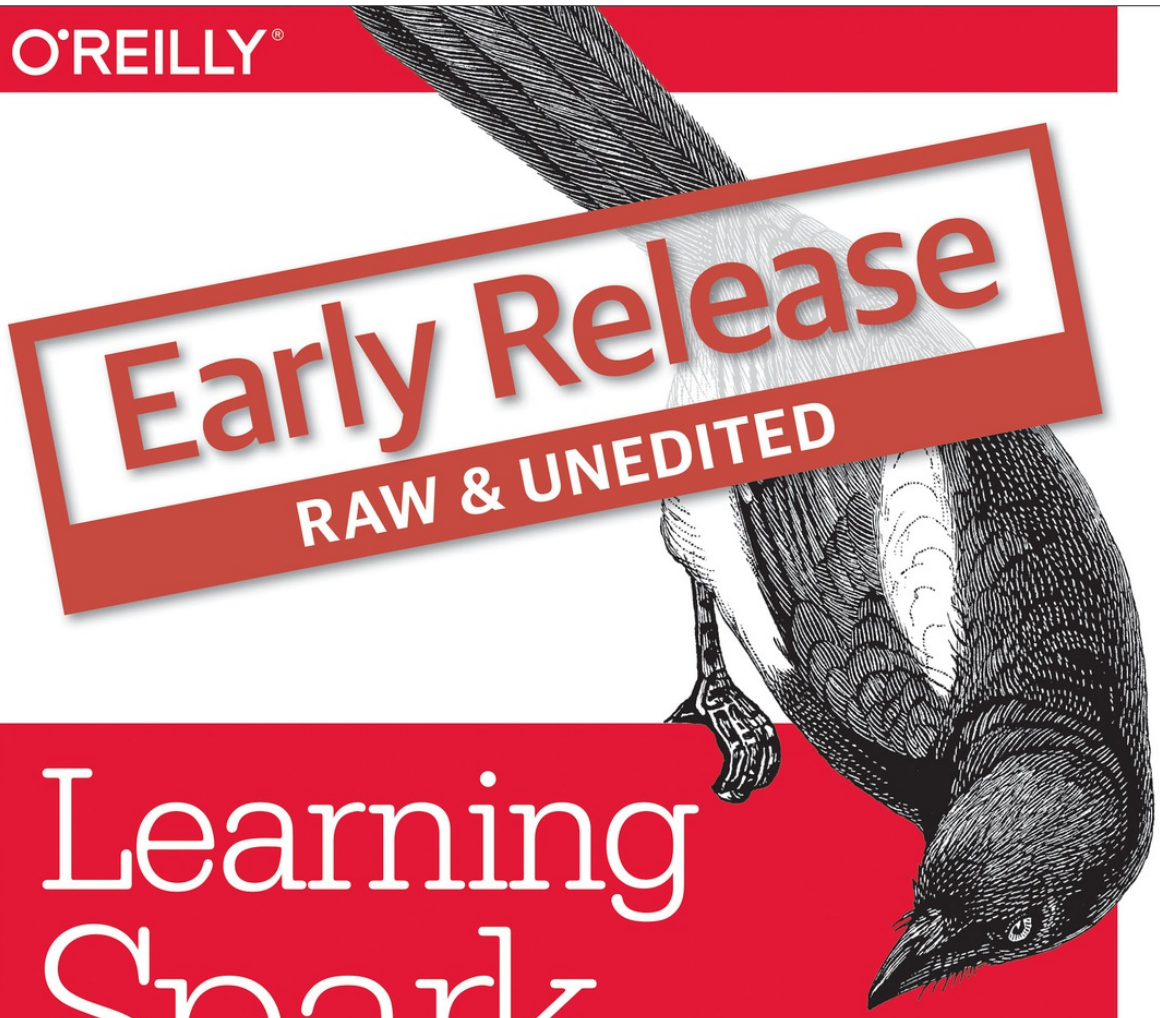
O'REILLY®



Early Release

RAW & UNEDITED

Learning Spark Streaming

BEST PRACTICES FOR SCALING AND OPTIMIZING APACHE SPARK

François Garillot & Gerard Maas

Learning Spark Streaming

First Edition

Francois Garillot and Gerard Maas

Learning Spark Streaming

by Francois Garillot and Gerard Maas

Printed in the United States of America.

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

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

- Editor: Shannon Cutt
- Production Editor: FILL IN PRODUCTION EDITOR
- Copyeditor: FILL IN COPYEDITOR
- Proofreader: FILL IN PROOFREADER
- Indexer: FILL IN INDEXER
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Rebecca Demarest

- December 2017: First Edition

Revision History for the First Edition

- 2017-06-19 First Early Release
- 2017-08-28: Second Early Release

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

978-1-491-94424-0

[FILL IN]

Chapter 1. Introducing Spark Streaming

Large-scale data analytics and Apache Spark

This book focuses on Spark Streaming, a streaming module for the Apache Spark computation engine, used in distributed data analytics. Spark's Streaming capabilities came essentially as an addition throughout the thesis of its first and main developer, Tathagata Das.

The concepts that Spark Streaming embodies were not all new at the time of its implementation, and it carries a rich legacy of learnings on how to expose an easy way to do distributed programming at a massive scale. Chief among these heirlooms is MapReduce, the Google-created work born at Google, and that gave rise to Hadoop - and which concepts we will sketch in a few pages.

Users familiar with MapReduce may want to skip to "More than MapReduce : how the model came about and how Spark extends it.". For others, we will introduce here the main tenets of Spark Streaming: an expressive, MapReduce-inspired programming API (Spark) and its resiliency primitives, extended by a micro-batching approach to streaming (Spark Streaming).

Note

MapReduce, while an important precursor in the domain of distributed data analysis, and while sharing some base concepts with Spark, is both a framework an a major implementation (Hadoop) which are outside the scope of this book.

Its open-source API and implementation language is Java, a programming language that this book does not readily cover. Moreover, the somewhat verbose nature of the API would force our examples to be rather long. Hence, we have made the choice to be faithful to the concepts of MapReduce and not to the code, and to show examples in Scala pseudo-code.

We hope that the Scala experts and the Hadoop practitioners will readily grasp the mapping from our examples to the exact Java code. All will enjoy succinct and simple examples along the way.

Finally, MapReduce sometimes has a slightly different language to describe very similar concepts (e.g. slaves for what Spark would call executors). Since this book is about Spark Streaming, we will name the MapReduce concepts using the Spark terminology when no confusion is possible.

More than MapReduce : how the model came about and how Spark extends it.

The history of programming for distributed systems experienced a notable event in February of 2003. Jeff Dean and Sanjay Gemawhat, after going through a couple iterations of rewriting Google's crawling and indexing systems, started noticing some operations that they could expose through a common interface. In their minds started to crystallize the operations for what was to become MapReduce, a system for distributed processing on large clusters at Google.

What is now clear (???, ???), is that one major driver for the innovations behind MapReduce was scalability, and in particular fault tolerance.

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

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

It was not the first — nor would it be the last — to tackle distributed computation. But it had going for it a number of points that made this approach very attractive:

- a significant offloading of the difficulty of distributing a program, from the shoulders of the programmer to those of the library designer
- a very high expressivity
- a simple API

Conceptually, many other APIs, such as the message-passing interface (MPI), or actors, also alleviated the complexity involved in running applications on multiple machines. But they still required the developer to orchestrate the coordination within the participating machines.

In many cases, that level of fine-grained control is useful or sought. But for data analytics, there is an advantage in just requiring the programmer to be an expert at extracting insight out of data through computation without also requiring her to be an expert at dealing with complicated parallel processing models.

This simple model therefore builds on very few elements:

A Fault-tolerant MapReduce cluster

A MapReduce cluster involves a master and several slaves, with the master scheduling jobs — i.e. stages of the user-specified computation — on the slaves. The user talks to the cluster through the intermediary of the master, usually from a personal workstation. This later machine, on which this program runs from, is the driver.

The architecture layout features in the following figure. Note that this organization has the potential to cater for a multi-tenant system from the first step: as multi-machine clusters are expensive, they invite sharing from the beginning — though the multi-tenant facilities in systems inspired by the MapReduce model underwent significant evolution.

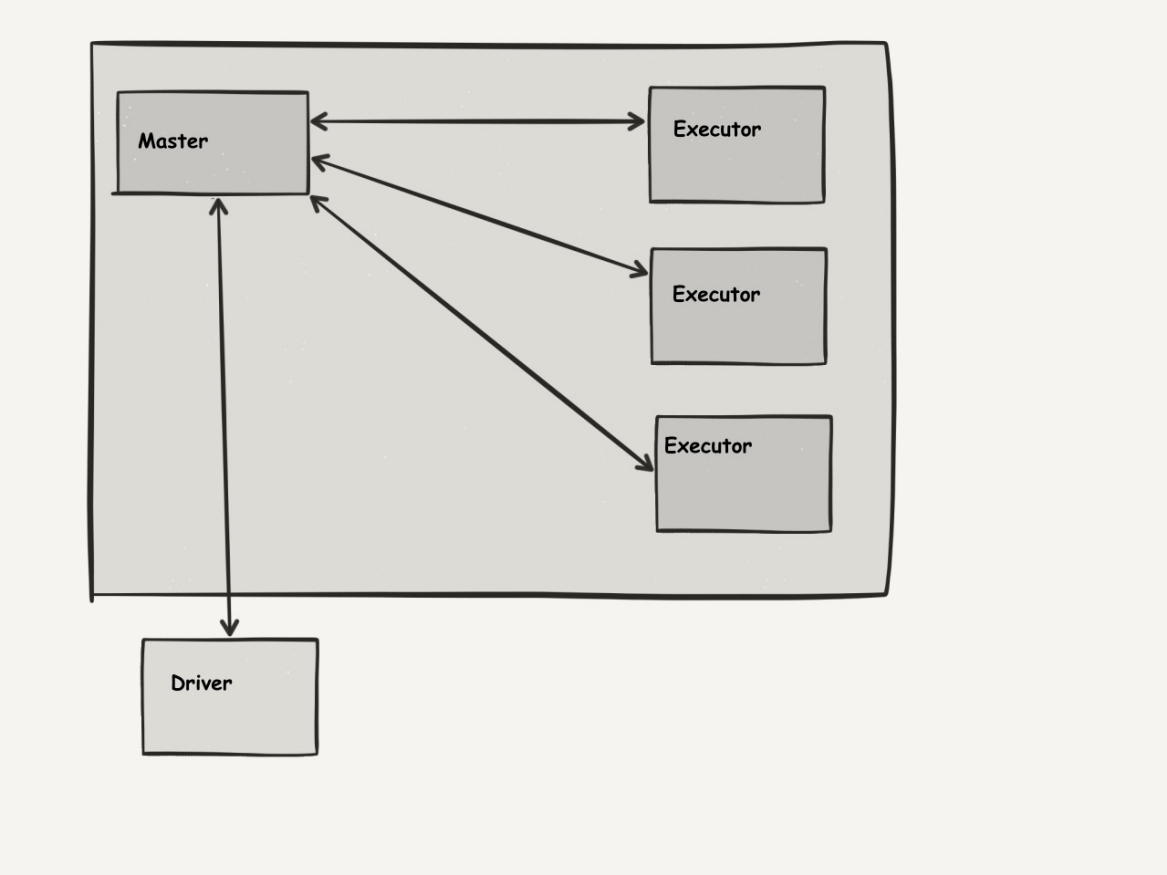


Figure 1-1. MapReduce architecture in a cluster

Finally, note the advantage of this architecture is a guarantee of fault tolerance, with an automated restart in case of a failure. Indeed, the responsibility for of not losing any data falls on HDFS. For any data-carrying executor that may crash, the data it contained when it does crash has a copy on another machine. As a result, it is enough to re-launch the task that was running on this executor. Since the master keeps track of that task, that does not pose any particular problem other than rescheduling.

