formats = DefaultFormats
5.       val query = parse(rec) \ "query"
6.       ((query \ "results" \ "quote").children)
7.         .map(rec => ((rec \ "symbol").extract[String], ((rec \ "LastTradePriceOnly").
           extract[String].toFloat, (rec \ "Volume").extract[String].toLong)))
8.     })
9.     .updateStateByKey(updateState)
10.    .print()
11.
12.  def updateState(values: Seq[(Float, Long)], state: Option[(Long, Long, Long)]):
     Option[(Long, Long, Long)] = {
13.    val volumes = values.map(s => s._2)
14.    val localMin = volumes.min
15.    val localMax = volumes.max
16.    val localCount500 = values.map(s => s._1).count(price => price > 500)
17.    val globalValues = state.getOrElse((Long.MaxValue, Long.MinValue, 0L)).
       asInstanceOf[(Long, Long, Long)]
18.    val newMin = if (localMin < globalValues._1) localMin else globalValues._1
19.    val newMax = if (localMax > globalValues._2) localMax else globalValues._2
20.    val newCount500 = globalValues._3 + localCount500
21.    return Some(newMin, newMax, newCount500)
22.  }
```

[23]Don't forget to set a checkpoint directory.

`updateStateByKey()` works well if the number of keys and the amount of RDD-invariant state is small. This is primarily because RDDs and `DStreams` are immutable. On the plus side, this simplifies fault-tolerance: Spark can regenerate any lost RDD by replaying data from the checkpoint. On the downside, this means any state maintained via `StateDStream` (the `DStream` that makes `updateStateByKey` operations possible) needs to be regenerated in every batch. Imagine if you are tracking a million keys but only a handful of them need to be updated in every micro-batch. It is clearly overkill and suboptimal to create new copies of keys that have not been mutated. Can you do better?

Accumulators

Accumulators, if you recall, are shared variables that support associative operations. This enables them to perform operations in parallel. One key property of accumulators is that only the driver program can read the value of an accumulator; workers can only add to it. They are extremely useful for calculating counts and sums across RDDs. In addition to these simple operations, accumulators can also contain Scala collections. For instance, you can create a `HashMap`-based accumulator as follows

```
mapAcc  = StreamingContext#sparkContext.accumulableCollection(mutable.HashMap[String, Int]())
```

and then add items to it:

```
mapAcc += (keyStr -> valInt)
```

This insertion of values is typically performed in a `foreachRDD`.

In the example use case, you need to keep track of a few statistics. So rather than embedding this logic in the core streaming flow, let's create a custom accumulator to provide this functionality. Spark provides two interfaces for implementing custom accumulators: `AccumulatorParam` and `AccumulableParam`. The former is used when the value to be added to the accumulator is the same as the accumulated value. For example, if the accumulator is a 2-tuple of `ints`, then only 2-tuple `ints` can be added to it. In contrast, `AccumulableParam` allows a different type for the added value. In fact, under the hood, `AccumulatorParam` is just syntactic sugar on top of `AccumulableParam`.

You need to use `AccumulableParam`, because the input value (`"stockPrice": Float, "stockVolume": Long`) is different than the values you need to accumulate (`"maxVolume": Long, "minVolume": Long, "priceCounter": Long`). In addition, these metrics need to be maintained per stock symbol.

Each concrete implementation of `AccumulableParam` needs to override the following three methods:

- `zero()`: The identity value of the accumulator.
- `addAccumulator()`: Adds a single value.
- `addInPlace()`: Merges two accumulators. Invoked each time the values of sharded accumulators from different tasks need to be aggregated.

The code for the custom `StockAccum` accumulator is presented in Listing 6-22. Internally, it maintains a hash map indexed by the stock symbol to hold per-stock stats. The second `type` parameter to `AccumulableParam` needs to represent the value to be added, which in this case is a 2-tuple of this form:

```
(String, (Float, Long)): ("stockSym", ("stockPrice", "stockVolume"))
```

In the identity-initialization method (line 2), you create a new HashMap. To enable two accumulators to be merged, the addInPlace method (line 5) compares each stock symbol in the two maps and copies over the max and min of the two to the first map, respectively. For the counter, it simply adds the values from the maps. Finally, the addAccumulator method (line 15) adds the values to the previous set of values for a particular stock symbol (or initializes it if the symbol is encountered for the first time). The counter is incremented by checking whether the current stock price exceeds 500 (line 19).

Listing 6-22. Custom Accumulator to Keep Track of Global Stock Stats

```
1.   object StockAccum extends AccumulableParam[mutable.HashMap[String, (Long, Long, Long)],
     (String, (Float, Long))] {
2.     def zero(t: mutable.HashMap[String, (Long, Long, Long)]): mutable.HashMap[String,
       (Long, Long, Long)] = {
3.       new mutable.HashMap[String, (Long, Long, Long)]()
4.     }
5.     def addInPlace(t1: mutable.HashMap[String, (Long, Long, Long)], t2: mutable.
       HashMap[String, (Long, Long, Long)]): mutable.HashMap[String, (Long, Long, Long)] = {
6.       t1 ++ t2.map {
7.         case (k, v2) => (k -> {
8.           val v1 = t1.getOrElse(k, (Long.MaxValue, Long.MinValue, 0L))
9.           val newMin = if (v2._1 < v1._1) v2._1 else v1._1
10.          val newMax = if (v2._2 > v1._2) v2._2 else v1._2
11.          (newMin, newMax, v1._3 + v2._3)
12.        })
13.      }
14.    }
15.    def addAccumulator(t1: mutable.HashMap[String, (Long, Long, Long)], t2:
       (String, (Float, Long))): mutable.HashMap[String, (Long, Long, Long)] = {
16.      val prevStats = t1.getOrElse(t2._1, (Long.MaxValue, Long.MinValue, 0L))
17.      val newVals = t2._2
18.      var newCount = prevStats._3
19.      if (newVals._1 > 500.0) {
20.        newCount += 1
21.      }
22.      val newMin = if (newVals._2 < prevStats._1) newVals._2 else prevStats._1
23.      val newMax = if (newVals._2 > prevStats._2) newVals._2 else prevStats._2
24.      t1 += t2._1 -> (newMin, newMax, newCount)
25.    }
26.  }
```

Listing 6-23 shows the use of the accumulator. It replaces the foreachRDD transform in Listing 6-20. In the inner foreach (which is executed on worker nodes), you add values to the accumulator (line 4). These values are subsequently displayed in the driver process by printing the hash map from the accumulator (line 7).

Listing 6-23. Adding Values to an Accumulator

```
val stateAccum = ssc.sparkContext.accumulable(new mutable.HashMap[String,
(Long, Long, Long)]())(StockAccum)
...
.foreachRDD(rdd => {
  rdd.foreach({ stock =>
    stateAccum += (stock._1, (stock._2._1, stock._2._2))
  })
  for ((sym, stats) <- stateAccum.value.to) printf("Symbol: %s, Stats: %s\n",
  sym, stats)
})
```

As you can see, accumulators are very easy to use and reason about. But at the same time, their functionality is limited: only the driver process can read from them, and only associative operations can be performed. What if the application needs to keep arbitrary state and operations?

External Solutions

One option for storing global values is to explicitly turn them into side effects and keep them in external storage. With this design, in each batch, previous state is read from external storage, transformed, and then written back. At the same time, this external storage needs to ensure low latency to match the performance of native in-JVM data structures. One such option is Redis, an in-memory key-value store.

Redis

Redis (REmote DIctionary Server) is simply an in-memory data-structure directory. It supports a wide range of common data types including lists, sets, and hash maps. In addition, Redis contains out-of-the-box implementations of advanced structures such as bitmaps and HyperLogLogs. It also enables direct manipulation of these data types. For instance, the hash data structure supports `set` and `get` operations. All of these structures are stored in memory for efficient lookup. To ensure fault-tolerance, they are periodically synced with disk.

Transactions are atomic at the command level, but different commands can be clumped explicitly into a single atomic transaction. Redis uses the asynchronous master-slave replication mode for redundancy and scalability. Furthermore, it supports a cluster mode wherein data is sharded across nodes.

Continuing the example application, you can use Redis to store stock volume and price metrics. For each stock symbol, the application will store the minimum and maximum volume and the price counter in a hash map. Client libraries for Redis exist for all major programming languages. For Java, the package of choice is Jedis. Once Redis has been set up,[24] add the following to your build definition file:

```
libraryDependencies += "redis.clients" % "jedis" % "2.7.3"
```

[24]Download Redis (ver 3.0.5) from `http://redis.io/download`, and build the project (`make`). Post-build run it with `$REDIS_HOME/src/redis-server`. To access the console, use `$REDIS_HOME/src/redis-cli`.

Listing 6-24 shows how Redis can be used to store arbitrary data structures from Spark Streaming applications. In the per-partition `foreach` (line 2), you connect to the Redis server using a Jedis client connection object that takes the hostname of the server as input (line 3). Then, for each record, you first need to check whether the stock symbol key exists in Redis (lines 4–5). If it does not, the `value` tuple is initialized with default values. Otherwise, the previous values stored in Redis are updated (lines 8–17) and written back (line 18). In each batch interval, these values are also emitted to standard output in the driver JVM (lines 24–26). If the number of keys is small, it may be more efficient to batch (or partition) them in a single pipelined call to Redis (obtained via `Jedis#pipelined()`). Wrapping Redis operations in a `foreachRDD` operation allows you to use the former to store boundless, batch-invariant state.

Listing 6-24. Keeping Spark Streaming Application State in Redis

```
1.   .foreachRDD(rdd => {
2.     rdd.foreachPartition({ part =>
3.       val jedis = new Jedis(hostname)
4.       part.foreach(f => {
5.         val prev = jedis.hmget(f._1, "min", "max", "count")
6.         if (prev(0) == null) {
7.           jedis.hmset(f._1, mutable.HashMap("min" -> Long.MaxValue.toString,
             "max" -> Long.MinValue.toString, "count" -> 0.toString))
8.         } else {
9.           val prevLong = prev.toList.map(v => v.toLong)
10.          var newCount = prevLong(2)
11.          val newPrice = f._2._1
12.          val newVolume = f._2._2
13.          if (newPrice > 500.0) {
14.            newCount += 1
15.          }
16.          val newMin = if (newVolume < prevLong(0)) newVolume else prevLong(0)
17.          val newMax = if (newVolume > prevLong(1)) newVolume else prevLong(1)
18.          jedis.hmset(f._1, mutable.HashMap("min" -> newMin.toString, "max" ->
             newMax.toString, "count" -> newCount.toString))
19.        }
20.      })
21.      jedis.close()
22.    })
23.
24.    val jedis = new Jedis(hostname)
25.    jedis.scan(0).getResult.foreach(sym => println("Symbol: %s, Stats: %s".format
       (sym, jedis.hmget(sym, "min", "max", "count").toString)))
26.    jedis.close()
27.  })
```

Summary

In Spark Streaming, side effects facilitated by `foreachRDD` signpost explicit points in the data flow to talk to external services and maintain cross-RDD state. The same interface can be used to integrate streaming applications with HBase, Cassandra, and Redis, among many others. Combined with the ability to ingest data from disparate sources, this empowers Spark Streaming to be the centerpiece of real-time processing pipelines from any domain. The use of external NoSQL solutions permits applications to persist their output while achieving high scalability, throughput, and availability.

Having examined different ingress and egress options for Spark Streaming, you can move on to the next leg of writing scalable streaming applications: logging, monitoring, and general optimizations.

CHAPTER 7

Getting Ready for Prime Time

Given enough eyeballs, all bugs are shallow.

—Linus's Law

Application development is an incremental and continuous process: once an application has been designed, implemented, and deployed, it needs to be constantly monitored and improved. The same applies to real-time pipelines, with additional variables: scalability and capacity. There may be an increase in the volume and velocity of the incoming data or lower latency requirements. Over time, as requirements change, initial design choices need to be reevaluated. Developers and infrastructure engineers clamor to squeeze the last bit of performance out of both the software stack and the hardware. Regardless of the cause and effect, all such projects require rigorous and generous instrumentation—from logging and monitoring to alerting and metrics.

This chapter equips you with frameworks, tools, and techniques to master Spark Streaming application instrumentation and management. The example is a clickstream analysis of data from Wikipedia. You kick off the chapter by offloading RDD management from executors to Tachyon, a general-purpose, in-memory file system. A deep dive into the Spark UI, as an enabler of application optimization, is also on the menu. Subsequent sections introduce and hone the instrumentation-first mantra. Now, let's make all bugs shallow.

Every Click Counts

Online businesses are interested in learning the behavior, habits, and wants of their visitors. This enables them to provide a personalized experience to each customer with the goal of increasing conversions via both targeting and re-targeting. Clickstream data (mashed up with third-party sources) underpins this category of analytics. A clickstream, as the name suggests, is a detailed timeline of the interaction of a user with the artifacts on a web site. Traditionally, these were stored in server logs for offline analysis; but in recent years, thanks to solutions like Kafka, Flume, and Scribe, to name a few, this event stream coupled with direct data from JavaScript code is transported in real-time to analytics solutions.

Clickstream data is cobbled together to construct a complete profile of a user. The ensuing micro-segmentation drives user engagement, predictive analytics, and server and web site design optimization. A typical record in the stream contains a timestamp, details about the event (click, purchase, form entry, and so on), the ingress and egress points, and user machine information (device type, operating system, and so on). Ingress information is important because it permits a web site to reason about how visitors land on a certain page—who the referrers are. For instance, if the top referrals come from Google, then search engine optimization (SEO) is working well and should be given more resources. On the other hand, if most visitors (or lack therefore) come via social media—which has definitely turned into a force to be reckoned with on this front—then that should drive future strategy.

Z. Nabi, *Pro Spark Streaming*, DOI 10.1007/978-1-4842-1479-4_7

Detailed clickstream data generally does not reside in the public/open data sphere for proprietary and privacy reasons. Fortunately, Wikipedia released two months (January and February 2015) of clickstream data from its request logs.[1] Table 7-1 lists the six fields in the dataset.

Table 7-1. *Fields in the Wikipedia Clickstream Dataset*

Field #	Name	Description
1	`prev_id (referrer)`	The ID of the previous Wikipedia article the user was on. Empty if a non-Wikipedia source.
2	`curr_id (resource)`	The ID of the current page.
3	`n`	The number of occurrences of this referrer, resource pair.
4	`prev_title`	The title of the referrer English Wikipedia article, or marked as `other-wikipedia`, `other-empty`, `other-internal`, `other-google`, `other-yahoo`, `other-bing`, `other-facebook`, `other-twitter`, or `other-other`.
5	`curr_title`	The title of the current article.
6	`type`	The type of the referrer: • `link`: Both the referrer and resource are Wikipedia articles. • `redlink`: The referrer is a Wikipedia page, but the target does not exist. • `other`: The referrer and target are both Wikipedia articles, but the referral came through an external source or a search.

Tachyon (Alluxio)

RDDs and other application state in Spark are co-located in executor JVMs. Because executors are per-application entities, this prevents applications from sharing RDDs. A corollary is that no cross-platform state (for example, between Spark and Hadoop) can be shared. More important, RDDs consume the lion's share of the heap space, thus potentially competing with other objects. The previous chapter looked at how Spark Streaming state can be offloaded to an in-memory solution like Redis. But this is not a general-purpose solution, because storing elements that constitute a `DStream` and in turn an RDD in external storage kills the utility of RDDs, especially the lineage and checkpointing-based fault-tolerance model. Enter Tachyon.

Tachyon is a general-purpose RDD store. Similar to native Spark, it uses a combination of lineage and checkpointing to ensure availability in the face of failure.[2] Unlike Spark, which uses job semantics (end of a batch, for example) to checkpoint files to negate unbounded recomputation, Tachyon has no such application-level information. Instead, it selectively and asynchronously checkpoints files at the leaves of the lineage graph while bearing priority in mind. Additionally, it uses a resource-allocation strategy that is application-level job-scheduler aware to allocate resources for recomputation.

Feature wise, Tachyon works out of the box with a number of frameworks including Hadoop, Spark, and Flink, while using a wide range of storage systems including HDFS, Amazon S3, and the local file system. For general-purpose applications, Tachyon has a native API that mimics the standard Java `java.io.File` interface. Furthermore, it takes advantage of I/O performance differences between memory, SSD, and HDD to realize tiered block storage.

[1]Ellery Wulczyn and Dario Taraborelli, "Wikipedia Clickstream," *Figshare*, January 4, 2016, http://figshare.com/articles/Wikipedia_Clickstream/1305770.
[2]Haoyuan Li et al., "Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks," *Proceedings of SOCC '14* (ACM, 2014).

As always, let's use a real example to understand the dynamics of RDD storage in Tachyon. As mentioned earlier, understanding the details of the channels through which a user lands on a web site has tremendous business value. In light of this, the example ranks the referrers to all Wikipedia articles in the dataset by frequency. It also ranks the different referrers for the Apache Spark Wikipedia article. First things first: set up Tachyon 0.6.4,[3] which is the version that works out of the box with Spark 1.4.0.

Download the Wikipedia dataset to your local machine/cluster. Use the custom socket driver program from Chapter 5 to feed the dataset to Spark Streaming. Only a single change is required: the original driver program expects the data to be in zipped form, whereas the Wikipedia data is gzipped. Simply insert the code from Listing 7-1 at line 67 of Listing 5-4.

Listing 7-1. Tweaking the Socket Driver to Support Gzipped Files

```
else if (ext.equals("gz")) {
    LOG.info(String.format("Feeding file %s", f.getName()));
    try (BufferedReader br = new BufferedReader(
        new InputStreamReader(new GZIPInputStream(new FileInputStream(f))))) {
    // skip header
    br.readLine();
    String line;
    while ((line = br.readLine()) != null) {
        sendRecord(line);
    }
}
```

Once Tachyon is up and running, execute the snippet of code from Listing 7-2. It reads clickstream data from a socket (line 1) and tokenizes its fields, which are tab separated. You need to process this stream twice—once to rank the referrals for all articles and then for referrals to the Apache Spark article—so it makes sense to persist the records in memory. Recall from Chapter 4 that each output action results in a separate job starting from the bifurcation point. If you do not persist the output of the map function on line 2, it will be executed twice. Rather than persist it in the same JVM as the Spark application, you offload it to Tachyon. This is as simple as providing a storage level of `OFF_HEAP` to the `persist` function. Make sure you set `spark.externalBlockStore.url` to your Tachyon server with the format `tachyon://hostname:port` in your configuration object.

The rest of the code maps the occurrence of referrals for all articles[4] (lines 5–12) and `Apache_Spark` (lines 14–16). The code uses a helper function `saveTopKeys` to aggregate ranked keys across batches and write them to secondary storage (lines 18 and 20). The code for the helper function is provided in Listing 7-3.

Listing 7-2. Offloading RDDs from a Clickstream Analysis Application to Tachyon

```
1.   val clickstream = ssc.socketTextStream(hostname, port.toInt)
2.     .map(rec => rec.split("\\t"))
3.     .persist(StorageLevel.OFF_HEAP)
4.
5.   val topRefStream = clickstream
```

[3]Download Tachyon version 0.6.4, and fill in `$TACHYON_HOME/conf/tachyon-env.sh` with values appropriate for your setup. If running it on top of the local file system, make sure `TACHYON_UNDERFS_ADDRESS` is set to a local file system folder, such as `/tmp`. Like any other file system, Tachyon first needs to be formatted via `$TACHYON_HOME/bin/tachyon format` before being executed: `$TACHYON_HOME/bin/tachyon-start.sh [all|local]`.
[4]Instead of keeping track of each individual article, the code aggregates all of them under the key "wikipedia".

```
    .map(rec => {
      var prev_title = rec(3)
      if (!prev_title.startsWith("other")) {
        prev_title = "wikipedia"
      }
      (prev_title, rec(2).toInt)
    })

  val topSparkStream = clickstream
    .filter(rec => rec(4).equals("Apache_Spark"))
    .map(rec => (rec(3), rec(2).toInt))

  saveTopKeys(topRefStream, outputPathTop)

  saveTopKeys(topSparkStream, outputPathSpark)
```

Listing 7-3. Helper Function for Clickstream Analysis

```
  def saveTopKeys(clickstream: DStream[(String, Int)], outputPath: String) {
    clickstream.updateStateByKey((values, state: Option[Int]) => Some(values.sum +
    state.getOrElse(0)))
      .repartition(1)
      .map(rec => (rec._2, rec._1))
      .transform(rec => rec.sortByKey(ascending = false))
      .saveAsTextFiles(outputPath)
  }
```

As per the results, the top five referrers overall include Wikipedia, other-empty, other-google, other-wikipedia, and other. The referrers for `Apache_Spark` are related to different Apache-centric articles including Apache_Hadoop, MapReduce, and other-google. It makes sense that users would jump from one Wikipedia article to another from the same domain: Big Data systems, in this case. other-google is by far the top referrer, which means most people land on the Wikipedia page for Spark via Google.

Tachyon can also be useful if RDDs need to be shared across different applications, even outside of Spark. Other applications can use the native Java API for Tachyon to access these files. Additionally, it comes packed with a command-line tool to manipulate the data store. It can be accessed via `$TACHYON_HOME/bin/tachyon tfs`, which enables simple file system operations. For instance, use this to copy an RDD generated by the application that you just executed[5] to the local file system:

```
$TACHYON_HOME/bin/tachyon tfs copyToLocal /tmp_spark_tachyon/spark-b5ad8845-eb30-4e9c-94e9-
1d772bd151ee/0/spark-tachyon-20151120000110-1009/20/rdd_81_9/ ./rdd_81_9
```

Note that `tmp_spark_tachyon` is the default block-store location and can be set via `spark.externalBlockStore.baseDir`.

Spark Web UI

The Spark UI is the central portal for drilling down into the details of applications. It can be really handy for diagnosing problems and improving performance. You can use it to, say, check whether the chosen batch interval is appropriate. Listing 7-4 contains example code for an application that this section dissects using the UI.

[5]The application id and rdd id would obviously vary from execution to execution.

Online businesses are always interested in knowing whether more users come to their web site via social media or a search engine. This enables them to figure out which channel to focus on to increase traffic. This application keeps track of the number of exclusive social media and search engine referrers—that is, articles for which the referrers are from either social media or a search engine but not both.

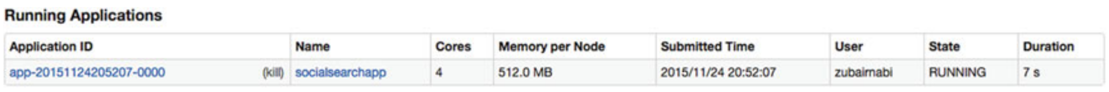
Listing 7-4. Referrer Channel Division Application

