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

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.