You may notice that this leaves the question of what to do when the master itself crashes. That did create a problem for the first version of open-source implementations of the MapReduce model, which did not have an answer. After a few releases, Masters stopped being single points of failure, being co-maintained with a hot-swappable clone: the cluster and afferent systems maintain two copies of the master, one ready to take over in case the first one would crash. In the first years of existence of Hadoop, some users — with runtime demand perhaps lesser than those of the web giants — felt this was a somewhat manageable liability.

A distributed file system

A few years before MapReduce, Google had devised the Google File System, a distributed system intended to fit the scale where Google stores data. More interestingly, it offered storage of data in a distributed way with fault tolerance, since each block of data had a fallback copy. Given that writing data to GFS was taking place in a distributed fashion, data distribution occurred very early, on the instant of saving this data.

Two higher-order functions

MapReduce consists of two transformations applied to data in the form of a distributed collection. That distributed collection can be any data chunked in the distributed file system where MapReduce executes. For the sake of simplicity, it's easier to think of that collection as a distributed array.

Map

> The map operation takes as an argument a function to be applied to every element of the collection. The collection's elements are read in a distributed manner, through the distributed file system, one chunk per executor machine. Then, all the elements of the collection that reside in the local chunk see the function applied to them, and the executor emits the result of that application, if any.
> Defining `map` as a new operation on Scala lists (to simplify the reasoning with respect to distribution) is shown on the following page.

Reduce

> The reduce operation takes two arguments: One is a neutral element, which is what the *reduce* operation would return if passed an empty collection. The other is an aggregation operation, that takes the current value of an aggregate, a new element of the collection, and lumps them into a new aggregate.
> Again, as defined in Scala, on lists, it is simple to specify and shown on the next page.

```
1 def map[T](l: List[T])(f: T -> U): List[U] =
2   for (i <- l) yield f(i)
```

```
def reduce[T](l: List[T])(neutral: T)(op: (T,T) => T) : T =
  l match {
    case Nil => neutral
    case h :: t =>  reduce(t)(op(neutral, h))(op)
  }
```

Note

One significant detail is that `Map`, just as `Reduce` and other constructs in the MapReduce world, operates on key-value pairs rather than just "*Lists*" of elements.

Those two higher-order functions are powerful enough to express every operation one would want to do on the types that would naturally constitute a distributed collection. This statement reflects a theorem, which is that *catamorphisms induced by a strict function on any bifuctor defined on a free type admits a MapReduce factorization* (???).

In layman's terms, it means that every computation:

- definable on a collection that could contain any type of information (i.e. such as arrays, which nearly every programming language defined as able to contain any kind of type)

- and which proceeds by inspecting all the elements of the collection in an order implied by the structure of the collection is definable in the MapReduce model.

Note that in no way does this imply that you would **want** to define that computation using the MapReduce model. When a computer scientist tells MapReduce is an expressive model, she makes a parallel statement to the one she might make by saying that a given language is Turing-complete, invoking the Church-Turing thesis : a language capable of encoding the very crude ribbon-writing binary operations of a Turing machine is — according to Church-Turing — expressive enough to express any computable function. But it would be foolish to want to use a Turing machine directly to compute anything significant, just as it is sometimes unwise to shoehorn some computation in the model offered by MapReduce — and we'll show some examples of these suboptimal cases later.

Let's now see the model in a little more applied fashion, with the classic example of the MapReduce world of counting occurrences of words, in the "normal" fashion, that is, as defined on key-value pairs. Assume `l` is a list of words, given as `String` elements in Scala. This is how the operation of counting words would go:

```
l.map{ (s) =>
  s.split(" ").map((s) => (s, 1))
}.reduceByKey(0){
  (count: Int, partialCount: Int) => count + partialcount
}
```

How would that operation execute on a cluster ? A keystone argument about the distribution of operations in a data-centric computation is that data is usually much more hefty than the description of the operation that is to be performed on it. Hence, we want to bring the computation to where the data is, instead of the classical way around. This is touted as *data locality* and is an important feature in the MapReduce infrastructure as it enables the distribution of tasks to the right executors where the data can be processed locally.

In classical data-oriented systems, databases are the usual data containers. They implement efficient access methods to single units of data or small datasets: e.g. lookup the user's preferences in a web application or select last's month transactions in an accounting application. The data is requested to the database, typically in the form of a query, and sent back to the client, who further operates on it to apply the necessary business-logic for the use case at hand.

In contrast, in a MapReduce application, we want to apply the business logic to a large dataset. Instead of requesting the data from the client application, we want to send our computation to those nodes in the cluster where our target data is located.

What is being sent across the network to the executors is the operation. More specifically, what is being sent is executable code, containing two elements:

- the function that is to be applied to the data.
- an pointer to the data that this function should take as argument.

From that pointer, the cluster can deduce which machine to send this to. It's the task of the distributed file system manager, which knows on which machine each chunk of data is — or at worse how to retrieve it. This does not mean that data transfer is prohibited throughout the

network. But any data transfer avoided is time saved. When operations are scheduled to be executed on the machine that already stores the data, we say we have achieved *data locality*.

A key property of the `map` operation is that is doesn't particularly care about which element of the collection it operates on, since the same operation is performed on each element. So the *function* part of the closure is always the same. Thanks to the distributed file system, the data is spread more or less evenly across the cluster. This means that executors most likely receive offsets for whatever data they already have stored locally.

For the `reduce` operation, however, things are not so simple. The most elementary parts of the computation have to go, at execution, through one processor and one processor only. So that in the end, the values that have to be composed to create this final value have to be moved locally on the same machine.

This creates intense traffic on the network for every machine, as a chunk of a distributed collection has to be sent to another machine having a different chunk so that those chunks can be composed into a final result.

Having an efficient `reduce` execution is therefore predicated on a few optimizations and conditions that we will detail below.

Optimizations in a `reduce` operation

Associativity : a necessary condition.

We have seen how to use reduce to compute the sum a simple word count above. Summing integers is both an associative and commutative operation, meaning respectively:

Equation 1-1. Associativity

$$(x + y) + z = x + (y + z)$$

Equation 1-2. Commutativity

$$x + y = y + x$$

In the context of MapReduce, associativity is the most important operation of the two. Assume that, for the sake of simplicity, each of the values that you have to sum is on a single machine. If your operation is associative, it means you can tell executors to pair the values they need to share and sums to create intermediate sums in parallel. Then all you need to do is to sum the intermediate results. You need to proceed in sequence, however, pairing the machine containing an integer with that containing either the next value in the sum, or an intermediate sum (itself computed in sequence) obtained from it. That ordering is represented in below.

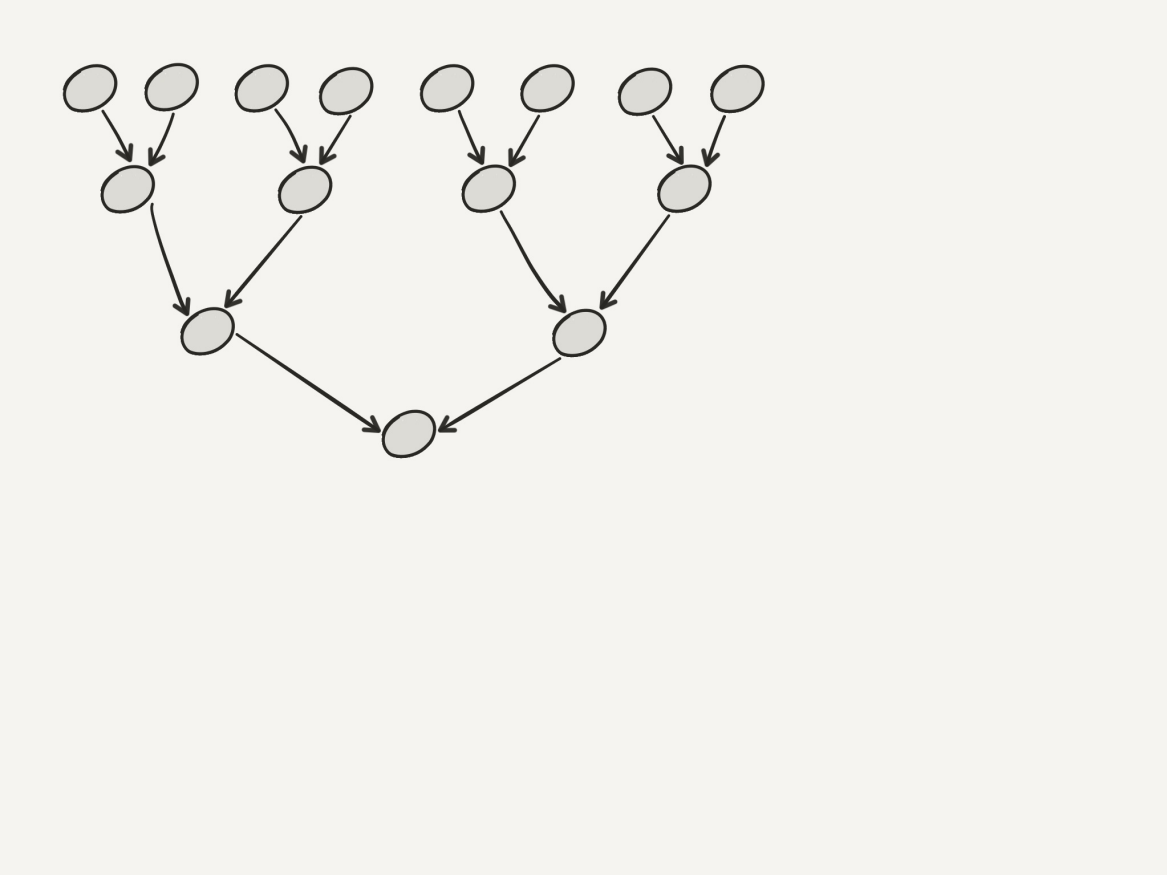


Figure 1-2. Associative Operation in parallel

When an operation is commutative, you can pair the machines in any way, irrespective of their place in the process.

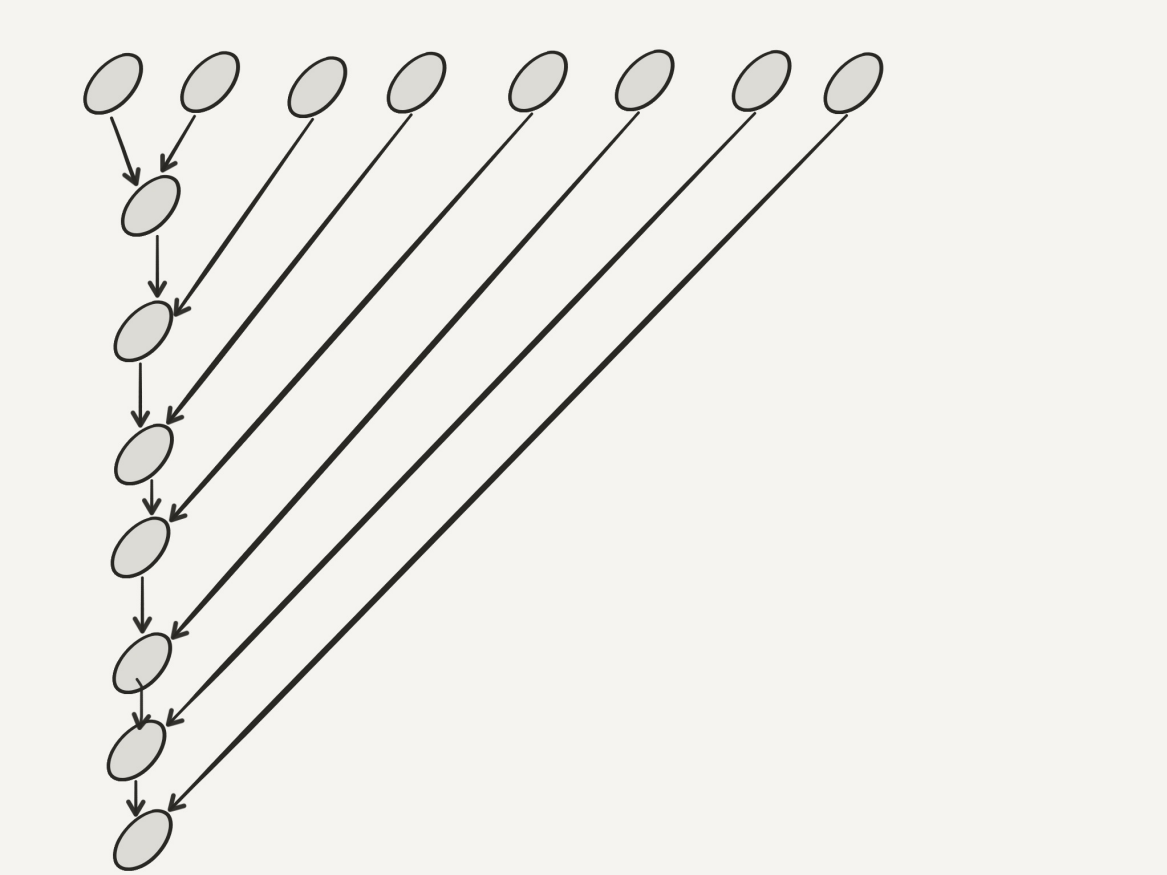


Figure 1-3. Non-Associative Operation in parallel

MapReduce will not spontaneously optimize a reduction for an associative operation, but it will assume your operation is associative, since it is a prerequisite to distributing your operation efficiently across a cluster.