```
1.   val countSearch = new AtomicLong(0)
2.   val countSocial = new AtomicLong(0)
3.
4.   val titleStream = ssc.socketTextStream(hostname, port.toInt)
5.     .map(rec => rec.split("\\t"))
6.     .filter(_(3) match {
7.       case "other-google" | "other-bing" | "other-yahoo" | "other-facebook" | "other-
         twitter" => true
8.       case _ => false
9.     })
10.    .map(rec => (rec(3), rec(4)))
11.    .cache()
12.
13.  val searchStream = titleStream.filter(_._1 match {
14.    case "other-google" | "other-bing" | "other-yahoo" => true
15.    case _ => false
16.  })
17.    .map(rec => rec._2)
18.
19.  val socialStream = titleStream.filter(_._1 match {
20.    case "other-facebook" | "other-twitter" => true
21.    case _ => false
22.  })
23.    .map(rec => rec._2)
24.
25.  val exclusiveSearch = searchStream.transformWith(socialStream,
26.    (searchRDD: RDD[String], socialRDD: RDD[String]) => searchRDD.subtract(socialRDD))
27.    .foreachRDD(rdd => {
28.      countSearch.addAndGet(rdd.count())
29.      println("Exclusive count search engines: " + countSearch)
30.    })
31.
32.  val exclusiveSocial = socialStream.transformWith(searchStream,
33.    (socialRDD: RDD[String], searchRDD: RDD[String]) => socialRDD.subtract(searchRDD))
34.    .foreachRDD(rdd => {
35.      countSocial.addAndGet(rdd.count())
36.      println("Exclusive count social media: " + countSocial)
37.    })
```

Following the mantra of early projection, you cull the dataset early on by only keeping records for social media and search engines, and retaining only those fields that you need: `prev_title` and `curr_title` (lines 6–11). Because the application has to treat the social and search streams differently, you need to bifurcate the input stream. You achieve this via two filter transformations (lines 13–17 and 19–23) and only keep article names in both streams. DStreams out of the box do not have a set difference operation, but RDDs do. One way to use it is to invoke the transformWith transformation on DStreams; it takes as input another DStream and enables RDD-wise operations. In this transformation, you subtract the elements of each RDD (essentially article titles) separately to achieve set-difference operations (lines 25–26 and 32–33). Finally, to maintain a global count of these differences across RDDs, you keep this running tally in two static atomic variables and print it every batch (lines 27–30 and 34–37). The results show that for the Wikipedia dataset, search engine referrers exceed social media ones by many orders of magnitude.

■ **Note** transformWith only allows two-way DStream operations. For an *n*-way operation, use StreamingContext#transform(seq[Dstream[T]], transformFunc).

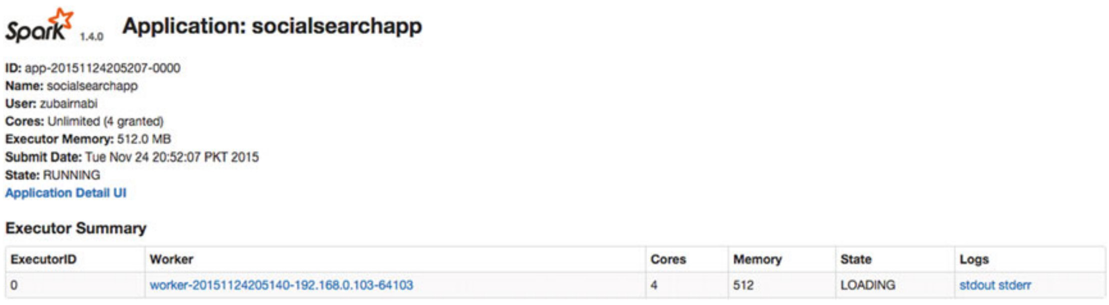
Now let's use the Spark UI to study the behavior of this application. After launching the application, go to your browser and enter the location of the UI. The UI typically runs on port 8080 on the same machine as the Spark master. The main page holds cluster-wide information such as the details of connected workers and running and completed applications. You can click each of these entities to access more in-depth information. For instance, to drill down into the details of the social media versus search engine referrals app, click its application ID or name as shown in Figure 7-1.


Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20151124205207-0000 (kill)	socialsearchapp	4	512.0 MB	2015/11/24 20:52:07	zubairnabi	RUNNING	7 s

Figure 7-1. *Running Applications in the Spark UI*

The summary page for each application lists its status and its launched executors, as shown in Figure 7-2. You can also jump to application details via the Application Detail UI link. Each SparkContext launches a separate Application Detail UI with ports starting with 4040.



Application: socialsearchapp

ID: app-20151124205207-0000
Name: socialsearchapp
User: zubairnabi
Cores: Unlimited (4 granted)
Executor Memory: 512.0 MB
Submit Date: Tue Nov 24 20:52:07 PKT 2015
State: RUNNING
Application Detail UI

Executor Summary

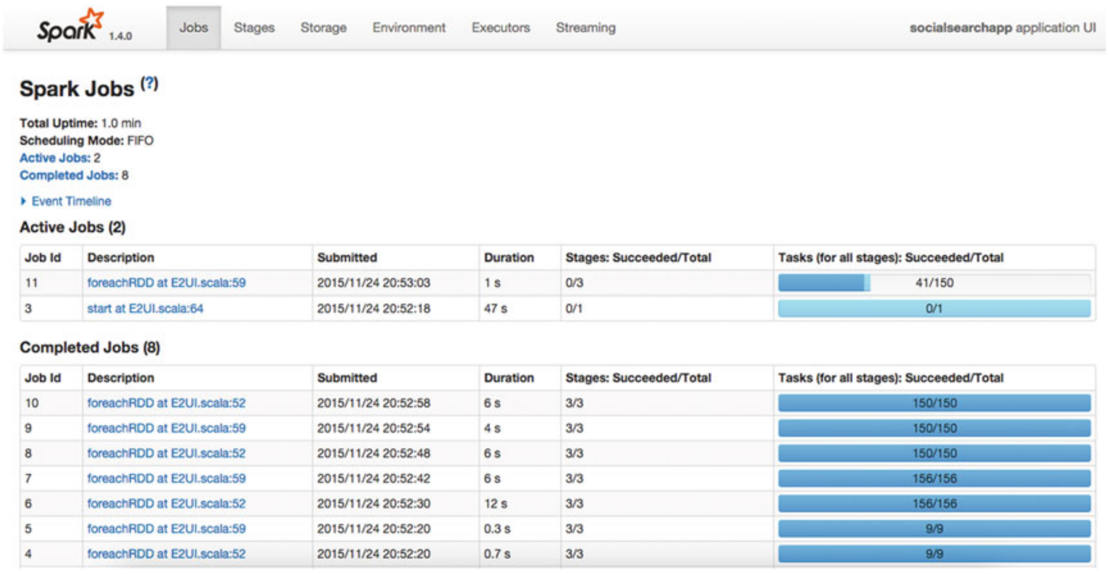
ExecutorID	Worker	Cores	Memory	State	Logs
0	worker-20151124205140-192.168.0.103-64103	4	512	LOADING	stdout stderr

Figure 7-2. *Summary page for an application*

Let's go to the application detail UI (Figure 7-3). It gives the constantly increasing job breakdown for the application. Each job is represented by the title of its last stage. In this case, the last stage ends with a foreachRDD action. Jobs are grouped as active or completed. This page gives you some useful information about the application. For starters, each job has three stages: one each for the two parallel streams (social and search) and one for the subtract operation. This is because the former entails a shuffle. A job called start runs continuously to execute the streaming scheduler. In addition, only a single user code job is active at a time. This is due to `spark.streaming.concurrentJobs` being set to 1 by default. Increase this number to 2 to enable more than one job to execute in parallel, as described in Chapter 4. Remember, each output action results in a separate job.

The last column in the table contains the overall task progress for each job. Note that the number, 150 in this case, is across all stages. This application has three stages, which means the number of tasks per stage is 50. As you saw in Chapter 4, the number of tasks in a stage is equal to the number of partitions in the last RDD in that stage. The number of partitions in an RDD, on the other hand, is determined by the number of partitions in its input RDD (or the output of the previous stage). If you walk up the stage graph, you reach the application input stream, which is a `SocketInputDStream`. Recall from Chapter 5 that the number of partitions of any receiver-based `InputDStream` is decided by the batch interval and the block interval. The application was executed with a batch interval of 10 seconds and a block interval of 200 ms (the default value), which gives you 50 (10 s/200 ms).

The Duration column in the table gives you the time taken by each job. To decide on optimum values for both batch and block intervals, play with their values and see how they affect the stage duration. This obviously depends on application complexity, data properties, and the cluster setup.



Spark Jobs (?)

Total Uptime: 1.0 min
Scheduling Mode: FIFO
Active Jobs: 2
Completed Jobs: 8

▸ Event Timeline

Active Jobs (2)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
11	foreachRDD at E2UI.scala:59	2015/11/24 20:53:03	1 s	0/3	41/150
3	start at E2UI.scala:64	2015/11/24 20:52:18	47 s	0/1	0/1

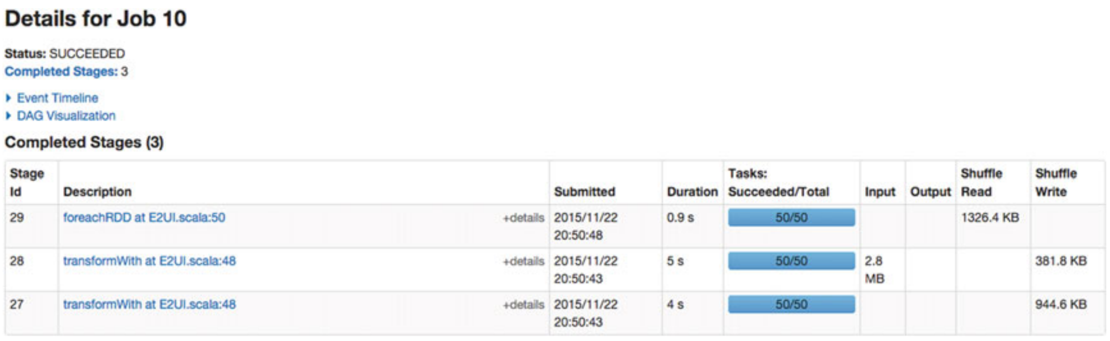
Completed Jobs (8)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
10	foreachRDD at E2UI.scala:52	2015/11/24 20:52:58	6 s	3/3	150/150
9	foreachRDD at E2UI.scala:59	2015/11/24 20:52:54	4 s	3/3	150/150
8	foreachRDD at E2UI.scala:52	2015/11/24 20:52:48	6 s	3/3	150/150
7	foreachRDD at E2UI.scala:59	2015/11/24 20:52:42	6 s	3/3	156/156
6	foreachRDD at E2UI.scala:52	2015/11/24 20:52:30	12 s	3/3	156/156
5	foreachRDD at E2UI.scala:59	2015/11/24 20:52:20	0.3 s	3/3	9/9
4	foreachRDD at E2UI.scala:52	2015/11/24 20:52:20	0.7 s	3/3	9/9

***Figure 7-3.** Detailed job information for an application*

You can use the tabs at the top of this page to jump to different status pages. The Storage tab gives RDD-level information, the Environment tab displays configuration information, and the Executors tab gives details of each executor for this application. All of these tabs apply equally to Spark and Spark Streaming except the Streaming tab.

You can also get stage-level information by clicking each job. The information for a particular job is shown in Figure 7-4. For each stage, it shows the submission time, duration, and data size. Stages are grouped by the name of the last transformation. It is obvious that more time is taken by stages that read the `SocketInputDStream` and bifurcate it into search engine and social media referral streams in relation to other stages. Stage boundaries are also revealed by shuffle behavior. The `transformWith` stage writes shuffle data, which is read by the subsequent `foreachRDD` stage. Column 5 shows that, as mentioned earlier, each stage consists of 50 tasks.



Details for Job 10

Status: SUCCEEDED
Completed Stages: 3

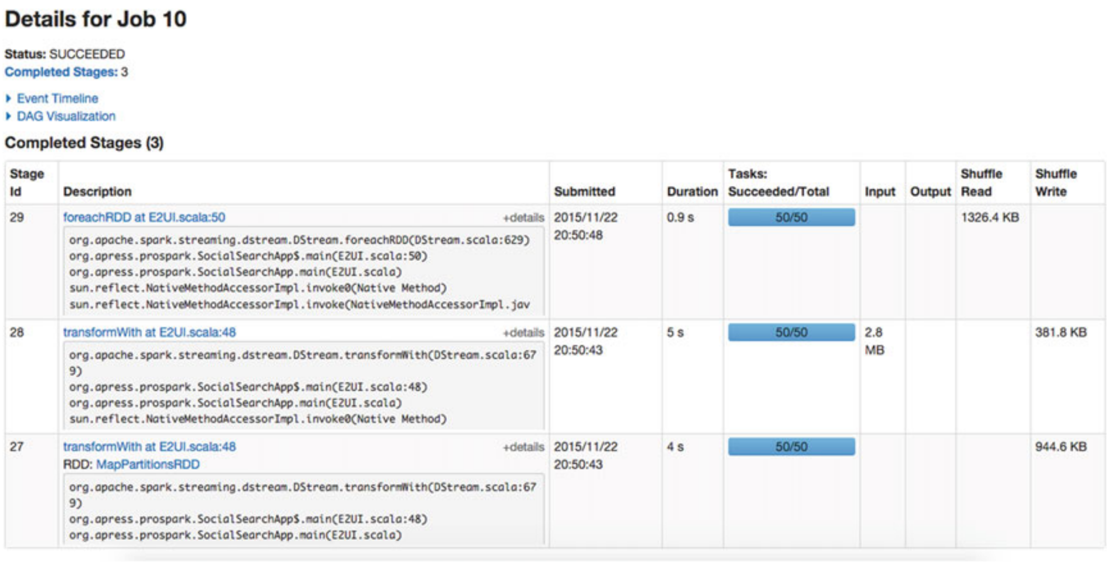
▸ Event Timeline
▸ DAG Visualization

Completed Stages (3)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
29	foreachRDD at E2UI.scala:50	+details	2015/11/22 20:50:48	0.9 s	50/50			1326.4 KB	
28	transformWith at E2UI.scala:48	+details	2015/11/22 20:50:43	5 s	50/50	2.8 MB			381.8 KB
27	transformWith at E2UI.scala:48	+details	2015/11/22 20:50:43	4 s	50/50				944.6 KB

Figure 7-4. *Detailed stage information for a job*

The code-level trigger for a stage (a shuffle boundary) can also be viewed by clicking the details option, as shown in Figure 7-5.



Details for Job 10

Status: SUCCEEDED
Completed Stages: 3

▸ Event Timeline
▸ DAG Visualization

Completed Stages (3)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write

29	foreachRDD at E2UI.scala:50
	`org.apache.spark.streaming.dstream.DStream.foreachRDD(DStream.scala:629)`
	`org.apress.prospark.SocialSearchApp$.main(E2UI.scala:50)`
	`org.apress.prospark.SocialSearchApp.main(E2UI.scala)`
	`sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)`
	`sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.jav`
		+details	2015/11/22 20:50:48	0.9 s	50/50			1326.4 KB	

28	transformWith at E2UI.scala:48
	`org.apache.spark.streaming.dstream.DStream.transformWith(DStream.scala:67`
	`9)`
	`org.apress.prospark.SocialSearchApp$.main(E2UI.scala:48)`
	`org.apress.prospark.SocialSearchApp.main(E2UI.scala)`
	`sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)`
		+details	2015/11/22 20:50:43	5 s	50/50	2.8 MB			381.8 KB

27	transformWith at E2UI.scala:48
	RDD: MapPartitionsRDD
	`org.apache.spark.streaming.dstream.DStream.transformWith(DStream.scala:67`
	`9)`
	`org.apress.prospark.SocialSearchApp$.main(E2UI.scala:48)`
	`org.apress.prospark.SocialSearchApp.main(E2UI.scala)`
		+details	2015/11/22 20:50:43	4 s	50/50				944.6 KB

Figure 7-5. *Code-level information for a stage*

There is also an option to view the graph for each job by clicking DAG Visualization (the link above Completed Stages). The DAG for one representative job for this application is shown in Figure 7-6. Each blue box represents a transformation, and the arrows represent the flow. Each black dot is an RDD. Notice the single green dot on each of the `transformWith` stages: these are RDDs that you explicitly cached to negate redundant processing. You can also see that Spark pipelines non-shuffle triggering transformations in the same task to reduce interprocess communication.[6] Because both subtract operations rely on the same input data (social minus search and search minus social), they are pipelined into the same stage.

[6]Andrew Or, "Understanding Your Spark Application through Visualization," *Databricks*, June 22, 2015, https://databricks.com/blog/2015/06/22/understanding-your-spark-application-through-visualization.html.

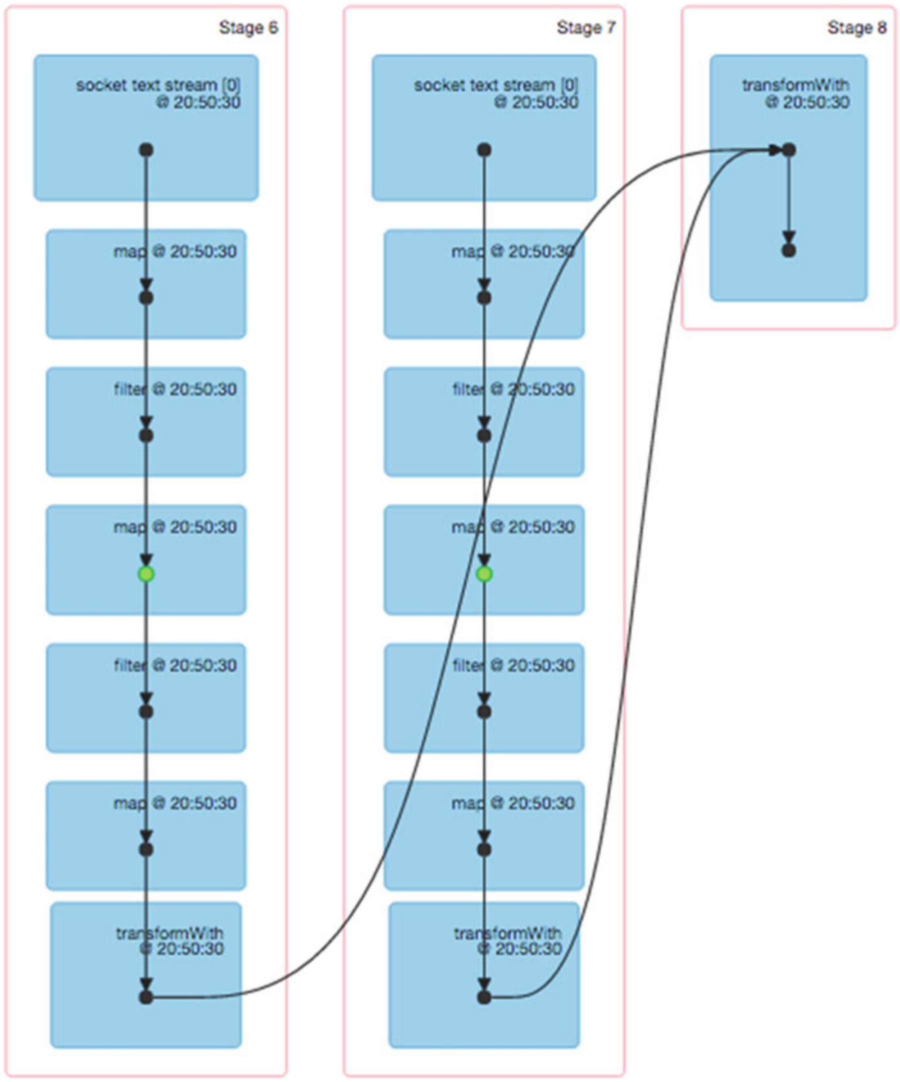


Figure 7-6. *DAG visualization for a job*

You can also display the RDD-level flow for each stage as shown in Figure 7-7. In the stage flow on the left, you can see that the RDDs for the `InputDStream` reside in the block store—hence the name `BlockRDD`. The rest of the flow emits and consumes `MapPartitionsRDD`s. The `foreachRDD` stage on the right, on the other hand, creates a `SubtractedRDD` based on the subtract operation in the `transformWith` transformation.

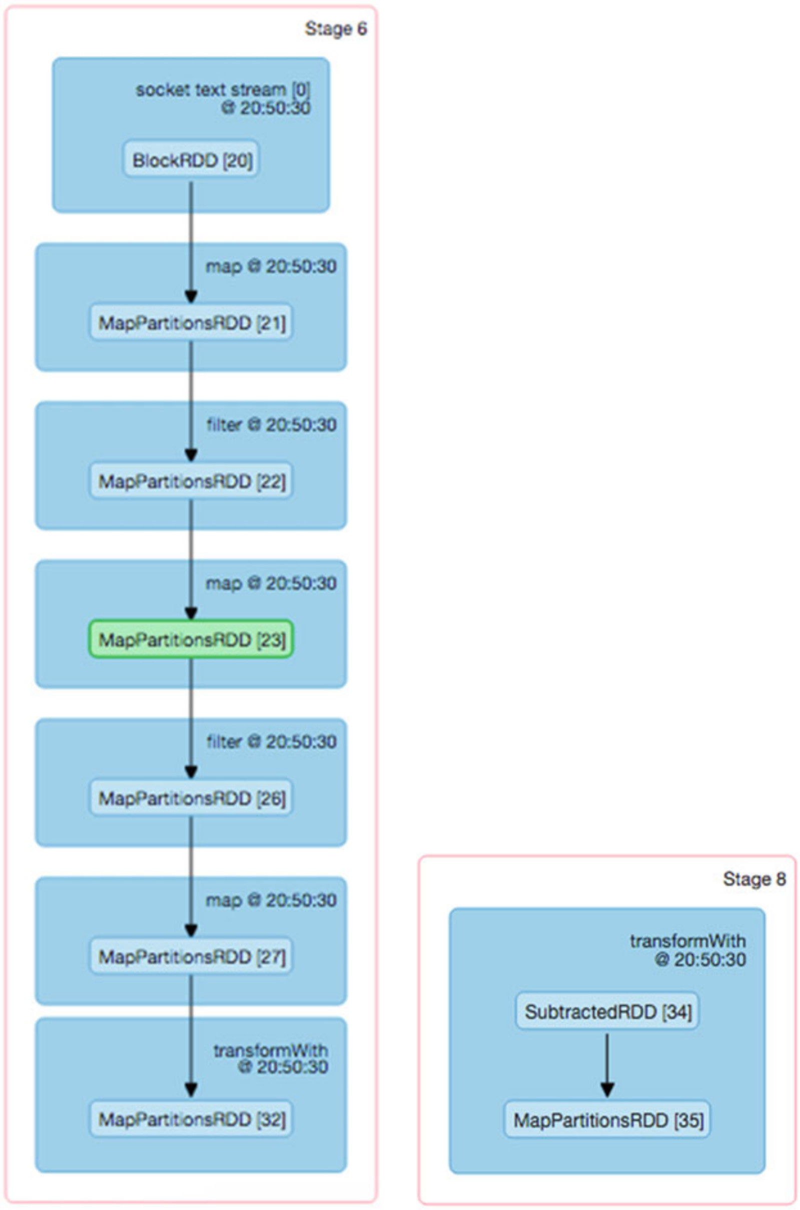


Figure 7-7. *RDD flow visualization*

The other visualization enabled by the UI is timeline based; it allows you to see a temporal flow of events. These events include the addition and removal of executors, and scheduling of jobs, stages, and tasks. Let's first analyze the job-level timeline graph. To generate it, click the Event Timeline link in Figure 7-3. As shown in Figure 7-8, it provides a complete timeline for the life cycle of executors and jobs.

The legend on the left of the timeline in Figure 7-8 signposts the color coding of the different events. A number of things stand out. First, as expected, the executors for the application are launched before everything else. Second, the "start at" job keeps running throughout, because it is in charge of scheduling jobs across batches. Finally, jobs are scheduled in series, one after the other. Clicking a particular job takes you to the stage timeline, similar to the one in Figure 7-9. As expected, the two `transformWith` stages execute in parallel, and upon their completion, the `foreachRDD` stage is kicked off.

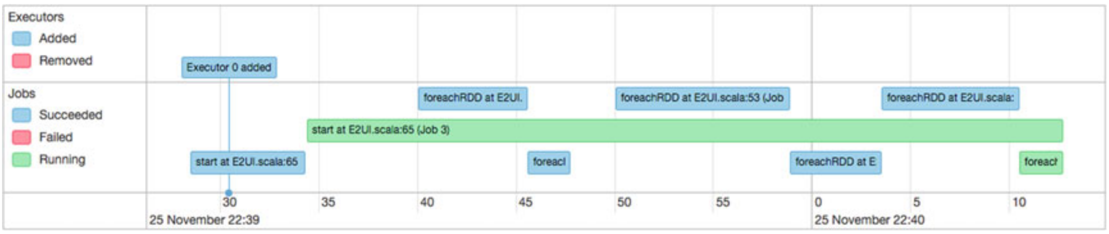


Figure 7-8. *Event timeline of jobs*

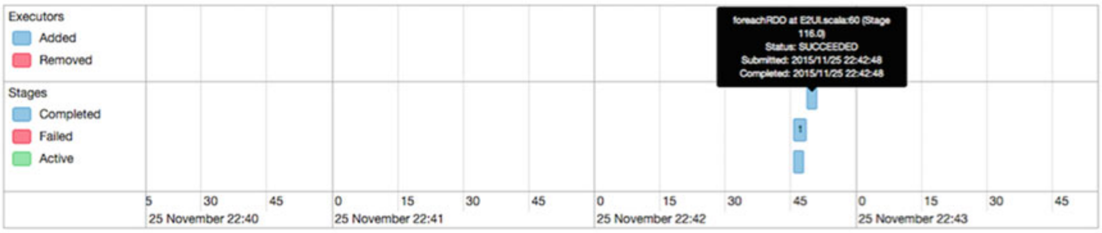


Figure 7-9. *Event timeline of stages in a job*

A more telling and useful timeline is for each task in a stage. You can view the graph in Figure 7-10 by clicking any stage in the event timeline visualization (or by directly jumping to any stage via the Stages tab). The spectrum of colors in each task reflects the cost of the different phases in its execution. This visual provides a number of insights. First and foremost, all the tasks are CPU bound, because Executor Computing Time takes the lion's share of their duration. In addition, toward the tail end of the stage, tasks start experiencing longer Scheduler Delay, which suggests that throwing more machines (and, in turn, executors) at them will help improve performance. The same page also contains summary metrics for this stage in tabular form (see Figure 7-11). In addition, aggregated task-execution statistics and shuffle metrics per executor are shown. This view also provides information such as the fact that garbage-collection time is not a dominant factor in the application, which means the number of JVM heap objects is under control.

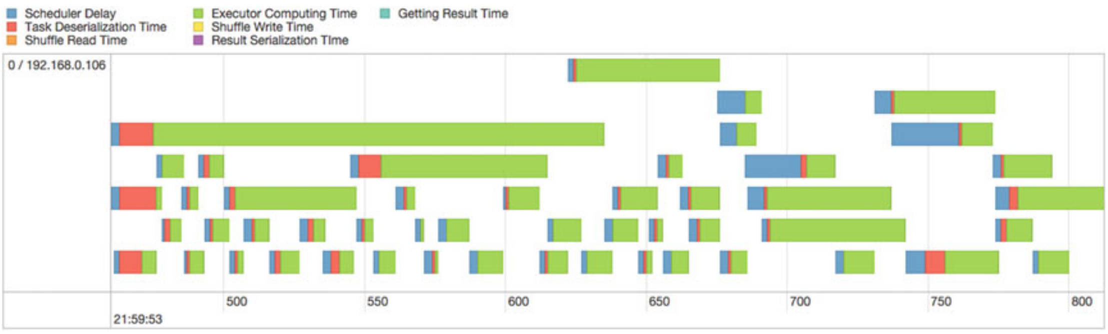
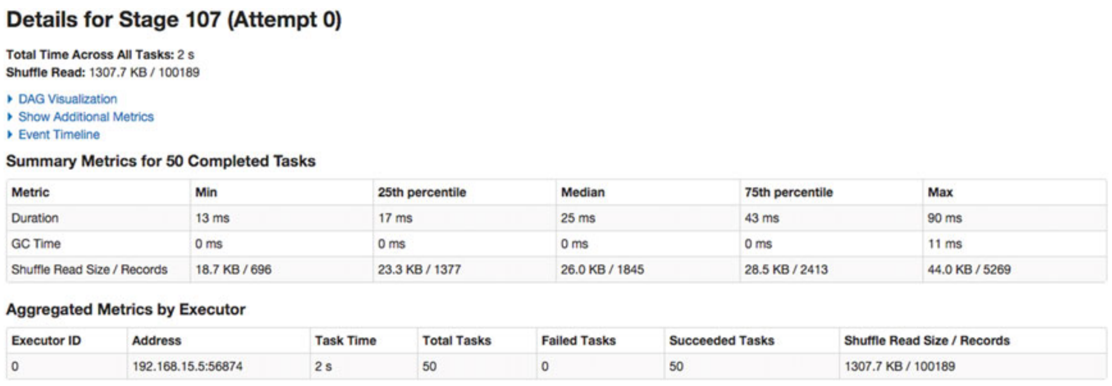


Figure 7-10. *Task execution state timeline*



Details for Stage 107 (Attempt 0)

Total Time Across All Tasks: 2 s
Shuffle Read: 1307.7 KB / 100189

▸ DAG Visualization
▸ Show Additional Metrics
▸ Event Timeline

Summary Metrics for 50 Completed Tasks

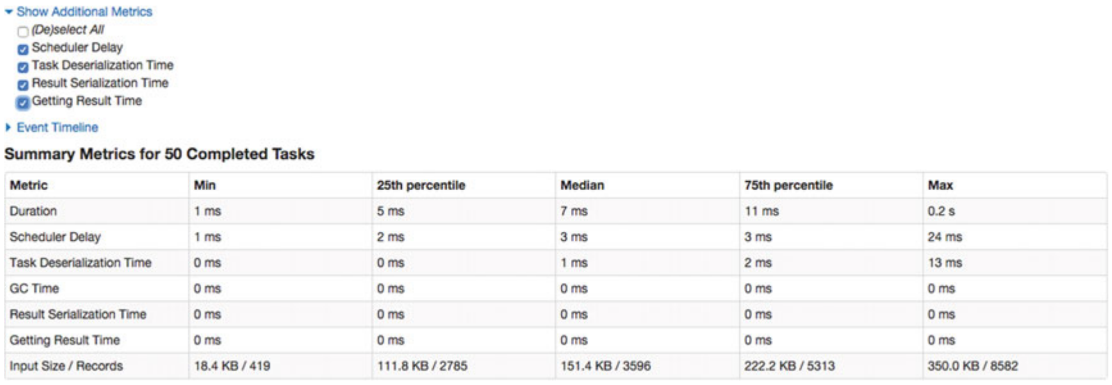
Metric	Min	25th percentile	Median	75th percentile	Max
Duration	13 ms	17 ms	25 ms	43 ms	90 ms
GC Time	0 ms	0 ms	0 ms	0 ms	11 ms
Shuffle Read Size / Records	18.7 KB / 696	23.3 KB / 1377	26.0 KB / 1845	28.5 KB / 2413	44.0 KB / 5269

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Shuffle Read Size / Records
0	192.168.15.5:56874	2 s	50	0	50	1307.7 KB / 100189

Figure 7-11. *Summary metrics for tasks in a stage*

The metrics highlighted in Figure 7-11 can be enriched with more attributes by clicking the Show Additional Metrics option and selecting appropriate metrics, as shown in Figure 7-12.



▾ Show Additional Metrics
☐ (De)select All
☑ Scheduler Delay
☑ Task Deserialization Time
☑ Result Serialization Time
☑ Getting Result Time

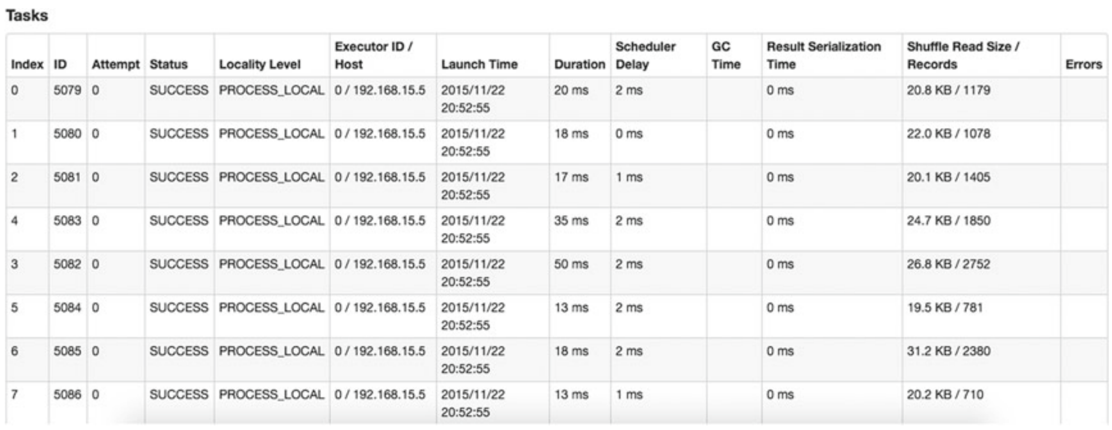
▸ Event Timeline

Summary Metrics for 50 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	1 ms	5 ms	7 ms	11 ms	0.2 s
Scheduler Delay	1 ms	2 ms	3 ms	3 ms	24 ms
Task Deserialization Time	0 ms	0 ms	1 ms	2 ms	13 ms
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms
Result Serialization Time	0 ms	0 ms	0 ms	0 ms	0 ms
Getting Result Time	0 ms	0 ms	0 ms	0 ms	0 ms
Input Size / Records	18.4 KB / 419	111.8 KB / 2785	151.4 KB / 3596	222.2 KB / 5313	350.0 KB / 8582

Figure 7-12. *Additional task-level summary metrics*

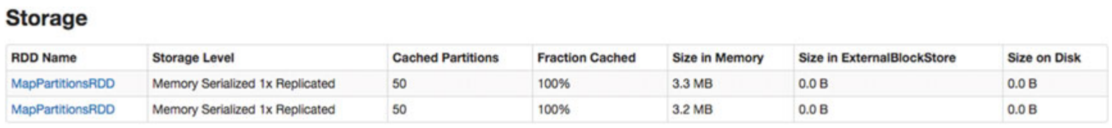
Scrolling down on the same page provides individual task-level metrics (Figure 7-13). These are helpful for learning that, for instance, the shuffle read size is similar across tasks, which suggests that the data distribution is fairly even.


Tasks

Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	Scheduler Delay	GC Time	Result Serialization Time	Shuffle Read Size / Records	Errors
0	5079	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.15.5	2015/11/22 20:52:55	20 ms	2 ms		0 ms	20.8 KB / 1179	
1	5080	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.15.5	2015/11/22 20:52:55	18 ms	0 ms		0 ms	22.0 KB / 1078	
2	5081	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.15.5	2015/11/22 20:52:55	17 ms	1 ms		0 ms	20.1 KB / 1405	
4	5083	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.15.5	2015/11/22 20:52:55	35 ms	2 ms		0 ms	24.7 KB / 1850	
3	5082	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.15.5	2015/11/22 20:52:55	50 ms	2 ms		0 ms	26.8 KB / 2752	
5	5084	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.15.5	2015/11/22 20:52:55	13 ms	2 ms		0 ms	19.5 KB / 781	
6	5085	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.15.5	2015/11/22 20:52:55	18 ms	2 ms		0 ms	31.2 KB / 2380	
7	5086	0	SUCCESS	PROCESS_LOCAL	0 / 192.168.15.5	2015/11/22 20:52:55	13 ms	1 ms		0 ms	20.2 KB / 710	

Figure 7-13. *Metrics for individual tasks*

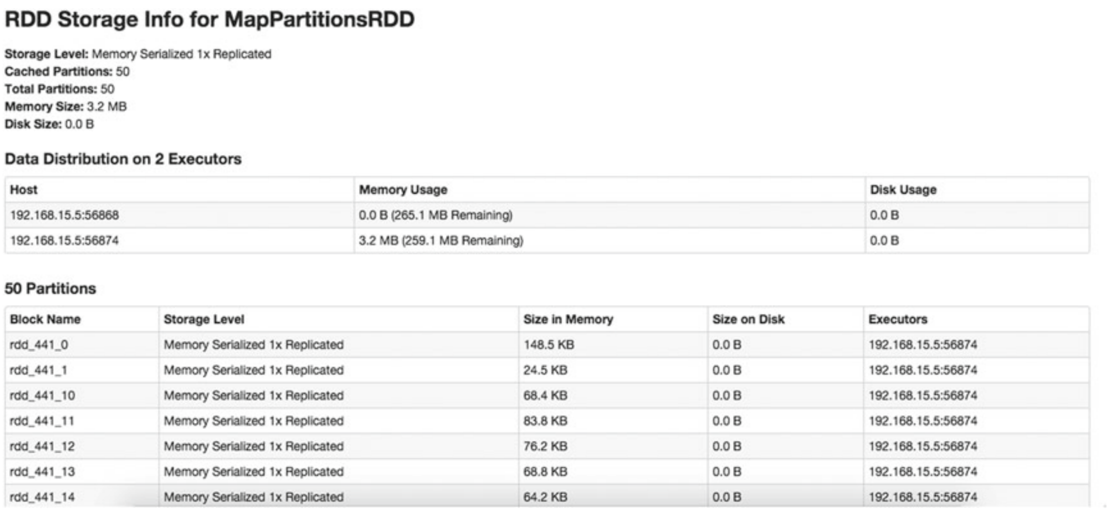
Now let's explore some other tabs from the top menu bar, beginning with Storage. It contains details of RDDs that have a configured storage level other than the default (which triggers regeneration under failure). As shown in Figure 7-14, you see a complete breakdown of its caching properties. For instance, the two RDDs shown in the figure have a storage level of `StorageLevel.MEMORY_ONLY_SER`. This is due to the `cache` operation on line 11 of the code (Listing 7-4). That is why the RDDs are stored entirely in memory. If disk-based or `OFF_HEAP` storage had been chosen, it would have been reflected in the last two columns in this table.


Storage

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in ExternalBlockStore	Size on Disk
MapPartitionsRDD	Memory Serialized 1x Replicated	50	100%	3.3 MB	0.0 B	0.0 B
MapPartitionsRDD	Memory Serialized 1x Replicated	50	100%	3.2 MB	0.0 B	0.0 B

Figure 7-14. *Storage properties of cached RDDs*

Clicking any RDD in the table takes you to the detailed per-partition breakdown in Figure 7-15. You can see that each RDD contains 50 partitions, which is what you expect due to the selected block interval and batch interval.



RDD Storage Info for MapPartitionsRDD

Storage Level: Memory Serialized 1x Replicated
Cached Partitions: 50
Total Partitions: 50
Memory Size: 3.2 MB
Disk Size: 0.0 B

Data Distribution on 2 Executors

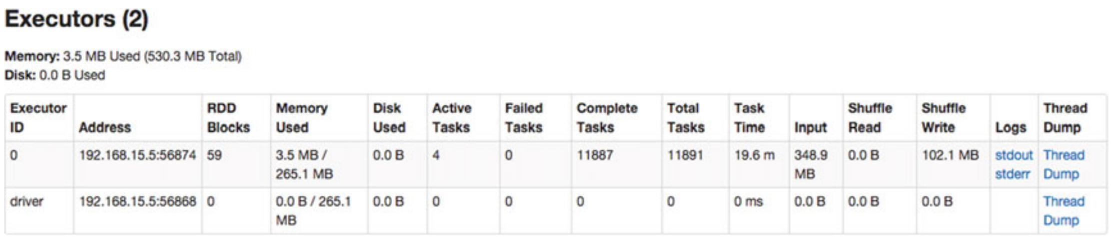
Host	Memory Usage	Disk Usage
192.168.15.5:56868	0.0 B (265.1 MB Remaining)	0.0 B
192.168.15.5:56874	3.2 MB (259.1 MB Remaining)	0.0 B

50 Partitions

Block Name	Storage Level	Size in Memory	Size on Disk	Executors
rdd_441_0	Memory Serialized 1x Replicated	148.5 KB	0.0 B	192.168.15.5:56874
rdd_441_1	Memory Serialized 1x Replicated	24.5 KB	0.0 B	192.168.15.5:56874
rdd_441_10	Memory Serialized 1x Replicated	68.4 KB	0.0 B	192.168.15.5:56874
rdd_441_11	Memory Serialized 1x Replicated	83.8 KB	0.0 B	192.168.15.5:56874
rdd_441_12	Memory Serialized 1x Replicated	76.2 KB	0.0 B	192.168.15.5:56874
rdd_441_13	Memory Serialized 1x Replicated	68.8 KB	0.0 B	192.168.15.5:56874
rdd_441_14	Memory Serialized 1x Replicated	64.2 KB	0.0 B	192.168.15.5:56874

Figure 7-15. *Partition breakdown of an RDD*

The Executors tab displays statistics about each running executor (Figure 7-16). It is important to highlight that the Memory Used tab does not reflect the memory used by the executor but rather the memory used by cached RDDs (obtained by invoking `cache()` or `persist()`). The same table allows you to obtain a thread-level dump of the executor (Figure 7-17).



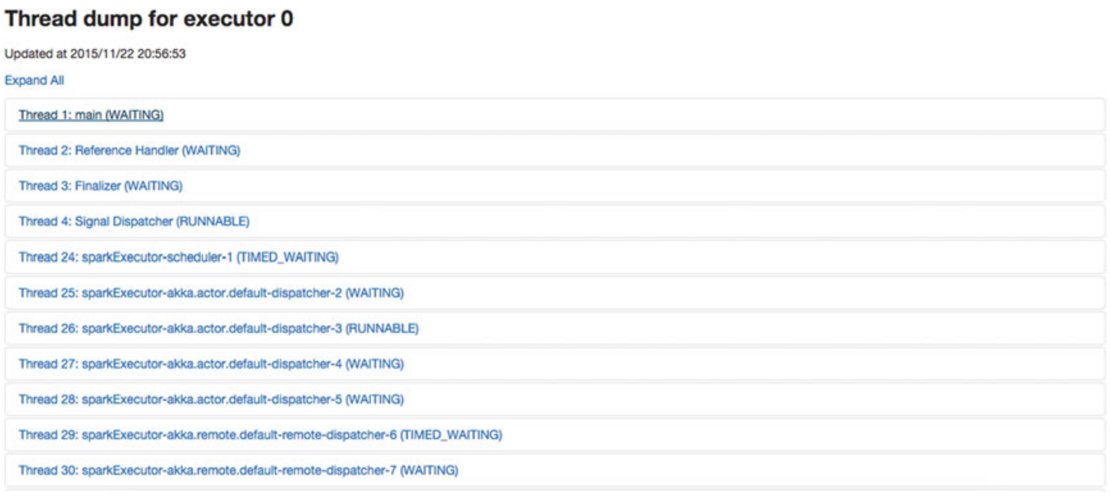
Executors (2)

Memory: 3.5 MB Used (530.3 MB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Memory Used	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
0	192.168.15.5:56874	59	3.5 MB / 265.1 MB	0.0 B	4	0	11887	11891	19.6 m	348.9 MB	0.0 B	102.1 MB	stdout stderr	Thread Dump
driver	192.168.15.5:56868	0	0.0 B / 265.1 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B		Thread Dump

Figure 7-16. *High-level executor metrics*

The thread-level dump is handy for figuring out the status of each thread. For custom threads, you can check their behavior in order to avoid deadlocks.


Thread dump for executor 0

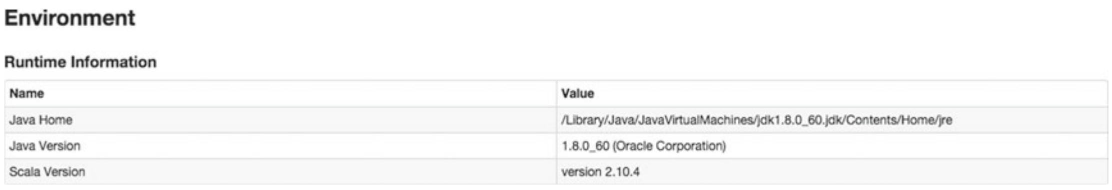
Updated at 2015/11/22 20:56:53

Expand All

Thread 1: main (WAITING)
Thread 2: Reference Handler (WAITING)
Thread 3: Finalizer (WAITING)
Thread 4: Signal Dispatcher (RUNNABLE)
Thread 24: sparkExecutor-scheduler-1 (TIMED_WAITING)
Thread 25: sparkExecutor-akka.actor.default-dispatcher-2 (WAITING)
Thread 26: sparkExecutor-akka.actor.default-dispatcher-3 (RUNNABLE)
Thread 27: sparkExecutor-akka.actor.default-dispatcher-4 (WAITING)
Thread 28: sparkExecutor-akka.actor.default-dispatcher-5 (WAITING)
Thread 29: sparkExecutor-akka.remote.default-remote-dispatcher-6 (TIMED_WAITING)
Thread 30: sparkExecutor-akka.remote.default-remote-dispatcher-7 (WAITING)

Figure 7-17. *Thread dump for a running executor*

The Environment tag is useful for figuring out whether configurable parameters are being properly set and their default values. In addition, you can get system-wide information such as JVM version (as shown in Figure 7-18) and the JARs on the classpath.


Environment

Runtime Information

Name	Value
Java Home	/Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/jre
Java Version	1.8.0_60 (Oracle Corporation)
Scala Version	version 2.10.4

Figure 7-18. *Snippet of the environment information page*

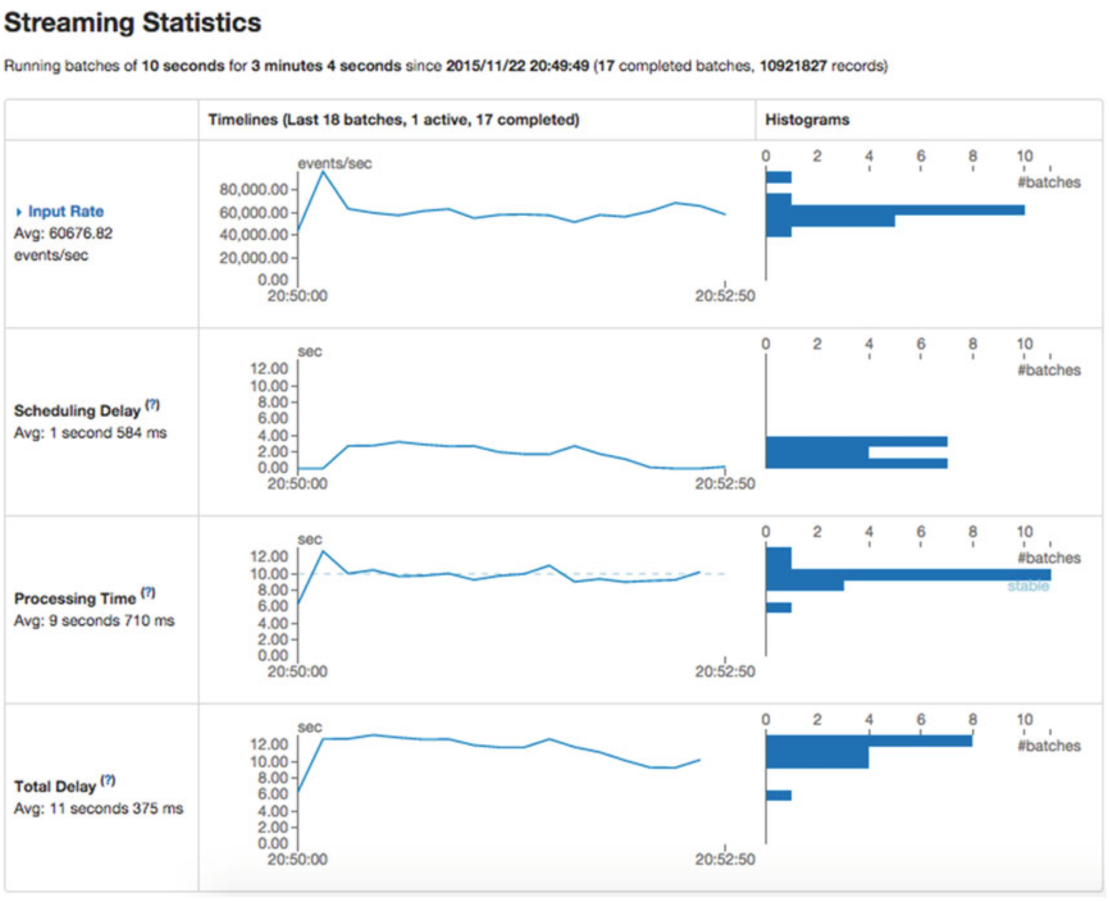
The Spark Streaming micro-batch-processing model imposes an additional burden in terms of choosing the batch interval (and block interval, for receiver-based streaming). If there is a mismatch between the batch interval and input data rate, the application will be unhealthy and may crash due to queue pressure. A useful utility to aid you in choosing these settings is Streaming Statistics; Figure 7-19 shows the graphs for the example clickstream application. The Input Rate[7] and Processing Time trends are similar, which indicates that the chosen batch interval of 10 seconds is appropriate. But the Total Delay on average is greater than 10 seconds, which means the application is lagging behind by more than a second on average. Moreover, Processing Time has a horizontal “stable” application marker: on average, the trend should be below it, but in this case every batch misses the line.

For healthy, stable applications, Scheduling Delay (the amount of time RDDs stay in the queue before being processed) should be close to zero on average, and Total Delay should be less than the batch interval on average. The histograms for Processing Time and Total Delay also tell you that the distribution of these times is toward the high end: both numbers are near or greater than 10 seconds most of the time. For a stable application, the distribution should be lopsided toward the low end—the histogram should be laterally flipped.

[7]You can click Input Rate to see a breakdown per receiver.

Recall from Chapter 4 that increasing the batch interval increases latency but reduces task/RDD setup time. On the other hand, lowering the value decreases end-to-end latency but leads to higher scheduling and RDD-creation overheads. Therefore, for optimum performance, you need to find the sweet spot between the pipeline's batch interval and processing time. In Figure 7-19, you can see that RDDs spend time in the queue waiting to be processed, which indicates that the data is created at a much faster rate than it can be processed by the application. To reduce this delay, you need to decrease the number of partitions the application has to process in each batch; you can do this by reducing the batch interval.

Let's try to achieve that by lowering the batch interval to 1 second. The Streaming Statistics are shown in Figure 7-20.



Streaming Statistics

Running batches of **10 seconds** for **3 minutes 4 seconds** since **2015/11/22 20:49:49** (17 completed batches, **10921827** records)

	Timelines (Last 18 batches, 1 active, 17 completed)	Histograms
Input Rate Avg: 60676.82 events/sec		
Scheduling Delay (?) Avg: 1 second 584 ms		
Processing Time (?) Avg: 9 seconds 710 ms		
Total Delay (?) Avg: 11 seconds 375 ms		

Figure 7-19. Streaming Statistics for an unhealthy application

You can see that both the processing time and total delay are well below 1 second on average. In addition, the scheduling delay is on the order of tens of milliseconds for most batches. Additional positive indicators are the histograms for processing time and total delay, which are now concentrated toward lower values. Just by knowing that reducing the number of partitions leads to lower processing time, and therefore reducing the batch interval, you have stabilized a misconfigured application.

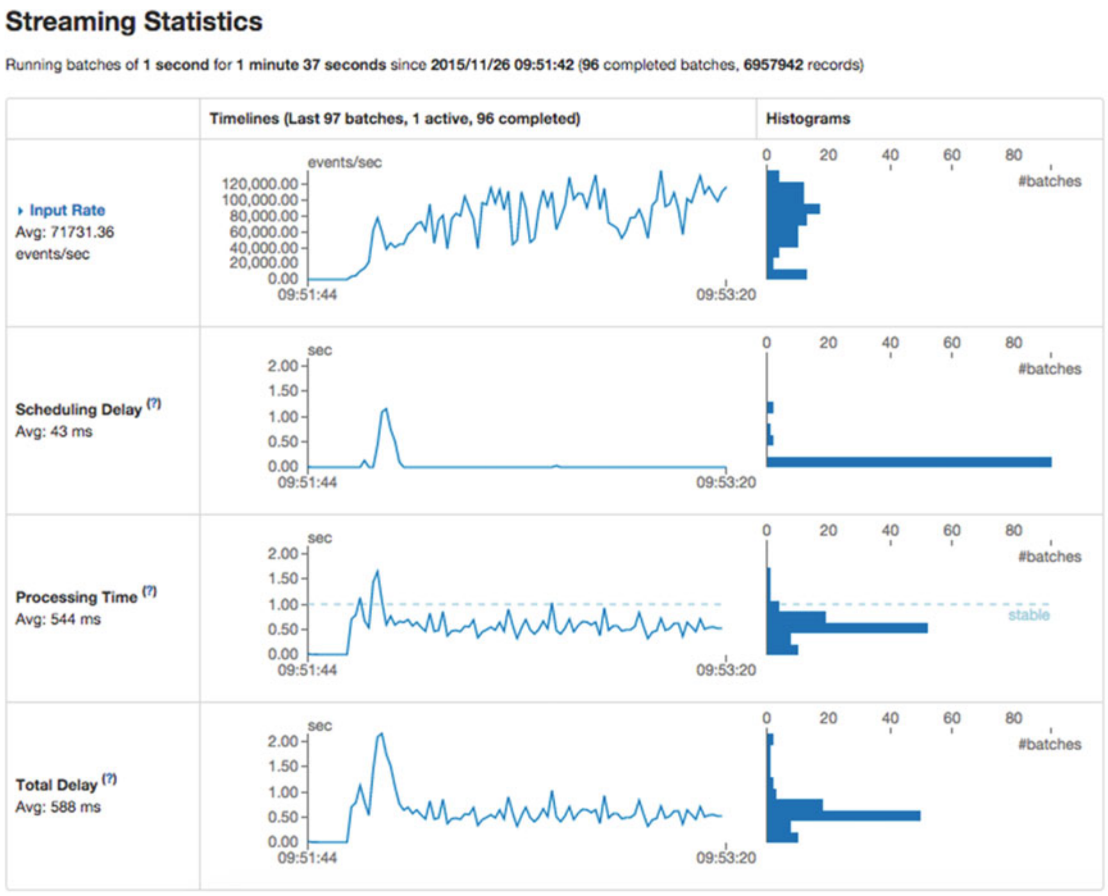
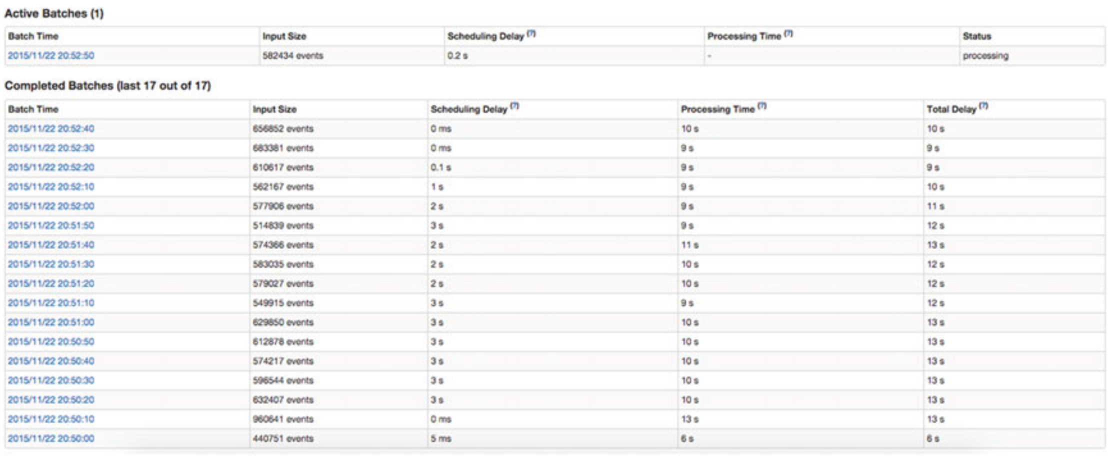


Figure 7-20. *Streaming Statistics for a healthy application*

The same information is also shown in tabular form on the same page, as shown in Figure 7-21. Overall, the Spark UI is an extremely useful tool for debugging and optimizing applications. It makes a number of systemwide statistics readily available.



Active Batches (1)

Batch Time	Input Size	Scheduling Delay (?)	Processing Time (?)	Status
2015/11/22 20:52:50	582434 events	0.2 s	-	processing

Completed Batches (last 17 out of 17)

Batch Time	Input Size	Scheduling Delay (?)	Processing Time (?)	Total Delay (?)
2015/11/22 20:52:40	656852 events	0 ms	10 s	10 s
2015/11/22 20:52:30	683381 events	0 ms	9 s	9 s
2015/11/22 20:52:20	610617 events	0.1 s	9 s	9 s
2015/11/22 20:52:10	562167 events	1 s	9 s	10 s
2015/11/22 20:52:00	577906 events	2 s	9 s	11 s
2015/11/22 20:51:50	514839 events	3 s	9 s	12 s
2015/11/22 20:51:40	574366 events	2 s	11 s	13 s
2015/11/22 20:51:30	583035 events	2 s	10 s	12 s
2015/11/22 20:51:20	579027 events	2 s	10 s	12 s
2015/11/22 20:51:10	549915 events	3 s	9 s	12 s
2015/11/22 20:51:00	629850 events	3 s	10 s	13 s
2015/11/22 20:50:50	612878 events	3 s	10 s	13 s
2015/11/22 20:50:40	574217 events	3 s	10 s	13 s
2015/11/22 20:50:30	596544 events	3 s	10 s	13 s
2015/11/22 20:50:20	632407 events	3 s	10 s	13 s
2015/11/22 20:50:10	960641 events	0 ms	13 s	13 s
2015/11/22 20:50:00	440751 events	5 ms	6 s	6 s

Figure 7-21. *Streaming Statistics in tabular form*

Historical Analysis

By default, detailed statistics (those you can see by clicking Application Detail UI) for applications are lost after the application finishes its execution. That means no after-the-fact analysis is possible—but don't panic. Spark has a logging mechanism to enable historical analysis that is very easy to use: set `spark.eventLog.enabled` to `true` to start logging events. The location for the log is determined by `spark.eventLog.dir`, which has a default value of `file:/tmp/spark-events`. You can then use the standard Application Detail UI to view the metrics for these instrumented applications. This only applies to standalone mode.

For operation under Mesos and YARN, Spark has a historical server you can use to replay events from the log. To analyze these logs post hoc, run the historical server via `$SPARK_HOME/sbin/start-history-server.sh`. This runs a web server at port 18080 (`spark.history.ui.port`) on that particular machine. The interface of this tool is almost identical to the real-time UI. Other useful configuration parameters include `spark.history.retainedApplications`, which has a default value of 50 and determines the number of applications for which events are retained; `spark.history.fs.update.interval` (default: 10 seconds), which dictates the event-logging interval; and `spark.history.fs.logDirectory` (default: `file:/tmp/spark-events`), which is the directory from which the history server loads event logs.

RESTful Metrics

All the numbers in the UI are also exposed as a RESTful API in JSON format to enable external applications to consume and visualize them in different ways. The base URL for the REST API is `http://<master_node_hostname>:4040/api/v1`. For the history server, the URL is `http://<history_server_hostname>:18080/api/v1`. Listing 7-5 shows how to access a list of all applications.

Listing 7-5. Accessing the RESTful API for Spark Metrics