It is always useful to keep in mind which exact properties an operation exhibits before committing to its implementation on a cluster. To this effect, we have enumerated a few examples in the following table.

operation	associative	commutative
sum	yes	yes
string concatenation	yes	no
set union	yes	yes
set difference	no	no

Shuffling

The pairing of executors based on the split of data they can access creates a number of intermediary results, which means these temporary, intermediary results have to be sent across the network.

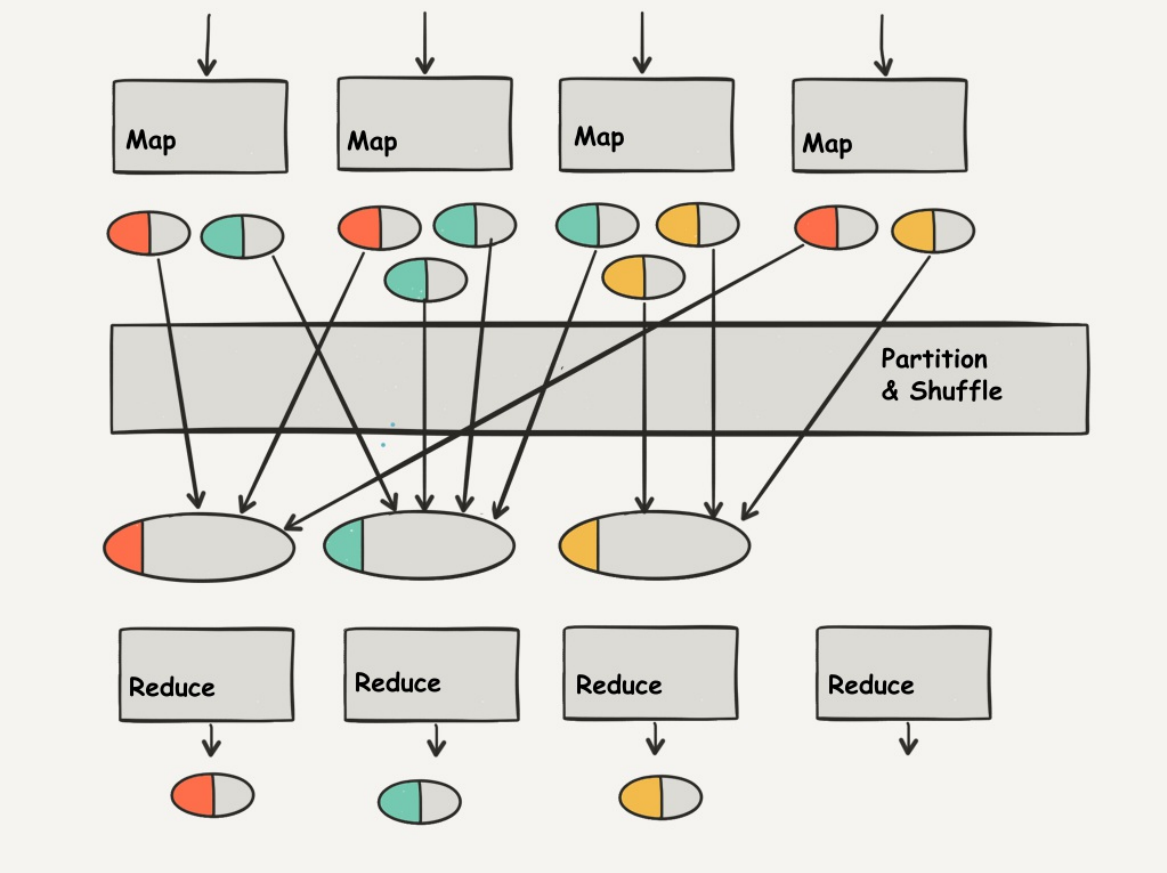


Figure 1-4. ReduceByKey groupings a MapReduce cluster

The pattern through which this is done in the word count operation is represented in the figure above. MapReduce guarantees that all the values that correspond to a given key will end up on the same machine. To achieve that, MapReduce will pick up the results of a `map` operation, and assemble them together on executors based on their key.

In that case, it means that we should always keep data skew in mind : the idea here is that, as is frequent in natural language-based text, some words are way more frequent than others, we have the potential for overwhelming the particular executor that will receive, say, the common word “the”. Indeed, the distribution of words in natural language obeys a Zipfian law, which means its distribution is not conducive to an equal share of work being sent to executors.

Note

Zipf’s law states that given some corpus of natural language utterances, the frequency of any word is inversely proportional to its rank in the frequency table.

To compensate for that, a MapReduce implementation will inevitably have to distribute the data encountered for one particularly “loaded” key to different executors; another point where associativity comes in handy. Because of this skew, we can also observe that some executors in one given stage of an operation may be much slower than others. This is a bad case as a stage of a MapReduce operation is as slow as its slowest worker.

Map-side combiner

Finally, to help with the large volume of intermediate data being sent across the network, MapReduce offers another optimization. It consists essentially in summarizing intermediate results as early as possible, before paying the price of sending data across the network.

You'll notice that in the regular functioning of the shuffle in a reduce operation above, the counts for each word are being sent across the network by default. However, a single executor may contain several counts for the same word. If you remember the shape of distribution of words in natural language, you'll notice that this is in fact very often the case. But for an executor to send several counts obtained from local processing for a single key (word) makes no sense: the executor can sum them within the confines of its own processing power, since it generated those counts in the first place.

To be able to aggregate the intermediate results it has obtained, all the executor needs is an idea of the operation to be performed — in fact, in many cases, the reduce operation itself. The resulting optimization results in the processing schematized below.

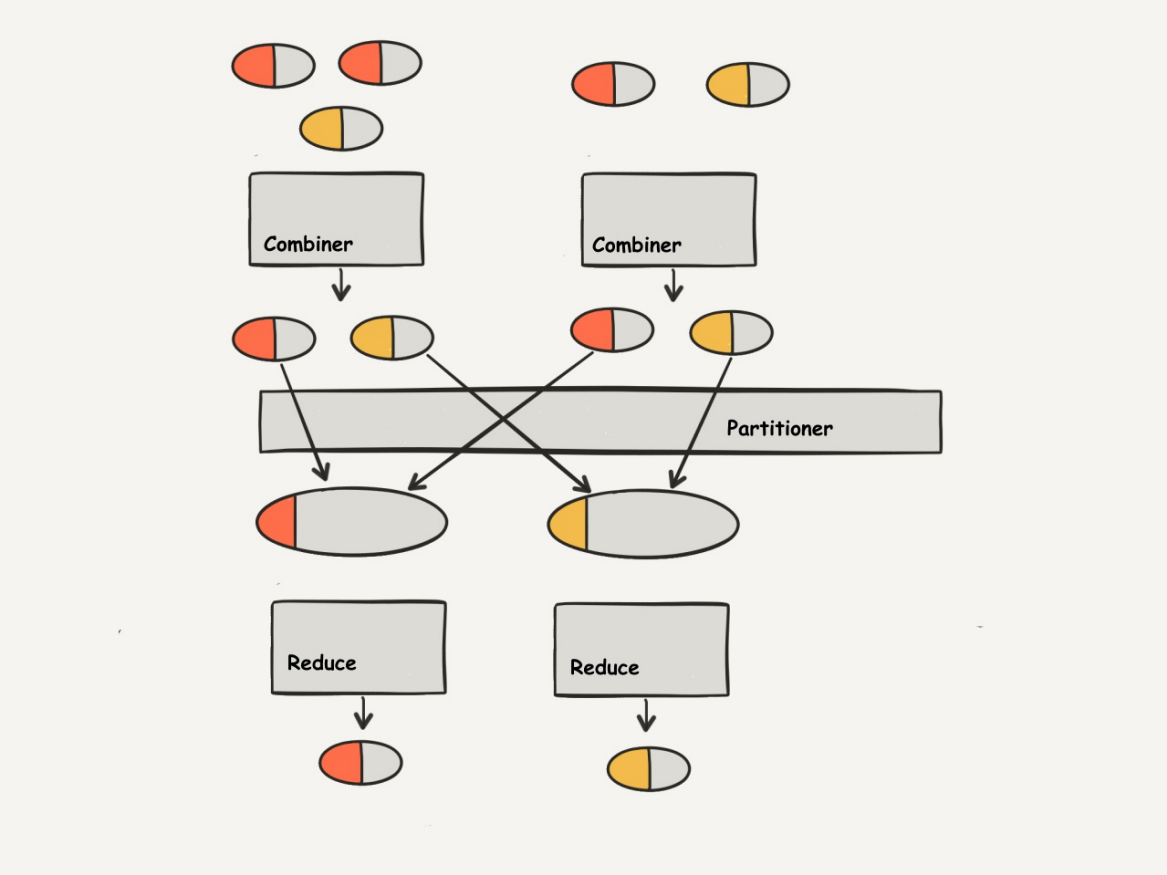


Figure 1-5. Map-Side combiners in a MapReduce cluster

To Learn more about MapReduce

The MapReduce model was introduced in ???, but finds a more detailed (though still high-level) explanation in ???. More information about the origins of MapReduce can be found in ???.

We have briefly touched on aspects of functional programming traditionally introduced through category theory, to explain the generality of the MapReduce model with sufficient precision. The references on those aspects abound and just listing the most accessible ones would be a bibliography by itself. We will just mention a more exhaustive study of the model than our pedagogical one, by ???, and a concrete outlook on how the notion of monoid (outside the scope of this book) helps the design of MapReduce programs, by ???.

The Spark ecosystem, approach and polyglot APIs

Spark appeared first in 2010, in the original technical report for the Mesos two-level scheduler. Wait, what's a two-level scheduler, you ask ?

Multiple frameworks, and a framework scheduler

By 2010, the MapReduce model, introduced in its first implementation at Google, had known an open-source sister in the labs of Yahoo Inc. That open-source effort was quite faithful to the original design. It was based on:

- a distributed file system, the Hadoop distributed File System (HDFS)
- an implementation of the MapReduce API (MapReduce)
- and as early as November 2011, a cluster manager called YARN (Yet Another Resource Negotiator) was to know its first release.

With the new resource manager and satellite open-source projects present to tackle points such as fault tolerance deficiencies (due to a single point of failure at the name node), Hadoop was fast becoming an established data processing framework.

Besides it, the older but more customizable MPI framework was trudging along as computation platform for academic experiments. This caused resource fragmentation: if you had a cluster in an academic lab, you would find it difficult to share computing resources between teams of people using different frameworks. The only alternative at the time was partitioning the cluster either in space — so that some machines run a given type of workload — or in time, running some workloads at scheduled times only.

Mesos was created as a meta-framework aimed at providing an efficient way to run heterogeneous workloads on a shared physical cluster infrastructure. Using process isolation techniques known as *containerization*, Mesos could dynamically match available resources in the cluster with the requirements coming from the participating workloads at any point in time. Mesos showed it could achieve higher average resource utilization on the overall cluster using that dynamic negotiation.

The first managed cluster resources were CPU and RAM. While any data analytics job is using the CPU, it must have been tempting to see what use could be made of executor's RAM.

Mesos' scheduling proficiency was hence demonstrated using a minimalistic framework called Spark, built to showcase the efficiency of caching. Spark was indeed presented at first as an experimental framework to illustrate the iterative nature of some machine learning algorithms, where the same dataset is mutated by several different processes many times over such as linear regression.

It is to be noted that since Hadoop relies on its distributed file system for fault tolerance, it has to save its intermediate results to disk on every iteration of the many implied by an algorithm such as linear regression. Spark changed that premise by noticing that with in-memory replication (and having the hard drive as an asynchronous fall-back), you could cache intermediate results in memory and still have fault tolerance. That allowed it to offer performance soaring above Hadoop, a point we'll come back to in an upcoming section.

However, the birth of Spark as a technology demonstrator for Mesos had another important consequence for its scope and development.

A Data Processing engine

When people consider Spark and Hadoop, they often wonder which hey should choose among the two. The question itself is ill-posed, since those are different pieces of software made to fulfill distinct tasks. Choosing between them makes as much sense as asking yourself if you should choose between a sedan and a V8 car engine. While the V8 engine may be more performant than the average sedan's engine, it will never replace the car itself.

Distributed data analytics imply that it will involve several machines. Because even commodity hardware is somewhat expensive, it frequently happens that these machines are to be shared among employees of a company, or teams of a given academic lab.

Thus appears the very problem that Mesos meant to tackle: multi-tenancy. It's not only that several teams want to run some computation on the cluster, it's that this computation differs significantly from one team to the next. This has forced people using large clusters to deal with this heterogeneity with modularity, making several blocks emerge as interchangeable pieces of a Big Data solution puzzle.

This puzzle is presented in the following figure. Going from the bare metal level at the bottom of the schema to the actual computation demanded by a business hypothesis, you could find:

The hardware level

> Potentially virtualized in homogeneous cloud solutions (such as the T-shirt size offerings of the giant Amazon), with a base operating system installed.

The persistence level

> On top of that, it is often expected that machines will offer a shared interface to a persistence solution offering to store the results of their computation, as well as perhaps its input. This is the level you would find HDFS at — among many other distributed file systems. Note however that some use a NoSQL database as a persistence layer. Others again are content to do without one.

The resource manager

> After persistence, most cluster architectures offer a single point of negotiation to submit jobs to be executed on the cluster. This is the resource manager, and the more evolved offspring of the *schedulers* of grid computing — except that on top of being an interface, it has an important role of meta-scheduling and compartimentation, ensuring that several heterogeneous workloads run efficiently and securely. Here you would find YARN or Mesos, among other alternatives.

The execution engine

> At an even higher level, there is the execution engine, which is tasked with executing the

actual computation. Its defining characteristic is that it holds the interface with the programmer's input, and describes the data manipulation. Spark or MapReduce would be examples of this.

A data ingestion component

Besides the execution engine, you could find a data ingestion server, that could be plugged directly into that engine. Indeed, the old practice of reading data from a distributed file system is often supplemented or even replaced by another data source which may be queried in real time. The realm of messaging systems or log processing engines in particular, is set at this level.

A processed data sink

On the output side of an execution engine, it is frequent to find a high-level data sink, which may be either another analytics system (in the case of an execution engine tasked with an ETL, *Extract, Transform and Load*, job), a No SQL database, or some other service.

A visualization layer

It is to be noted that since the results of computation are only useful if they are integrated in a larger framework, those results are often plugged into a visualization. Nowadays, since the data being analyzed evolves quickly, that visualization has moved away from the reporting of old, and towards more real-time interfaces, often using some web-based technology.

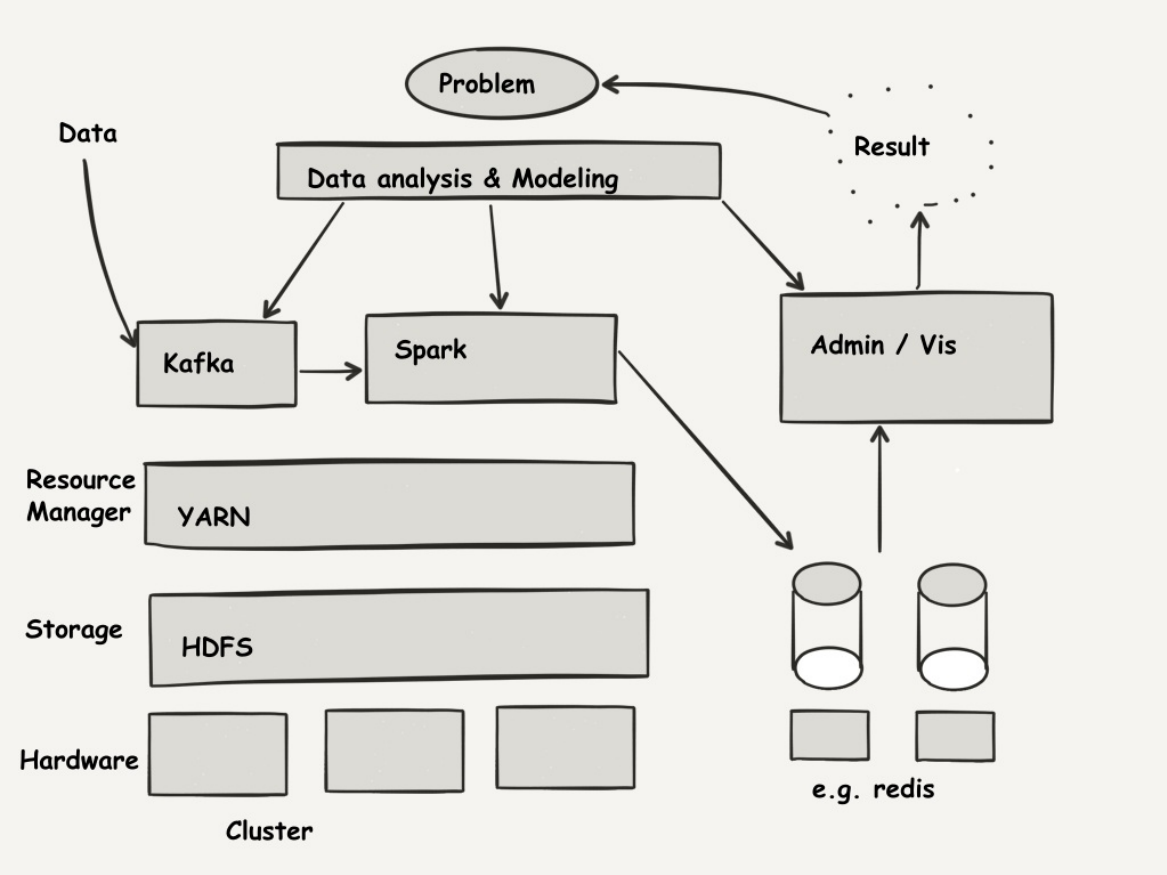


Figure 1-6. The Parts of a Big Data Solution

Within the scope of big data architecture, it is important to recognize the potential role of Spark. As an execution engine, which purpose was specifically to play well with other persistence layers, resource schedulers, input layers, and output layers, it never wanted to replace Hadoop as a whole.

Nonetheless, being conscious of its particular place in the Big Data picture, Spark concentrates in having excellent interfaces with the other blocks of the picture. In particular, it maintains great interfaces with YARN and MESOS as resource managers, has many data sources covered and new ones are easily added through an easy to extend API, and interfaces very well with output data sinks.

Finally, Spark is particularly obsessed with the most important interface of all: that which occurs with the programmer himself.

A polyglot API

Spark was first coded as a Scala-only project. But far from being just a library for distributing computation, it has become a polyglot framework that the user can interface with using Scala, Java, Python or the R language. The development language is still Scala, and this is where the main innovations come in. The coverage of the Java API has for a long time been fairly in sync with Scala, owing to the nice Java compatibility offered by the Scala language. And while in Spark 1.3 and earlier versions, Python was lagging behind in terms of functionalities, it is now mostly caught up. The newest addition is R, for which feature-completeness is an enthusiastic work in progress.

More interestingly, this versatile interface has let programmers of various levels and backgrounds flock to Spark for implementing their own data analytics solution. The amazing and growing richness of the contribution to the Spark open-source project are a testimony to the strength of Spark as a federating framework.

But Spark’s approach to best catering to its user’s computing needs goes beyond letting them use their favorite programming language.

A MapReduce extension

MapReduce, for all its simplicity, offered a relatively Spartan interface: while map and reduce are known to be two versatile functions, there is something to be said for extending the functions offered to the programmer to other high-level instructions that can be represented using the same distributed notions.

In fact, the notion of operations on a cached, distributed collections can be generalized to two rough types, reflecting the dichotomy between the map and reduce operations of the old model:

- the operations that do not particularly care which portion of the data they are processing. Those are said to be *embarrassingly parallelizable* applications, or equivalently, locality-agnostic operations. One example not readily provided by the MapReduce interface is the `filter` operation, which only returns the elements of a collection that follow a given predicate.

- the operation that need to exchange some data with other executors to compute their result, which require a shuffle. The canonical example is the frowned upon `groupBy` , but also, more simply, `distinct`, which returns the elements of the collection without duplicates.

Spark makes a strong effort to provide most of the operations one can find over Scala immutable collections, but over their own, distributed collection, the `RDD` . Moreover, it provides operations on key-value pairs, owning to the convenience of dealing with such mappings in every day data analysis.

A SQL interface, expanding into a DataFrame interface.

Since its very beginnings, large scale data analysis engines have not been the prerogative of programmers. Enterprises have long had business intelligence (BI) analysts that would employ a simpler language than Scala, Python or R. They rather use a simpler declarative language, akin to SQL, to query the large, distributed dataset they were working on.

That explains the fast development of Hive, a SQL query interface for Hadoop originally developed at Facebook. And since Spark aims at versatility, it explains while it first aimed at its source-compatible SQL-like interface, called Shark.

But the confines of this engine soon seemed difficult to keep as the Hive legacy of new Spark adopters became less and less relevant. In particular, this opened the door for an engine rewrite, which led to SparkSQL, which supports the full query language of Hive (HiveQL), and extends it. The Catalyst engine, running under the hood of SparkSQL, is now mature and feature-complete.

Moreover, thanks to Scala's versatility at defining domain-specific languages and a hefty use of the macro system, ways to interface with Catalyst have appeared outside of a dedicated SQL console. In particular, SQL queries can be addressed directly in Spark code. The recent practice of data science in dynamic languages (mostly Python and R) has revealed there was something more fundamental to the notion of tables in a dataset, leading to the work crystalized into *DataFrames*, an abstraction analogous to the R dataframes and the *pandas* library in Python.

DataFrames were introduced as API elements in Spark version 1.5.0. They allow the user to manipulate table-like datasets with an expressive API, irrespective of her language of choice. The code generated from the user's commands leverages the Catalyst execution engine, and allows for fast and uniform performance, thanks to numerous built-in optimizations. All the user has to do is to express the logic of her operation in a SQL-like DSL and she doesn't have to relinquish her favorite programming language to do so.