```
$ curl http://localhost:4040/api/v1/applications
[ {
  "id" : "socialsearchapp",
  "name" : "socialsearchapp",
  "attempts" : [ {
    "startTime" : "2015-11-26T16:46:04.363GMT",
    "endTime" : "1969-12-31T23:59:59.999GMT",
    "sparkUser" : "",
    "completed" : false
  } ]
} ]
```

Table 7-2 lists some of the important endpoint paths that are available. Instead of describing them, the table links to information from the earlier UI figures. In addition to these, you can download the logs for an application in a zipped file from `/applications/<app_id>/logs`. For YARN applications, all of these application paths contain an additional segment, `attempt_id`, because YARN applications can be attempted more than once. For instance, the location of the zipped file under YARN is `/applications/<app_id>/<attempt_id>/logs`.

Table 7-2. *REST API for Spark Application Metrics*

Path	Information in This Figure
/applications/<app_id>/jobs	7-3
/applications/<app_id>/jobs/<job_id>	7-4
/applications/<app_id>/stages/<stage_id>/<stage_attempt_id>/taskSummary	7-12
/applications/<app_id>/stages/<stage_id>/<stage_attempt_id>/taskList	7-13
/applications/<app_id>/storage/rdd	7-14
/applications/<app_id>/storage/rdd/<rdd_id>	7-15
/applications/<app_id>/executors	7-16

Logging

It is always a good idea to instrument applications to generate detailed logs for debugging and auditing, But obviously there is no such thing as a free lunch—or free storage, in this case. It is very easy to run out of space on your cluster once logs begin to fill up. To circumvent this problem, logs should be rolled periodically. Spark provides this functionality out of the box via the configuration parameters listed in Table 7-3.

Table 7-3. *Logging Configuration Parameters*

Parameter	Details
`spark.executor.logs.rolling.strategy`	The strategy to use for rolling. Options include `time` and `size`. For `size`, set `spark.executor.logs.rolling.maxSize`; and for `time`, set `spark.executor.logs.rolling.time.interval`.
`spark.executor.logs.rolling.maxRetainedFiles`	The maximum number of log files to retain.
`spark.executor.logs.rolling.maxSize`	The maximum size of a log file in bytes after which rolling kicks in.
`spark.executor.logs.rolling.time.interval`	The rolling interval. Possible values include `daily`, `hourly`, `minutely`, and any second interval.

Logging in Spark applications by default is very chatty, which causes useful information such as the output from the running application to be lost in the clutter. Spark uses Log4j, which means its log behavior can easily be configured. A reasonable setup is to suppress `DEBUG` and `INFO` messages and log the rest to an external file. The configuration in Listing 7-6 does just that. Copy it over to `$SPARK_HOME/conf` on all your machines. By default, it logs messages to `/tmp/spark.log`. Change `log4j.appender.FILE.File` to another location if desired.

Listing 7-6. Log4j Configuration to Make Spark Applications Less Chatty

```
log4j.rootLogger=WARN, FILE
log4j.rootCategory=WARN, FILE

log4j.appender.FILE=org.apache.log4j.FileAppender
log4j.appender.FILE.File=/tmp/spark.log
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.ConversionPattern=%d{yy/MM/dd HH:mm:ss} %p %c{1}: %m%n

log4j.logger.org.spark-project.jetty=WARN
log4j.logger.org.spark-project.jetty.util.component.AbstractLifeCycle=ERROR
log4j.logger.org.apache.spark.repl.SparkIMain$exprTyper=INFO
log4j.logger.org.apache.spark.repl.SparkILoop$SparkILoopInterpreter=INFO
```

External Metrics

Along with UI-enabled metrics, Spark can optionally emit metrics to a number of external sinks. The mechanism works out of the box with some popular metrics consumers such as JMX and Graphite. This is enabled via the Dropwizard metrics library.[8] Metrics to zero or more sinks are emitted for different processes including master, workers, executors, applications, and drivers.

[8]`http://metrics.dropwizard.io/3.1.0/`.

Metrics configuration resides at `$SPARK_HOME/conf/metrics.properties`. To turn on the Graphite sink, for example, add the configuration information from Listing 7-7 to the `metrics.properties` file and fill in `graphite_hostname` and `graphite_port`.

Listing 7-7. Graphite Metrics Sink Configuration

```
*.sink.graphite.class=org.apache.spark.metrics.sink.GraphiteSink
*.sink.graphite.host=<graphite_hostname>
*.sink.graphite.port=<graphite_port>
*.sink.graphite.period=10
```

Let's view Graphite metrics for the application from Listing 7-4. Graphite[9] is a system for storing and visualizing metrics from various applications. It consists of three major components: Carbon, a series of daemons to collect data; Whisper, a database to store time series; and web-app, a Python web service to render metrics. Graphite is great at collecting and storing data, but its visualization features are limited. To overcome this, you can use Grafana,[10] which has richer graphing and aggregation options and lets you create dynamic dashboards. Grafana seamlessly integrates with data sources including Graphite, which permits you to publish metrics to Graphite but visualize them via Grafana.

Both Graphite and Grafana have a number of dependencies (the former is in Python, and the latter is in Go), so setting them up is nontrivial. Instead of taking the tedious manual-installation route, let's use a Docker image with Graphite and Grafana preinstalled. Docker[11] images simplify the deployment of applications because each image is self contained and runs cross-platform.[12] These images are hosted in a central repository from where they are fetched and executed.

Let's use a publicly available Graphite + Grafana image.[13] Execute it by running

```
docker run -v /data/graphite:/var/lib/graphite/storage/whisper -p 80:80 -p 3000:3000 -p
2003:2003 -p 2004:2004 -p 7002:7002 -p 8125:8125/udp -p 8126:8126 -d nickstenning/graphite
```

The command fetches and launches the image and also sets up iptables forwarding rules on the host machine. The various ports, which are exposed in the host machine, belong to the running services. Note that the `<graphite_hostname>` in Listing 7-7 should be the IP of the Docker container.[14] The image comes fully loaded, and only Grafana needs to be configured to consume Graphite data.[15] Start the Spark application, head over to the Grafana web app running on `container_hostname:3000` in your browser, and sign in (default login and password: admin).

Setting up graphs to view different Spark metrics requires the creation of at least one dashboard in Grafana. Create a dashboard, and add a graph to it. The format for all metrics for streaming applications is `app_id.process.app_name.StreamingMetrics.streaming.metric`. Figure 7-22 shows the graph for `totalDelay`. Use a wildcard (`app_id.process.app_name.StreamingMetrics.streaming.*`) to enlist all the available metrics.

[9]http://graphite.readthedocs.org/en/latest/index.html.
[10]http://grafana.org/.
[11]www.docker.com/.
[12]Install Docker on a *nix system by running `wget -qO- https://get.docker.com/ | sh`. Warm up Docker by executing `docker-machine env default` followed by `eval "$(docker-machine env default)"`.
[13]https://github.com/SamSaffron/graphite_docker.
[14]Find out the IP of a container via `docker-machine ip <container_id>`.
[15]Use your browser to jump to the Grafana dashboard on port 3000, and choose Data Sources from the menu on the extreme left. Then click Add New, and enter http://localhost:80 as the URL.

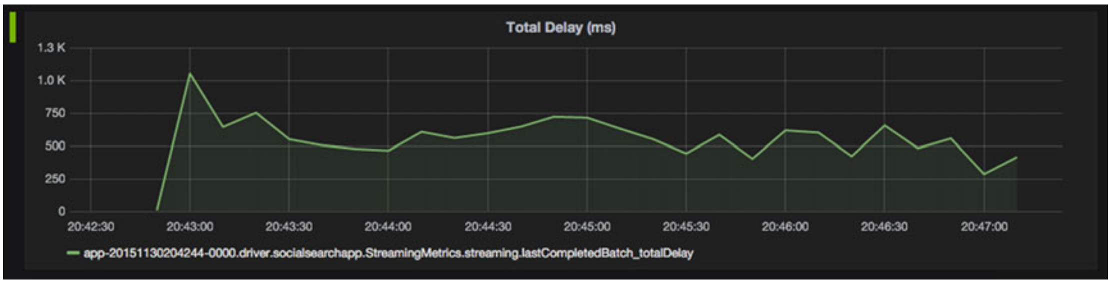


Figure 7-22. *Grafana graph for a Spark Streaming application time series*

System Metrics

The system-resource utilization of applications can also give you a window into their dynamics and behavior. A number of tools exist for collecting and visualizing these system-wide numbers and they have their own dependencies. It is always a good idea to keep centralized dashboards for all types of metrics. So, let's visualize system metrics in Grafana as well. How do you collect and send these metrics to Grafana?

collectd[16] is a lightweight Unix daemon that collects and ships out various configurable system stats.[17] Each type of metric is collected via a plug-in. For instance, standalone plug-ins exist for CPU, memory, and network metrics. In addition, these numbers can be sent to external systems, such as Graphite/Grafana for visualization, also via plug-ins. For example, to send metrics to Graphite/Grafana, you need to enable the plug-in for it,[18] point it to your Grafana setup as shown in Listing 7-8, and fill in `grafana_docker_ip`. In addition, let's configure collectd to send CPU and memory numbers to Grafana for visualization: just add `LoadPlugin cpu` and `LoadPlugin memory` to `collectd.conf`.

Listing 7-8. Graphite/Grafana Plug-in Configuration for collectd

```
<Plugin write_graphite>
  <Node "example">
    Host grafana_docker_ip
    Port "2003"
    Protocol "tcp"
    LogSendErrors true
    Prefix "collectd"
    Postfix "collectd"
    StoreRates true
    AlwaysAppendDS false
    EscapeCharacter "_"
  </Node>
</Plugin>
```

Figure 7-23 shows the CPU and memory graphs for the social versus search referral application. The application was kicked off at 12:13; hence the decrease in CPU idle type and an increase in system and user time. System-resource utilization can be useful for working out whether an application is CPU or IO bound and taking the appropriate action: for instance, scaling out by adding machines with more memory.

[16]`https://collectd.org/`.
[17]To set up collectd, download it from `https://collectd.org/download.shtml` and install it on all machines in your cluster via `./configure; make all install`.
[18]Add `LoadPlugin write_graphite` to your `collectd.conf`, which is typically located at `/etc/collectd.conf`.

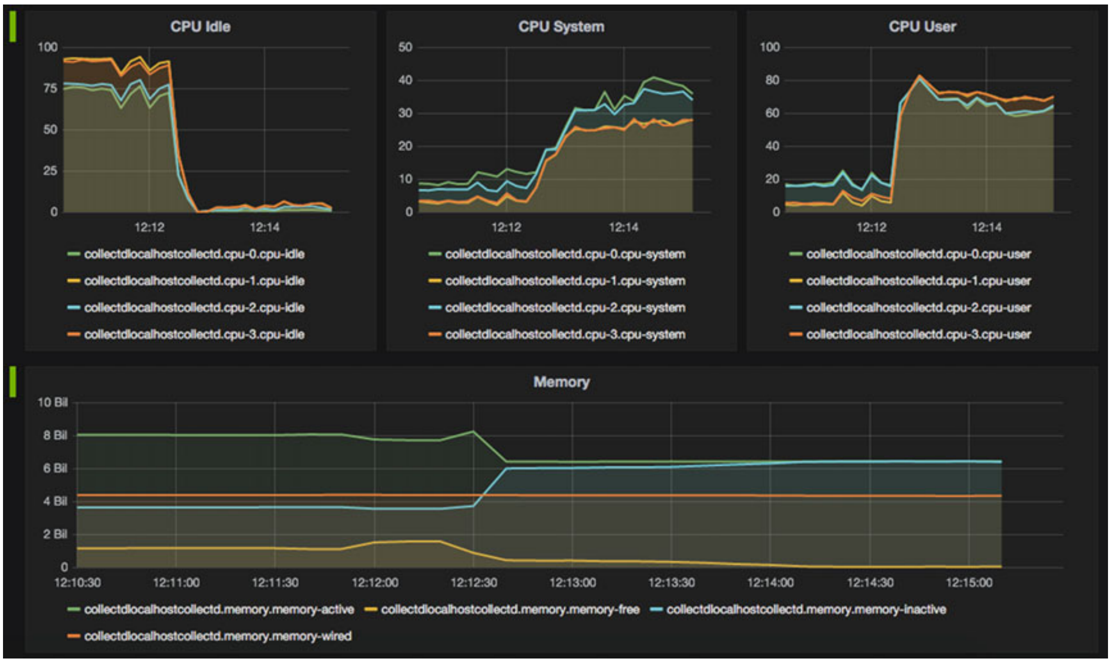


Figure 7-23. *Grafana screenshot of system metrics collected via collectd*

Monitoring and Alerting

The last piece of the puzzle is monitoring and alerting. Systems and applications fail all the time—especially in cluster environments where commodity off-the-shelf components have been cobbled together. Automatic monitoring tools can be configured to raise a flag when something is amiss and alert you. The de facto standard for monitoring and alerting in a distributed environment is Nagios.[19] It comes prepackaged with a number of checks, including ping, number of total processes, and swap usage.[20] In addition, Nagios Exchange contains hundreds of plug-ins customized for each individual system/service,[21] including Big Data systems, databases, and web servers. It can also run arbitrary Perl scripts for service checks. Fortunately, the exchange also contains a service check script for Spark.[22] It simply uses the Spark HTTP interface to check for master, worker, and application status.

To configure Nagios to monitor Spark, you first need to add the `script`[23] command (Listing 7-9) to `command.cfg`, which is typically located at `/etc/nagios/objects/commands.cfg`.

[19]`https://www.nagios.org/`.

[20]Download Nagios Core from `https://www.nagios.org/downloads/nagios-core`. The installation process is pretty standard: `./configure` followed by `make all` and `make install`. Please refer to the Nagios web site for more information.

[21]`https://exchange.nagios.org/directory/Plugins`.

[22]`https://exchange.nagios.org/directory/Plugins/Clustering-and-High-2DAvailability/check_spark_cluster-2Epl-(Advanced-Nagios-Plugins-Collection)/details`.

[23]`git clone https://github.com/harisekhon/nagios-plugins`.

Listing 7-9. Nagios Command Configuration for Spark

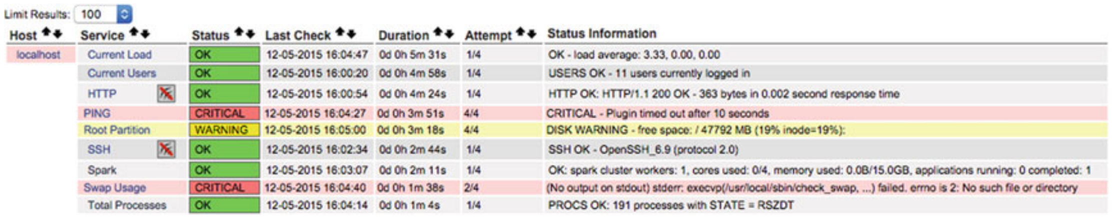
```
define command{
        command_name    check_spark_cluster
        command_line    perl -T <path/to>/nagios-plugins/check_spark_cluster.pl -H
        $ARG1$
        }
```

Each host in a Nagios cluster needs to have a configuration file wherein it lists all the services and commands for which it needs to be monitored. This file is typically located at `/etc/nagios/objects/localhost.cfg`. Let's configure one machine to invoke the Spark script, as shown in Listing 7-10.

Listing 7-10. Nagios Spark Command Configuration for a Host

```
define service{
        use                         local-service
        host_name                   localhost
        service_description         Spark
        check_command               check_spark_cluster!localhost
        notifications_enabled       1
        }
```

You can now use Nagios to monitor and raise alerts for Spark. Figure 7-24 shows service monitoring for a host. To access this page, go to the Nagios web service[24] and select Services from the menu. You can see that Spark is running correctly but the machine seems to be running out of swap space and is not responding to pings. These alerts are priceless for diagnosing machine failure.

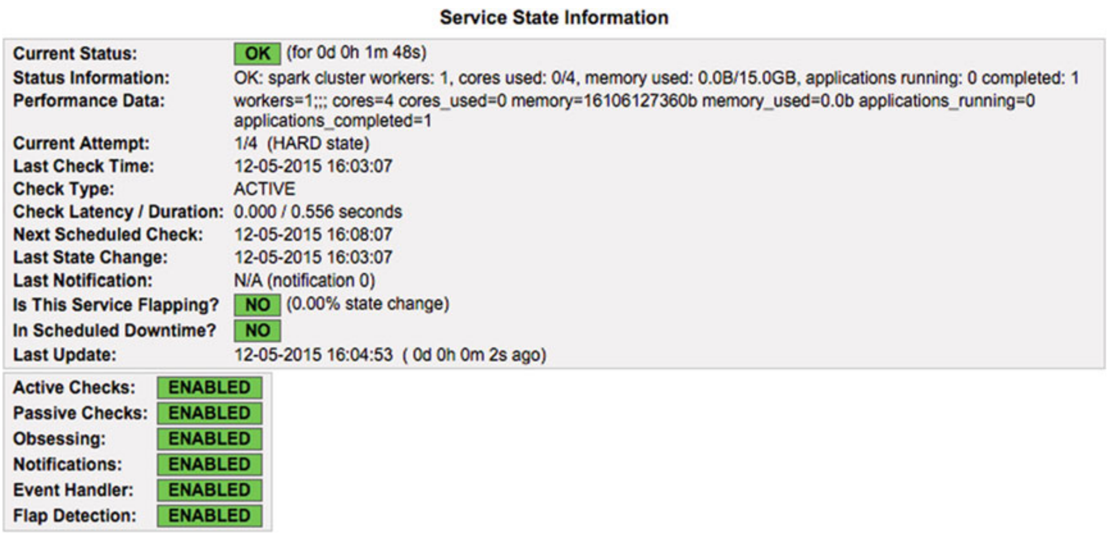

Limit Results: 100

Host	Service	Status	Last Check	Duration	Attempt	Status Information
localhost	Current Load	OK	12-05-2015 16:04:47	0d 0h 5m 31s	1/4	OK - load average: 3.33, 0.00, 0.00
	Current Users	OK	12-05-2015 16:00:20	0d 0h 4m 58s	1/4	USERS OK - 11 users currently logged in
	HTTP	OK	12-05-2015 16:00:54	0d 0h 4m 24s	1/4	HTTP OK: HTTP/1.1 200 OK - 363 bytes in 0.002 second response time
	PING	CRITICAL	12-05-2015 16:04:27	0d 0h 3m 51s	4/4	CRITICAL - Plugin timed out after 10 seconds
	Root Partition	WARNING	12-05-2015 16:05:00	0d 0h 3m 18s	4/4	DISK WARNING - free space: / 47792 MB (19% inode=19%):
	SSH	OK	12-05-2015 16:02:34	0d 0h 2m 44s	1/4	SSH OK - OpenSSH_6.9 (protocol 2.0)
	Spark	OK	12-05-2015 16:03:07	0d 0h 2m 11s	1/4	OK: spark cluster workers: 1, cores used: 0/4, memory used: 0.0B/15.0GB, applications running: 0 completed: 1
	Swap Usage	CRITICAL	12-05-2015 16:04:40	0d 0h 1m 38s	2/4	(No output on stdout) stderr: execvp(/usr/local/sbin/check_swap, ...) failed. errno is 2: No such file or directory
	Total Processes	OK	12-05-2015 16:04:14	0d 0h 1m 4s	1/4	PROCS OK: 191 processes with STATE = RSZDT

Figure 7-24. *Nagios service monitoring for a host*

You can drill down into the details of a service by clicking its name. Figure 7-25 shows the Service State Information page for Spark. You can see that it is running as expected and all checks have passed. In a production-grade Spark environment, monitoring and alerting should be enabled for all components including HDFS, any data sources such as Kafka, and individual applications.

[24] `http://<nagios_host>/nagios`.



Service State Information

Current Status:	OK (for 0d 0h 1m 48s)
Status Information:	OK: spark cluster workers: 1, cores used: 0/4, memory used: 0.0B/15.0GB, applications running: 0 completed: 1
Performance Data:	workers=1;;; cores=4 cores_used=0 memory=16106127360b memory_used=0.0b applications_running=0 applications_completed=1
Current Attempt:	1/4 (HARD state)
Last Check Time:	12-05-2015 16:03:07
Check Type:	ACTIVE
Check Latency / Duration:	0.000 / 0.556 seconds
Next Scheduled Check:	12-05-2015 16:08:07
Last State Change:	12-05-2015 16:03:07
Last Notification:	N/A (notification 0)
Is This Service Flapping?	NO (0.00% state change)
In Scheduled Downtime?	NO
Last Update:	12-05-2015 16:04:53 (0d 0h 0m 2s ago)

Active Checks:	ENABLED
Passive Checks:	ENABLED
Obsessing:	ENABLED
Notifications:	ENABLED
Event Handler:	ENABLED
Flap Detection:	ENABLED

Figure 7-25. *Spark Service State Information via Nagios*

Summary

Stream-processing applications have become mission critical for a number of industry verticals. But to have business value, each application needs to be optimized for a particular use case. Once it has been deployed, it also needs to be monitored to ensure application availability and stability. In case of failure, the appropriate alerts must be sent out so that the right action can be taken for service recovery. This chapter explored all of these topics, ranging from application performance and optimization through the use of Tachyon and the Spark UI to metrics visualization via Grafana and collectd, and monitoring and alerting through Nagios.

So far in this book, you have employed the Spark Scala API to implement applications. Although it is an extremely powerful mechanism to manipulate RDDs to realize user logic, it is still too low-level for common ETL and statistical tasks. Data scientists prefer an interface that simplifies data cleaning and analysis. Additionally, data-mining and machine-learning applications are common. So, the next chapter introduces SparkSQL and SparkR.

CHAPTER 8

Real-Time ETL and Analytics Magic

When Jeff has trouble sleeping, he MapReduces sheep.

—Jeff Dean Facts

Data (big or otherwise) has been woven into the fabric of most businesses. The world is at a stage where Big Data directly drives corporate strategy. To maintain a competitive edge, most businesses try to run their analytics pipeline in near real-time. Although this captures the behavior of a large class of applications that rely on unstructured data, it is not exhaustive: a significant chunk of data sources are structured, and their analysis applications require data-warehousing capabilities. One way to handle these requirements is to blend the existing Spark API with an external warehousing solution such as Hive, but this is a marriage of convenience rather than a natural fit: data must be copied back and forth, not to mention the burden of maintaining two different APIs. A better solution is Spark SQL.

Spark SQL, as the name suggests, is Spark's interface for analyzing structured data. This data resides in *data frames*, which are relational tables optimized for Spark. It also works out of the box with SQL, Hive, and R. Spark SQL by design permits you to blend declarative queries with existing Spark code in Scala. It is the topic of discussion in this chapter. The dataset you use in the chapter represents one of the major success stories for Big Data: telecommunications. You begin by getting your hands dirty with the DataFrame API followed by a primer in using traditional SQL to analyze streaming data. Once you've gotten a hang of Spark SQL, you interface it with Hive before using SparkR. The latter will whet your appetite for analytics before you tackle data science in detail in the next chapter.

The Power of Transaction Data Records

The most powerful and richest source of data lurks right under your nose. You use, generate, and contribute to it on an almost minute-by-minute basis. What is this fabled data?

Every time you make a call, send an SMS, or browse the Internet, you generate a transaction data record (TDR) or call data record (CDR). Each record typically contains information about the transaction metadata, network conditions, application status, and location. This rich set of details can be used to determine the network experience of each user. For instance, a telco can predict when a user is going to experience deteriorated performance while browsing the Internet. Based on this, the customer care department can proactively reach out to the customer. Similarly, another application can predict when a user is thinking of churning (switching to another network carrier). The action on part of the telco in this case can be to, say, offer the user a discount.

The potential of CDRs applies beyond network-quality improvement and customer care applications. Their diverse array of attributes are useful outside the telco world. This is primarily because 50% percent of the world's population now lives in urban areas,[1] and the number of cell phones exceeds the number of people on Earth.[2] This makes CDRs one of the most representative and inclusive datasets of people around the world. The location information from these CDRs has been used to implement applications that can predict disease outbreaks, analyze mobility patterns, study crowd dynamics, and help with transportation planning, to name a few. And this data can be married with other datasets to achieve novel use cases. For example, CDR mobility data coupled with social media data can aid in working out the geography of social networks, such as how online behavior results in purchasing in the real world. Another class of applications revolves around analyzing the browsing, search, and social interaction behavior of users based on their CDRs and micro-segmenting them based on these patterns. This information can then be used to improve the accuracy of customized ad serving.

The dataset used in this chapter is from Telecom Italia. In 2014, Telecom Italia organized a Big Data Challenge using data from CDR, grid, and other sources from November and December, 2013, spanning the cities of Milan and Trento.[3] Each tab-separated record in the CDR set has eight fields, as listed in Table 8-1.

Table 8-1. *Attributes in Each Record in the Telecom Italia Call Data Record Dataset*

Field #	Name	Description
1	`square_id`	Grid ID. All data is spatially aggregated into a grid whose exact coordinates are presented in a separate grid dataset.
2	`time_interval`	The Unix epoch timestamp of the event. Events are aggregated into temporal windows of 10 minutes. The end time interval can be obtained by adding 600,000 milliseconds to this value.
3	`country_code`	The country calling code of the event (either in or out).
4	`sms_in_activity`	The incoming SMS activity.
5	`sms_out_activity`	The outgoing SMS activity.
6	`call_in_activity`	The incoming call activity.
7	`call_out_activity`	The outgoing call activity.
8	`internet_traffic_activity`	The Internet upload and download activity.

Note that the dataset has been both spatially aggregated (into a grid) and temporally aggregated (into 10-minute windows). For obvious anonymity reasons, the activity values do not represent an exact volume but rather a normalized value. A CDR is generated each time an SMS is sent or received or a call is placed or received. On the other hand, a CDR for Internet activity is generated each time a session is started or ended. In addition, an Internet activity CDR is generated every 15 minutes or every 5 MB of download/upload in a session. The mapping of grid ID to coordinates is available in another dataset in the same repository. For each grid ID, it encodes the coordinates in the form of a polygon in GeoJSON[4] format. You focus on the Milan dataset in this chapter.

Without further adieu, let's implement your first streaming Spark SQL application.

[1]Charles Clover, "Urban Population to Exceed 50 Percent," *The Telegraph*, June 27, 2007, "`www.telegraph.co.uk/news/earth/earthnews/3298527/Urban-population-to-exceed-50-per-cent.html`.

[2]Joshua Pramis, "Number of Mobile Phones to Exceed World Population by 2014," Digital Trends, February 28, 2013, `www.digitaltrends.com/mobile/mobile-phone-world-population-2014/`.

[3]Open Big Data, *Dandelion*, `https://dandelion.eu/datamine/open-big-data/`.

[4]`http://geojson.org/`.

First Streaming Spark SQL Application

Inter-country communication is generally more expensive than local communication. To determine whether users are placing international calls and sending international texts, telcos are interested in a real-time dashboard where they can get this information. This is exactly what the Spark SQL code in Listing 8-1 achieves. Note that the application expects its input stream via a socket. To feed the socket, use your trusty friend SocketDriver, from Chapter 5. It reads the zipped CDR files and replays them over the socket to the streaming application (Listing 8-1) at the other end.

Listing-8-1. Using Data Frames to Analyze Streaming CDRs