A Real Time processing engine

Finally, Spark has the opportunity, through low latencies and fast processing, to ingest and process data in a continuous, real-time fashion. This means that Spark can perform, in other words, stream processing. Spark Streaming is an API and a set of connectors, where a Spark script is being served fresh data in a continuous manner, performs a given computation and returns a result every time the stream of data receives new elements.

The interface that Spark offers on this front is particularly rich, to the point where this book is essentially centered on explaining it. We'll come more specifically to the basic properties that allow Spark to perform this feat in a moment.

As far as the purpose of Spark Streaming is concerned, it has become clear that it answers a developing need of the data analytics industry — in a fashion quite similar to what the foray into SQL and data frames is intended to be. As data analysts try to answer more questions on large amounts of data that may not be represented in an SQL database, so do programmers and analysts try to discover insights in a real time fashion, with a fast feedback loop.

In some industries, such as higher-frequency trading, monitoring and in the interaction with real time systems, the use and sense behind using streaming in analytics is obvious. But streaming analytics is pervading more and more business and scientific models as part of a wider trend: tightening the feedback loop between a model and the phenomenon being analyzed.

If you come back to the diagram in the big data solution diagram above, you will notice that an actionable insight (which often means revenue) is generated every time a loop — starting and ending at the business (or scientific) problem — is traveled fully. In sum, this loop is a crude representation of the experimental method, going through observation, hypothesis, experiment, measure, interpretation and conclusion.

But rather than just proposing to travel this loop many times per day, stream processing offers the possibility of getting intermediate results quickly, when testing slightly different versions of an hypothesis. This allows for fast course corrections, both on the phenomenon which is observed (think of an analysis doing fault detection on a cluster, where a detection triggers a response) and on the way it is observed (think of a prediction engine that consistently overshoots or undershoots, and of the speed at which a correction would ensue).

In fact, it could be argued that the movement of MapReduce towards simplifying distributed programming for the broader software engineering community, the push for higher-level interfaces and higher-level languages, the use of distributed computing to speed up computation on large amounts of data, and finally the ever-present search for the fastest software and hardware is a show of impatience in wanting to extract insight from data. Stream processing, which conclusions are delivered in a near real-time basis, is just the logical pinnacle of that "*impatience*".

In-memory computing, with impact on processing speed and latency

MapReduce and memory legacy

One of the first large performance improvements that Spark could show with respect to MapReduce was due to the fact that it made better use of the memory of the cluster. It is still the case today.

Indeed, MapReduce writes a lot of data to its distributed file system on every operation, and in between every operation. You can see the chronology of such an operation with disk reads and writes underlined in the next figure. We can see that the disk is accessed at the beginning and at the end of every operation.

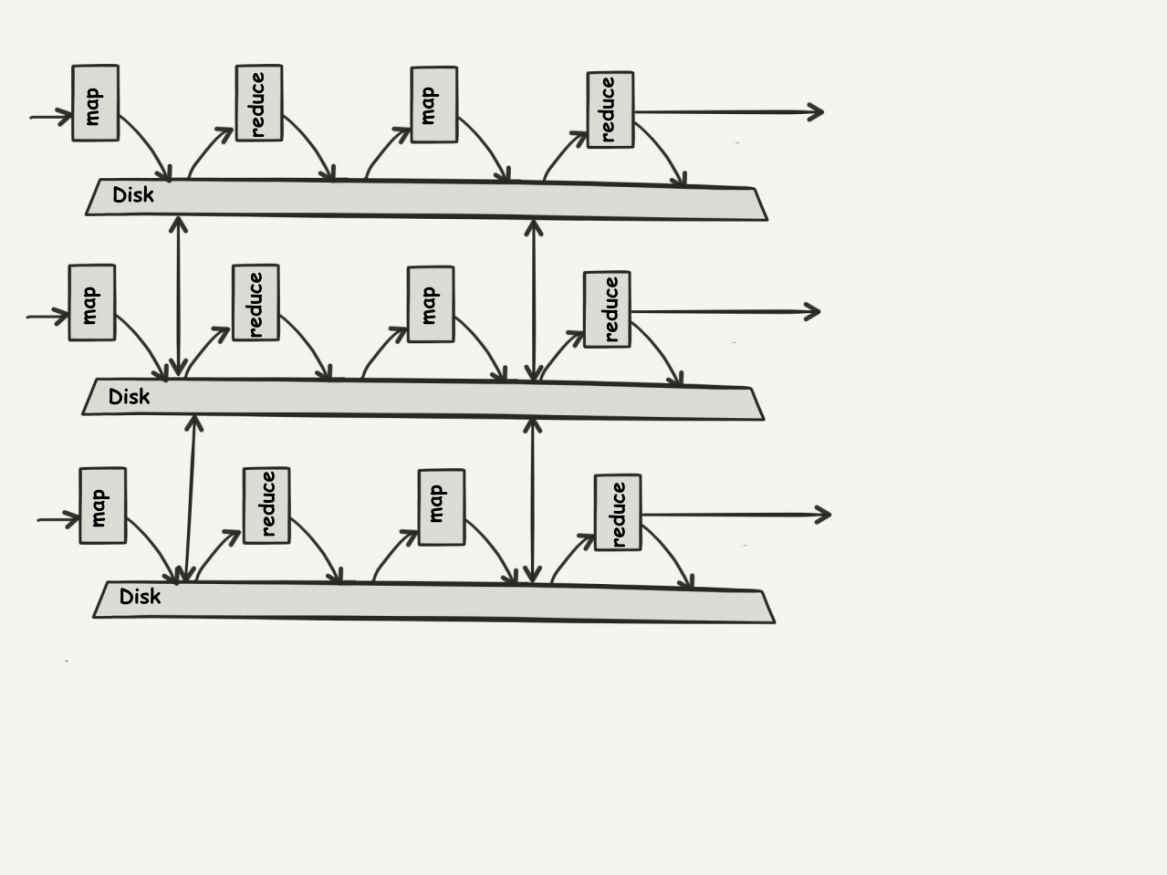


Figure 1-7. Disk usage in an example MapReduce operation

The advantage of this is clear: the data persistence part of fault tolerance is not a problem of the MapReduce engine anymore, since it can be cleanly delegated to the underlying distributed file system. The good thing is also that, behind the scenes, the input and output of any MapReduce task is well-defined: it's a distributed file. But the consequence is that all of these reads and writes to the secondary storage take time. Using the random access memory (RAM) of a computer is orders of magnitude faster.

Look at this 2002 blog post from Peter Norvig to find a comparison of the order of magnitude of these operations.

instructions	execute typical instruction
1/1,000,000,000 sec = 1 nanosec	fetch from L1 cache memory
0.5 nanosec	branch misprediction
5 nanosec	fetch from L2 cache memory

7 nanosec	Mutex lock/unlock
25 nanosec	fetch from main memory
100 nanosec	send 2K bytes over 1Gbps network
20,000 nanosec	read 1MB sequentially from memory
250,000 nanosec	fetch from new disk location (seek)
8,000,000 nanosec	read 1MB sequentially from disk
20,000,000 nanosec	send packet US to Europe and back

[1]

The choice of delegating data persistence to the distributed file system is readily explainable: from the original inventor (Google) to the open-source implementor (Yahoo), MapReduce is a framework designed for the “giants” of the web. And those “giants” are often touted as doing “big data”. For that size of data, only hard drives distributed across a cluster would offer an amount of storage of a suitable size. But can we make this intuition more precise?

In general, when someone says they are tackling a big data problem, it is common to assume that the size of their dataset will be beyond the processing capacity of a single machine, in terms of processing power, storage size and memory. Sometimes, this is taken to be the signal where one should go for a distributed solution. Others are often content, if their algorithm allows it, to only load a part of the dataset being worked on in the main memory, at any given point.

Spark’s genius in memory usage was two-fold: everybody knew that memory offers a much faster access than a hard drive. The first surprising idea was that *there is a place for large but not outlandishly so datasets*. Not everybody is a web giant trying to index the world wide web. Datasets that do not fit in the memory of one machine, can fit in the sum of the available memory of a cluster of machines. And as long as each executor can compute effectively based on this spliced fragment of the dataset’s size, all executors can contribute to solve the problem simultaneously.

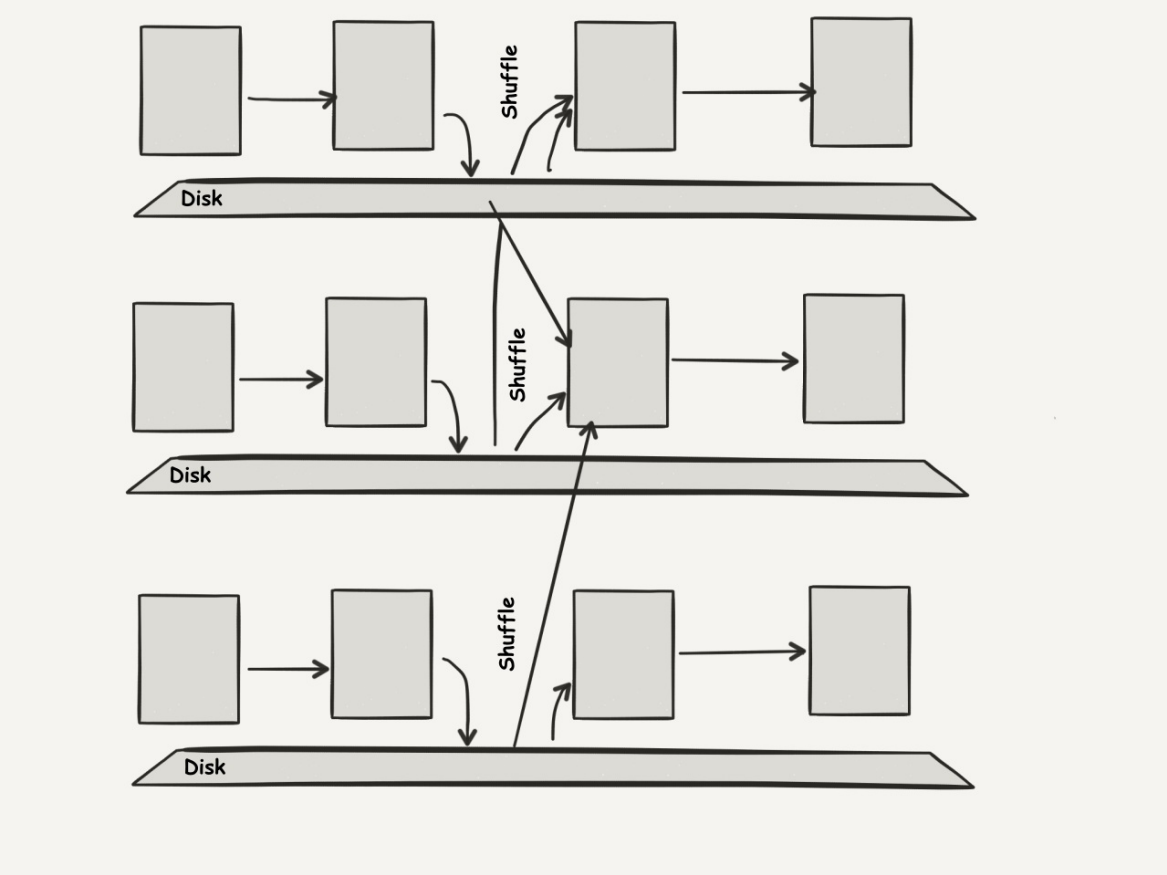


Figure 1-8. Disk usage in an example Spark operation

Another insight, which follows from the first, is that *if a problem can be tackled with out-of-core methods* — that is, if it can be spliced — *it usually means that chunks of it can be stored and processed in memory*. So even when this dataset has a size beyond the cluster's total available memory, this usually means that the use of memory as a storage layer offers performance boosts.

To wit, the original Mesos paper explains:

> We evaluated the benefit of running iterative jobs using the specialized Spark framework we developed on top of Mesos (Section 5.3) over the general-purpose Hadoop framework. We used a logistic regression job implemented in Hadoop by machine learning researchers in our lab, and wrote a second version of the job using Spark. We ran each version separately on 20 EC2 nodes, each with 4 CPU cores and 15 GB RAM. Each experiment used a 29 GB data file and varied the number of logistic regression iterations from 1 to 30 (see Figure 9).
>
> With Hadoop, each iteration takes 127s on average, because it runs as a separate MapReduce job. In contrast, with Spark, the first iteration takes 174s, but subsequent iterations only take about 6 seconds, leading to a speedup of up to 10x for 30 iterations. This happens because the cost of reading the data from disk and parsing it is much higher than the cost of evaluating the gradient function computed by the job on each iteration.

Hadoop incurs the read/parsing cost on each iteration, while Spark reuses cached blocks of parsed data and only incurs this cost once. The longer time for the first iteration in Spark is due to the use of slower text parsing routines.

Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center

Spark’s Memory Usage

Spark offers in-memory storage of slices of a dataset, which has to be initially loaded from a data source, which can be a distributed file system, or another storage medium. The Spark form of in-memory storage is the very simple operation of caching data.

Indeed, many algorithms in machine learning are iterative, and their expression of reuse of data can often be interpreted as caching. For example, linear regression has as the goal to compute an optimal set of weights, which minimize an error over a prediction that these weights produce. To do that, the most common method in a distributed setting is stochastic gradient descent, where a part of the dataset is chosen at random, the weights are tested on that fraction to see how well they perform on this subset, and if their performance is suboptimal, they are shifted in a direction that lets them reach a better result on this fragment of the data.

The set of weights represent the intermediate state of the algorithm along the training process — equivalently, it means that each step in the training depends on the weights obtained at the previous stage. Thus, where the intermediate result is always stored to disk in MapReduce, Spark can be thought of as saving an in-memory cache of the latest state of these weights. /// check this: Only the weights are cached? also (part of) the dataset is.

A customizable cache

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

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

However, since memory is such a valuable, but limited commodity, Spark makes — by default — the cache short-lived.

As we'll see in greater detail later on, a good part of the operations that can be defined on values in Spark's storage have a lazy execution, and it is the execution of a final, eager output operation that will trigger the actual execution of computation in a Spark cluster, since the very first step reading data from a source. It's worth noting that, if a program consists of a series of linear operations, with the previous one feeding in to the next, the intermediate results *disappear* right after said next step has consumed its input.

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

The data flow of such an operation is represented in the next figure.

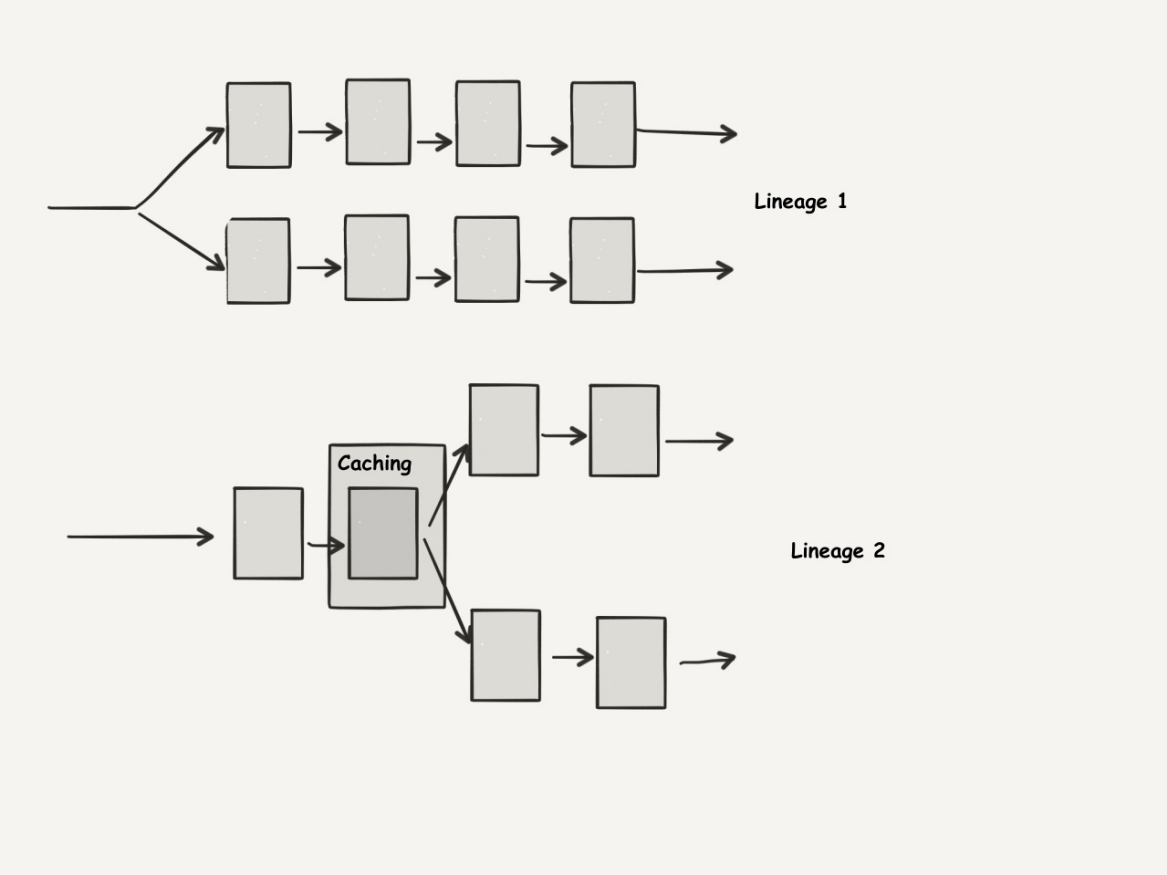


Figure 1-9. Operations on cached values

And finally, Spark offers the opportunity to spill the cache to secondary storage in case it runs out of memory on the cluster.

Operation Latency

One well-known issue in the Hadoop environment is the *small-file problem*: if your processing engine is reading small files as an input to its processing, the overhead of reading those small files is usually bigger than that of actually performing the computation on them.

One example of such an operation is analyzing the daily logs of a number of web servers. While the total sum of the logs may be quite big, the individual daily logs for one given server could be quite small, leading to a very large number of small input files. Since, for various reasons, Hadoop introduces some processing overhead at the file level, this results in a lot of time lost finding, opening and parsing files.

One usual solution to the small-file problem is to aggregate those logs once in a single file, as a pre-processing step, but this begs the question of redoing the analysis on small increments of data, such as the batch of logs for the newly elapsed day. What if inspecting a single day is enough to do the work, such as in anomaly detection ? You shouldn't have to analyze the logs since the beginning of time to determine whether some anomaly occurred yesterday — or in the last hour.

This question of small increments is, in fact, better served by a layer that can deliver these files with a minimum of latency — that is, a streaming data source. But serving this small increment — just a few files — was not a good use case for Hadoop, for which significant latencies were introduced by repeatedly writing and reading those few files at each intermediate step of the computation.

In fact, the best match for such a streaming server, delivering data in small increments, is some system that introduces as little latency as possible in between computation steps. In sum, we would be looking for a low-latency system. And Spark, with its ability to keep intermediate results in memory without ever needing the disk, fits this description.

Spark Streaming can be seen as an interface that reads a very small incremental chunk of the dataset, over the wire, at regular but small intervals. And the fact that Spark introduces very few other latencies that computation itself, allows for not only this incremental approach, but even permits a more extreme take on it, where computation is reduced to the minimum : Extract, Transform & Load (ETL) tasks. Those tasks have as a goal to prepare data for ingestion by another system, often performing relatively menial tasks in the process: for example, examples include parsing (extract), formatting interesting data (transform), and serving this data to the downstream consumer (load).

In a system with a high latency such as Hadoop, where most of the time spent in a job would be starting the job and reading files, we would not have a good fit. The fact that Spark is being used for these tasks is a testimony to the transformational aspect of the low latency it offers.

How Spark Streaming fits in the Big Picture

Micro-batching

Spark is, by design, a system that is really good at processing on a distributed collection. And yet streaming is about processing every value coming from a stream of data, making maximal use of the cluster. The two notions seem antagonistic at first: if the goal is to be really quick at computing on values coming in over the wire, the natural thinking goes towards something that looks like load balancing:

In network routing, or in reactive programming, the flow of data often looks like there are decision nodes that direct data to machines which process the in one or another specialized way. For simple load balancing, this is about spreading the incoming data evenly across the available machines. For reactive, distributed systems, it's more about sending the data to where the best use can be made of it.

Spark's notion of data flow is more about considering a pre-formed collection, and distributing it evenly across the cluster, and processing it in a quick and fault-tolerant manner. That even spread seems at first not to really make sense : for the spread to be even, you would need to have several elements, forming the collection to be divided evenly. Except a stream only delivers, by nature, one element at a time. And since that value is atomic, spreading it across the network is impossible.

So, as a solution to this conundrum, Spark will hold for a while, collecting data, until it has a certain number of elements — optimally, until those elements form a collection of a size suitable to be sent across the cluster.

The measure Spark decides to split the collection on is time, a measure which will make more sense below. Hence, Spark Streaming fundamentally waits for a fixed (but configurable) amount of time, before distributing the collection across the network and starting processing. For a high-volume data stream, this is actually a short amount of time. And since there is very little overhead in creating such a temporary collection, Spark will be able to process it quickly. This approach is known as micro-batching.

Note

You might want to consider why batches are bounded by time rather than a fixed number of elements. A fixed number (or size) of elements might be considered a superior solution from the point of view of the implementation: it ensures the division of the task across executors is more fair. and the processing time for those elements, in quite a similar way, becomes more predictable. But in a highly variable throughput, the time it would take to receive that many elements has no particular bound. In sum, that looks much more like arbitrary batches but by losing the time-based constraint, it has lost any meaningful connection to the real-time processing of a *stream*.

A strong Streaming characteristic

Spark Streaming can thus, as a whole, be seen as a bulk-synchronous system: every time the clock hits a certain mark, a new collection is formed from the data collected since the last tick, and sent across the network. One very important aspect to understand is that this mechanism is not interruptible, in quite the way that regular streaming systems can not be prevented from receiving elements, albeit one by one.

Streaming is very much the “show must go on” approach to computing. In fact, it could be argued that the basic tenet of stream processing is that it is built on the premise of computing for an infinite amount of data coming in through a stream. But the immediate consequence of this — since our computers are finite machines that are not able to store an infinity — is that the machine must be continually receiving new elements. With successive batches separated by time, that is even more easy to observe : when a new clock tick comes, the system must be finished with what it was doing with the last batch of data, and ready to process the new one from scratch.

Why is that? Because any system that accumulates a small amount of *lateness*, or that has any sort of unfinished processing to do with batch (n-1) when batch n comes, will have to either:

- recuperate by being faster to process the next batch than it was last time — but this is unlikely to always happen, after all batch (n-1) was a “slow” batch
- or carry this lateness over to batch (n). If this is not about being late, but some kind of resource consumption (e.g. memory usage), the extra memory used by this late processing of batch (n-1) may be freed by the end of batch (n), but surely batch (n) will consume some extra memory itself, that it may not release by the time of batch (n+1).

Hence, a system that does not *finish its homework* between a clock tick and the next, will carry an ever-growing lateness and grind to a halt, or see itself starved of resources by processing dedicated to stale batches and eventually will crash.