```
package org.apress.prospark

import scala.reflect.runtime.universe

import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.SQLContext
import org.apache.spark.sql.functions.desc
import org.apache.spark.streaming.Seconds
import org.apache.spark.streaming.StreamingContext

object CdrDataframeApp {

  case class Cdr(squareId: Int, timeInterval: Long, countryCode: Int,
    smsInActivity: Float, smsOutActivity: Float, callInActivity: Float,
    callOutActivity: Float, internetTrafficActivity: Float)

  def main(args: Array[String]) {
    if (args.length != 4) {
      System.err.println(
        "Usage: CdrDataframeApp <appname> <batchInterval> <hostname> <port>")
      System.exit(1)
    }
    val Seq(appName, batchInterval, hostname, port) = args.toSeq

    val conf = new SparkConf()
      .setAppName(appName)
      .setJars(SparkContext.jarOfClass(this.getClass).toSeq)

    val ssc = new StreamingContext(conf, Seconds(batchInterval.toInt))

    val sqlC = new SQLContext(ssc.sparkContext)
    import sqlC.implicits._

    val cdrStream = ssc.socketTextStream(hostname, port.toInt)
      .map(_.split("\\t", -1))
      .foreachRDD(rdd => {
        val cdrs = seqToCdr(rdd).toDF()

41.         cdrs.groupBy("countryCode").count().orderBy(desc("count")).show(5)
42.       })
43.
44.     ssc.start()
45.     ssc.awaitTermination()
46.   }
47.
48.   def seqToCdr(rdd: RDD[Array[String]]): RDD[Cdr] = {
49.     rdd.map(c => c.map(f => f match {
50.       case x if x.isEmpty() => "0"
51.       case x => x
52.     })).map(c => Cdr(c(0).toInt, c(1).toLong, c(2).toInt, c(3).toFloat,
53.       c(4).toFloat, c(5).toFloat, c(6).toFloat, c(7).toFloat))
54.   }
55. }
```

The application starts by creating a `StreamingContext` (line 31), because you need to consume streaming data. The SQL interface is driven by the `SQLContext`, which takes as input a `SparkContext` object (line 33). The application listens on the socket for CDRs (line 36) and parses the fields of each tab-delimited record. It is important to mention that if any activity type is missing during a time interval, the dataset adds an empty string for it. To preserve these empty strings in the tokenized array, you pass -1 to the String `split` method (line 37).

Individual RDDs can be converted directly into data frames via implicit conversions (line 34), but the API does not provide any functionality for the conversion of `DStreams`. To overcome this, a `foreachRDD` action can be used to convert each RDD in the stream to a data frame prior to further processing (line 38). You also preprocess the fields to make sure you replace each empty field with a value of 0 (lines 49–51) in a custom function so that type conversion can be simplified in the next step. Records in data frames need to be structured—that is, they need to have typed key-value semantics. To achieve this, the application creates a Scala `case class` to represent a CDR (line 15). Each record on the wire is converted to this format (line 52). The `toDF` method is used to convert an RDD to a data frame (line 39). Under the hood, Spark SQL infers the column types via reflection.

All this can be considered job setup boilerplate code, because the core logic of the application is encapsulated in a single line. Line 41 performs a `groupBy` operation on the data frame with `countryCode` as the grouping key. You then invoke the `count` method to count the frequency of each `countryCode` before ordering the records by `count` in descending order. You peek into these ordered rows by viewing the top five (Listing 8-2). `show` is an output action that causes the entire query to be executed.

Listing 8-2. Output of One Batch Interval for the Country Code Ranking Application

```
+-----------+------+
|countryCode| count|
+-----------+------+
|         39|157323|
|          0|125838|
|         46| 27309|
|         33| 18145|
|         49| 14939|
+-----------+------+
```

That's it! It really is that simple to use data frames to manipulate structured data. But the DataFrame API is not the only way to analyze structured data. In fact, Spark SQL enables you to directly invoke standard SQL, as you see shortly.

To run the application, add the following to the simple build-tool definition file:

```
libraryDependencies += "org.apache.spark" %% "spark-sql" % "1.4.0"
```

The execution mechanism remains the same as that for standard Spark applications. The results are as expected: the top five codes are all in Europe, and the lion's share of calls are intra-Italy (country code: 39).

Now let's get friendly with `SQLContext`.

SQLContext

`SQLContext` is to Spark SQL as `SparkContext` is to batch Spark and `StreamingContext` is to streaming data. This means it is the central entity to create data frames. These data frames can be converted from RDDs, various Hive table formats, and external databases. `SQLContext` is also in charge of query scheduling and execution. You walk through the many facets of `SQLContext` next.

Data Frame Creation

Data frames constitute the recommended and optimum mechanism to manipulate structured data in Spark. Data frames can be created from various other formats using `SQLContext`. In addition to direct conversion from RDDs, as illustrated in Listing 8-1, the following options are available.

Existing RDDs

```
createDataFrame[A <: Product](rdd: RDD[A]): DataFrame
```

Explicitly converts an RDD to a data frame, as shown in Listing 8-3. It is a drop-in replacement for line 39 in Listing 8-1.

Listing 8-3. Explicitly Converting an RDD to a Data Frame

```
val cdrs = sqlC.createDataFrame(seqToCdr(rdd))
```

Dynamic Schemas

```
createDataFrame(rowRDD: RDD[Row], schema: StructType): DataFrame
```

Transforms an RDD of type `Row` to a data frame. `Row` is a generic representation of a row in a data frame. The most powerful feature of this variant of `createDataFrame` is that it can be provided with a schema on the fly. This is useful for scenarios where you do not know the schema of the input data up front and hence cannot create a case class for it as you did in Listing 8-1. This schema needs to be a special Spark SQL type: `StructType`. Listing 8-4 illustrates how you can read a JSON schema from a file (line 1) and convert it to a `StructType` object (line 2). You can then use this schema to dynamically create data frames (line 7). Listing 8-5 contains the JSON schema for the CDR data.

Listing 8-4. Dynamically Supplying the Schema of a Data Frame

```
1.   val schemaJson = scala.io.Source.fromFile(schemaFile).mkString
2.   val schema = DataType.fromJson(schemaJson).asInstanceOf[StructType]
3.
4.   val cdrStream = ssc.socketTextStream(hostname, port.toInt)
5.     .map(_.split("\\t", -1))
```

```
    .foreachRDD(rdd => {
      val cdrs = sqlC.createDataFrame(rdd.map(c => Row(c: _*)), schema)

      cdrs.groupBy("countryCode").count().orderBy(desc("count")).show(5)
    })
```

Listing 8-5. JSON Schema of the CDR Dataset

```
{
  "type": "struct",
  "fields": [
    {
      "name": "squareId",
      "nullable": false,
      "type": "integer"
    },
    {
      "name": "timeInterval",
      "nullable": false,
      "type": "long"
    },
    {
      "name": "countryCode",
      "nullable": true,
      "type": "string"
    },
    {
      "name": "smsInActivity",
      "nullable": true,
      "type": "float"
    },
    {
      "name": "smsOutActivity",
      "nullable": true,
      "type": "float"
    },
    {
      "name": "callInActivity",
      "nullable": true,
      "type": "float"
    },
    {
      "name": "callOutActivity",
      "nullable": true,
      "type": "float"
    },
    {
      "name": "internetTrafficActivity",
      "nullable": true,
      "type": "float"
    }
  ]
}
```

In this example, you read the JSON schema from a file; in practice, it could dynamically come from another input stream. This highlights the fact that the data in the main stream can change on the fly, and you can switch its schema in every batch interval. At any point in the lifecycle of a data frame, its schema can be printed by calling the `printSchema()` function.

Scala Sequence

`createDataFrame[A <: Product](data: Seq[A]): DataFrame`

Turns a Scala sequence into a data frame (see Listing 8-6). Note that this snippet of code is only for illustrative purposes. It should not be used in production, because it reads an entire RDD into the driver memory before converting it to a data frame.

Listing 8-6. Creating a Data Frame from a Scala Sequence

```
val cdrs = sqlC.createDataFrame(seqToCdr(rdd).collect())
```

RDDs with JSON

`read.json(jsonRDD: RDD[String]): DataFrame`

Converts an RDD of stringified JSON objects to a data frame (see Listing 8-7). Another drop-in replacement for Listing 8-1, line 39.

Listing 8-7. Converting an RDD of JSON Objects to a Data Frame

```
val cdrsJson = seqToCdr(rdd).map(r => {
  implicit val formats = DefaultFormats
  write(r)
})
val cdrs = sqlC.read.json(cdrsJson)
```

`read` returns a `DataFrameReader` object, which can be used to create data frames from many input sources. More examples follow.

External Database

`read.jdbc(url: String, table: String, properties: Properties): DataFrame`

Connects to the database located at `url` via JDBC while using connection `properties`. Converts the referenced `table` to a data frame.

Parquet

`read.parquet(paths: String*): DataFrame`

Returns the Parquet file located at `paths`[5] as a data frame. Parquet is a compressed columnar data representation used by a number of Hadoop projects include Hive and Pig.

[5]A Parquet table is typically made up of more than one file, which may be located at multiple locations.

Hive Table

```
read.table(tableName: String): DataFrame
```

Fetches the table `tableName` from the Hive metastore and returns it as a data frame. Note that this feature requires the use of a `HiveContext` object, which inherits from `SQLContext`, as described later in this chapter.

SQL Execution

Data frames provide natural primitives for manipulating structured data, but some analysts prefer to use SQL directly. This may also be preferable for legacy reasons: an organization may have hundreds of SQL queries, and it could require a bit of engineering effort to translate them to the data frame API. For such scenarios and more, `SQLContext` lets you invoke SQL queries via `SQLContext#sql("<sql query>")`. Listing 8-8 contains replacement code for line 41 in Listing 8-1. Instead of using the DataFrame API to achieve application logic, it uses SQL while keeping the output semantics the same. Note that because SQL can only be executed against an existing table, you save the CDRs data frame (line 39 in Listing 8-1) as a temporary in-memory table (line 1) first. You can then reference it as table `cdrs` in the SQL (line 2). In the next section, you learn how to save tables externally.

Listing 8-8. Running a SQL Query Using Spark SQL

```
1.   cdrs.registerTempTable("cdrs")
2.   sqlC.sql("SELECT countryCode, COUNT(countryCode) AS cCount FROM cdrs GROUP BY
     countryCode ORDER BY cCount DESC LIMIT 5").show()
```

Providing the same name to `registerTempTable` in every batch causes it to overwrite the previous table. If your logic requires you to maintain an individual table for each interval, give it a unique name by, say, using

```
cdrs.registerTempTable("cdrs" + rdd.id)
```

Be cautious with the fact that temporary tables remain in memory and can overwhelm the JVM heap. To negate that, `SQLContext` allows tables to be dropped via

```
dropTempTable(tableName: String): Unit
```

Listing 8-9 shows how you can drop the temporary table that you created for SQL analysis in Listing 8-8 after you are done with it in every batch interval.

Listing 8-9. Dropping a Temporary, In-Memory Table

```
sqlC.dropTempTable("cdrs")
```

Configuration

Configuration parameters for Spark SQL can be set (`setConf(key: String, value: String): Unit`) and retrieved (`getConf(key: String): String`) via `SQLContext`. In general, none of the Spark SQL configuration parameters need to be manually tweaked, because Spark SQL performs most optimizations under the hood, as you see shortly. The only parameter that may need to be explicitly tweaked is `spark.sql.shuffle.partitions`, which controls the number of partitions used for joins and aggregations (default value: 200). If you perform frequent joins, you should also play with `spark.sql.autoBroadcastJoinThreshold` (default value: 10 MB), which controls the use of broadcast hash joins by the underlying query optimizer for smaller tables.

Other potential targets are parameters that relate to the Parquet format. Look for `spark.sql.parquet.*` configuration keys.

User-Defined Functions

In most applications, SQL and the DataFrame API do not suffice to realize user logic. For instance, in the previous example, you tallied the top five countries by their SMS, call, and Internet usage activity. The dataset only contains country-calling codes, so an analyst going through the results needs to manually look up the country associated with each code. It would be handy to somehow convert these codes to country names. SQL allows user-defined functions (UDFs) to be embedded in a query. These user-defined functions are pushed down into the database layer for efficient processing. `SQLContext` also permits the registration and application of UDFs:

```
udf.register(name: String, func: ())
```

`name` can then be used in the SQL query to refer to the UDF. Note that `func` can have up to 22 arguments. Listing 8-10 contains code for creating and registering a UDF to map country-calling codes to country names.

Listing 8-10. Creating and Registering a UDF

```
1.   def getCountryCodeMapping() = {
2.     implicit val formats = org.json4s.DefaultFormats
3.     parse(Source.fromURL("http://country.io/phone.json").mkString).extract[Map[String,
       String]].map(_.swap)
4.   }
5.
6.   def getCountryNameMapping() = {
7.     implicit val formats = org.json4s.DefaultFormats
8.     parse(Source.fromURL("http://country.io/names.json").mkString).extract[Map[String,
       String]]
9.   }
10.
11.  def getCountryName(mappingPhone: Map[String, String], mappingName: Map[String, String],
     code: Int) = {
12.    mappingName.getOrElse(mappingPhone.getOrElse(code.toString, "NotFound"), "NotFound")
13.  }
14.
15.  val getCountryNamePartial = getCountryName(getCountryCodeMapping(),
     getCountryNameMapping(), _: Int)
16.
17.  sqlC.udf.register("getCountryNamePartial", getCountryNamePartial)
```

The code first uses a function to retrieve a mapping of country codes to country-name abbreviations from an online source (lines 1–4). It also contains another function to map country-name abbreviations to full country names, again using an online source (lines 6–9). The `getCountryName` method takes as input the previous two maps and a country code and returns the name of the country that corresponds to that code (line 11–13). Invoking this method for each output row is very inefficient, because it requires two downloads per invocation. To remedy this, you create a partially applied version of the function that is prepopulated with the two maps (line 15). You finally register this function as a UDF with `SQLContext` with the name `getCountryNamePartial` (line 17).

Listing 8-11 updates the SQL query from Listing 8-8 to use the UDF, resulting in output tables of the form shown in Listing 8-12.

Listing 8-11. Invoking a UDF in a SQL Query

```
sqlC.sql("SELECT getCountryNamePartial(countryCode) AS countryName, COUNT(countryCode)
AS cCount FROM cdrs GROUP BY countryCode ORDER BY cCount DESC LIMIT 5").show()
```

Listing 8-12. Output for One Batch of the Country Activity-Ranking Application

```
+-----------+------+
|countryName|cCount|
+-----------+------+
|      Italy|180653|
|   NotFound|139872|
|     Sweden| 28130|
|     France| 16970|
|    Germany| 16528|
+-----------+------+
```

UDFs are extremely powerful for pushing down simple logic into the database layer for execution, but their utility is limited in cases with more complex transformation scenarios. For such applications, Spark SQL lets you blend data frames seamlessly with RDD operations, as you see shortly.

Catalyst: Query Execution and Optimization

`SQLContext` under the hood uses an optimizer dubbed Catalyst that converts queries into logical plans. For a given query, it can generate a number of physical plans from the logical plan, out of which it selects one as the optimum physical plan based on a cost model. This physical plan is then sent off to the Spark processing engine. Catalyst uses features of Scala to generate byte code for queries. It also tries to enforce predicate pushdown to improve performance. By design, it is extensible: users can add support for different data sources and new data types. This interface has been used to add external support for various sources including Avro and CSV.[6]

HiveContext

`HiveContext` is an extension of `SQLContext` that gives you access to HiveQL in addition to all the features provided by `SQLContext`: it is a drop-in replacement for `SQLContext`. Including `spark-hive` as a build dependency can enable it. In fact, it is the recommended choice, because it contains a superset of the features of `SQLContext` in addition to the rich features of HiveQL, existing Hive UDFs, and access to existing Hive data tables. To explore some of these extra features, let's write code that implements another application to tap the telco dataset.

Cellular networks rely on accurate traffic-demand numbers to optimize current systems and plan for future capacity. In the short term, traffic volume can drive, say, bandwidth distribution and power-supply management. This can be done on an hourly scale due to the availability of real-time network data. In the long term, these statistics can be used to increase the number of base stations and overall network coverage. Bearing this mind, let's use HiveQL to determine the hourly network activity (call + SMS + Internet) in the Telecom Italia data (Listing 8-13).

[6]`http://spark-packages.org/package/databricks/spark-csv`.

Listing 8-13. Calculating Hourly Network Activity Using HiveQL

```
1.   val cl = Thread.currentThread().getContextClassLoader()
2.   val hiveC = new HiveContext(ssc.sparkContext)
3.   Thread.currentThread().setContextClassLoader(cl)
4.
5.   import hiveC.implicits._
6.
7.   val cdrStream = ssc.socketTextStream(hostname, port.toInt)
8.     .map(_.split("\\t", -1))
9.     .foreachRDD(rdd => {
10.      seqToCdr(rdd).toDF().registerTempTable("cdrs")
11.
12.      hiveC.sql("SET DATE_FMT='yy-MM-dd|HH'")
13.      hiveC.sql("SELECT from_unixtime(timeInterval, ${hiveconf:DATE_FMT}) AS TS,
         SUM(smsInActivity + smsOutActivity + callInActivity + callOutActivity +
         internetTrafficActivity) AS Activity FROM cdrs GROUP BY from_unixtime(timeInterval,
         ${hiveconf:DATE_FMT}) ORDER BY Activity DESC").show()
14.    })
```

The first order of the day is to create a `HiveContext` from a `SparkContext` (line 2). Note that you deliberately save and reload the context class loader around `HiveContext` due to a subtle bug in Spark 1.4.0.[7] The rest of the application setup is the same as that in the previous examples: reading data from a socket (line 7) and doing the conversion to data frames (line 10) in a `foreachRDD` clause (line 9).

The key update is the use of HiveQL to implement the application logic. You need to aggregate activity by hour, so you must convert the Unix timestamp for each record into a human-readable representation. The code uses a function from the rich suite of UDFs provided by Hive to perform the timestamp standardization. You need to perform the conversion more than once (in the `SELECT` clause and then in the `GROUP BY`), so it makes sense to define the format up front and then reuse it. The `from_unixtime(bigint unixtime[, string format])` UDF takes as input an optional Java format string. Hive also allows you to set variables that can be reused in the same context. Using this feature, you first define the date format on line 12. You then use standard HiveQL (in fact, SQL) syntax to aggregate network activity by hour (line 13).

Note that using `HiveContext` also gives you the freedom to read from and write to Hive tables. This also enables applications outside of Spark to use the same data. You will look at saving tables when the chapter drills down into the details of data frames.

Data Frame

Inspired by data frames in Pandas and R, Spark SQL data frames expose the same widely used interface loved by data scientists while using the powerful Spark processing engine underneath. This results in a marriage between the best of both worlds. Conceptually a data frame is just an RDD of `Row` objects. Data frames are interoperable with other data-storage models such as RDDs. In fact, under the hood, a data frame simply maintains a reference to the underlying RDD and can access its fields in place without the need for a verbatim data copy. Similar to RDDs and `DStreams`, data frames are evaluated in a lazy fashion: an action is the only trigger for the execution of a physical plan. Logical plans, on the other hand, are eagerly evaluated to catch errors early, such as a mismatch between column names.

[7] https://issues.apache.org/jira/browse/SPARK-8368.

Data frames are also interoperable with MLlib, as you see in the next chapter, which means they can be used as a common substrate for both ETL as well as sophisticated machine learning. Let's look at some of the features of data frames in detail.

Types

The data model mimics the nested one from Hive with a number of native and complex types. Primitive types include `BooleanType`, `ByteType`, `DoubleType`, `FloatType`, `IntegerType`, `LongType`, `ShortType`, and `StringType`. The grammar also contains a few domain-specific primitive types: `BinaryType` (an array of bytes), `DateType` (year, month, and day), `DecimalType` (equivalent to `java.math.BigDecimal`), and `Timestamp` (year, month, day, hour, minute, and second). Finally, the complex types provided by data frames are `ArrayType`, `MapType`, and `StructType` (a structure with elements of type `StructField`).

Query Transformations

Each query transformation mimics a SQL statement. In the average case, these transformations suffice to implement user logic. Let's start with the simplest and the most ubiquitous one: `select`.

For all transformations, this section also presents illustrative examples. Each snippet of code assumes the existence of a CDR data frame similar to the one at line 39 of Listing 8-1. So far, you have seen only one data frame output action, `show`. Use it to view the output for the following queries if desired. You look at more output actions in the next section.

select(col: String, cols: String*): Data Frame

Selects one or more columns from a data frame. Listing 8-14 projects three columns from the CDR dataset.

Listing 8-14. Using a `select` to Project Three Columns from a Data Frame

```
cdrs.select("squareId", "timeInterval", "countryCode")
```

Another variant of `select` uses expressions to perform the selection instead of column names:

select(cols: Column*): DataFrame

Under the hood, each expression is a `Column` object. Listing 8-15 implements the same logic as Listing 8-14 using expressions. An expression starts with a dollar sign ($) and can also be turned into a Boolean condition, as you see shortly.

Listing 8-15. Column Selection Based on Expressions

```
cdrs.select($"squareId", $"timeInterval", $"countryCode")
```

filter(conditionExpr: String): DataFrame

Only keeps rows that match `conditionExpr`. Listing 8-16 uses it to restrict the data frame to records that originate from `squareId: 5`. Multiple conditions can be clumped together using logical expressions in the condition or by chaining multiple filter operations.

Listing 8-16. Filtering a Data Frame Based on a Specific Condition

```
cdrs.filter("squareId = 5")
```

drop(colName: String): DataFrame

Drops a column `colName` from the data frame, as shown in Listing 8-17.

Listing 8-17. Dropping a Column from an Existing Data Frame

```
cdrs.drop("countryCode")
```

where(condition: Column): DataFrame

A variant of filter that relies on column-based conditions. Under the hood, it makes a call to `filter()`. For instance, the code in Listing 8-18 achieves the combined effect of Listings 8-15 and 8-16.

Listing 8-18. Chaining Projection and Filtering Together

```
cdrs.select($"squareId", $"timeInterval", $"countryCode").where($"squareId" === 5)
```

limit(n: Int): DataFrame

Limits the number of rows in the output data frame to `n` (Listing 8-19).

Listing 8-19. Limiting the Number of Rows in a Data Frame to a Specific Number

```
cdrs.limit(5)
```

withColumn(colName: String, col: Column): DataFrame

Adds `col` with `colName` to an existing data frame. To illustrate the use of this operation, the code in Listing 8-20 adds a column to the CDRs dataset with the end time of each record.

Listing 8-20. Adding a Column to an Existing Data Frame

```
cdrs.withColumn("endTime", cdrs("timeInterval") + 600000)
```

groupBy(col1: String, cols: String: GroupedData

Groups data based on one or more columns and returns a `GroupedData` object, which allows different aggregations. Listing 8-21 uses a `groupBy` operation followed by a `count` aggregation to count the frequency of each square/grid. Table 8-2 lists all aggregation functions that can be invoked on `GroupedData`.

Listing 8-21. `groupBy` Followed by an Aggregation

```
cdrs.groupBy("squareId").count()
```

Table 8-2. *Aggregation Functions for GroupedData*

Method	Description	Example
`avg(colNames: String*): DataFrame`	Calculates the average value of colNames in a group. All columns are used if no colNames is specified.	`cdrs.groupBy("countryCode").avg("internetTrafficActivity")`
`mean(colNames: String*): DataFrame`	An alias for avg().	Same as above
`max(colNames: String*): DataFrame`	Finds the max of each column in colNames.	`cdrs.groupBy("countryCode").max("callOutActivity")`
`min(colNames: String*): DataFrame`	Same as the previous but calculates min.	`cdrs.groupBy("countryCode").min("callOutActivity")`
`sum(colNames: String*): DataFrame`	Calculates the sum of columns in colNames.	`cdrs.groupBy("squareId").sum("internetTrafficActivity")`
`agg(aggExpr: (String, String), aggExprs: (String, String)*): DataFrame`	Invokes all aggExprs (org.apache.spark.sql.functions).	`cdrs.groupBy("squareId").agg(sum("callOutActivity"), sum("callInActivity"), sum("smsOutActivity"), sum("smsInActivity"), sum("internetTrafficActivity"))`

agg(aggExpr: (String, String), aggExprs: (String, String)*): DataFrame

Applies one or more aggregation expressions to the data frame. This is similar in spirit to the agg() function for GroupedData, but instead of aggregated grouped data, this is applied to columns in their current form across all rows. See Listing 8-22 for an example.

Listing 8-22. Applying Aggregation Expressions across All Rows

```
cdrs.agg(sum("callOutActivity"), sum("callInActivity"), sum("smsOutActivity"),
sum("smsInActivity"), sum("internetTrafficActivity"))
```

orderBy(sortCol: String, sortCols: String*): DataFrame

Sorts the data frame by one or more columns in ascending order. To order the data in descending order, wrap up the column name in org.apache.spark.sql.functions.desc as shown in Listing 8-23. Note that orderBy is just an alias for sort(sortCol: String, sortCols: String*): DataFrame.

Listing 8-23. Ordering the Results of a groupBy in Descending Order

```
cdrs.groupBy("countryCode").sum("internetTrafficActivity").orderBy(desc("SUM(internet
TrafficActivity)"))
```

rollup(col1: String, cols: String*): GroupedData

Similar to a groupBy operation, but a subsequent aggregation also spits out a cumulative aggregation for the passed columns. For instance, the snippet of code in Listing 8-24 groups square IDs and country codes and calculates the number of rows in each group (similar to a standard groupBy). In addition, for each square ID, it also contains a cumulative group and subsequently a count for all country codes. The order of the input columns matters in this case, because switching them would create the cumulative group based on country code instead of square ID.

Listing 8-24. Performing a rollup Operation for Cumulative Grouping

```
cdrs.rollup("squareId", "countryCode").count()
```

cube(col1: String, cols: String*): GroupedData

Similar to a rollup but creates groups for all permutations of columns. As a result, the order of the input columns does not matter. See Listing 8-25 for an example.

Listing 8-25. Performing a cube Operation for Grouping across All Columns

```
cdrs.cube("squareId", "countryCode").count()
```

dropDuplicates(colNames: Seq[String]): DataFrame

Drops rows in which the row values in the provided columns, colNames, are the same. For example, Listing 8-26 drops all rows where the incoming and outgoing call activity are the same.

Listing 8-26. Using dropDuplicates to Cull Rows Where the Columns Have the Same Value

```
cdrs.dropDuplicates(Array("callOutActivity", "callInActivity"))
```

sample(withReplacement: Boolean, fraction: Double): DataFrame

Returns a data frame with a random fraction of the rows from the current data frame, as shown in Listing 8-27.

Listing 8-27. Taking a Random Sample of Rows from a Data Frame

```
cdrs.sample(true, 0.01)
```

except(other: DataFrame): DataFrame

Only returns rows that are present in the calling data frame but not in other. Listing 8-28 compares the RDD from the current and previous batches. Your goal is to print only the square ID, country code pairs that have occurred in the current batch interval but not in the previous one.

Listing 8-28. Subtracting One Data Frame Row from Another

```
var previousCdrs: Option[DataFrame] = None

val cdrStream = ssc.socketTextStream(hostname, port.toInt)
```

```
4.     .map(_.split("\\t", -1))
5.     .foreachRDD(rdd => {
6.       val cdrs = seqToCdr(rdd).toDF().select("squareId", "countryCode").dropDuplicates()
7.       previousCdrs match {
8.         case Some(prevCdrs) => cdrs.except(prevCdrs).show()
9.         case None => Unit
10.      }
11.      previousCdrs = Some(cdrs)
12.    })
```

intersect(other: DataFrame): DataFrame

As the name suggests, this operation performs a `set` intersection on the calling data frame and `other`. Replace `except` on line 8 in Listing 8-28 to get a hang of its usage.

unionAll(other: DataFrame): DataFrame

Performs a union between the rows of this data frame and `other`. Tweak Listing 8-28 to see an example.

join(right: DataFrame, joinExprs: Column): DataFrame

Similar to a database join, allows you to perform an inner join between this and the `right` data frame. `joinExprs` specifies the join column from either data frame. Let's join the CDR dataset with the grid dataset using the square ID as the joining column. The grid dataset is encoded in GeoJSON, which you read in the driver program (line 1) in Listing 8-29 and then flatten to extract polygon coordinates (lines 2–7). After converting it to a data frame (line 9), you join it with the per-interval CDRs data frame in line 15, using the square ID as the key.

Listing 8-29. Inner Join Between Two Data Frames

```
1.   val gridFile = scala.io.Source.fromFile(gridJsonPath).mkString
2.   val gridGeo = (parse(gridFile) \ "features")
3.   val gridStr = gridGeo.children.map(r => {
4.     val c = (r \ "geometry" \ "coordinates").extract[List[List[List[Float]]]].flatten.
       flatten.map(r => JDouble(r))
5.     val l = List(("id", r \ "id"), ("x1", c(0)), ("y1", c(1)), ("x2", c(2)), ("y2",
       c(3)), ("x3", c(4)), ("y3", c(5)), ("x4", c(6)), ("y4", c(7)))
6.     compact(render(JObject(l)))
7.   })
8.
9.   val gridDF = sqlC.read.json(ssc.sparkContext.makeRDD(gridStr))
10.
11.  val cdrStream = ssc.socketTextStream(hostname, port.toInt)
12.    .map(_.split("\\t", -1))
13.    .foreachRDD(rdd => {
14.      val cdrs = seqToCdr(rdd).toDF()
15.      cdrs.join(gridDF, $"squareId" === $"id").show()
16.    })
```

na

Returns a `DataFrameNaFunctions` object for manipulating missing data. Table 8-3 contains functions provided by this object along with examples.

Table 8-3. *`DataFrameNaFunctions` Functions for Working with Missing Data in Data Frames*

Method	Description	Example
`drop(how: String): DataFrame`	Drops rows with a null value based on the `how` criteria. A value of `all` and `any` for `how` dictates whether all or any columns should be considered. Other variants of this function also allow column selection.	`cdrs.na.drop("any")`
`fill(value: String, cols: Seq[String]): DataFrame`	Fills in nulls in `cols` with `value`. Variants exist for filling in numeric values and specific columns.	`cdrs.na.fill (0, Array("squareId"))`
`replace[T](cols: Seq[String], replacement: Map[T, T]): DataFrame`	Replaces specific values in `cols`. `replacement` contains the replacement values in which the key is the value to be replaced, and the value is the replacement.	`cdrs.na.replace ("squareId", Map(0 -> 1))`

stats

Returns a `DataFrameStatFunctions` object to perform various statistical analyses. Refer to Table 8-4 for examples and a list of methods.

Table 8-4. *`DataFrameStatFunctions` Functions for Performing Various Statistical Analyses*

Method	Description	Example
`corr(col1: String, col2: String): Double`	Finds the correlation between `col1` and `col2` using the Pearson correlation coefficient.	`cdrs.stat.corr ("smsOutActivity", "callOutActivity")`
`cov(col1: String, col2: String): Double`	Calculates the covariance between `col1` and `col2`.	`cdrs.stat.cov ("smsInActivity", "callInActivity")`
`crosstab(col1: String, col2: String): DataFrame`	Works out the cross tabulation of `col1` and `col2`. Also known as a contingency table.	`cdrs.stat.crosstab ("squareId", "countryCode")`
`freqItems(cols: Seq[String], support: Double): DataFrame`	Calculates frequent items in the sequence of `cols`, where `support` is the minimum frequency for a column to be considered.	`cdrs.stat.freqItems (Array("squareId", "countryCode"), 0.1)`

Actions

As you saw earlier, data frames are lazily evaluated: an action causes a chain of data-frame operations to be shipped off for execution. Along the same lines as RDDs and DStreams, these actions are of two types: those that ingest data into the driver program and those that send output to external storage. Table 8-5 lists the former category of actions. Test these by replacing show() on line 41 in Listing 8-1.

Table 8-5. *Internal Actions for Data Frames*

Method	Description
show(numRows: Int): Unit	Prints the first numRows in the data frame in a tabular form.
show(): Unit	Alias for show(20).
head(n: Int): Array[Row]	Returns the first n rows in the data frame as an array.
take(n: Int): Array[Row]	Alias for head(n).
head(): Row	Returns the first row in the data frame.
first(): Row	Alias for head().
count(): Long	Returns the count of rows.
collect(): Array[Row]	Converts the data frame into an array and returns it.
collectAsList(): List[Row]	Similar to collect() but returns a list instead of an array.
describe(cols: String*): DataFrame	Returns a new data frame, which contains the count, mean, standard deviation, minimum, and maximum of numerical cols. For instance, cdrs.describe("smsInActivity", "smsOutActivity", "callInActivity", "callOutActivity", "internetTrafficActivity").

Output actions for external storage are provided by DataFrameWriter, which can be obtained by invoking write on any data frame. Let's walk through its functions.

format(source: String): DataFrameWriter

Sets the output format for the writer. source can be set to json or parquet, with parquet as the default (see Listing 8-30).

Listing 8-30. Setting the Output Format for a Data Frame to JSON