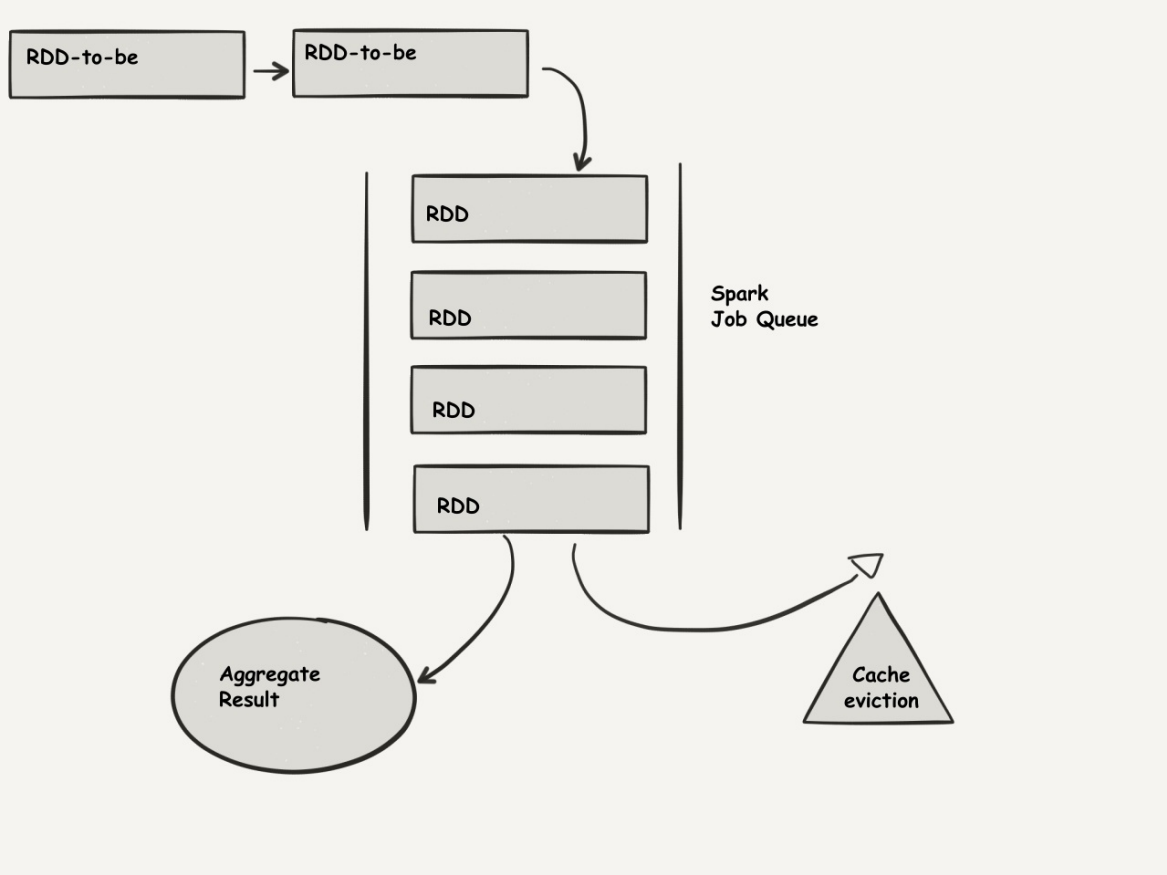


Figure 1-10. Streaming : receiving data in a continuous fashion

This reality should not be a surprise to people familiar with synchronous systems, as it is a golden rule of this kind of processing, designed with a bounded latency in mind.

A minimal delay

A consequence of micro-batching is hence that delays in processing that extends beyond the interval between two batches (the *batch interval*) is a bad idea. At most, it can happen on a very occasional basis if the average processing time is strictly less than this batch interval. In that exceptional regimen, the “longer than usual” batch introduces some latency (or resource reservation) that gets slowly reclaimed by the — on average — idle portion of the batch interval that is not usually taken up by processing.

But another consequence of micro-batching, which consequences are perhaps harder to see than the premise, is that any micro-batch delays the processing of any particular element of a batch by at most the batch interval.

Firstly, micro-batches create a minimal latency. The jury is still out on how small it is possible to make this latency, though 2 seconds is a relatively common number for the lower bound. For most applications, a latency in the space of a few minutes is sufficient: having a dashboard that refreshes you on key performance indicators of your website over the last few minutes is usually well sufficient. If you are doing predictive maintenance for a fleet of vehicles which won’t be available for maintenance until they finish their next trip of a few minutes anyway, there’s no point in having the information in the first few seconds where it’s mathematically possible to obtain it.

But for some low-latency application, such as high-frequency training where one single event may trigger a market reaction, or anomaly detection on board on a vehicle, where the result may trigger an immediate course correction, that delay may be a deal-breaker. The tightness of the streaming feedback loop is a careful point to consider when chosing Spark Streaming as a platform.

All the more since, if you remember, the potential delay of a batch interval between data availability and processing has an impact on resources : setting a small batch interval is not without consequences. With that setting, you should also ensure your resources are beefy enough to finish processing within the interval.

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

Throughput-oriented tasks

All in all, where Spark Streaming succeeds is with throughput-oriented data analytics. Some real time approaches offer gains when they offer the fastest answer. Others succeed when they're able to do more number-crunching, through more examples than the competition.

That distinction became clearer when Google released a paper called The Unreasonable Effectiveness of Data. In this paper (???) Halexy et al. argue that large volumes of training examples allow some relatively simpler models to generalize better than more advanced counterparts, provided they are confronted with a large amount of examples. And that argument was backed up by the various machine learning and analytics tasks Google was running on massive amounts of data, with cutting-edge results across a number of fields.

And the intuition is relatively simple to explain. If machine learning is about inferring a function of a given class given some examples on its output, a massively more important number of outputs will provide a much more precise approximation of this function — up to the limits of what the assumed function class (e.g. linear functions) can infer.

Moreover, the stability constraints of Spark Streaming, which drive the programmer to have his or her analysis perform well under the time limit of the batch interval, tends to let the programmer use relatively simple models, than do on-line, incremental processing rather than going through the whole history once again when a new training example is received.

Why you would want to use Spark Streaming

Building a pipeline

Machine Learning applications are often described as a pipeline — albeit one carrying insights rather than oil or gas — due to the number of successive stages they involve. Data has to be ingested, often parsed and cleaned in the process, then it has to be normalized, before feeding it into a model. Even this feeding involves a complex data flow, where the data is split between training and testing. On the output side, the model has to run on testing data after training, the results on the testing set have to be evaluated for generalization, and if unsatisfactory this may trigger a parameter change in the model. Provided that we eventually reach a stage where we have satisfactory results, the conclusions often have to be brought to some form of visualization. And this is just an high-altitude overview. Based on whether the data has to be vetted, or tested for coherence, or anonymized and mixed with noise to ensure privacy, there may be many more stages to the pipeline.

One important characteristic of a Streaming application is that all these stages have to be automated. Indeed, streaming tasks tend to have running times larger than the bath ones, because they never cease receiving data. Moreover, they tend to run beyond the bed time of most engineers that would be able to support them, so that there is a benefit to treating them like a server and making sure every part of pipeline can run on its own and feed its output to the next one without supervision.

The building of this pipeline of components therefore creates a difficulty: it requires completing a pipeline before seeing any results. To evaluate a model, however sketchy its first results may be, you need to reach its output, with all the prior data transformation components written and used. Downstream components tend to want the prior components to be already written, because while it is possible to unit test a part of the data pipeline, and compare its behavior with respect to a specification, it is really running on the data that will, alone, say if this part of the pipeline is suitable. This may make the development of data pipelines linear, and therefore relatively long.

Another difficulty, finally, is that the pipeline has to run permanently, and therefore be made of very reliable software. Not only do we expect components to be high-quality software, but we expect a Streaming platform to hold them together in a consistent way over time, in the adverse condition of receiving an unexpectedly high volume of data, of unknown quality, according to an imprevisible throughput.

Productive deployment of pipelines

But despite the specificities that we have been describing throughout this chapter, machine learning software is still software, in the sense that its quality is still proportional to the effort and care that has been put into it. As a consequence, component reuse is a powerful — but sometimes elusive — productivity boost. As we will see throughout the next few chapters, the computation model, API, and commands of Spark Streaming are very close to those of Spark's batch processing core.

And while the data transformation tasks involved in a machine learning task are varied, there are also, in the least subtle parts of the processing, pretty standard components. With the strong impetus created around Spark in open source development, it is possible to expect a set of components to already be present, boosting the speed at which it is possible to build this pipeline.

Moreover, as we will see in chapter 3, a Streaming application sometimes makes sense alongside a batch application. The low-latency task, in Streaming, delivers fresh approximations on an analysis that can be performed in the same way with higher precision on a daily collection of the data. In that case, Spark allows the reuse of components between the batch and the Streaming version with surprising ease, simplifying the code base to the point where users have the impression of maintaining one program, with slightly different running modes, and not two versions of the same analysis.

Productive implementation of data analysis

Spark's advantages in developing a *streaming* data analytics pipeline go beyond offering a concise, high-level API in Scala and compatible APIs in Java and Python. It also offers the simple model of Spark as a conceptual shortcut throughout the development process. If developing with a distributed collection is understood by most programmers, the mental leap to Spark Streaming — where one new such distributed collection comes in on every batch interval is easy.

Component reuse with Spark is a valuable asset, as is access to the Java ecosystem of libraries for Machine Learning and many other fields. As an example, Spark lets users benefit from say, the Stanford CoreNLP library with ease, avoiding the user the painful task of writing a tokenizer. All in all, this lets Spark Streaming users quickly prototype their data pipeline solution, getting first results quickly enough to choose the right components at every step of the pipeline development.

Finally, Spark Streaming lets users benefit from its model of fault tolerance, leaving them with the confidence that faulty machines are not going to bring the streaming application to its knees. This is an important boon, since a streaming application will potentially run for a very long time — and the probability of equipment fault, however small, augments with the length of time for which the application runs. If Spark users have enjoyed the automatic restart of failed jobs, they will doubly appreciate that automatic resiliency when tackling a streaming operation.

Beyond this standard aspect of resiliency, we will see that Spark Streaming has considerable advantages with respect to runtime, including back-pressure and straggler mitigation through speculative execution.

With its back-pressure support, Spark Streaming measures and reacts to the number of elements it sees at its input. It determines if the adequation between its processing capacities and the size of the input data, at the very instant it receives it, is optimal. And if is not, Spark Streaming limits the number of input elements it ingests, letting it avoid the accumulation of resource reservation and increasing delays by oversized data batches.

Using speculative execution, Spark measures the run time of successive tasks through the cluster, and reacts to any task running slower than comparable ones. If such a slow — but normally-sized — task, called a straggler, exists, Spark can re-launch it speculatively on another machine of the cluster, hoping for it to achieve a result quicker in a different environment.

In conclusion, Spark Streaming is a platform that, while making trade-offs in latency, optimizes for building a data analytics pipeline with urgency: fast prototyping in an environment with linear constraints, and stable run performance under adverse conditions are problems it recognizes and tackles head on, offering users significant advantages.

To learn more about Spark

This book is focused on Spark Streaming. As such, it will go quickly on the Spark-centric concepts, in particular about batch processing. The most detailed reference is ???.

On a more low-level approach, the Spark Programming guide is another accessible must-read.

Conclusion

In this chapter, we have learnt about Spark and where it came from.

- seen the base elements of the MapReduce model, and how it is dependent on a simple API, that produces programs executed in a cluster organized in a master-slave architecture, with a heavy reliance on a distributed file system.
- We have seen how Spark extends that model with key performance improvements, notably in-memory computing, as well as how it expands on the API, with new higher-order functions.
- Finally we have also considered how Spark integrates into the modern ecosystem of big data solutions, including the smaller footprint it focuses on, when compared to its older brother Hadoop.
- We have focused on Spark Streaming and, in particular, on the meaning of its micro-batching approach, what uses it should be appropriate for, as well as the applications it would not serve well.
- We have finally considered Spark Streaming in the context of Spark, and how building a pipeline with urgency, along with a reliable, fault toleran deployment is its best use case.