```
cdrs.groupBy("countryCode").count().orderBy(desc("count")).write.format("json")
```

save(path: String): Unit

Saves the data frame at path. Listing 8-31 shows how you can use the combination of format() and save() to save a data frame in JSON format to any Hadoop-compatible file system.

Listing 8-31. Saving a Data Frame in JSON Format

```
cdrs.groupBy("countryCode").count().orderBy(desc("count")).write.format("json").save(path)
```

parquet(path: String): Unit

Convenience function for `format("parquet").save(path)`.

json(path: String): Unit

Convenience function for `format("json").save(path)`.

saveAsTable(tableName: String): Unit

Saves the data frame as a persistent table with `tableName` in the Hive metastore. Be sure to use `HiveContext` for this. Listing 8-32 saves the country counts of CDRs as a table.

Listing 8-32. Saving a Data Frame as a Hive Table

```
cdrs.groupBy("countryCode").count().orderBy(desc("count")).write.format("parquet").
saveAsTable("count_table")
```

mode(saveMode: SaveMode): DataFrameWriter

Decides the behavior if the output table (when using `saveAsTable()`) or file (when using `save()`) already exists. `saveMode` can be `Ignore`, `Append`, `Overwrite`, or `ErrorIfExists`. Listing 8-33 appends data-frame output to the same output file(s). This is handy for streaming applications where incremental output is created in each batch.

Listing 8-33. Setting the Output Mode for the Writer

```
cdrs.groupBy("countryCode").count().orderBy(desc("count")).write.mode(SaveMode.Append).
save(path)
```

partitionBy(colNames: String*): DataFrameWriter

Partitions the data frame by `colNames` before writing it.

insertInto(tableName: String): Unit

Inserts the data frame into an existing table. Note that the table should have the same schema as the data frame.

jdbc(url: String, table: String, connectionProperties: Properties): Unit

Ships out the data frame to `table` over a JDBC connection `url` using `connectionProperties`, as shown in Listing 8-34.

Listing 8-34. Writing a Data Frame to an External Database over JDBC

```
val prop: java.util.Properties = new java.util.Properties()
counts.write.jdbc("jdbc:mysql://hostname:port/cdrsdb", "count_table", prop)
```

RDD Operations

Each data-frame object also has associated RDD operations enabled by the method `rdd`, which returns an RDD of `Rows`. The first time it is called, it creates the RDD, which is memoized, so subsequent calls return this cached instance. All standard RDD operations can then be applied. Let's mix RDD operations with data-frame queries to determine the disparity between SMS usage and calls. Specifically, you need to bifurcate the dataset by whether there was more SMS activity or call activity. Then, for either dataset, you need square ID and country-code pairs that are exclusive to that data frame. The snippet of code in Listing 8-35 does just that.

Listing 8-35. Mixing RDD Operations with Data-Frame Operations

```
val highInternet = sqlC.createDataFrame(cdrs.rdd.filter(r => r.getFloat(3) +
r.getFloat(4) >= r.getFloat(5) + r.getFloat(6)), schema)
val highOther = cdrs.except(highInternet)
val highInternetGrid = highInternet.select("squareId", "countryCode").dropDuplicates()
val highOtherGrid = highOther.select("squareId", "countryCode").dropDuplicates()
highOtherGrid.except(highInternetGrid).show()
highInternetGrid.except(highOtherGrid).show()
```

Persistence

The persistence level of data frames can also be configured via calls to familiar functions: `cache()` and `persist(newLevel: StorageLevel)`. If so desired, a previously persisted data frame can be uncached via a call to `unpersist()`.

Best Practices

- *Avoid shuffling.* As you saw in Chapter 4, shuffling is one of the most expensive steps in distributed computing. Therefore, if possible, you should avoid it at all costs. The same applies to Spark SQL and data frames. When performing joins, make copious use of broadcast hash joins to minimize shuffle time.
- *Use code generation.* For complex or recurring queries, it is generally more efficient to convert them to byte code for faster execution. This can be achieved by setting `spark.sql.codegen` to `true`.
- *Cache aggressively.* Similar to RDDs and `DStreams`, if a data frame needs to be accessed multiple times, it should be cached in memory. This has a number of advantages including efficient column lookups, compression, and minimal garbage-collection pressure.

This brings you to the end of the discussion of data frames. Data frames are not the only mechanism enabled by Spark for analytics. There is another that's dearly loved by data scientists: R.

SparkR

R is the language of choice for most data scientists. It is geared toward statistical analysis and graphical visualization of data. R is an interpreted, scripted language with its own shell. The functionality of the core runtime can be extended using external packages. Although R is an extremely powerful tool for data analysis, its utility is limited by the capabilities of the machine it is running on. A number of enhancements are available to enable R to take advantage of parallel and distributed execution,[8] but none of them have the muscle to turn R into a general-purpose distributed environment for Big Data analysis. That is where SparkR comes in.

SparkR is an R package that acts as a bridge between R and Spark. You can manipulate RDDs and data frames in R while using the full capabilities of Spark. It is part of the standard Spark distribution and can easily be spun up to get the best of both worlds. Let's kill the suspense and set up SparkR.

INSTALLING SPARKR

Download R, and install it if you have not done so already.

Run R. If you are running from the command line, simply type **R**.

SparkR relies on `rJava` as the glue between R and Spark and `devtools` for package installation. Install both using `install.packages("rJava")` and `install.packages("devtools")` in the R shell.

Check that both dependencies have been installed properly by loading them: `library(rJava)` and `library(devtools)`.

Outside of the R shell, jump to the `$SPARK_HOME/R/lib/SparkR` folder and execute the following:

```
R -e "devtools::install('.')"
```

This installs SparkR as an R package.

To verify that the installation is correct, load the package: `library(SparkR)`.

This also gives you access to SparkR from within RStudio if that is your weapon of choice for data science.

First SparkR Application

To get your hands dirty with SparkR, let's reimplement the very first application from this chapter (Listing 8-1) in R, with a caveat: SparkR does not support Spark Streaming out of the box. To account for that, the application reads data directly from the CDR dataset TSVs. Don't panic, though; in the next section, you analyze a design pattern that lets you handle streaming data in SparkR. For now, let's go back to the first example, the code for which is shown in Listing 8-36.

Listing 8-36. First SparkR Application

```
args <- commandArgs(trailingOnly = TRUE)
if(length(args) != 2) {
    stop("Usage: CdrSparkRApp <master> <filepath>")
}
library(SparkR)
Sys.setenv('SPARKR_SUBMIT_ARGS'='"--packages" "com.databricks:spark-csv_2.10:1.3.0"
"sparkr-shell"')
sc <- sparkR.init(master = args[1])
sqlContext <- sparkRSQL.init(sc)
```

[8]https://cran.r-project.org/web/views/HighPerformanceComputing.html.

```
9.   df <- read.df(sqlContext, args[2], source = "com.databricks.spark.csv", inferSchema =
     "true", delimiter = "\t")
10.  cnames <- c("squareId", "timeInterval", "countryCode", "smsInActivity",
     "smsOutActivity", "callInActivity", "callOutActivity", "internetTrafficActivity")
11.  for (i in 1:NROW(cnames)) {
12.      df <- withColumnRenamed(df, paste0("C", i - 1), cnames[i])
13.  }
14.  counts <- count(groupBy(df, "countryCode"))
15.  showDF(orderBy(counts, desc(counts$count)), numRows = 5)
16.  sparkR.stop()
```

The first four lines are boilerplate code to ensure that you provide the correct command-line arguments while line 5 loads the SparkR package. The input file from the Milan CDR dataset is a tab-separated collection of lines (after you unzip the archive). To read one of the files, you use the spark-csv package from Databricks. This package is included on line 6. Note that Spark also downloads and makes this package available automatically if it is not already available locally. The next order of the day is to initialize the Spark context (line 7) and Spark SQL context (line 8). The former requires the location of the Spark master, and the latter requires SparkContext for initialization. Now you are ready to create data frames, which is what you do in line 9. The read.df function requires a reference to SQLContext as well as the path to the file. In addition, it requires the name of the input source, which in this case is a CSV. The rest of the arguments contain formatting information, which is passed down to the input source.

Once you have a data frame in hand, you manually add column names, because the input CSV does not have header information. Repeated execution of withColumnRenamed() helps you to replace the generic column names with those specific to the logic (lines 10–13). Similar to the vanilla DataFrame API (used in Listing 8-1), the actual query is implemented in only two lines (lines 14–15) in which you rank the top five country codes by frequency. Note that each data-frame operation has an equivalent in Spark. Once the application has completed, sparkR.stop() enables you to bring down the driver program.

Execution

Save the code from Listing 8-36 in a file, and name it, say, CdrSparkRApp.R. You can then execute it from the command line:

```
Rscript CdrSparkRApp.R spark_master path_to_cdr_file_txt
```

You should obtain output similar to that in Listing 8-37.

Listing 8-37. Output of Your First SparkR Application

```
+-----------+-------+
|countryCode|  count|
+-----------+-------+
|         39|1439981|
|          0|1159632|
|         33| 210334|
|         46| 208713|
|         49| 165341|
+-----------+-------+
```

Streaming SparkR

As you saw in the last application, SparkR does not support Spark Streaming out of the box, but that does not mean you cannot use various other features of Spark to implement real-time applications. The design pattern in this section contains the following two key components:

- A Spark Streaming application that preprocesses the data and writes a new Hive table in every batch
- A stateless SparkR script that is invoked by the streaming application in each batch and that processes the per-batch Hive table

Listing 8-38 lists the main logic of the Spark Streaming end of the application.

Listing 8-38. Streaming Application That Creates a Per-Batch Data Frame and Invokes a SparkR Script to Consume It

```
1.   ssc.sparkContext.addFile(rScriptPath)
2.   val rScriptName = SparkFiles.get(Paths.get(rScriptPath).getFileName.toString)
3.   val master = hiveC.sparkContext.getConf.get("spark.master")
4.
5.   val cdrStream = ssc.socketTextStream(hostname, port.toInt)
6.     .map(_.split("\\t", -1))
7.     .foreachRDD((rdd, time) => {
8.       val iTableName = tableName + time.milliseconds
9.       seqToCdr(rdd).toDF().write.saveAsTable(iTableName)
10.      hiveC.sparkContext.parallelize(Array(iTableName)).pipe("%s %s".format(rScriptName,
         master)).saveAsTextFile(Paths.get(logsPath, iTableName).toString)
11.    })
```

The application ingests the CDR data from a socket and writes it out to a Hive table per batch (line 9). You append a batch timestamp, which you obtain from the `foreachRDD` operation (line 7), to each table name to temporally segregate the data (line 8). The fun part starts on line 10. You use a `PipedRDD` to kick off the SparkR script. Specifically, you create an RDD with just the per-batch table name and pipe it to the external script via a `pipe` call. The script requires the location of the Spark master as a command-line argument, so you provide it. Finally, the output (`stdout`) of the external script is piped back to the caller, which you save to a file. You see shortly that the SparkR script writes only log information to standard output, not actual data.

The SparkR side of the application (Listing 8-39) reads the name of the data frame from standard input (lines 9–11) and loads the corresponding data frame using a `HiveContext` (line 15). Once the top five country codes by occurrence have been obtained, the resulting data frame is written as a new output table to Hive (line 19).

Listing 8-39. SparkR Script That Is Invoked by the Streaming Application Every Interval to Manipulate Data Frames

```
1.   #!/usr/local/bin/Rscript
2.   args <- commandArgs(trailingOnly = TRUE)
3.   if(length(args) != 1) {
4.       stop("Usage: CdrStreamingSparkRApp <master>")
5.   }
6.   library(SparkR)
7.   sc <- sparkR.init(master = args[1])
8.   hiveContext <- sparkRHive.init(sc)
```

```
f <- file("stdin")
open(f)
while(length(tableName <- readLines(f, n = 1)) > 0) {
    tryCatch({
        tableName <- trimws(tableName)
        write(paste0("Processing table: ", tableName), stderr())
        df <- table(hiveContext, tableName)
        counts <- count(groupBy(df, "countryCode"))
        outputTable <- paste0(tableName, "processed")
        write(paste0("Output written to: ", outputTable), stderr())
        saveAsTable(limit(orderBy(counts, desc(counts$count)), 5), outputTable,
        "parquet", "error")
    }, error = function(e) {stop(e)})
}
close(f)
sparkR.stop()
```

Note that in order for this application to work, you need to run an external metastore for Hive. Specifically, the embedded Derby metastore in Spark SQL only supports single user/application operations. As a result, the SparkR script raises an exception telling you "Another instance of Derby may have already booted the database." This can be remedied either by running a network service version of Derby or by using a MySQL or Postgres instance as the metastore database. The following guides you through setting up the former.

INSTALLING AND RUNNING APACHE DERBY FOR HIVE

Download Derby version 10.10.1.1,[9] and unzip it in a suitable location: `DERBY_HOME`.

Edit `$JAVA_HOME/jre/lib/security/java.policy`, and add the following to it:

```
grant { permission java.net.SocketPermission "localhost:1527", "listen"; };
```

Create a `data` folder at `$DERBY_HOME/data`, and jump to it.

Execute the following:

```
../bin/startNetworkServer -h 0.0.0.0
```

You have Derby running as a network service on this machine.

If not already present, create `hive-site.xml` at `$SPARK_HOME/conf/hive.site.xml`, add the following configuration parameters to it, and replace `localhost` with the IP of the machine running the Derby server:

```
<configuration>
  <property>
    <name>javax.jdo.option.ConnectionURL</name>
    <value>jdbc:derby://localhost:1527/metastore_db;create=true</value>
  </property>
  <property>
```

[9]https://db.apache.org/derby/releases/release-10.10.1.1.html.

```
    <name>javax.jdo.option.ConnectionDriverName</name>
    <value>org.apache.derby.jdbc.ClientDriver</value>
  </property>
</configuration>
```

Add the Derby client JAR to the Spark classpath. Specifically, edit `$SPARK_HOME/conf/spark-env.sh`, and add

```
SPARK_CLASSPATH=$DERBY_HOME/lib/derbyclient.jar
```

It is important to highlight that this example is not for production use, because it creates a large number of tables in the Hive metastore. There are a number of ways to make this design pattern more robust and efficient. One method consists of writing the output of the Spark Streaming application to temporary files and then performing a bulk load into Hive for consumption by R. This obviously comes at the cost of higher latency. Alternatively, instead of creating a per-batch table in the Hive metastore, you can create a single table and keep appending to it in every batch interval with an additional timestamp column. At the R end, only records with a specific range of timestamps are processed each time the script is invoked.

Using this simple but effective design paradigm, you can analyze streaming data using R. The exciting prospect is the ability to use standard R packages and constructs for data science. You can use the entire suite of machine learning, data mining, and graphing tools enabled by R to suit your needs. The combination of Spark and R is just what the doctor ordered for large-scale Big Data processing and data science.

Summary

Spark SQL simplifies the analysis of structured data using Spark. In this chapter, you saw how to use it to perform ETL for streaming data via data frames: RDDs for structured data. The support for HiveQL allows you to use the trusted suite of Hive queries and UDFs and run your existing queries almost untouched. The cherry on top is SparkR, which completes the trio of choice for data science: SQL, Hive, and R. You looked at multiple design patterns to extend the applicability of these tools to real-time data. But you only scratched the surface in terms of hardcore data science, which revolves around machine learning. This chapter mentioned that data frames are also used by MLlib: the equivalent of Mahout for Spark, to which Chapter 9 is dedicated.

CHAPTER 9

Machine Learning at Scale

There is an art to flying. The knack lies in learning how to throw yourself at the ground and miss.

—Douglas Adams, *Life, the Universe and Everything*

Data by itself is a static, lifeless entity. You need analytics to breathe life into it and make it talk or even sing. The most sophisticated and popular class of such analytics revolves around nowcasting, forecasting, and recommendations, more generally known as *machine learning* and *data mining*. Machine-learning algorithms learn patterns in data and can then be used to make predictions, whereas data mining helps extract structure from unstructured data. Using the power of both, electricity providers can predict the network load and control power generation accordingly, a clothing line can figure out the standard t-shirt sizes for a new market, oil companies can choose the location of their next drilling operation, and health practitioners can diagnose diseases without a physical checkup. This is easier said than done, though, because of the sheer size of the data: in some cases, the dataset can exceed petabytes. Consequently, machine learning at scale is the key to practical predictions and recommendations, which are essential to drive the needs of consumers: commercial, academic, or scientific.

The scale and complexity of some of these analytics is magnified in real-time scenarios where an answer is needed as soon as possible, exposing the inherent trade-off between accuracy and training time in statistical models. Fortunately, MLlib—the suite of machine-learning algorithms for Spark—supports streaming analytics out of the box. In this chapter, you use some of these analytics to model IoT data. You start off with statistical analysis to learn the distribution of data and get a feel for its attributes, followed by feature-selection algorithms. The bulk of the chapter is dedicated to various learning algorithms covering regression, classification, clustering, recommendation systems, and frequent pattern matching. The chapter wraps up with the Spark ML package, which simplifies the implementation of end-to-end learning pipelines.

Sensor Data Storm

By 2020, 50 billion sensors will be connected to the Internet.[1] This Internet of Things (IoT) revolution has already started. Today, GE receives 50 million readings from 10 million sensors deployed atop devices and machinery worth more than $1 trillion.[2] These sensors are used to instrument devices as diverse as toothbrushes and bulldozers. In these devices, sensors monitor a rich constellation of phenomena: the environment, movement, weather, you name it. A single device can have hundreds of sensors, which generate

[1]Plamen Nedeltchev, "The Internet of Everything Is the New Economy," *Cisco*, September 29, 2015, www.cisco.com/c/en/us/solutions/collateral/enterprise/cisco-on-cisco/Cisco_IT_Trends_IoE_Is_the_New_Economy.html.
[2]Heather Clancy, "How GE Generates $1 Billion from Data," *Fortune*, October 10, 2014, http://fortune.com/2014/10/10/ge-data-robotics-sensors/.

Z. Nabi, *Pro Spark Streaming*, DOI 10.1007/978-1-4842-1479-4_9

hundreds of thousands of data points per time interval. Typically, the applications that consume this data have a strict Service Level Agreement (SLA) and need to produce results with a specific latency. This is more pronounced in online learning scenarios, which have both a time budget and an accuracy requirement for predictive models.

One major class of sensor-based analytics uses data from the rich set of sensors in smartphones to improve the wellness of users: for instance, predicting how much a user will walk over the course of a year and recommending an appropriate diet for them. Similarly, another application can cluster users based on their heart rate and other vitals and notify them when their health profile changes. Sensor-based healthcare analytics have become prevalent due to the popularity of devices such as Fitbit and Apple Watch. For obvious privacy reasons, the data from these devices is locked away behind multiple layers of safeguards. Fortunately, a number of such datasets are available in the public domain, albeit from limited, controlled environments.

One such dataset, which you use in this chapter, logs the physical activity of nine individuals via inertial and heart-rate monitoring sensors.[3] Each individual wore three inertial measurement units (IMUs), located on different parts of the body: hand, chest, and ankle. Each IMU contains four 3D sensors: two accelerometers, a gyroscope, and a magnetometer.

In total, ten hours of activity is recorded in the data. The physical activities are categorized using 18 labels, which include walking, running, cycling, lying down, sitting, standing, and so on. The dataset is divided into nine text files (with `.dat` extension), one per subject. Each space-separated record consists of 52 attributes, 1 timestamp, and 1 label, which are summarized in Table 9-1.

Table 9-1. *Summary of the Fields in the Sensor Activity Dataset*

Field #	Name	Description
1	Timestamp	The timestamp of the record in seconds
2	Activity ID	The tag assigned to the activity: 1. Lying, 2. Sitting, 3. Standing, 4. Walking, 5. Running, 6. Cycling, 7. Nordic walking, 9. Watching TV, 10. Using a Computer, 11. Driving, 12. Ascending stairs, 13. Descending stairs, 16. Vacuum cleaning, 17. Ironing, 18. Folding laundry, 19. House cleaning, 20. Playing soccer, 21. Rope jumping, and 0. Invalid
3	Heart rate	The heart rate in beats per minute
4–20	IMU Hand	The inertial measurement unit attached to the hand (detailed breakdown in Table 9-2)
21–37	IMU Chest	Self-explanatory
38-54	IMU Ankle	Self-explanatory

[3]PAMAP2 Physical Activity Monitoring Data Set, *UC Irvine Machine Learning Repository*, `https://archive.ics.uci.edu/ml/datasets/PAMAP2+Physical+Activity+Monitoring`.

The breakdown of the attributes of each IMU is presented in Table 9-2.

Table 9-2. *Summary of the IMU Attributes*

Field #	Name	Description
1	Temperature	The temperature in Celsius
2–4	3D Accelerometer	Speed in m/s with a range of 16 g (g-force)
5–7	3D Accelerometer	Speed in m/s with a range of 6 g[4]
8–10	3D Gyroscope	Orientation of the device in radians per second
11–13	3D Magnetometer	Magnetic field sensor with micro Tesla units
14–17	Orientation	Skipped in this dataset

This dataset is a good representative of a plurality of sensors that can be used to gauge human activity. Let's get going.

Streaming MLlib Application

Listing 9-1 contains the code for your very first machine-learning application. The goal is to use linear regression to model the target sensor data by using heart rate, temperature, and acceleration.

Listing 9-1. First Streaming MLlib Application

```
package org.apress.prospark

import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.regression.StreamingLinearRegressionWithSGD
import org.apache.spark.rdd.RDD
import org.apache.spark.rdd.RDD.doubleRDDToDoubleRDDFunctions
import org.apache.spark.streaming.Seconds
import org.apache.spark.streaming.StreamingContext

object LinearRegressionApp {

  def main(args: Array[String]) {
    if (args.length != 4) {
      System.err.println(
        "Usage: LinearRegressionApp <appname> <batchInterval> <hostname> <port>")
      System.exit(1)
    }
    val Seq(appName, batchInterval, hostname, port) = args.toSeq

```

[4]This sensor was not calibrated properly while taking measurements, so the use of the 16 g attribute is recommended for any analytics.

```
    val conf = new SparkConf()
      .setAppName(appName)
      .setJars(SparkContext.jarOfClass(this.getClass).toSeq)

    val ssc = new StreamingContext(conf, Seconds(batchInterval.toInt))

    val substream = ssc.socketTextStream(hostname, port.toInt)
      .filter(!_.contains("NaN"))
      .map(_.split(" "))
      .filter(f => f(1) != "0")

    val datastream = substream.map(f => Array(f(2).toDouble, f(3).toDouble, f(4).
    toDouble, f(5).toDouble, f(6).toDouble))
      .map(f => LabeledPoint(f(0), Vectors.dense(f.slice(1, 5))))
    val test = datastream.transform(rdd => rdd.randomSplit(Array(0.3, 0.7))(0))
    val train = datastream.transformWith(test, (r1: RDD[LabeledPoint], r2:
    RDD[LabeledPoint]) => r1.subtract(r2)).cache()
    val model = new StreamingLinearRegressionWithSGD()
      .setInitialWeights(Vectors.zeros(4))
      .setStepSize(0.0001)
      .setNumIterations(1)

    model.trainOn(train)
    model.predictOnValues(test.map(v => (v.label, v.features))).foreachRDD(rdd =>
    println("MSE: %f".format(rdd
      .map(v => math.pow((v._1 - v._2), 2)).mean())))

    ssc.start()
    ssc.awaitTermination()
  }

}
```

Once the model has been trained using temperature and accelerometer attributes, you can use it to predict the heart rate for each record. In each batch, the application uses 70% of the dataset for training and 30% for testing. The first order of business is to push data to Spark, for which you again repurpose your trusted SocketDriver from Chapter 5 by augmenting AbstractDriver with Listing 9-2 to read plain-text files.

Listing 9-2. Augmenting AbstractDriver from Chapter 5 to Support .dat files

```
else if (ext.equals("dat")) {
    LOG.info(String.format("Feeding dat file %s", f.getName()));
    try (BufferedReader br = new BufferedReader(new InputStreamReader(new
    FileInputStream(f)))) {
        String line;
        while ((line = br.readLine()) != null) {
            sendRecord(line);
        }
    }
}
```

Feed the socket before running the Spark Streaming from Listing 9-1. You read records from the socket (line 29) and filter out records that have invalid numbers (`NaN`) or where the activity itself is invalid (0) on lines 30–32. You need to predict the heart rate based on temperature and acceleration, so you project fields that might be handy for this purpose: the heart rate (index 2), temperature (index 3), and 16 g accelerometer (index 4–6). MLlib algorithms require data in the form of a `Vector` type with each feature as a `Double`. For training, each record needs to be converted to a `LabeledPoint` whose first parameter is a label for the record and second is a `Vector` type (line 35). In this case, the label is the heart rate and the rest are features.

Once each record has been converted to `LabeledPoint`, the input stream needs to be divided into training and test. For this you use the `randomSplit` method for RDDs with a `transform` operation to divide the stream and get the `test` stream (line 36). The code then uses another `transform` operation over the `test` stream and the original stream to get the difference of the two, which carves out the `train` stream (line 37). You cache this dataset because it will be used across iterations. The data is now ready for training.

The dataset has continuous values, so you use a streaming regression model: `StreamingLinearRegressionWithSGD`. The model requires a few configuration parameters including the initial weight of the features, the step size (learning rate for stochastic gradient descent [SGD]), and the number of iterations. You assign initial values of zero for the features (line 39). The step size and the number of iterations along with other parameters are very sensitive to the data being modeled. The values were selected via trial and error. For other datasets, they might be very different. Refer to Table 9-3 for more parameters and their default values.

After the model has been calibrated with various parameters, it is ready to receive training data (line 43). As mentioned earlier, the application uses 70% of the input stream for training and then validates the learning by predicting the activity label of the other 30%. `predictOnValues` uses the model to predict the label of the new data. In this case, because you already know the activity types of the 30% streams as well, you can use it to come up with an error rate for the prediction via mean square error (MSE); this basically gives you the mismatch between the actual value and the predicted value. `predictOnValues` performs the prediction but keeps the actual activity label (the key) intact. You use this functionality to calculate a per-batch-interval MSE (lines 44–45). When you run the application, notice that this error goes down over time, as the model starts accurately predicting the regression line. This is due to the online, accumulative nature of this algorithm, which is updated in each microbatch based on `miniBatchFraction`.

Build and run it like any standard Spark Streaming application.[5]

Table 9-3. *Parameters to Calibrate Streaming Linear Regression*

Parameter	Set	Default	Description
`initialWeights`	`setInitialWeights (initialWeights: Vector)`	na	The initial weight for each feature vector. This is typically initialized to 0 unless you have performed an advance analysis to precalculate a value.
`miniBatchFraction`	`setMiniBatchFraction (miniBatchFraction: Double)`	1.0	The fraction of the data in each batch to use to update the model.
`numIterations`	`setNumIterations (numIterations: Int)`	50	The number of iterations per update.
`stepSize`	`setStepSize(stepSize: Double)`	0.1	The gradient descent step size.

[5]Make sure you add `libraryDependencies += "org.apache.spark" %% "spark-mllib" % "1.4.0"` to your build specification file.

MLlib is self-contained in the `org.apache.spark.mllib` package. Let's explore its other features in depth.

MLlib

MLlib covers the entire spectrum of machine-learning algorithms and tools to enable predictions, recommendations, and feature extraction at scale. The list of supported algorithms includes classification, clustering, collaborative filtering, dimensionality reduction, frequent pattern mining, and regression. Another suite of machine-learning tools was introduced in Spark 1.2 in the `org.apache.spark.ml` package to facilitate the creation of machine-learning pipelines. The next section introduces it, so for now let's focus on the core MLlib library. The functionality of MLlib can be grouped into three main categories: statistical analysis and preprocessing, feature selection and extraction, and learning algorithms. Before you look at each in turn, let's begin with the data types inherent to MLlib.

Data Types

Similar to Mahout and other machine-learning libraries, MLlib has two major data types: vectors and matrices. As the names suggest, a vector is a one-dimensional representation of `Doubles`, and a matrix covers two dimensions. These data types are either stored in a single executor process or distributed across many executors in a cluster. This leads to a combination of types, as listed in Table 9-4. For each example, assume that the input stream is the same as that on line 29 in Listing 9-1.

Table 9-4. *Examples of the Data Types Provided by MLlib*

Type	Description	Example
Local dense vector	Stored locally as an array of doubles. Use this vector when your data does not have many zero values.	**Signature:** `Vectors.dense(values: Array[Double]): Vector` `val denseV = substream.map(f => Vectors.dense(f.slice(1, 5)))`
Local sparse vector	Stored locally as two parallel arrays: one for indexes and the other for actual values. Zero values are skipped in this representation. Use this type if your data contains many zero values.	**Signature:** `Vectors.sparse(size: Int, elements: Seq[(Int, Double)]): Vector` `val sparseV = substream.map(f => f.slice(1, 5).toList).map(f => f.zipWithIndex.map { case (s, i) => (i, s) }) .map(f => f.filter(v => v._2 != 0)).map(l => Vectors.sparse(l.size, l))`
Labeled point	Attaches a label to a vector (dense or sparse). This is used for learning algorithms in which the label trains the model for the provided attributes.	**Signature:** `new LabeledPoint(label: Double, features: Vector)` `val labeledP = substream.map(f => LabeledPoint(f(0), Vectors.dense(f.slice(1, 5))))`
Local matrix	A dense matrix with integer row and column indexes and double values.	**Signature:** `Matrices.dense(numRows: Int, numCols: Int, values: Array[Double]): Matrix` `val denseM = substream.map(f => Matrices.dense(3, 16, f.slice(3, 19) ++ f.slice(20, 36) ++ f.slice(37, 53)))`

(*continued*)

Table 9-4. (*continued*)

Type	Description	Example

Distributed `RowMatrix`	A matrix that is distributed across the cluster. Each row is a local vector.	**Signature:** `new RowMatrix(rows: RDD[Vector])`
		
		`denseV.foreachRDD(rdd => {`
		`     val rowM = new RowMatrix(rdd)`
		`})`

Distributed `IndexedRowMatrix`	Similar to a distributed `RowMatrix`, with long type column indexes. Each row is an `IndexedRow` type, which is a two tuple of `(Long, Vector)`.	**Signature:** `new IndexedRowMatrix(rows: RDD[IndexedRow])`
		
		`denseV.foreachRDD(rdd => {`
		`     val iRdd = rdd.zipWithIndex.map(v`
		`     => new IndexedRow(v._2, v._1))`
		`     val iRowM = new`
		`     IndexedRowMatrix(iRdd)`
		`})`

Distributed `CoordinateMatrix`	Similar to a distributed `IndexedRowMatrix` with column indexes. Under the hood, it is an RDD of `MatrixEntry`, where `MatrixEntry` is a 3-tuple of `(Long, Long, Vector)`. This is useful to store high-dimensionality, sparse data.	**Signature:** `new CoordinateMatrix(entries: RDD[MatrixEntry])`
		
		`substream.foreachRDD(rdd => {`
		`     val entries = rdd.zipWithIndex.`
		`     flatMap(v => List(3, 20, 37).`
		`     zipWithIndex.map(i => (i._2.toLong,`
		`     v._2, v._1.slice(i._1, i._1 + 16).`
		`     toList))) .map(v => v._3.map(d =>`
		`     new MatrixEntry(v._1, v._2, d))).`
		`     flatMap(x => x) val cRowM = new`
		`     CoordinateMatrix(entries)`
		`})`

Distributed `BlockMatrix`	Arranges the matrix as an RDD of type `MatrixBlock`. Each MatrixBlock is a sub-matrix with row and column indexes of the sub-matrix and the contents of the sub-matrix itself. It can be created by directly converting from an `IndexedRowMatrix` or a `CoordinateMatrix`.	**Signature:** `new BlockMatrix(blocks: RDD[((Int, Int), Matrix)], rowsPerBlock: Int, colsPerBlock: Int)`
		
		Or
		`CoordinateMatrix#toBlockMatrix(): BlockMatrix`
		`substream.foreachRDD(rdd => {`
		`     val entries = rdd.zipWithIndex.`
		`     flatMap(v => List(3, 20, 37).`
		`     zipWithIndex.map(i => (i._2.toLong,`
		`     v._2, v._1.slice(i._1, i._1 + 16).`
		`     toList))) .map(v => v._3.map(d =>`
		`     new MatrixEntry(v._1, v._2, d))).`
		`     flatMap(x => x)`
		`     val blockM = new`
		`     CoordinateMatrix(entries).`
		`     toBlockMatrix`
		`})`

Statistical Analysis

MLlib enables a number of statistics over vectors and matrices. These are useful for understanding the distribution and layout of the input dataset: for instance, finding correlations between different attributes can assist in feature selection for learning. Helper functions are provided in the `org.apache.spark.mllib.stat.Statistics` package. One of the most basic operations is to study the distribution of the attributes of a dataset in terms of their mean, maximum, minimum, and so on. This is typically the first step for any data scientist, because it enables you to get a feel for the data. Listing 9-3 shows how to perform these calculations for the sensor-readings dataset. Once again, assume that `substream` comes from line 29 in Listing 9-1. `colStats` takes an RDD of vectors as input.

Listing 9-3. Calculating Basic Statistics for a `DStream` of Vectors

```
substream.map(f => Vectors.dense(f.slice(1, 5))).foreachRDD(rdd => {
  val stats = Statistics.colStats(rdd)
  println("Count: " + stats.count)
  println("Max: " + stats.max.toArray.mkString(" "))
  println("Min: " + stats.min.toArray.mkString(" "))
  println("Mean: " + stats.mean.toArray.mkString(" "))
  println("L1-Norm: " + stats.normL1.toArray.mkString(" "))
  println("L2-Norm: " + stats.normL2.toArray.mkString(" "))
  println("Number of non-zeros: " + stats.numNonzeros.toArray.mkString(" "))
  println("Varience: " + stats.variance.toArray.mkString(" "))
})
```

Another useful operation is finding the correlation between different attributes. This is handy for feature selection: you can leave out attributes that are not correlated with the outcome. Correlations in MLlib can be calculated using either Pearson or Spearman correlation coefficients. The use of both is illustrated in Listing 9-4. Note that the output of the correlation is in the form of a matrix, as shown in Table 9-5. The first column contains the heart rate, and the other three contain acceleration across the three axes. The same goes for rows. Self-correlation obviously results in a positive correlation of 1. From this output, for instance, you can see that heart rate and acceleration across the z-axis are negatively correlated.

Listing 9-4. Finding the Correlation Between Different Attributes in a Vector

```
substream.map(f => Vectors.dense(f.slice(1, 5))).foreachRDD(rdd => {
  val corrSpearman = Statistics.corr(rdd, "spearman")
  val corrPearson = Statistics.corr(rdd, "pearson")
  println("Correlation Spearman: \n" + corrSpearman)
  println("Correlation Pearson: \n" + corrPearson)
})
```

Table 9-5. *Spearman and Pearson Correlation Matrices from One Microbatch*

Correlation Spearman			
1.0	0.17973751928417933	0.1636219656601942	-0.47253208575624295
0.17973751928417933	1.0	0.012457932551072698	-0.3863339273787353
0.1636219656601942	0.012457932551072698	1.0	0.0780129101161669
-0.47253208575624295	-0.3863339273787353	0.0780129101161669	1.0
Correlation Pearson			
1.0	0.152651154439714	0.27941329240545676	-0.4661000937754152
0.152651154439714	1.0	-0.2771174159022592	-0.42022123940447803
0.27941329240545676	-0.2771174159022592	1.0	-0.03222384704608242
-0.4661000937754152	-0.42022123940447803	-0.03222384704608242	1.0

Correlations highlight the relationships among variables, which are useful for determining how a trend in one variable affects another. These can be compounded with hypothesis testing, which tells you whether the results are statistically significant or happened by pure chance. For instance, can the sensor data actually help you predict whether a subject is running or walking, or are the outcomes simply random? The chi-square test is widely used for two main functions: goodness of fit (does the model reflect the data?), and independence (are two variables related?). These are enabled by the `Statistics.chiSqTest()` method. The type of the input decides whether you are testing for independence (vector or RDD of `LabeledPoint`) or goodness of fit (`Matrix`). Listing 9-5 uses it to determine the column-wise independence values.

Listing 9-5. Performing the Chi-Square Test for Independence

```
substream.map(f => Array(f(1).toDouble, f(2).toDouble, f(4).toDouble, f(5).toDouble,
f(6).toDouble))
  .filter(f => f(0) == 4.0 || f(0) == 5.0)
  .map(f => LabeledPoint(f(0), Vectors.dense(f.slice(1, 5))))
  .foreachRDD(rdd => {
    Statistics.chiSqTest(rdd).zipWithIndex.foreach(v => println("%s, column no. %d".
    format(v._1, v._2)))
  })
```

Preprocessing

In a large number of cases, data needs to be preprocessed before it can be used for learning. A common task is to normalize and scale the data to negate the effect of features with high variance. This also enables some learning algorithms to perform better. One of the most prevalent preprocessing algorithms is `StandardScaler`, which scales the variance and/or mean of numerical data. The snippet of code in Listing 9-6 uses `StandardScaler` to scale the sensor dataset readings to unit standard deviation. `StandardScaler` by default scales by standard deviation. Scaling by mean can be enabled by passing `withMean = true` to the constructor. `StandardScaler` has two phases: fitting the data (line 5) and transforming the data based on the fit (line 6). The scaled data can then be fed to any machine-learning algorithm for modeling.

Listing 9-6. Using `StandardScaler` to Scale Data to Unit Standard Deviation

```
substream.map(f => Array(f(2), f(4), f(5), f(6)))
  .map(f => f.map(v => v.toDouble))
  .map(f => Vectors.dense(f))
  .foreachRDD(rdd => {
    val scalerModel = new StandardScaler().fit(rdd)
    val scaledRDD = scalerModel.transform(rdd)
  })
```

Feature Selection and Extraction

Datasets in general have a large number of features or attributes. Not all of them may be relevant or even useful for learning. For instance, a timestamp field may not be relevant for figuring out whether a specific item is defective in manufacturing data, or two attributes may be very similar and thus one can be dropped. Having extra fields can affect both the quality of the prediction and the performance of the learning algorithm. This is exacerbated in real-time processing of Big Data because it can adversely affect client SLAs. Let's look first at a technique for feature selection (chi-square selection) followed by extraction (principal component analysis).

Chi-Square Selection

Feature selection is the process of reducing the number of features in a dataset. One such methodology is feature ranking, in which all the features are ranked via an algorithm and only the top k features are kept intact. The chi-square test can also be used for feature selection because it can find the degree of dependence between features and the target variable. Features with a low degree can be culled. Note that chi-square selection only applies to categorical data. Therefore, all continuous data must be discretized via binning or some other mechanism.

In the case of streaming data, the effectiveness of features can dynamically change over time: a feature that is essential right now may be irrelevant 30 minutes later. Therefore, feature selection needs to be a constantly evolving process. Fortunately, the microbatch processing model in Spark Streaming simplifies this process by enabling you to select features in every batch. Listing 9-7 shows how you can use chi-square selection for streaming data. This example is confined to the three 16 g accelerometers attached to the hand, chest, and ankle of the subjects and uses the activity as the predicted feature (line 1). `ChiSqSelector` fits the model on `LabeledPoint` objects, so you convert the data stream to that format (lines 2–3). To discretize the accelerometer data, you split it into 2,048 buckets. For the 16 g accelerometer limit, the sensitivity (or range) is 2,048. You use that information to perform the binning in line 3.

`ChiSqSelector` takes as input the number of required features, which you set to a modest value of 5 (line 6). It exposes a `fit` method, which takes as input an RDD of `LabeledPoints`. You use a `foreachRDD` operation to fit an RDD for selection in every batch. Once the model has been trained (line 7), it can be used to filter values from the input RDD (line 8). The filtered data can readily be used to perform learning using any model.

Listing 9-7. Using Chi-Square for Feature Selection

```
1.  val datastream = substream.map(f => Array(f(1), f(4), f(5), f(6), f(20), f(21), f(22),
    f(36), f(37), f(38)))
2.    .map(f => f.map(v => v.toDouble))
3.    .map(f => LabeledPoint(f(0), Vectors.dense(f.slice(1, f.length).map(f => f / 2048))))
4.
```

```
5.   datastream.foreachRDD(rdd => {
6.     val selector = new ChiSqSelector(5)
7.     val model = selector.fit(rdd)
8.     val filtered = rdd.map(p => LabeledPoint(p.label, model.transform(p.features)))
9.   })
```

Principal Component Analysis

Feature extraction is another dimensionality reduction mechanism in which data is transformed from one coordinate system or space to another. One of the most popular algorithms for feature extraction is principal component analysis (PCA). As the name suggests, PCA aims to bring out the features that have the most variance by transforming them into a new space: that is, the features with the most variance are the principal components. In contrast to feature selection, à la chi-square, feature extraction not only reduces the number of features but also enhances the effect of the reduced feature set.

As Listing 9-8 shows, similar to any learning algorithm, for PCA you first need to convert the data to `LabeledPoints` (lines 1–3). PCA takes the number of expected principal components as input. You set this to half of the total number of features from the accelerometer readings (line 5). You then train PCA to fit the features from the input data stream (line 6). As before, the data is split into 70% training and 30% testing. This PCA model can now be used to transform the split data (lines 9–10). This data, which requires less space, can now readily be used as input to a learning algorithm for better accuracy and performance.

Listing 9-8. Transforming Features into Their Principal Components Using PCA

```
1.   val datastream = substream.map(f => Array(f(1), f(4), f(5), f(6), f(20), f(21), f(22),
     f(36), f(37), f(38)))
2.     .map(f => f.map(v => v.toDouble))
3.     .map(f => LabeledPoint(f(0), Vectors.dense(f.slice(1, f.length))))
4.
5.   datastream.foreachRDD(rdd => {
6.     val pca = new PCA(rdd.first().features.size / 2)
7.       .fit(rdd.map(_.features))
8.     val testTrain = rdd.randomSplit(Array(0.3, 0.7))
9.     val test = testTrain(0).map(lp => lp.copy(features = pca.transform(lp.features)))
10.    val train = testTrain(1).map(lp => lp.copy(features = pca.transform(lp.features)))
11.  })
```

Learning Algorithms

The machine-learning universe contains a plethora of learning algorithms, each with a varying set of accuracy, performance, and amenability to parallelism and online learning. Their applicability also depends on the type of input data (discrete or continuous) and whether the use case is supervised or unsupervised. In supervised learning, you know upfront what you are looking for as you train the model based on past observations. For example, assume that your input data stream consists of pictures of sports balls. Under supervised learning, you provide the learning algorithm a set of prelabeled balls—say, marked as golf, cricket, soccer, football, and baseball. The algorithm then scores new incoming data based on various attributes, such as size, shape, and color. In contrast, under unsupervised learning, you have no advance information about any classification. Based on each attribute in the dataset, the unsupervised learning algorithm groups the data. It can figure out that a golf ball is radically different than a football but can't assign it a human-readable tag.

Another set of methods is applicable to recommendation systems, which, as the name suggests, try to match users with items, such as products, or people, such as potential friends: for instance, recommending which sports balls a user should buy. The algorithms used for such systems can broadly be put into two categories: collaborative filtering and content-based filtering. Collaborative filtering uses the past behavior of a user or a set of users to make the recommendation: for example, the ball-purchasing history of similar users. Content-based filtering, on the other hand, uses the user's profile and preferences to drive the recommender. An example of content-based filtering would be recommendations fueled by the type of sports columns, Facebook pages, and blogs the user follows.

Finally, frequent pattern mining represents an important strand of data mining that is essential for association rule mining. These are itemsets, sequences, and substructures that appear to have an interesting relationship: for instance, in a supermarket, items that are frequently purchased together in the same transaction, such as penne pasta and pesto sauce. Frequent sequences, on the other hand, occur over more than one transaction, For example, buying a smartphone, then a tablet, followed by a smart watch, constitutes a sequence. Such analysis is imperative for inventory management and shopping-cart analysis.

Note that MLlib currently has limited support for streaming/online-learning algorithms. Support exists only for streaming linear regression, logistic regression, and k-means clustering. This does not necessarily mean other algorithms cannot be used, though. In fact, any model from MLlib can be used as long as the application only requires per-batch modeling. This is achieved by invoking a stateless training/scoring cycle in each batch in a `foreachRDD` construct. You use this design for recommendation systems and frequent pattern mining in this chapter, because these methods do not have streaming variants in MLlib.

Supervised learning is divided into two classes based on whether the data is continuous (regression) or discrete (classification). You have already seen an example of regression, so let's look at classification next.

Classification

Linear regression is applicable to scenarios where the value to be predicted is continuous, such as heart rate in the chapter's first example. In cases where the predicted variable has a limited set of values, such as on and off or low, medium, and high, you need to use classification algorithms. One such algorithm is *logistic regression*, which is the classification equivalent of linear regression. Let's use it to predict the activity type—walking or running—of the sensor data. The snippet of code in Listing 9-9 uses streaming logistic regression to enable this prediction application.

Listing 9-9. Logistic Regression Application to Predict Whether a User Is Walking or Running

```
val datastream = substream.map(f => Array(f(1).toDouble, f(2).toDouble, f(4).toDouble,
f(5).toDouble, f(6).toDouble))

val walkingOrRunning = datastream.filter(f => f(0) == 4.0 || f(0) == 5.0).map(f =>
LabeledPoint(f(0), Vectors.dense(f.slice(1, 5))))
val test = walkingOrRunning.transform(rdd => rdd.randomSplit(Array(0.3, 0.7))(0))
val train = walkingOrRunning.transformWith(test, (r1: RDD[LabeledPoint], r2:
RDD[LabeledPoint]) => r1.subtract(r2)).cache()
val model = new StreamingLogisticRegressionWithSGD()
  .setInitialWeights(Vectors.zeros(4))
  .setStepSize(0.0001)
  .setNumIterations(1)

model.trainOn(train)
model.predictOnValues(test.map(v => (v.label, v.features))).foreachRDD(rdd =>
println("MSE: %f".format(rdd
  .map(v => math.pow((v._1 - v._2), 2)).mean())))
```

As a first step, the code projects relevant fields: activity ID, heart rate, and acceleration (line 1). You are interested only in walking and running activity, so you only keep records that represent those two types and convert them to `LabeledPoints` (line 3). The rest of the code is identical to Listing 9-1, with the only difference being the use of logistic regression instead of its linear counterpart. As with the previous example, the MSE decreases over time as the model improves its accuracy.

Clustering

Clustering is a prime example of unsupervised learning in which data points are placed into clusters based on certain attributes. For instance, baseballs and cricket balls might end up in the same cluster because of their similar size. One of the most popular algorithms for clustering is *k-means*. In k-means clustering, each data point is assigned to a cluster based on some distance metric, and the center of the cluster is recomputed. This process is iteratively computed until the points converge or a specific number of steps have been completed. `K` is the number of clusters, which you need to specify up front.

The streaming version of k-means in MLlib works in a similar fashion with one caveat: the notion of forgetfulness.[6] This parameter accounts for the dynamic nature of streaming data, in which newer data may have radically different properties than data, say, 10 minutes ago. This may lead to an abrupt recalibration of the cluster centers. Forgetfulness controls the weight that the model assigns to newer data as opposed to older data points. You can use this setting to treat all data points since the beginning of time equally (value of 0) or to consider only the most recent data (value of 1). In addition, the algorithm exposes another parameter to set forgetfulness, called the *half-life*. The half-life value determines how many batches or data points it would take for the impact of the past set of data points to drop to half its vaue. This value is set via `setHalfLife(halfLife: Double, timeUnit: String)`, where `timeUnit` can either be `batches` or `points`.

The physical activity monitoring dataset includes accelerometer and gyroscope readings from three sensors (attached to the hand, chest, and ankle of the subject). Each accelerometer and gyroscope notes acceleration and orientation along the three axes: x, y, and z. Let's use these features to determine whether a record represents a user in the lying down, sitting, or standing position. At the outset you do not know which readings represent which outcome, so you rely on k-means clustering to group data points into clusters. Note that k-means clustering still cannot tell you which cluster represents, say, lying down, only that all data points in a cluster represent similar activity. Without any further delay, let's get clustering (Listing 9-10).

Listing 9-10. K-means to Gauge Whether a User Is Lying Down, Sitting, or Standing Based on Data from the Three Orientation Sensors

```
val orientationStream = substream
  .map(f => Seq(1, 4, 5, 6, 10, 11, 12, 20, 21, 22, 26, 27, 28, 36, 37, 38, 42, 43,
  44).map(i => f(i)).toArray)
  .map(arr => arr.map(_.toDouble))
  .filter(f => f(0) == 1.0 || f(0) == 2.0 || f(0) == 3.0)
  .map(f => LabeledPoint(f(0), Vectors.dense(f.slice(1, f.length))))
val test = orientationStream.transform(rdd => rdd.randomSplit(Array(0.3, 0.7))(0))
val train = orientationStream.transformWith(test, (r1: RDD[LabeledPoint], r2:
RDD[LabeledPoint]) => r1.subtract(r2)).cache()
val model = new StreamingKMeans()
  .setK(3)
```

[6]Jeremy Freeman, "Introducing Streaming k-means in Spark 1.2," *Databricks*, January 28, 2015, https://databricks.com/blog/2015/01/28/introducing-streaming-k-means-in-spark-1-2.html.

```
10.    .setDecayFactor(0)
11.    .setRandomCenters(18, 0.0)
12.  model.trainOn(train.map(v => v.features))
13.  val prediction = model.predictOnValues(test.map(v => (v.label, v.features)))
```

The application first projects accelerometer and gyroscope readings and converts them into `double` values (lines 1–3). You are only interested in three activities—lying down, standing up, and sitting down—so the code then filters out records representing those activities (line 4). After converting the data points to `LabeledPoints` and splitting the stream for testing and training, you initialize a `StreamingKMeans` model, which requires a few parameters for tuning. `K` is set to the number of expected clusters: 3, in this case, for each activity. The data does not change drastically over time, so you set the forgetfulness or decay factor to 0, which forces the model to consider all points from the beginning of time equally (line 10). You also need to initialize the cluster center for each feature (18 total), which is what you do on line 11. (Table 9-6 discusses all the parameters.) After the application is executed, `prediction` contains the cluster ID as the label and the attributes as features (line 14).

Table 9-6. *Parameters for Streaming K-means* `clustering`

Parameter	Set	Default	Description
`decayFactor`	`setDecayFactor(a: Double)`	1.0	The weight to assign to newer data points in comparison to older ones: the forgetfulness value
`halfLife`	`setHalfLife(halfLife: Double, timeUnit: String)`	na	The decay factor as a half-life value: the number of batches or data points it would take for the impact of the past set of data points to drop to half its value
`centers`	`setInitialCenters(centers: Array[Vector], weights: Array[Double])`	na	The initial cluster center for each feature vector
`centers`	`setRandomCenters(dim: Int, weight: Double, seed: Long = Utils.random.nextLong)`	na	Instead of setting exact centers, populates them with random values
`k`	`setK(k: Int)`	2	The number of clusters to create

Recommendation Systems

Recommender systems have achieved wide traction due to the popularity of e-commerce and online retailing. Some of the recommendations are responsible for billions of dollars' worth of sales for some of the top brands in the world. From LinkedIn connection suggestions to Facebook newsfeed prioritization and from Amazon and Netflix product recommendations to YouTube content matching, all of these rely on recommendation systems (recsys) in some form. Collaborative filtering is one of the most popular methods for recsys because of its reliance on crowd-sourced behavior. One class of collaborative filtering algorithms, aptly called the *model-based approach*, uses machine-learning techniques for training and predictions. Most algorithms can be broken down into matrix-decomposition problems in which the job of the algorithm is to minimize a loss function. MLlib uses the *alternating least squares* (ALS) algorithm for collaborative filtering, which minimizes the loss function using a simple repeated technique: fix half the parameters, and optimize the other half.

The ALS implementation in MLlib has two modes: explicit and implicit. The explicit model can be used when the dataset contains an explicit user rating for each item. Unfortunately, that rating is not always available in the real world. To account for such scenarios, implicit feedback uses observed preferences and confidence values.

At first glance, the physical activity dataset does not seem amenable to recsys: there are no items or ratings in the data. You can remedy that with a little preprocessing, using activity ID as a product and the number of times a user has performed it as its rating. The snippet of Spark code in Listing 9-11 performs this preprocessing by reading all input files and manipulating their data. `iPath` needs to point to the folder with the `.dat` data files. The data is divvied up into one file per subject, so the code needs to dynamically add the name of each input file to each record. You use the `mapPartitionsWithInputSplit` method provided by `HadoopRDD`, which provides the `map` function with a reference to the input split and an iterator for the data in the partition (line 6).[7] The path to each input can be acquired from the input split, which can then be parsed to get only the filename (line 7). From each record, you only need to extract the activity ID (line 7). The next series of transforms removes invalid activities (line 8) and counts the frequency of each subject-activity pair (lines 9–10). To add some semblance of randomness to the data, you sample 70% of it (line 12) and write it to a file.

Listing 9-11. Preprocessing the Physical Activity Dataset to Enable Collaborative Filtering

```
1.   val delim = " "
2.
3.   val sc = new SparkContext(conf)
4.   sc.hadoopFile(iPath, classOf[TextInputFormat], classOf[LongWritable], classOf[Text],
     sc.defaultMinPartitions)
5.     .asInstanceOf[HadoopRDD[LongWritable, Text]]
6.     .mapPartitionsWithInputSplit((iSplit, iter) =>
7.       iter.map(splitAndLine => (Files.getNameWithoutExtension(iSplit.
         asInstanceOf[FileSplit].getPath.toString), splitAndLine._2.toString.split(" ")
         (1))))
8.     .filter(r => r._2 != "0")
9.     .map(r => ((r._1, r._2), 1))
10.    .reduceByKey(_ + _)
11.    .map(r => r._1._1.replace("subject", "") + delim + r._1._2 + delim + r._2)
12.    .sample(false, 0.7)
13.    .coalesce(1)
14.    .saveAsTextFile(oPath)
```

Once you have preprocessed the data, you can use ALS to drive recommendations. As shown in Listing 9-12, you first read the preprocessed data stream and convert each record to a `Rating` type with three fields: `subject`, `activity`, and `frequency`. These represent user, item, and rating, respectively. ALS requires each data point to conform to this class. Similar to other machine-learning methods, you divide the dataset for testing and training and use the latter to fine-tune and calibrate the model. The ALS model requires the rank of the factorization, the number of iterations to perform, and the regularization parameter lambda for configuration. The rest of the tunable parameters are listed in Table 9-7.

[7]"Getting the Current Filename with Spark and HDFS," *The Modern Life*, September 28, 2014, http://themodernlife.github.io/scala/spark/hadoop/hdfs/2014/09/28/spark-input-filename/.

Listing 9-12. ALS-Powered, Collaborative, Filtering-Based Recommendations

```
val ratingStream = ssc.textFileStream(iPath).map(_.split(" ") match {
  case Array(subject, activity, freq) =>
    Rating(subject.toInt, activity.toInt, freq.toDouble)
})

val rank = 10
val numIterations = 10
val lambda = 0.01
ratingStream.foreachRDD(ratingRDD => {
  val testTrain = ratingRDD.randomSplit(Array(0.3, 0.7))
  val model = ALS.train(testTrain(1), rank, numIterations, lambda)
  val test = testTrain(0).map {
    case Rating(subject, activity, freq) =>
      (subject, activity)
  }
  val prediction = model.predict(test)
})
```

Notice that the recommendation in each batch is independent of the other batches due to the stateless nature of Spark Streaming. This means ALS in MLlib is not true online learning. If your application logic can tolerate per-batch stateless modeling, then this current design will work well for you. Having said that, future releases of MLlib will hopefully contain native support for online ALS.[8]

Table 9-7. *Configuration Parameters for ALS-Based Collaborative Filtering*

Parameter	Set	Default	Description
alpha	setAlpha(value: Double)	1.0	Rate of increase of the confidence level in the case of implicit feedback.
implicitPrefs	setImplicitPrefs(value: Boolean)	false	Whether to use implicit or explicit feedback. Explicit by default.
maxIter	setMaxIter(value: Int)	10	The maximum number of iterations to run.
nonnegative	setNonnegative(value: Boolean)	false	Enables non-negative constraints.
numBlocks	setNumBlocks(value: Int)	-1	The number of user and product blocks for parallelism. -1 enables auto-tuning.
rank	setRank(value: Int)	10	The number of latent factors.
regParam/lambda	setRegParam(value: Double)	0.01	The regularization factor.

[8]https://issues.apache.org/jira/browse/SPARK-6407.

Frequent Pattern Mining

Frequent pattern mining has very rapidly become the most sought-after application of data mining due to its direct influence on sales. A number of algorithms discover such patterns. For instance, the Apriori family of algorithms uses a "downward closure property"; sub-item sets of an itemset are also frequent. For example, if books, tea, and gin are frequent itemsets, then so are books and tea, and tea and gin. This property is used to quickly build itemsets: performing a scan of the data and first finding one-itemsets, then two-itemsets, and so on. This process requires multiple scans of the data. The FP-growth technique remedies this by first finding item frequencies and then generating a prefix tree of transactions. This prefix tree is used to generate candidate sets. MLlib contains a parallel FP-growth approach, which achieves parallelism by partitioning the transaction prefix space.

As with recommendation systems, the physical activity sensor dataset out of the box does not seem like a good candidate for frequent pattern mining. You need to make a few cosmetic changes to it before you can apply FP-growth, but do not despair: this book has another trick up its sleeve. This example finds frequent activities. The code in Listing 9-13 does so by preprocessing the dataset to emit the sequence of activities performed by each user. The code is very similar to Listing 9-11, with the difference that instead of `<user, activity, frequency>` triples, each line contains unique activity IDs per subject. You can feed this data to an FP-growth model, which is what you do in the snippet of code in Listing 9-14.

Listing 9-13. Preprocessing the Sensor Activity Dataset for Frequent Itemset Mining

```
sc.hadoopFile(iPath, classOf[TextInputFormat], classOf[LongWritable], classOf[Text], sc.defaultMinPartitions)
  .asInstanceOf[HadoopRDD[LongWritable, Text]]
  .mapPartitionsWithInputSplit((iSplit, iter) =>
    iter.map(splitAndLine => (Files.getNameWithoutExtension(iSplit.asInstanceOf[FileSplit].getPath.toString), splitAndLine._2.toString.split(" ")(1))))
  .filter(r => r._2 != "0")
  .map(r => (r._1, r._2))
  .distinct()
  .groupByKey()
  .map(r => r._2.mkString(" "))
  .sample(false, 0.7)
  .coalesce(1)
  .saveAsTextFile(oPath)
```

Similar to Listing 9-12, the FP-growth-based code in 9-14 also refreshes its model in every batch interval. Minimum support dictates how many transactions an item needs to occur in before it can be considered frequent (lines 1 and 7). You pass the per interval transactions RDD to the `FPGrowth#run` method, which returns the frequent itemsets (line 8). The rest of the code simply prints these itemsets and their frequencies (lines 10–13).

Listing 9-14. FP-Growth-Based Mining to Determine Frequent Activities in the Physical Activity Dataset

```
1.   val minSupport = 0.4
2.
3.   ssc.textFileStream(iPath)
4.     .map(r => r.split(" "))
5.     .foreachRDD(transactionRDD => {
6.       val fpg = new FPGrowth()
7.         .setMinSupport(minSupport)
```

```
    val model = fpg.run(transactionRDD)

    model.freqItemsets
      .collect()
      .foreach(itemset => println("Items: %s, Frequency: %s".format(itemset.items.
      mkString(" "), itemset.freq)))
  })
```

So far, you have only looked at machine-learning tasks—preprocessing, feature extraction, prediction, and so on—in isolation. In the real world, these need to be executed as a pipeline of tasks: one following the other. Spark 1.2 introduced a new package, ML, which simplifies the construction of such pipelines. Let's see what it has to offer.

Streaming ML Pipeline Application

To get your hands dirty with ML, let's revisit one of the examples you implemented earlier: predicting whether a sensor reading represents walking or running. The end-to-end ML version of the code is presented in Listing 9-15.

Listing 9-15. First Streaming ML Pipeline Application

```
package org.apress.prospark

import scala.reflect.runtime.universe

import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.feature.Normalizer
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.regression.RandomForestRegressor
import org.apache.spark.sql.SQLContext
import org.apache.spark.streaming.Seconds
import org.apache.spark.streaming.StreamingContext

object MLPipelineApp {

  case class Activity(label: Double,
    accelXHand: Double, accelYHand: Double, accelZHand: Double,
    accelXChest: Double, accelYChest: Double, accelZChest: Double,
    accelXAnkle: Double, accelYAnkle: Double, accelZAnkle: Double)

  def main(args: Array[String]) {
    if (args.length != 4) {
      System.err.println(
        "Usage: MLPipelineApp <appname> <batchInterval> <hostname> <port>")
      System.exit(1)
    }
    val Seq(appName, batchInterval, hostname, port) = args.toSeq

    val conf = new SparkConf()
31.        .setAppName(appName)
32.        .setJars(SparkContext.jarOfClass(this.getClass).toSeq)
33.
34.      val ssc = new StreamingContext(conf, Seconds(batchInterval.toInt))
35.
36.      val sqlC = new SQLContext(ssc.sparkContext)
37.      import sqlC.implicits._
38.
39.      val substream = ssc.socketTextStream(hostname, port.toInt)
40.        .filter(!_.contains("NaN"))
41.        .map(_.split(" "))
42.        .filter(f => f(1) == "4" || f(1) == "5")
43.        .map(f => Array(f(1), f(4), f(5), f(6), f(20), f(21), f(22), f(36), f(37), f(38)))
44.        .map(f => f.map(v => v.toDouble))
45.        .foreachRDD(rdd => {
46.          if (!rdd.isEmpty) {
47.            val accelerometer = rdd.map(x => Activity(x(0), x(1), x(2), x(3), x(4), x(5),
             x(6), x(7), x(8), x(9))).toDF()
48.            val split = accelerometer.randomSplit(Array(0.3, 0.7))
49.            val test = split(0)
50.            val train = split(1)
51.
52.            val assembler = new VectorAssembler()
53.              .setInputCols(Array(
54.                "accelXHand", "accelYHand", "accelZHand",
55.                "accelXChest", "accelYChest", "accelZChest",
56.                "accelXAnkle", "accelYAnkle", "accelZAnkle"))
57.              .setOutputCol("vectors")
58.            val normalizer = new Normalizer()
59.              .setInputCol(assembler.getOutputCol)
60.              .setOutputCol("features")
61.            val regressor = new RandomForestRegressor()
62.
63.            val pipeline = new Pipeline()
64.              .setStages(Array(assembler, normalizer, regressor))
65.            val model = pipeline.fit(train)
66.            val prediction = model.transform(test)
67.            prediction.show()
68.          }
69.        })
70.
71.      ssc.start()
72.      ssc.awaitTermination()
73.    }
74.
75.  }
```

You have already seen variants of lines 39–44 in the other examples in this chapter, so they should be fairly familiar. The real magic happens in the foreachRDD block. ML relies on data frames as the common data substrate. Therefore, the first order of the day is to convert the input records to a data frame. The application uses the implicit conversion from a case-class method as described in Chapter 8 (line 47).

The conversion to a data frame has a positive side effect that you can now use data frame methods to manipulate the data. For instance, this example uses the `randomSplit` data frame method to divide the dataset for testing and training.

The application has three stages: conversion to vector, normalization, and prediction. Each needs to be initialized separately. For the first part of the pipeline, you use `VectorAssembler`, which takes as input an array of data-frame columns and clumps them together into a vector column (lines 52–57). Note that all components in a pipeline operate on the same data frame—more columns are added to it as it moves down the pipeline. In the example pipeline, the next task is to normalize the data to ensure similar ranges across the three accelerometers. The input to the normalizer is the output of the assembler. Learning algorithms in ML expect an input column called `features` in the data frame; so, the name `features` for the output column of the normalizer is intentional. Similarly, learning algorithms require a `label` column during the training phase; that is why the `Activity` case class for the main data frame contains a `label` variable for the activity.

The learning algorithm uses a random forest of regression trees (line 61), because they tend to perform better due to their ensemble approach as opposed to a single regression model. Now that all three parts are ready, they need to be cobbled together to form a pipeline. The `Pipeline` object (line 63) accepts as argument the different stages it needs to run (line 64) and executes them from left to right in the array. The `fit` method for the `Pipeline` object takes as input a data frame for training (line 65). Once the learning is complete, you can start scoring test data (line 66) via the `transform` method of the model. That's it. It is that easy to create and execute an end-to-end ML pipeline. The next section fleshes out some details of the ML package.

ML

The ML package is designed for complete workflows of machine-learning tasks. These workflows encompass end-to-end processing of machine-learning pipelines. From normalization and feature extraction to learning and recommendations, ML covers the entire spectrum. Some of these components are built on top of MLlib building blocks. To get a handle on the different concepts involved, let's revisit the example from Listing 9-15. Figure 9-1 shows the same example in the form of a block flow. Each arrow represents a data frame. There are two types of high-level components in ML: transformers and estimators (represented by red and green boxes, respectively, in Figure 9-1). Transformers and estimators in ML-speak are called *pipeline stages*, and a pipeline is defined as a *flow* of stages. So what are transformers and estimators, exactly?

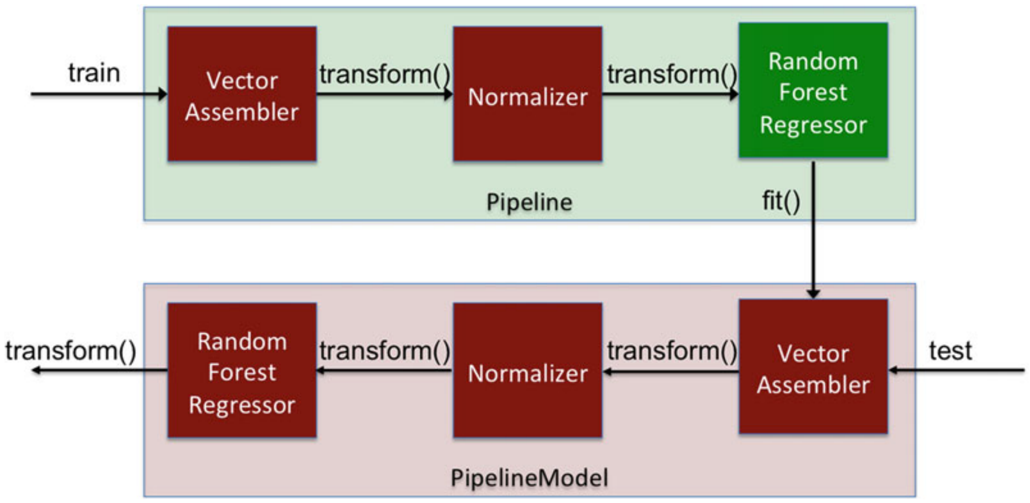


Figure 9-1. *Block diagram of the ML application*

Transformers, intuitively, transform a data frame by appending one or more columns to it. For instance, the VectorAssembler transforms one or more columns by converting them into a single-vector column. Similarly, the Normalizer transforms a single-vector column into a new column where the values have been normalized. Under the hood, each transformer implements a transform method that is invoked by the pipeline. In contrast, estimators are trained on data to create transformers. All learning models are estimators. For instance, RandomForestRegressor trains on the data frame features column and produces a PipelineModel object, which can be used for scoring new data. In essence, the PipelineModel reimplements the entire pipeline by turning all estimators into transformers (the bottom set of blocks in Figure 9-1). Each estimator implements a fit method to fit a data frame. The ML package contains a number of transformers and estimators to enable a wide range of machine-learning applications, covering feature selection, classification, regression, and recommendations.

Both transformers and estimators use a common API for parameters, where each parameter has the type Param. Parameters to them can be supplied via traditional setters on their objects: for instance, in Listing 9-15, if you wanted to set the value of P for the normalizer, you could invoke setP(Double) on its object. The alternative is to use a ParamMap, which as the name suggests is a map of parameters; it can be passed to the fit() method as shown in Listing 9-16.

Listing 9-16. Setting ML Parameters via ParamMap

```
val pMap =  ParamMap(normalizer.p -> 1.0)
val model = pipeline.fit(train, pMap)
```

Cross-Validation of Pipelines

Cross-validation is the process of measuring how well a model will predict unseen data. In addition to pipelined execution, ML can be used to cross-validate parameters and settings for a pipeline and automatically select the best set of settings based on an evaluator. Sticking to the example-first approach, let's jump into code to learn the ropes. Listing 9-17 extends the previous example (Listing 9-15) with cross-validation. You create a pipeline as before (line 19). The CrossValidator (line 22) needs to be given the estimator (pipeline in this case) under evaluation and a metric for evaluation. The latter is provided by an Evaluator, which takes as input a data frame with labels and predictions and returns a scalar metric to rate the quality of the prediction. ML provides two evaluators out of the box that cater to regression and classification, respectively: RegressionEvaluator and BinaryClassificationEvaluator.

CrossValidator validates a complete pipeline by accepting a series of parameters and evaluating the provided estimator. ML uses the k-folds cross-validation methodology, which divides the dataset into k equal-sized chunks. Out of these k samples, k-1 are used for training and one is used for testing. This is done iteratively k times (hence the name *k-folds*) so that each of the k samples is used as a testing dataset. The parameter exploration space is provided to CrossValidator via an array of ParamMap, for which you can use a builder (ParamGridBuilder); its use is shown on lines 25–28. Your evaluation space spans the Normalizer P-value and RandomForestRegressor number of trees, so you provide an array of values for each. You also set the value of k to 5 (line 30). Fitting the validator returns the best-performing calibrated pipeline (line 32), which can then be used to score new data (line 33).

Listing 9-17. Using ML Cross-Validation for Calibrated Model Selection

```
1.   .foreachRDD(rdd => {
2.     if (!rdd.isEmpty) {
3.       val accelerometer = rdd.map(x => Activity(x(0), x(1), x(2), x(3), x(4), x(5), x(6),
         x(7), x(8), x(9))).toDF()
4.       val split = accelerometer.randomSplit(Array(0.3, 0.7))
5.       val test = split(0)
```

```
      val train = split(1)

      val assembler = new VectorAssembler()
        .setInputCols(Array(
          "accelXHand", "accelYHand", "accelZHand",
          "accelXChest", "accelYChest", "accelZChest",
          "accelXAnkle", "accelYAnkle", "accelZAnkle"))
        .setOutputCol("vectors")
      val normalizer = new Normalizer()
        .setInputCol(assembler.getOutputCol)
        .setOutputCol("features")
      val regressor = new RandomForestRegressor()

      val pipeline = new Pipeline()
        .setStages(Array(assembler, normalizer, regressor))

      val validator = new CrossValidator()
        .setEstimator(pipeline)
        .setEvaluator(new RegressionEvaluator)
      val pGrid = new ParamGridBuilder()
        .addGrid(normalizer.p, Array(1.0, 5.0, 10.0))
        .addGrid(regressor.numTrees, Array(10, 50, 100))
        .build()
      validator.setEstimatorParamMaps(pGrid)
      validator.setNumFolds(5)

      val bestModel = validator.fit(train)
      val prediction = bestModel.transform(test)
      prediction.show()
    }
  })
```

This was a primer in Spark ML. The package itself has a lot to offer. Now that you've had a taste of its use, you can implement your own machine-learning pipelines for real-time data.

Summary

Machine learning fuels data-driven operations across businesses and operations. Every time you surf the Internet, an entire battery of machine-learning algorithms behind the scenes enhances your experience. They control everything from your Facebook newsfeed to the Amazon main page and from personalized ads on any web site to spam email filtering.

This chapter looked at using Spark MLlib and ML to implement scalable predictive analytics and learning patterns from the data. Feature selection enabled you to reduce the number of features from the dataset as well as to enhance the impact of features. The majority of the chapter walked you through the implementation of machine-learning applications encompassing a number of learning algorithms and data properties. You finished the chapter with the Spark ML package to architect end-to-end machine-learning pipelines and their calibration and selection.

You are almost at the end of your Spark Streaming journey. The final chapter of this book takes you into the land of clouds and lambdas. Buckle up.

CHAPTER 10

Of Clouds, Lambdas, and Pythons

Clouds come floating into my life, no longer to carry rain or usher storm, but to add color to my sunset sky.

—Rabindranath Tagore

In the real world, deployments of Big Data systems fit into a larger ecosystem that consists of managed cloud instances, cluster managers, data stores and warehouses, and so on. Cloud deployments enable organizations to pass the buck of DevOps to the cloud service provider and thus free them to focus on application development and business operations. Along with economies of scale, this provides on-demand horizontal elasticity and simplified scheduling. Similar to other systems, Spark can work in the cloud with fully managed instances provided by a number of companies including Google (Dataproc), Databricks, and IBM (Bluemix). No book about Spark would be complete without a discussion of running it in the cloud. In a similar vein, applications in an organization evolve over time and vary widely in terms of programming languages and models. This may be for legacy reasons (dragging an old stack), practitioner preference (data scientists love Python), or system limitations (a platform may only have an API in a particular language). For these reasons and more, Scala may not be the language of choice for Spark applications across the board. In this chapter, you also get your hands dirty with the Spark Python API.

Applications running in the cloud do not follow one monolithic design or ingest a single type of data. An increasingly large class of applications requires blending data in motion (real-time data) with data at rest (historical batch data). Examples of such applications include training a machine-learning model in batch mode with historical data and then using it in real time to score streaming data, filtering streaming data before landing it in a data warehouse for business intelligence, and serving requests from a real-time system while the batch system recalculates the view. The last example constitutes a popular paradigm known as the *Lambda Architecture*, which is another topic of discussion in this chapter.

Finally, many relations and datasets in the wild can be represented as large graphs and their applications as graph problems. All of Facebook can be turned into a massive graph of users and pages. Similarly, search engines like Google map the World Wide Web to a graph with 30 billion nodes. Algorithms similar to PageRank are then used to determine the popularity and relevance of every web page. The sizes of some of these graphs are on the order of petabytes, which means analyzing them requires the use of distributed Big Data platforms like Spark. To cater to such applications, this chapter wraps up by analyzing streaming graphs.

The very last set of applications in the book involve location-based services and, in particular, how they blend the offline and online worlds. For instance, a single star improvement in the rating of a small business on Yelp leads to a 5% to 9% increase in revenue.[1] To enable such analytics, in this chapter you use a publicly available dataset from Yelp with rich data for businesses, check-ins, users, tips, and reviews.

[1]Brad Plumer, "How Yelp Is Killing Chain Restaurants," *The Washington Post*, October 3, 2011, https://www.washingtonpost.com/blogs/ezra-klein/post/how-yelp-is-killing-chain-restaurants/2011/10/03/gIQAokJvHL_blog.html.

Z. Nabi, *Pro Spark Streaming,* DOI 10.1007/978-1-4842-1479-4_10

A Good Review Is Worth a Thousand Ads

Online social media is on course to replace traditional media as the focal point for marketing and advertising. This is due to its wider reach, richer semantics for microsegmentation, and dynamic personalization. In addition, review platforms such as Yelp have changed the face of how marketing works: rather than putting ads up, businesses can rely on the good word (hopefully) of ordinary users to attract more customers to their establishments. These reviews follow a five-star rating model along with more detailed freestyle textual reviews. Even simple check-ins are an effective medium for creating buzz about a business.

Yelp has been sharing a limited set of its data with the research community for crowd-sourced analytics as part of the Yelp Dataset Challenge. The data from the current round (7) of the challenge spans 10 cities across 4 countries (listed in Table 10-1).[2] The dataset consists of 5 files, each of which contains JSON records for 2.2 million reviews, 591,000 tips, 552,000 users, 77,000 businesses, and check-ins, respectively. It also cumulatively includes 566,000 attributes (for instance, being child friendly) for the businesses. Finally, because Yelp is also a social network, the data includes 3.5 million connections among the users. An auxiliary dataset also contains 200,000 pictures for the businesses in the main dataset. You drill down into the attributes of each data file as you use them over the course of this chapter.

Table 10-1. *Cities Represented in the Yelp Dataset Challenge*

Country	Cities
Scotland	Edinburgh
Germany	Karlsruhe
Canada	Montreal and Waterloo
United States	Charlotte, Las Vegas, Madison, Phoenix, Pittsburgh, Urbana-Champaign

Google Dataproc

Dataproc is Google's YARN-as-a-service platform. It has out-of-the-box support for Hadoop, Hive, Pig, and Spark. The unit of allocation in Dataproc is a *cluster*, which is a self-contained environment with a YARN resource manager and node managers. These are executed on Google Compute Engine virtual machine instances, which are provisioned on demand. Dataproc applications can also integrate with other Google cloud services such as Cloud Storage, BigQuery, and Cloud Bigtable.

Dataproc is one of the many services offered in the Google Cloud Platform, as listed in Table 10-2. Google Cloud Platform is rapidly turning into a one-stop shop for everything cloud related. Let's set up a Dataproc project and then run an application on top of it.

[2] https://www.yelp.com/dataset_challenge/.

Table 10-2. *Brief introduction to some of the services in Google Cloud Platform*

Service	Category	Description
App Engine	Compute	Platform as a service for web apps and back ends
Compute Engine	Compute	Infrastructure as a service with virtual machines to run arbitrary applications
Container Engine	Compute	Docker container execution as a service built on top of Kubernetes
HTTP load balancing	Network	Load balancing for HTTP traffic among Compute Engine/ Container Engine instances/containers
Network load balancing	Network	Load balancing for arbitrary TCP/UDP traffic
Cloud DNS	Network	Global DNS as a service
Cloud Bigtable	Storage	Bigtable (HBase) NoSQL data storage as a service
Datastore	Storage	ACID transactions on top of Bigtable
Cloud Storage	Storage	Cloud-based, durable, flat object storage
Cloud SQL	Storage	MySQL as a service
BigQuery	Big Data	SQL queries against petabyte-scale, append-only tables
Pub/Sub	Big Data	High-throughput messaging using the publisher/ subscriber model
Dataproc	Big Data	Hadoop, Spark, Pig, and Hive on top of YARN as a service
Dataflow	Big Data	Combined batch processing and stream processing with a FlumeJava-like API
Genomics	Big Data	Storage and processing of Genomics data using other services including BigQuery and Bigtable

SETTING UP GOOGLE CLOUD PLATFORM DATAPROC

Go to `https://cloud.google.com/dataproc/`, and click Try It Free.

Log in using your Google account credentials.

Sign up for a free Google Cloud Platform Account.[3] You need to provide debit/credit card details.

Install the `gcloud` command-line tool:

```
$ curl https://sdk.cloud.google.com | bash
$ exec -l $SHELL
$ gcloud init
```

Go to `https://console.cloud.google.com/`, and click the Products & Services tab (represented by three horizontal lines) as shown in Figure 10-1.

[3]At the time of writing, Google is offering $300 of free credit for 60 days for new accounts.

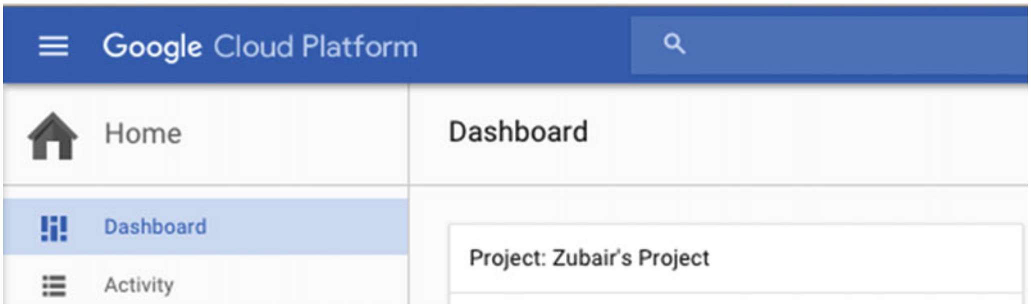


Figure 10-1. *Accessing the Products & Services menu for Google Cloud Platform*

Clicking the button displays a scrollable menu, as shown in Figure 10-2.

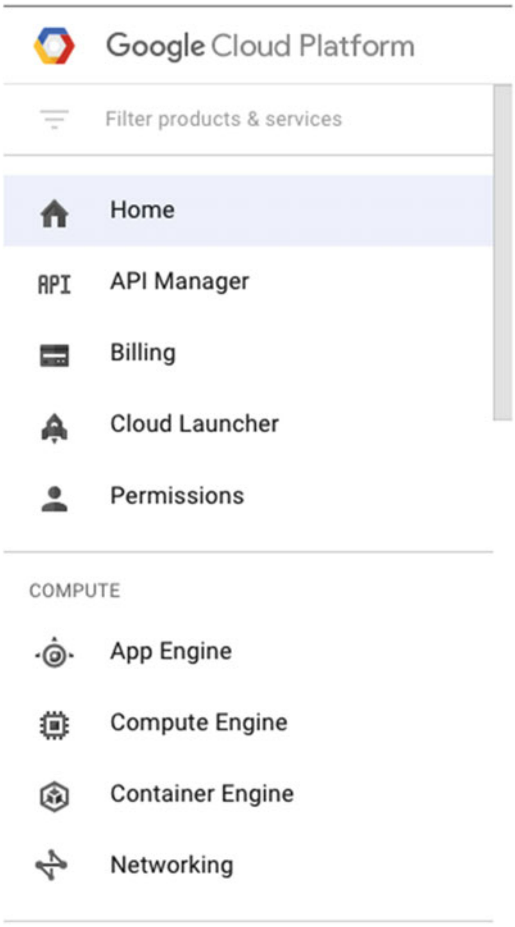


Figure 10-2. *GCP Products & Services menu*

Scroll down to the Big Data part of the menu, and select Dataproc (Figure 10-3).

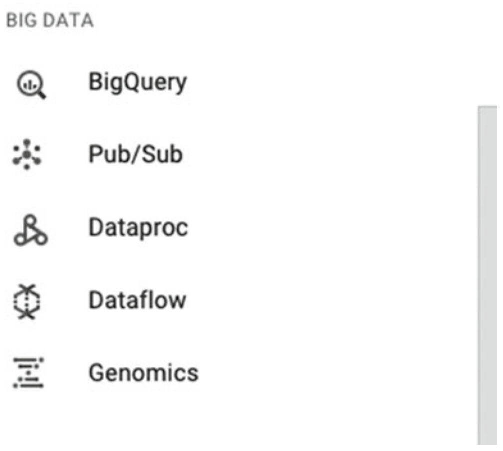
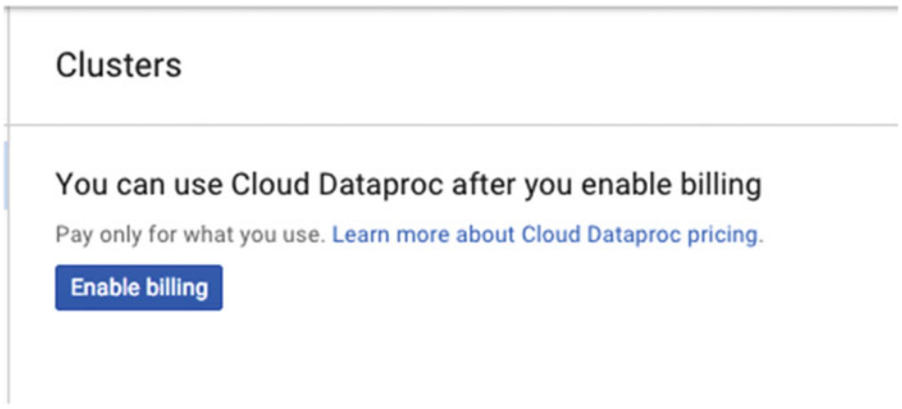


Figure 10-3. *Selecting Dataproc from the Big Data services provided by Google Cloud Platform*

Selecting Dataproc takes you to the screen in Figure 10-4, which tells you to enable billing before you can use Dataproc. If you are using a preconfigured GCP project, you can skip the next two steps.


Clusters

You can use Cloud Dataproc after you enable billing

Pay only for what you use. Learn more about Cloud Dataproc pricing.

Figure 10-4. *Billing needs to be enabled before using Dataproc*

Click Enable Billing to go to the Billing Account setup screen (Figure 10-5).

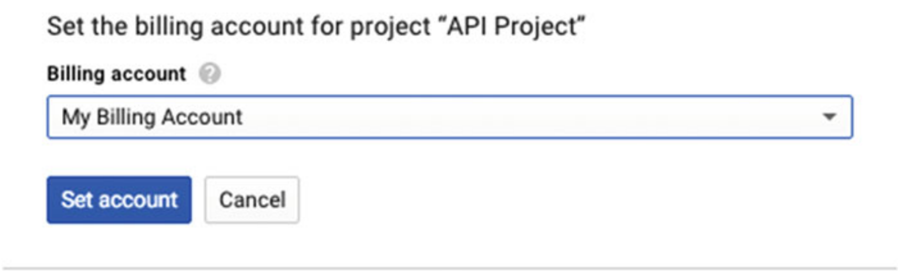
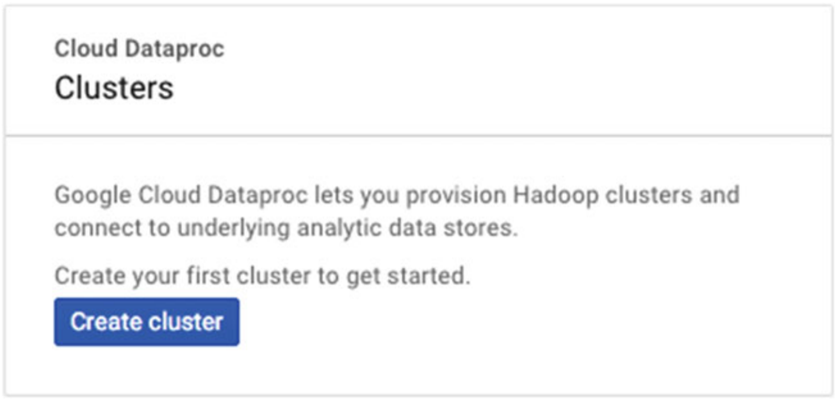

Set the billing account for project "API Project"

Figure 10-5. *Enabling a billing account for your GCP project*

Go back to the Dataproc dashboard, and you should be able to create a cluster, as shown in Figure 10-6.

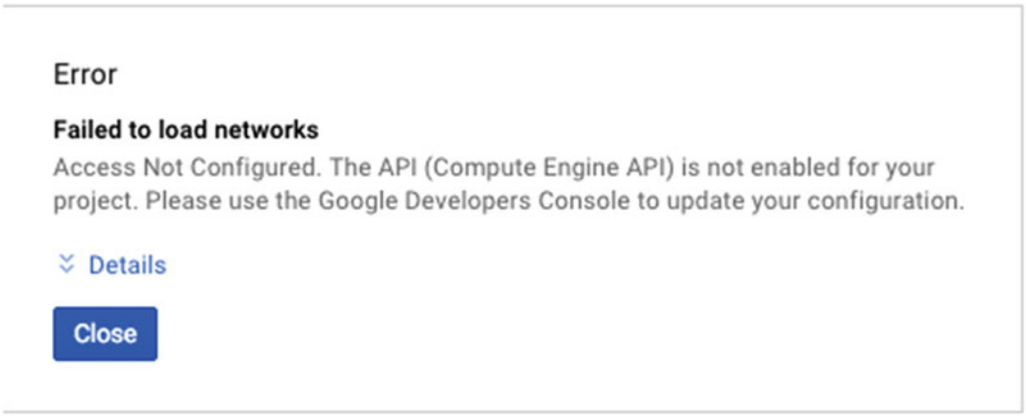


Google Cloud Dataproc lets you provision Hadoop clusters and connect to underlying analytic data stores.

Create your first cluster to get started.

Figure 10-6. *Dataproc cluster-creation screen*

Clicking Create Cluster takes you to the screen in Figure 10-7, because the Compute Engine API needs to be enabled first. Go to the Compute Engine dashboard to enable it.


Error

Failed to load networks

Access Not Configured. The API (Compute Engine API) is not enabled for your project. Please use the Google Developers Console to update your configuration.

Figure 10-7. *The Compute Engine API needs to be enabled before you can use Dataproc*

You should now be able to create a Dataproc cluster.

First Spark on Dataproc Application

The Yelp dataset also contains a number of features about businesses.[4] For each business, these features include its location, full address, and the reviews it has received. For restaurants, the dataset also contains qualitative information such as whether the location is child friendly and whether it accepts credit cards. Listing 10-1 contains the JSON template for this dataset.

Listing 10-1. JSON Blueprint of the Business Dataset

```
{
   "business_id":"<anonymized_id>",
   "full_address":"<street_address>",
   "hours":{
      "<day_of_week>":{
         "close":"<HH:MM>",
         "open":"<HH:MM>"
      }
   },
   "open":<true/false>,
   "categories":[
      <list_of_categories_such_as_restaurant>
   ],
   "city":"<self_explanatory>",
   "review_count":<number_of_reviews>,
   "name":"<self_explanatory>",
   "neighborhoods":[
      <list_of_neigborhood_names>
   ],
   "longitude":<self_explanatory>,
   "state":"<self_explanatory>",
   "stars":<star_count>,
   "latitude":<self_explanatory>,
   "attributes":{
      <key_value_pairs_of_attributes>
   },
   "type":"business"
}
```

One of the attributes is `Wi-Fi`, with three fairly obvious values: `no`, `free`, and `paid`. An interesting application would be to figure out whether there is any correlation between having WiFi access and the rating of an establishment. The code for this application is in Listing 10-2. It reads data from a socket (line 32) and converts each record to a JSON object (line 35). The application requires only two features, WiFi presence and star rating, so you confine yourself to records that contain the latter (lines 37–39) and convert them into key-value pairs where the key is the WiFi type and the value is the star rating (lines 40–43).

To calculate the average rating per WiFi type, you exercise a custom `combineByKey` transform to get an overall sum and count and a subsequent `map` operation to perform the actual average (recall Listing 3-18 in Chapter 3).

[4]Contained in `yelp_academic_dataset_business.json`.

Listing 10-2. First Spark on Dataproc Application to Compare the Ratings of Restaurants With and Without Free WiFi

```
package org.apress.prospark

import org.apache.spark.HashPartitioner
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.streaming.Seconds
import org.apache.spark.streaming.StreamingContext
import org.apache.spark.streaming.dstream.DStream.toPairDStreamFunctions
import org.json4s.DefaultFormats
import org.json4s.JsonAST.JNothing
import org.json4s.jvalue2extractable
import org.json4s.jvalue2monadic
import org.json4s.native.JsonMethods.parse
import org.json4s.string2JsonInput

object DataProcApp {

  def main(args: Array[String]) {
    if (args.length != 4) {
      System.err.println(
        "Usage: DataProcApp <appname> <batchInterval> <hostname> <port>")
      System.exit(1)
    }
    val Seq(appName, batchInterval, hostname, port) = args.toSeq

    val conf = new SparkConf()
      .setAppName(appName)
      .setJars(SparkContext.jarOfClass(this.getClass).toSeq)

    val ssc = new StreamingContext(conf, Seconds(batchInterval.toInt))

    ssc.socketTextStream(hostname, port.toInt)
      .map(r => {
        implicit val formats = DefaultFormats
        parse(r)
      })
      .filter(jvalue => {
        jvalue \ "attributes" \ "Wi-Fi" != JNothing
      })
      .map(jvalue => {
        implicit val formats = DefaultFormats
        ((jvalue \ "attributes" \ "Wi-Fi").extract[String], (jvalue \ "stars").extract[Int])
      })
      .combineByKey(
        (v) => (v, 1),
        (accValue: (Int, Int), v) => (accValue._1 + v, accValue._2 + 1),
        (accCombine1: (Int, Int), accCombine2: (Int, Int)) => (accCombine1._1 + accCombine2._1, accCombine1._2 + accCombine2._2),
        new HashPartitioner(ssc.sparkContext.defaultParallelism))
      .map({ case (k, v) => (k, v._1 / v._2.toFloat) })
      .print()

    ssc.start()
    ssc.awaitTermination()
  }

}
```

This application consumes the Yelp businesses JSON data via the SocketDriver. The only minor change you need to make is to replace line 1 in Listing 9-2 (AbstractDriver) with this:

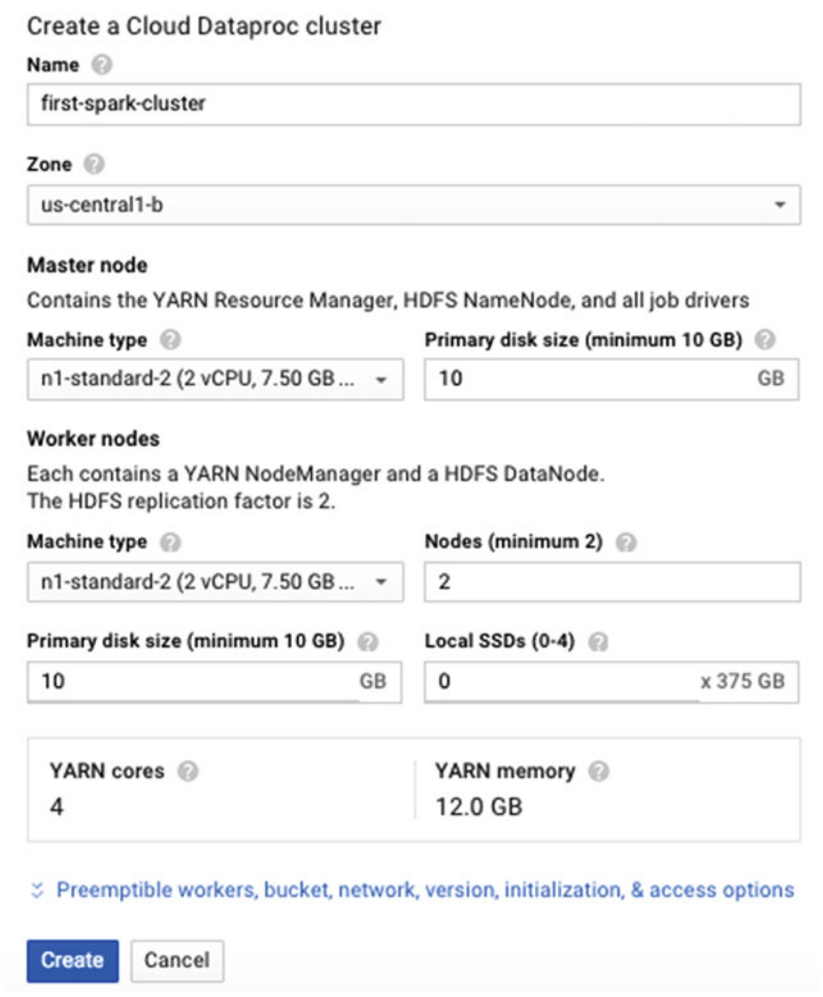
```
else if (ext.equals("dat") || ext.equals("json"))
```

To run the application on Dataproc, create a JAR from it using `sbt assembly`. To complete the story, let's create a Dataproc cluster.

CREATING A DATAPROC CLUSTER

Go to the Google Cloud Platform dashboard, and from the Big Data section of the Products & Services menu, select Dataproc. Finally, click Create cluster (Figure 10-6).

Create a cluster based on your requirements. Figure 10-8 shows the configuration for a bare minimum cluster.



Create a Cloud Dataproc cluster

Name: first-spark-cluster

Zone: us-central1-b

Master node

Contains the YARN Resource Manager, HDFS NameNode, and all job drivers

Machine type: n1-standard-2 (2 vCPU, 7.50 GB ...

Primary disk size (minimum 10 GB): 10 GB

Worker nodes

Each contains a YARN NodeManager and a HDFS DataNode.
The HDFS replication factor is 2.

Machine type: n1-standard-2 (2 vCPU, 7.50 GB ...

Nodes (minimum 2): 2

Primary disk size (minimum 10 GB): 10 GB

Local SSDs (0-4): 0 x 375 GB

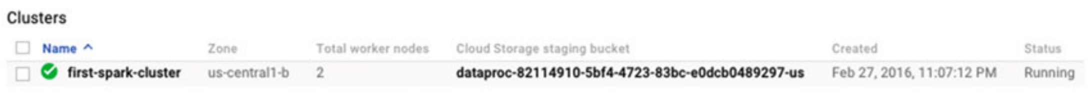
YARN cores: 4

YARN memory: 12.0 GB

Preemptible workers, bucket, network, version, initialization, & access options
Create
Cancel

Figure 10-8. *Creating a bare minimum cluster on Dataproc*

This creates and launches a cluster and takes you to a screen similar to the one in Figure 10-9.



Clusters

	Name ^	Zone	Total worker nodes	Cloud Storage staging bucket	Created	Status
☐	✓ first-spark-cluster	us-central1-b	2	dataproc-82114910-5bf4-4723-83bc-e0dcb0489297-us	Feb 27, 2016, 11:07:12 PM	Running

Figure 10-9. *Details of a running Dataproc cluster*

Clicking the name of the cluster leads you to the dashboard for that cluster (Figure 10-10). In this dashboard, you can see the resource consumption (CPU, disk, and memory) of the cluster at different time scales in the form of graphs. In addition, you can get details of running jobs, VM instances that constitute the cluster, and the configuration.

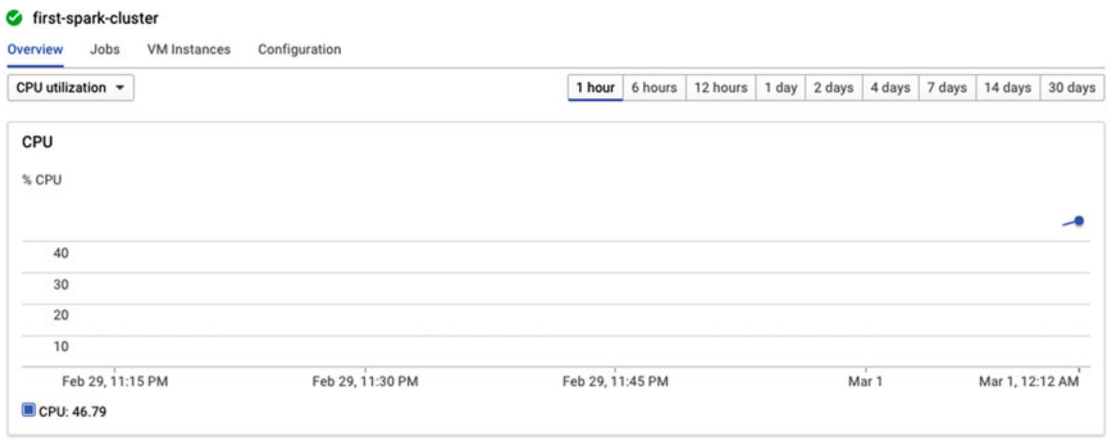
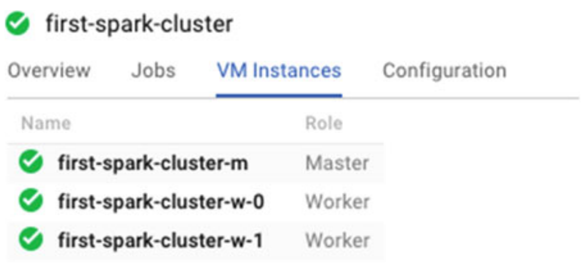


Figure 10-10. *Central dashboard for a Dataproc cluster*

The VM Instances tab lists all the instances in the cluster, as shown in Figure 10-11. As mentioned earlier, all of these VMs are Compute Engine instances. Therefore, selecting an instance takes you to its Compute Engine dashboard. You can see the three instances (one master plus two workers) that you just created.



first-spark-cluster

Overview Jobs VM Instances Configuration

Name	Role
✓ first-spark-cluster-m	Master
✓ first-spark-cluster-w-0	Worker
✓ first-spark-cluster-w-1	Worker

Figure 10-11. *VM instances in a Dataproc cluster*

RUNNING A SPARK APPLICATION ON DATAPROC

After creating a Dataproc cluster, you are ready to execute your Spark application on top of it. There are three ways to run an application on Dataproc:

- Dataproc console
- `gcloud` command-line tool
- REST API

To begin, let's use the console.

After creating a JAR from the application, you need to copy it over to the cluster. There are three locations to provide the JAR for execution:

- GCS
- HDFS
- Local FS on the cluster

This example takes the HDFS route. The first step is to copy the JAR to the cluster, for which you use the `gcloud` command-line tool. You simply transfer it to the master node (`first-spark-cluster-m` in this case) using

```
gcloud compute copy-files Chap10-assembly-1.0.jar first-spark-cluster-m:~/
```

Once the JAR has been uploaded to the cluster, you need to transfer it to HDFS. Let's first SSH into the master node:

```
gcloud compute --project "<your_gcloud_project_id>" ssh --zone "us-central1-b" "first-
spark-cluster-m"
```

Then, from that master node, you can transfer the JAR to HDFS:

```
hdfs dfs -copyFromLocal Chap10-assembly-1.0.jar /
```

You also execute the `SocketDriver` on the master node to feed data to your Spark Streaming application. Execute the `SocketDriver` before running the Spark application. You obviously first need to copy the data to the master node:

```
gcloud compute copy-files yelp_academic_dataset_business.json first-spark-cluster-m:~/
```

Now that everything has been set up, you can run the job. In the Dataproc dashboard, click the Jobs button on the left, and then click Submit Job. This is shown in Figure 10-12.

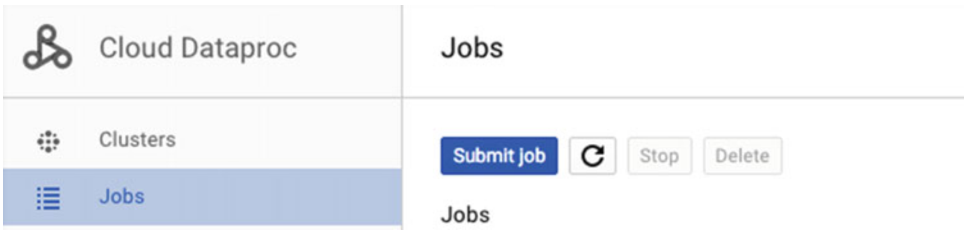
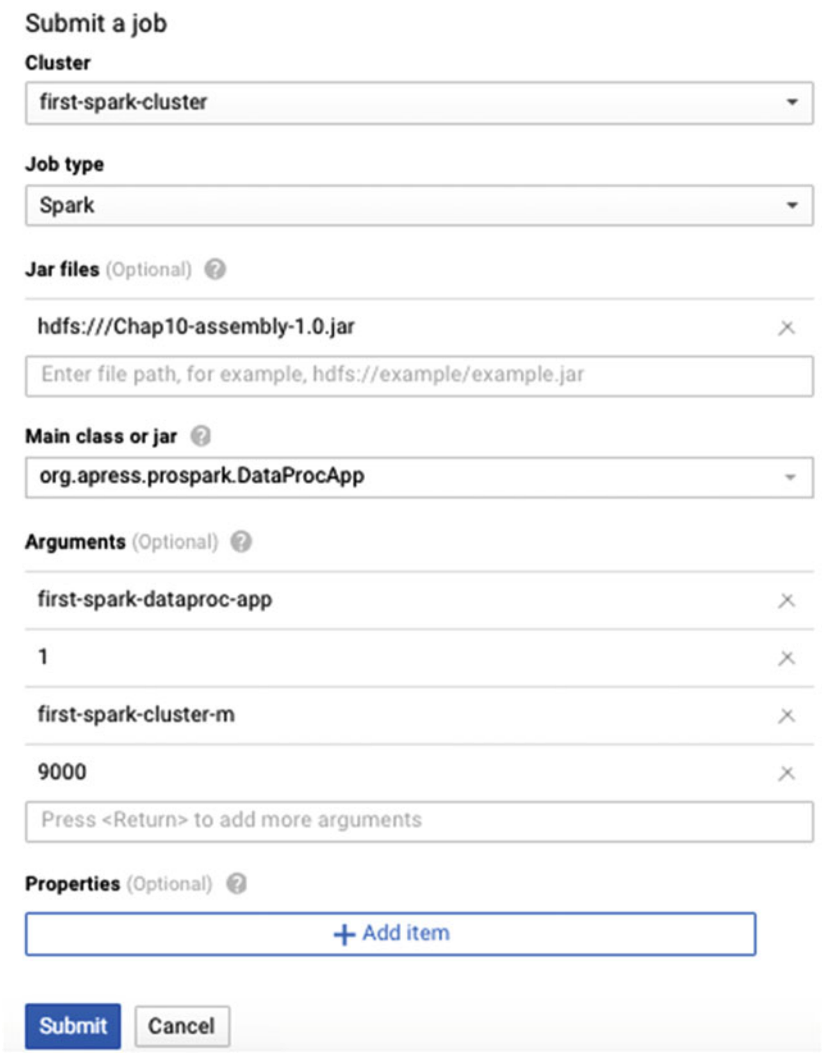


***Figure 10-12.** Submitting a job to Dataproc*

Fill out the prompt with parameters to run the JAR, as shown in Figure 10-13, and then click Submit.



Submit a job

Cluster: first-spark-cluster

Job type: Spark

Jar files (Optional): hdfs:///Chap10-assembly-1.0.jar

Enter file path, for example, hdfs://example/example.jar

Main class or jar: org.apress.prospark.DataProcApp

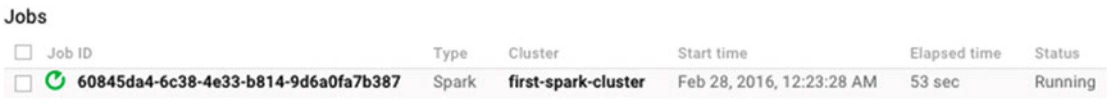
Arguments (Optional):
first-spark-dataproc-app
1
first-spark-cluster-m
9000

Press <Return> to add more arguments

Properties (Optional)

***Figure 10-13.** Entering details for the Spark job*

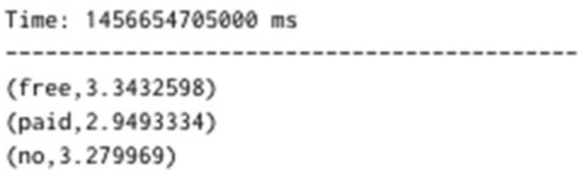
After the job is submitted, you are taken to the screen in Figure 10-14.



☐	Job ID	Type	Cluster	Start time	Elapsed time	Status
☐	✓ 60845da4-6c38-4e33-b814-9d6a0fa7b387	Spark	first-spark-cluster	Feb 28, 2016, 12:23:28 AM	53 sec	Running

Figure 10-14. *List of running Dataproc jobs*

Click the job ID to jump to the console output of the Spark application. You see output similar to that in Figure 10-15. As it turns out, restaurants where the WiFi is paid have a lower rating than establishments with no WiFi at all! This may be because customers believe the price of the WiFi should be factored into the price of the food, for instance. On the other hand, restaurants with free WiFi have the best star ratings on average.



```
Time: 1456654705000 ms
-------------------------------------------
(free,3.3432598)
(paid,2.9493334)
(no,3.279969)
```

Figure 10-15. *Output of the WiFi and star rating Spark application*

This was a first taste of Dataproc. In the next section, you kill two birds with one stone by using Dataproc to execute a Spark Streaming Python application.

PySpark

Although Spark is implemented in Scala, it has bindings for a number of languages include Java, Python, and C#. Of these, Python is a very popular choice especially in the data science community, due to its succinct syntax and rich set of powerful libraries such as NumPy, SciPy, matplotlib, and pandas. These libraries are limited by the capabilities of a single machine, whereas most datasets require parallel processing for timely and accurate results. That is where PySpark comes in. As the name suggests, PySpark is a Python front end for Spark. Under the hood, it uses py4j[5] to directly invoke Java objects from the Python interpreter. As a result, there is an almost one-to-one mapping between the Scala API and its equivalent in Python. As an example, let's reimplement the Scala code from Listing 10-2 in Python. The Python port is shown in Listing 10-3.

As you can see, the code is almost identical to the Scala version. The only two major differences are the lack of a fluent API in Python and the difference in lambda functions.

[5]https://www.py4j.org/.

Listing 10-3. Your First PySpark Application

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
from sys import argv, exit
try: import simplejson as json
except ImportError: import json

if len(argv) != 5:
    print 'Usage: yelp_pyspark.py <appname> <batchInterval> <hostname> <port>'
    exit(-1)

appname = argv[1]
batch_interval = int(argv[2])
hostname = argv[3]
port = int(argv[4])

sc = SparkContext(appName=appname)
ssc = StreamingContext(sc, batch_interval)

records = ssc.socketTextStream(hostname, port)
json_records = records.map(lambda rec: json.loads(rec))
restaurant_records = json_records.filter(lambda rec: 'attributes' in rec and 'Wi-Fi' in rec['attributes'])
wifi_pairs = restaurant_records.map(lambda rec: (rec['attributes']['Wi-Fi'], rec['stars']))
wifi_counts = wifi_pairs.combineByKey(lambda v: (v, 1),
                             lambda x, value: (x[0] + value, x[1] + 1),
                             lambda x, y: (x[0] + y[0], x[1] + y[1]))
avg_stars = wifi_counts.map(lambda (key, (sum_, count)): (key, sum_ / count))
avg_stars.pprint()

ssc.start()
ssc.awaitTermination()
```

You are going to deploy this application to Dataproc for execution using the `gcloud` command-line tool. Copy the code from Listing 10-3 to a file, say, `yelp_pyspark.py`. As before, run the `SocketDriver` on the master node first. Once it is up and running, submit the application to Dataproc using the following on your local machine:

```
gcloud beta dataproc jobs submit pyspark --cluster first-spark-cluster yelp_pyspark.py
first-pyspark-dataproc-app 1 first-spark-cluster-m 9000
```

The `gcloud` command-line tool takes care of transferring the Python code to the cluster and also connecting the output of the Spark console with your terminal.

You can alternatively run this application on a local cluster. You first need to install Py4J by using `pip` and then add PySpark to the `PYTHONPATH`:

```
pip install py4j
export PYTHONPATH=$SPARK_HOME/python/:$PYTHONPATH
```

Once PySpark is set up, submit the application by using the `spark-submit` script:

```
$SPARK_HOME/bin/spark-submit yelp_pyspark.py first-pyspark-dataproc-app 1 <hostname_of_
machine_running_socketdriver> 9000
```

It was that simple to write a Spark Streaming application in Python and execute it on top of a Dataproc cluster. This was just an introduction to both topics—both have much more to offer.

Lambda Architecture

Imagine you are working on an application, which will allow anyone to query the Yelp rating of a business broken down by a given timestamp. The granularity of timestamps can vary from minutes to days depending on the SLA. Let's assume that there is only one type of query: the number of positive or negative ratings for a given business ID. Considering that 30,000 reviews are posted every minute on Yelp and 25,000 pictures are uploaded each day,[6] the queries would be very expensive to compute on the fly. To remedy this, queries need to be precomputed.

In Chapter 1, you briefly touched upon the CAP theorem, which unpins the design of most contemporary distributed systems. Under its 2 out of 3 (consistency, availability, and partition tolerance) rule, users have to explicitly incorporate these trade-offs. Considering that partition tolerance is a given due to the commodity off the shelf hardware-based design of distributed systems, the actual trade-off is between consistency and availability. Going back to your example, if you focus on consistency that means all the queries will always get the correct rating but considering the sheer size of the data, precomputing views will take a very long time leading to bouts of service unavailability. On the other hand, implementing for availability would mean that some queries may not reflect the most recent ratings.

For web scale applications, the choice is obvious: availability. This has the implication that the system will be eventually consistent, that is, when all distributed replicas of data are able to talk to each other they will agree on a state of the system. But achieving consensus is easier said than done. Just like humans, it is non-trivial to get systems to agree on an answer as well. Should the most recent answer be considered? Or the average of all answers? Or the one with the highest cardinality? Clearly this cannot be decided upon by the storage system itself and the buck needs to be passed back to the user application. In turn the application would perform read or write repair in a lazy fashion using various algorithms, such as vector clocks. Unfortunately, these mechanisms are very hard to implement and maintain in user applications.

Most inconsistencies stem from the fact that applications perform append, update, and delete operations. What if you could relax those conditions to only allow append operations, essentially make the data immutable, wherein records are incrementally added to the data store. In this model, an update is implemented by appending a new record that supersedes the previous one by say maintaining an ordering via timestamps. The same goes for deletions: if an event is no longer valid, say that Barack Obama is the President of the US, rather than deleting its record, you can just append another record with the name of the new President. You can achieve eventual consistency by rerunning the query over the entire dataset periodically, that is, the only inconsistency would occur when new data arrives while you are in the precomputation phase.

Let's look at two different ends of the design spectrum for implementing such an application. Batch processing à la Hadoop fits the bill very well. It can process bulk data in a scalable fashion. As more data is appended, you can rerun the MapReduce job on the entire dataset, which in essence is a transaction. In case of failure, missing state can be recovered by re-executing failed tasks. The downside is that the query will become stale eventually. At the opposite end of the spectrum is stream processing, which can potentially bring down the latency of the query to near real time. This comes at a cost, though: the entire state must be maintained in memory, which requires an almost untenable amount of memory. What if you could get the best of both worlds? One such design is sketched out in Figure 10-16.

[6]Craig Smith, "By the Numbers: 50 Amazing Yelp Statistics," *DMR*, April 1, 2016, `http://expandedramblings.com/index.php/yelp-statistics/`.

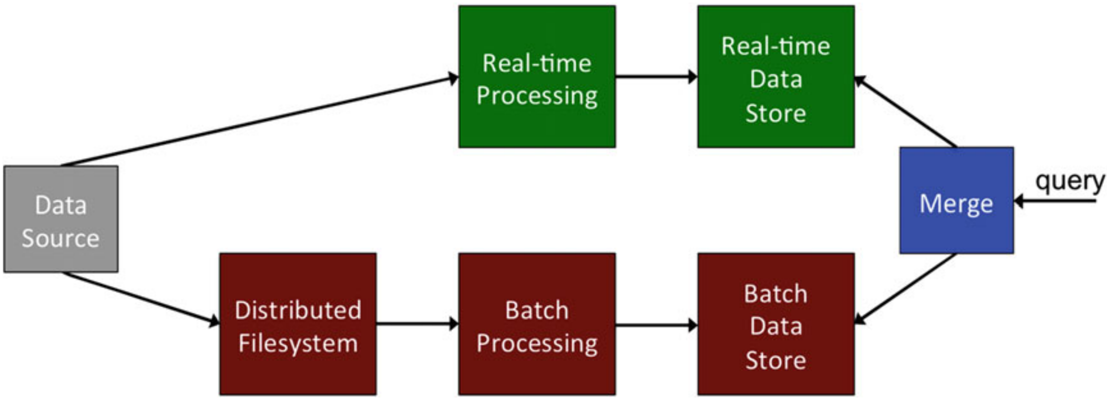


Figure 10-16. *Blending real-time processing with batch processing to implement a data-querying system*

The architecture in Figure 10-16 has three layers: the real-time or *speed layer*, which computes results for a specific number of time units; the *batch layer*, which computes results for the entire dataset (starting at T=0) periodically with newly appended data; and the *serving layer*, which serves queries by merging results from the online and offline views. Every time a new record is generated by the data source (the grey box in the figure), it makes its way to both layers. For the real-time layer, it is instantly consumed, and the result is written to a data store optimized for real-time computation. On the batch layer, the data is first written to an append-only distributed filesystem. A batch-processing job is kicked off periodically to recompute the view over the entire dataset and write it to a batch data store. The real-time layer is refreshed every time the batch job completes. Every time a user query comes in (at extreme right in the figure) the results for a key—the Yelp business ID in this case—is computed on the fly by merging values from the real-time and batch data stores. In essence, the real-time view masks away the latency of the batch layer.

This design is known as the Lambda Architecture.[7] It was conceived by Nathan Marz, the creator of the popular Apache Storm stream-processing system. Let's implement the Lambda Architecture using a combination of Spark Streaming and Google Cloud Platform. Spark is a great system to implement the Lambda Architecture because it provides a unified API and execution engine for both batch and real-time processing. In addition, Spark SQL simplifies the implementation of typical queries, which revolve around aggregations, rollups, and cubes. Finally, the integration of Spark with other Big Data systems, such as message queues, key value stores, and distributed file systems, enables end-to-end applications.

Lambda Architecture using Spark Streaming on Google Cloud Platform

Listing 10-4 provides the code for this implementation. For the real-time layer, you use Spark Streaming in concert with Cloud BigTable. The batch layer, on the other hand, is implemented using BigQuery. The application uses the Yelp reviews dataset to determine the positive and negative ratings of a business ID at different aggregation levels (basically, a SQL rollup operation). The application is ready to be deployed to Dataproc for execution.

Let's walk through the code to understand the specifics.

[7]Nathan Marz, "How to Beat the CAP Theorom," *Thoughts from the Red Planet*, October 13, 2011, `http://nathan-marz.com/blog/how-to-beat-the-cap-theorem.html`.

Listing 10-4. Lambda Architecture Using Spark Streaming, Cloud BigTable, BigQuery, and Dataproc

```
package org.apress.prospark

import org.apache.hadoop.conf.Configuration
import org.apache.hadoop.hbase.HBaseConfiguration
import org.apache.hadoop.hbase.client.Put
import org.apache.hadoop.hbase.mapreduce.TableOutputFormat
import org.apache.hadoop.hbase.util.Bytes
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.rdd.RDD.rddToPairRDDFunctions
import org.apache.spark.streaming.Seconds
import org.apache.spark.streaming.StreamingContext
import org.apache.spark.streaming.dstream.DStream.toPairDStreamFunctions
import org.json4s.DefaultFormats
import org.json4s.jvalue2extractable
import org.json4s.jvalue2monadic
import org.json4s.native.JsonMethods.parse
import org.json4s.string2JsonInput
import com.google.cloud.hadoop.io.bigquery.BigQueryConfiguration
import com.google.gson.JsonObject
import com.google.cloud.hadoop.io.bigquery.BigQueryOutputFormat
import org.apache.hadoop.io.Text

object LambdaDataprocApp {

  def main(args: Array[String]) {
    if (args.length != 14) {
      System.err.println(
        "Usage: LambdaDataprocApp <appname> <batchInterval> <hostname> <port> <projectid>"
          + " <zone> <cluster> <tableName> <columnFamilyName> <columnName> <checkpointDir>"
          + " <sessionLength> <bqDatasetId> <bqTableId>")
      System.exit(1)
    }
    val Seq(appName, batchInterval, hostname, port, projectId, zone, clusterId,
      tableName, columnFamilyName, columnName, checkpointDir, sessionLength,
      bqDatasetId, bqTableId) = args.toSeq

    val conf = new SparkConf()
      .setAppName(appName)
      .setJars(SparkContext.jarOfClass(this.getClass).toSeq)

    val ssc = new StreamingContext(conf, Seconds(batchInterval.toInt))
    ssc.checkpoint(checkpointDir)

    val statefulCount = (values: Seq[(Int, Long)], state: Option[(Int, Long)]) => {
      val prevState = state.getOrElse(0, System.currentTimeMillis())
      if ((System.currentTimeMillis() - prevState._2) > sessionLength.toLong) {
        None
      } else {
        Some(values.map(v => v._1).sum + prevState._1, values.map(v => v._2).max)
      }
    }

    val ratings = ssc.socketTextStream(hostname, port.toInt)
      .map(r => {
        implicit val formats = DefaultFormats
        parse(r)
      })
      .map(jvalue => {
        implicit val formats = DefaultFormats
        ((jvalue \ "business_id").extract[String], (jvalue \ "date").extract[String],
        (jvalue \ "stars").extract[Int])
      })
      .map(rec => (rec._1, rec._2, if (rec._3 > 3) "good" else "bad"))

    ratings.map(rec => (rec.productIterator.mkString(":"), (1, System.
    currentTimeMillis())))
      .updateStateByKey(statefulCount)
      .foreachRDD(rdd => {
        val hbaseConf = HBaseConfiguration.create()
        hbaseConf.set("hbase.client.connection.impl", "com.google.cloud.bigtable.
        hbase1_1.BigtableConnection")
        hbaseConf.set("google.bigtable.project.id", projectId)
        hbaseConf.set("google.bigtable.zone.name", zone)
        hbaseConf.set("google.bigtable.cluster.name", clusterId)
        hbaseConf.set(TableOutputFormat.OUTPUT_TABLE, tableName)
        val jobConf = new Configuration(hbaseConf)
        jobConf.set("mapreduce.job.outputformat.class", classOf[TableOutputFormat[Te
        xt]].getName)
        rdd.mapPartitions(it => {
          it.map(rec => {
            val put = new Put(rec._1.getBytes)
            put.addColumn(columnFamilyName.getBytes, columnName.getBytes, Bytes.
            toBytes(rec._2._1))
            (rec._1, put)
          })
        }).saveAsNewAPIHadoopDataset(jobConf)
      })

    ratings.foreachRDD(rdd => {
      val bqConf = new Configuration()

      val bqTableSchema =
        "[{'name': 'timestamp', 'type': 'STRING'}, {'name': 'business_id', 'type':
        'STRING'}, {'name': 'rating', 'type': 'STRING'}]"
      BigQueryConfiguration.configureBigQueryOutput(
        bqConf, projectId, bqDatasetId, bqTableId, bqTableSchema)
      bqConf.set("mapreduce.job.outputformat.class",
        classOf[BigQueryOutputFormat[_, _]].getName)
      rdd.mapPartitions(it => {
```

```
95.          it.map(rec => (null, {
96.            val j = new JsonObject()
97.            j.addProperty("timestamp", rec._1)
98.            j.addProperty("business_id", rec._2)
99.            j.addProperty("rating", rec._3)
100.           j
101.         }))
102.       }).saveAsNewAPIHadoopDataset(bqConf)
103.     })
104.
105.     ssc.start()
106.     ssc.awaitTermination()
107.
108.   }
109.
110. }
```

Similar to Listing 10-2, the application relies on the SocketDriver for data production. Therefore, before running the application, feed the SocketDriver the review dataset (yelp_academic_dataset_review.json). The structure of the JSON object in the review dataset is outlined in Listing 10-5.

Listing 10-5. JSON Blueprint of the Review Dataset

```
{
   "votes": {
      "funny": <count>,
      "useful": <count>,
      "cool": <count>
   },
   "user_id": "<anonymized_id>",
   "review_id": "<anonymized_id>",
   "stars": <count>,
   "date": "<YYYY:MM:DD>",
   "text": "<free form text>",
   "type": "review",
   "business_id": "<anonymized_id>"
}
```

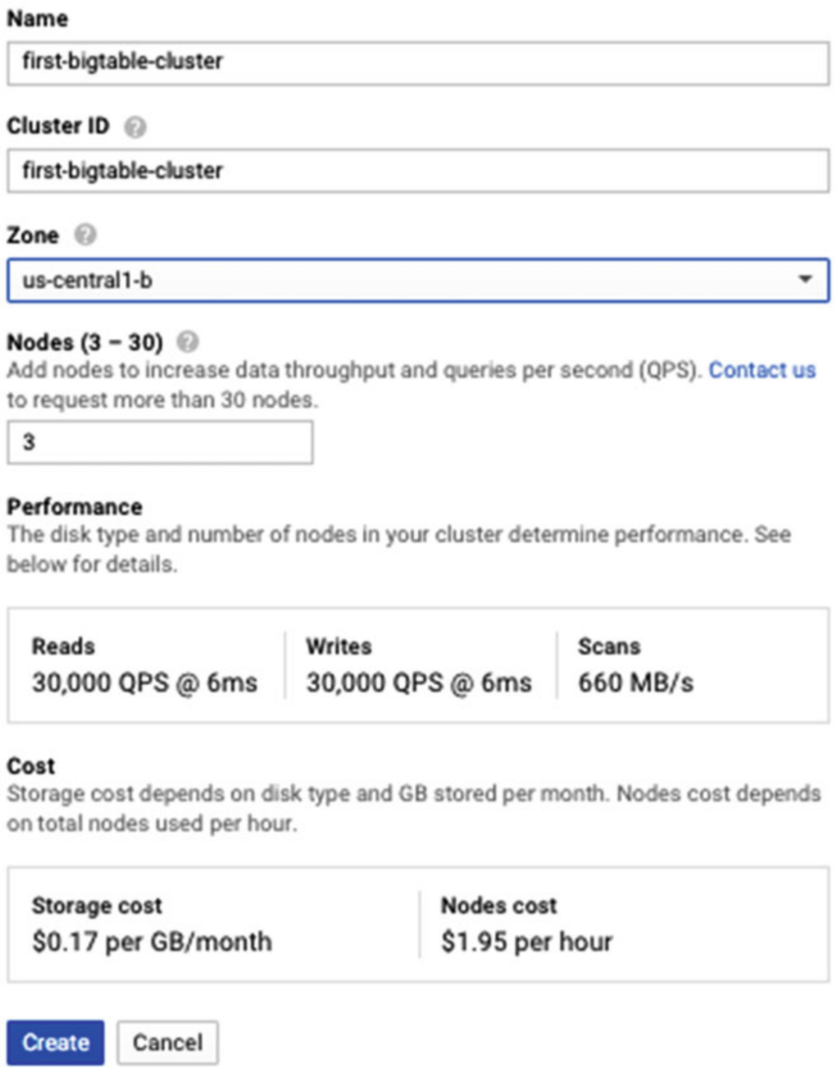
The application reads this data from a socket (line 54) and projects the business_id, date, and stars fields from the JSON object (lines 55–62). You then categorize records as *good* if the number of stars is greater than three or *bad* otherwise (line 63). This resulting stream is divided into two flows: one for real-time processing and the other for batch processing.

The real-time pipeline puts its data in Cloud BigTable, which is a fully managed, cloud-based version of BigTable from Google with an HBase-compatible API. You model the layout such that the row key is a concatenation of the business ID, timestamp, and rating category (good or bad) (line 65). With each key, you also need to maintain the count across batches, for which you use an updateStateByKey operation. The real-time pipeline needs to be flushed every time the batch view has been updated in BigQuery. Specifically, you need to remove keys from the updateStateByKey. So, you associate a session-length value with each record, which is simply the time before you expire a key. To implement this, you insert the current system time into each record (line 65) and remove the key if it has exceeded the session length (lines 47–49) in the statefulCount function. Ideally, this session-length value should be equal to the latency of the batch layer. For instance, if the batch job is kicked off every hour, the value of the session length should be 3600.

CREATING A CLOUD BIGTABLE CLUSTER

Log in to the Google Cloud Platform dashboard, and, from the Storage section of the Products & Services menu, select Bigtable. Click Create Cluster in the next screen.

Use Figure 10-17 as reference to set up your cluster.



Name

first-bigtable-cluster

Cluster ID

first-bigtable-cluster

Zone

us-central1-b

Nodes (3 – 30)

Add nodes to increase data throughput and queries per second (QPS). Contact us to request more than 30 nodes.

3

Performance

The disk type and number of nodes in your cluster determine performance. See below for details.

Reads	Writes	Scans
30,000 QPS @ 6ms	30,000 QPS @ 6ms	660 MB/s

Cost

Storage cost depends on disk type and GB stored per month. Nodes cost depends on total nodes used per hour.

Storage cost	Nodes cost
$0.17 per GB/month	$1.95 per hour

Figure 10-17. *Creating a Cloud BigTable cluster*

Go to the API Manager in the Products & Services menu, and enable the API for Cloud Bigtable.

You can also configure the HBase shell to talk to the Bigtable deployment.[8]

The rest of the real-time layer code writes the row keys and values to Bigtable using the HBase API. Notice that the code is almost identical to Listing 6-12 in Chapter 6. The only difference is in terms of additional configuration parameters for Bigtable, such as the project ID, cluster ID, and cluster zone (lines 69–72).

The batch part of the application relies on BigQuery, which is another fully managed storage service from Google that allows SQL queries against append-only tables. It achieves performance and scalability by using aggregation trees and columnar storage. The unit of creation in BigQuery is a dataset, which can contain many tables. The batch layer in Listing 10-4 (lines 88–103) simply writes each record verbatim to BigQuery. The BigQuery execution layer in SQL can then be used to periodically re-create precomputed views.

The BigQuery connector for Spark treats BigQuery as just another Hadoop-compatible storage system. This means you can use `saveAsNewAPIHadoopDataset` to write to BigQuery. The only thing you need to do in the `foreachRDD` clause is to provide a schema for the BigQuery table (line 89) and configuration information (lines 90–93). You follow a simple schema with a column each for business ID, timestamp, and rating type. For BigQuery, each record needs to be converted to JSON, which is what you do in lines 96–99. Voilà—your Lambda Architecture application is ready for execution.

To execute the application, first add the dependencies from Listing 10-6 to your `sbt` build definition file. Make sure you have enabled the APIs for both BigQuery and Bigtable in the GCP console. In addition, you need to create a table with a single column family in Bigtable. You can do so by executing the following in the HBase shell:

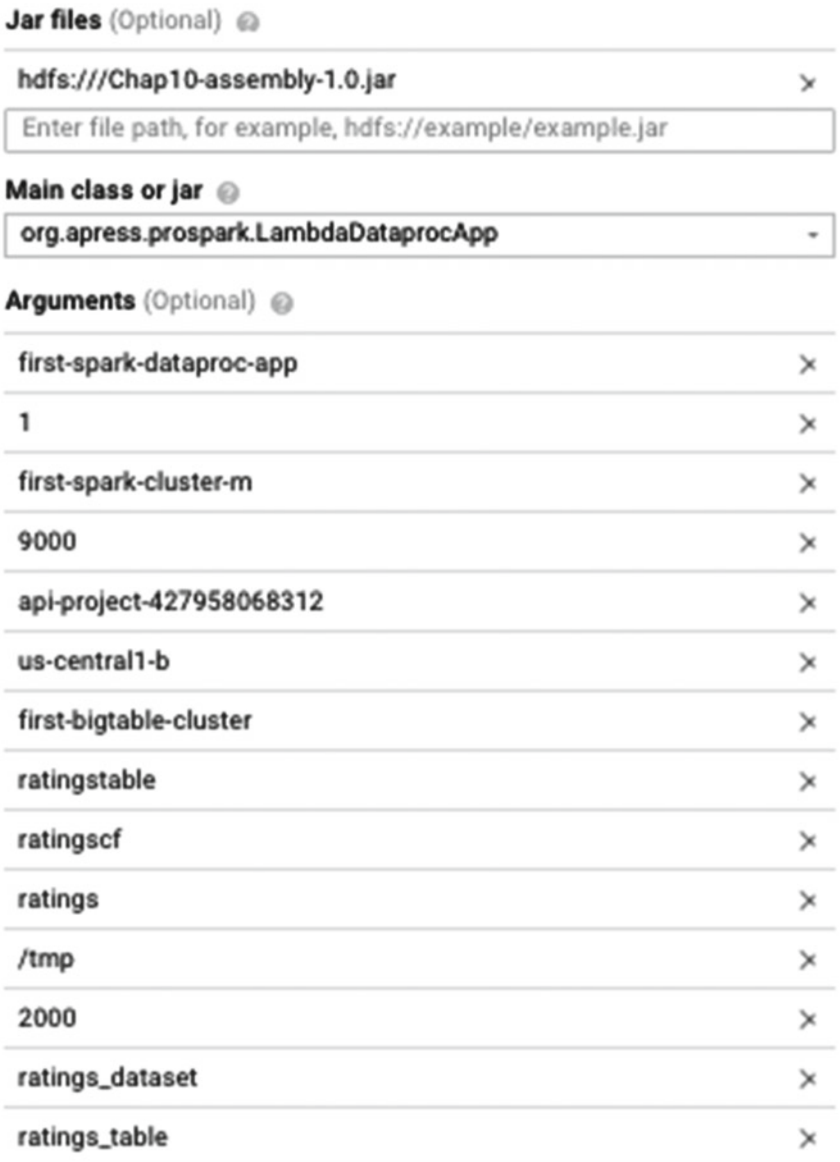
```
create 'ratingstable', 'ratingscf'
```

Listing 10-6. Dependencies Required for the Lambda Architecture Application

```
libraryDependencies += "org.json4s" %% "json4s-native" % "3.2.10"
libraryDependencies += "com.google.cloud.bigtable" % "bigtable-hbase-1.1" % "0.2.3"
exclude("com.google.guava", "guava")
libraryDependencies += "org.apache.hbase" % "hbase-server" % "1.1.2"
libraryDependencies += "org.apache.hbase" % "hbase-common" % "1.1.2"
libraryDependencies += "com.google.guava" % "guava" % "16.0"
libraryDependencies += "org.mortbay.jetty.alpn" % "alpn-boot" % "8.1.6.v20151105"
libraryDependencies += "com.google.cloud.bigdataoss" % "bigquery-connector" %
"0.7.4-hadoop2"
```

As before, create a JAR for the application, copy it to the HDFS deployment on your Dataproc cluster, and run it from the Dataproc UI. Refer to Figure 10-18 for command-line parameters that need to be passed to the application.

[8]https://cloud.google.com/bigtable/docs/installing-hbase-shell.


Jar files (Optional)

hdfs:///Chap10-assembly-1.0.jar

Enter file path, for example, hdfs://example/example.jar

Main class or jar

org.apress.prospark.LambdaDataprocApp

Arguments (Optional)

first-spark-dataproc-app
1
first-spark-cluster-m
9000
api-project-427958068312
us-central1-b
first-bigtable-cluster
ratingstable
ratingscf
ratings
/tmp
2000
ratings_dataset
ratings_table

Figure 10-18. *Arguments required by the Lambda Architecture application on top of Dataproc*

Once the application has started executing, you can run the query from Listing 10-7 periodically to create rollup values.

Listing 10-7. BigQuery SQL Query to Calculate Rollups for the Lambda Architecture Application

```
SELECT
  timestamp,
  business_id,
  rating,
  COUNT(1) AS COUNT
FROM
  [<your_dataset_id>.<your_table_id>]
GROUP BY
  ROLLUP(timestamp, business_id, rating)
```

And just like that, you have implemented a Lambda Architecture application to realize a highly available and eventually consistent query-serving system. Note that this application uses a common pipeline to generate real-time views as well as to route data to the batch layer. In a real-world deployment, these would need to be separated into two self-contained applications for fault-tolerance and performance. For instance, instead of publishing data to a socket, it can be published to a Kafka topic, and then these two separate applications can consume the topic via individual subscriptions.

Streaming Graph Analytics

Any dataset with relationships between entities can be modeled as a graph, and its analysis can be mapped onto graph problems. The Web is a graph of machines connected via the Internet, Facebook is a graph of users connected via friends, and locations on Google Maps lend themselves to a graph with connections provided by modes of transportation. The sheer size of some of these graphs negates the use of single-machine libraries. At the other end of the spectrum, standard Big Data systems such as Hadoop or Spark are too low level to capture the expressiveness required for graph processing. GraphX was designed to fill this gap by enabling graph-parallel computation on top of Spark. Like the rest of the book, this last topic is illustrated using an example.

Listing 10-8. JSON Structure of the Yelp User Dataset

```
{
   "yelping_since":"<YYYY-MM>",
   "votes":{
      "funny":<count>,
      "useful":<count>,
      "cool":<count>
   },
   "review_count":<count>,
   "name":"<first_name_of_user>",
   "user_id":"<anonymized_id>",
    "friends":[
       "<list_of_user_ids>"
    ],
    "fans":<count>,
    "average_stars":<average_star_count>,
    "type":"user",
    "compliments":{
       "photos":<count>,
       "hot":<count>,
       "cool":<count>,
       "plain":<count>
    },
    "elite":[
         <list_of_years>
    ]
 }
```

Finding influential users in social networks is an interesting topic due to its implications for online advertising. For instance, posts shared by influential users are widely disseminated in comparison to those of ordinary users. The Yelp dataset also contains friendship information for users (JSON attributes in Listing 10-8). This can be used to build a graph of relationships on Yelp. A number of algorithms such as PageRank and HITS can then be used to pinpoint influential users. The PageRank-based approach using GraphX is shown in Listing 10-9.

Listing 10-9. First Streaming GraphX Application

```
package org.apress.prospark

import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.graphx.Edge
import org.apache.spark.graphx.Graph
import org.apache.spark.graphx.Graph.graphToGraphOps
import org.apache.spark.streaming.Seconds
import org.apache.spark.streaming.StreamingContext
import org.json4s.DefaultFormats
import org.json4s.jvalue2extractable
import org.json4s.jvalue2monadic
import org.json4s.native.JsonMethods.parse
import org.json4s.string2JsonInput

 object UserRankApp {

   def main(args: Array[String]) {
     if (args.length != 4) {
       System.err.println(
         "Usage: UserRankApp <appname> <batchInterval> <hostname> <port>")
       System.exit(1)
     }
     val Seq(appName, batchInterval, hostname, port) = args.toSeq

     val conf = new SparkConf()
       .setAppName(appName)
       .setJars(SparkContext.jarOfClass(this.getClass).toSeq)

     val ssc = new StreamingContext(conf, Seconds(batchInterval.toInt))

     ssc.socketTextStream(hostname, port.toInt)
       .map(r => {
         implicit val formats = DefaultFormats
         parse(r)
36.         })
37.         .foreachRDD(rdd => {
38.           val edges = rdd.map(jvalue => {
39.             implicit val formats = DefaultFormats
40.             ((jvalue \ "user_id").extract[String], (jvalue \ "friends").
                extract[Array[String]])
41.           })
42.          .flatMap(r => r._2.map(f => Edge(r._1.hashCode.toLong, f.hashCode.toLong, 1.0)))
43.
44.           val vertices = rdd.map(jvalue => {
45.             implicit val formats = DefaultFormats
46.             ((jvalue \ "user_id").extract[String])
47.           })
48.             .map(r => (r.hashCode.toLong, r))
49.
50.           val tolerance = 0.0001
51.           val graph = Graph(vertices, edges, "defaultUser")
52.             .subgraph(vpred = (id, idStr) => idStr != "defaultUser")
53.           val pr = graph.pageRank(tolerance).cache
54.
55.           graph.outerJoinVertices(pr.vertices) {
56.             (userId, attrs, rank) => (rank.getOrElse(0.0).asInstanceOf[Number].
                doubleValue, attrs)
57.           }.vertices.top(10) {
58.             Ordering.by(_._2._1)
59.           }.foreach(rec => println("User id: %s, Rank: %f".format(rec._2._2,
              rec._2._1)))
60.         })
61.
62.       ssc.start()
63.       ssc.awaitTermination()
64.
65.     }
66.
67.   }
```

After reading the data from the socket and converting each record to JSON (lines 32–36), you implement the core logic of the application in a `foreachRDD` transform. GraphX does not have native support for online graph analysis, so using this approach means the state of analysis is confined to individual batches. This per-batch implementation is useful for a large class of applications, which only rely on the current state of the network.

As with other graph libraries, you need to create separate vertex and edge objects, which must be combined to generate the graph. Specifically, you need to create edge and vertex RDDs to spawn a graph object. For the edge records, you create `user_id:user_id` pairs from each JSON object, where each pair represents a friendship connection between users (lines 38–42). GraphX requires vertex IDs to be in the form of `long` values. Therefore, the application takes the hash code of user IDs and uses it for this purpose. In addition, you assign a weight of 1 to each friendship edge (line 42). Similarly, for the vertex RDDs, you emit the user ID hash as a `long` and its value in `string` form (lines 44–48). Both RDDs (edge and vertex) can then be used to create a graph object (line 51) with `defaultUser` as the default user ID for missing values. Because these default values are meaningless from your perspective (the influence of a `defaultUser` is useless), the `subgraph` method is used to filter by edges and vertices (line 52).

GraphX out of the box contains an implementation of PageRank, which you use on the graph with a convergence tolerance of 0.0001 (line 53). This returns vertex ID:rank pairs. These need to be joined with the original graph to get back the original user ID (lines 55–57). You then get the top 10 most influential users ordered by rank by iterating over the vertices of the graph (line 57-59). Finally, you print these values to standard output.

This example should have whetted your appetite for graph processing the Spark way. This was just the tip of the iceberg, though; GraphX is far richer in terms of algorithms, features, and transforms than were presented here. Refer to the official documentation for a deeper dive.

Summary

The days of on-premises data centers are numbered. Fully managed systems deployments are all the rage due to their simple cost model, elasticity, scalability, and out-of-the-box integration with the wider Big Data ecosystem. A number of such solutions also exist for Spark, and this chapter explored Dataproc. Python aficionados also got their hands dirty with the Spark Python API. To support low latency and highly available data querying, another topic of discussion in this chapter was the Lambda Architecture. Using a combination of Spark Streaming, Cloud Bigtable, and BigQuery, the nitty-gritty of the Lambda Architecture was laid bare. To wrap up the chapter as well as the book, graph processing using GraphX in tandem with Spark Streaming was introduced.

That's all, folks. With that, you come to the end of this "sparkling" journey!

Index