Bibliography

- [Dean2004] Jeff Dean and Sanjay Ghemawat, *MapReduce :Simplified Data Processing on Large Clusters* . OSDI'04 San Francisco, CA, December, 2004. *http://research.google.com/archive/mapreduce.html*

- [Halevy2009] Alon Halevy, Peter Norvig, and Fernando Pereira. *The Unreasonable Effectiveness of Data*. IEEE Intelligent Systems, March/April 2009 *http://research.google.com/pubs/archive/35179.pdf*

- [Lyon2013] Brad F Lyon. *Musings on the Motivations for Map Reduce*. URL *http://www.nowherenearithaca.com/2013/06/musings-on-motivation-for-map-reduce.html*

- [Karau2015] Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia. *Learning Spark : Lightning-Fast Big Data Analysis*. O'Reilly Media. 2015. ISBN 1-4493-5862-4

- [Lammel2007] Ralf Lämmel. *Google's MapReduce programming model — Revisited*. Science of Computer Programming, 68:3, Elsevier, October, 2007. doi:10.1016/j.scico.2007.07.001

- [Lin2010] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce* Morgan & ClayPool. 2010. doi:10.2200/S00274ED1V01Y201006HLT007

- [Lin2013] Jimmy Lin. *Monoidify! Monoids as a Design Principle for Efficient MapReduce Algorithms* arXiv:1304.7544, April 2013. *https://arxiv.org/abs/1304.7544*

- [Meijer1991] Erik Meijer, Maarten Fokkinga, Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*. pp. 121-144. Springer-verlag, NY, 1991 ISBN:0-387-54396-1

[1] Originally from Peter Norvig's "Teach Yourself Programming in Ten Years"

Chapter 2. Core Spark Streaming concepts

Apache Spark RDDs

A Spark Hello World

The reader unfamiliar with how to test a Spark application will refer to ???. However, let us give a quick rundown of how to launch a spark shell for testing.

Most distributions of Spark come with the `spark-shell` executable. Launching this executable creates an instance of the Scala REPL, with a custom spark context started for you.

This creates an instance of a `SparkSession` object, by default named `spark` and an instance of the legacy `SparkContext` with the alias `sc`. Since Spark 2.0, the `SparkSession` is the recommended way to interact with Spark, except for Spark Streaming, which relies on the `StreamingContext` as we will see later on. For the sake of this small intro, we will use the `SparkSession` or `spark` to interact with the Spark API.

The Spark Session needs a cluster to work. The easiest way to get started is using the default local cluster which gets automatically initialized at the start of the shell when no other cluster option is available. That local cluster uses the threads of the local machine (usually, if you are running this on a personal laptop, you can expect four threads for a two-core machine with an Intel processor and HyperThreading). Note the Spark shell can be and is used in deployments on a real cluster, though.

In the following, we will often indicate the use of completion to obtain method signatures. Please note therefore that when we put `[TAB]` in code examples, we indicate you should press the completion command at the point of insertion to obtain the same output.

The prompt at the launch of the Spark shell should look like this: (There might be small discrepancies depending on the specific Spark version used)

```
Spark context Web UI available at http://192.168.0.1:4040
Spark context available as 'sc' (master = local[*], app id = local-1491152898721).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.1.0
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_101)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

To avoid confusing the user with a lot of debug output, we have set the log level to `ERROR` . To see how to do that, consult the Spark documentationin the *logging* chapter.

Resilient Distributed Datasets

Spark has one single data structure as a base element of its API and libraries: the Resilient Distributed Dataset or RDD. This is a polymorphic collection which represents a bag of elements, in which the data to be analyzed is represented as an arbitrary Scala type. The dataset is distributed across the executors of the cluster and processed using those machines.

Using those RDDs involves — mostly — calling functions on the RDD collection type. The functions in this API are higher-order functions. In that sense, programming in Spark involves functional programming at its core: indeed, a programming language is considered to be functional when, in particular, it's able to define a function anywhere: as an argument, as a variable, or more generally as a syntax element. But more importantly, on the programming languages theory level, a language becomes a functional programming language only when it is able to pass functions as arguments. We will see in the following example how Spark lets you use an implementation of map to transform all the values of a collection by applying an arbitrary function to every single element.

```
scala> val names = sc.textFile("/home/learning-spark-streaming/names.txt")
names: org.apache.spark.rdd.RDD[String] =
      MapPartitionsRDD[1] at textFile at <console>:24
scala> names.take(10)
res0: Array[String] =
      Array(smith, johnson, williams, jones, brown, davis, miller,
            wilson, moore, taylor)
scala> val lengths = names.map(str => str.length )
lengths: org.apache.spark.rdd.RDD[Int] =
      MapPartitionsRDD[3] at map at <console>:27
```

Spark also lets you access the `reduce` function, which lets you aggregate key elements of a collection into another result, obtained through iterative composition. Let's have a look:

```
1 scala> val totalLength = lengths.reduce( (acc, newValue) => acc + newValue )
2 totalLength: Int = 606623
3 scala> val count = lengths.count()
4 count: Int = 88799
5 scala> val average = totalLength.toDouble / count
6 average: Double = 6.831417020461942
```

In these two examples, we are taking an RDD of `Strings` , read from a local text file of census data, and containing frequent last names. We are then summing their lengths in order to compute a mean of the length of last names in America.

It's worth noting that `reduce` requires that the `RDD` is non-empty. Otherwise, it will throw a `java.lang.UnsupportedOperationException` exception with the message: `empty collection`. This might seem like an extreme case, as we are discussing about processing large datasets, but it becomes necessary when we want to process incoming data in real time.

Spark implements a rich API inspired on the Scala collections API. It lets us expand our data processing vocabulary from the basic `map` and `reduce`. We could, for example, use `fold`, an aggregator similar to `reduce`, which lets you define an initial "zero" element to use with the aggregation function. `fold` and `reduce` both use an aggregation function closed over the RDD type. Hence, we could sum an RDD of Ints or calculate the `min` or `max` of an RDD of cartesian

coordinates according to a measure. There are cases when we would like to return a different type than the data represented by the RDD. The more general `aggregate` function lets you determine how to combine separate collections in an intermediate step:

```
1 scala> names.aggregate[TAB]
2 def aggregate[U](zeroValue: U)
3  (seqOp: (U, T) => U, combOp: (U, U) => U)(implicit scala.reflect.ClassTag[U]): U
4
5 scala> names.fold[TAB]
6    def fold(zeroValue: T)(op: (T, T) => T): T
```

It is the ease of use of this API that has won Spark RDDs the nickname of “ultimate Scala collections”. This reference to Spark’s original implementation programming language points at the library of Scala collections, which already let users benefit from functional programming on a single machine with a rich API.

The real genius of Spark is that it reproduces the ease-of-use of the Scala API, and scales it up to operate over a cluster of computing resources. Indeed, Spark does away with some of the restrictions of the MapReduce model: for instance, steps of the map operation do not have to be followed by a reduce. Several map operations can follow each other, in which case they are coalesced into a single stage, using a built-in optimization that is completely hidden from the user.

Moreover, Spark covers a richer API that is not limited to `Mapper` and `Reducers` or even to the extensions `fold` and `aggregate` of which we have had a glimpse right above.

Some operations, such as `filter` and `groupBy` are readily available. You can also consider `joins` and `cogroup`, that are interesting operations on several RDDs.

Transformations and Actions

Among the operations in the Spark RDD API, there are two broad categories. *Transformations*, on the one hand, take RDDs as their argument, and return another RDD. *Actions*, on the other hand, take a RDD as argument and return some local result. Examples include as diverse applications as `reduce`, which can sometimes output a single number, or `collect`, which returns a collection but one which is not distributed and rather stored in the memory of the driver machine. `count` will compute the number of elements in the collection and return a `Long`.

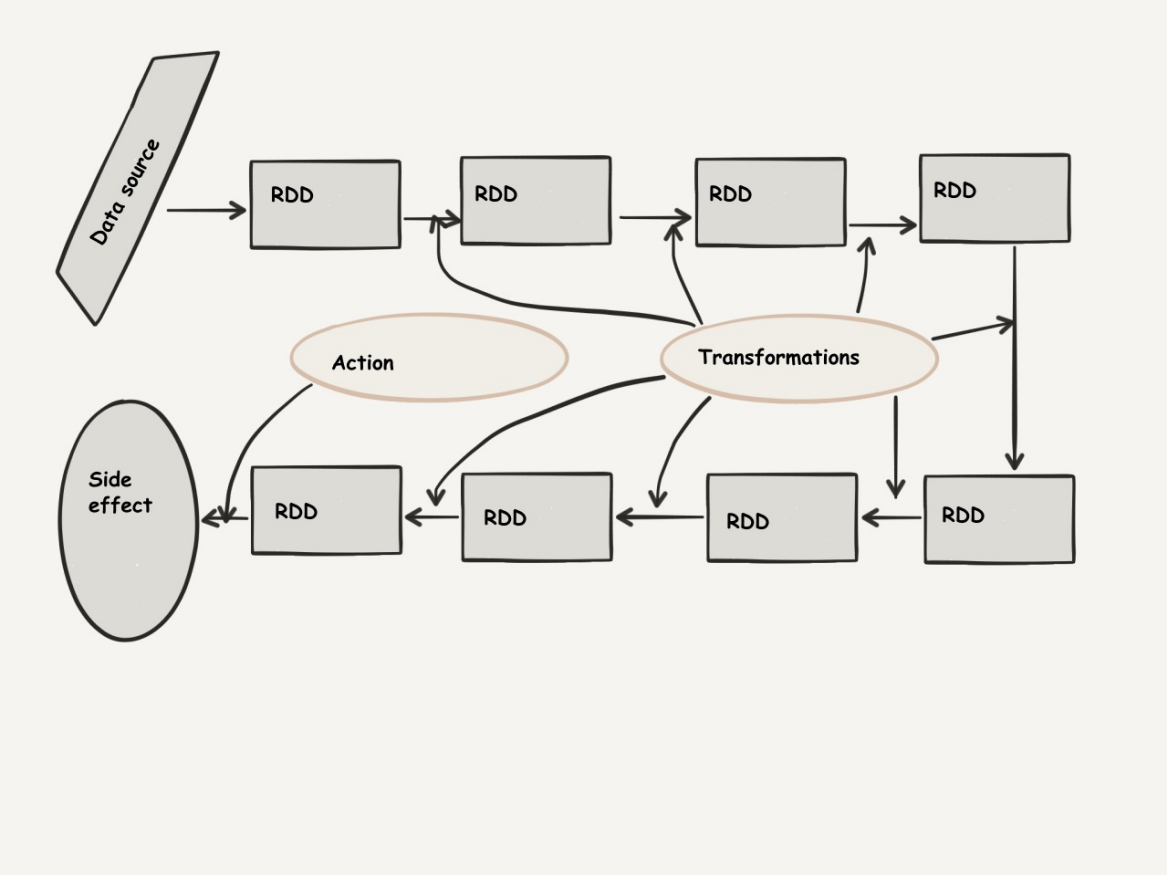


Figure 2-1. Transformations and Actions in Spark

The distinction between transformations and actions is important, since the writing of transformations in a Spark script does not trigger any computation, on the cluster or otherwise. Indeed, transformations simply define what is called *closures* which are a simple data structure that stores a function and its arguments (in a wide sense): in a nutshell, everything you needed to execute a computation, but not the actual execution itself. This closure is actually the payload which will be transported across the cluster network to later trigger computation on the executor nodes.

Closures can be composed — indeed, this is how `map` operations are coalesced into a single step — which means that when defining successive transformations within a script, Spark has several heuristics it can employ to combine, coalesce, or modify transformations as it needs for the purpose of fast computation.

But at some point, we would like to see results. This trigger for execution is where actions come in. Actions are actually cascading triggers, selecting in a lazy fashion which transformations are going to be used in producing a result. One very important point to notice here is that Spark actually computes the *to-do* list of tasks it will have to send on the cluster starting from the end: it starts from the action, calculating backwards which transformations need to actually be executed to provide the input to the next action. If the user writes his script linearly, applying successive transformations to its data, the scheduling process flows from the end of the script

where the action is defined, to the beginning where the required transformations (and transitively required operations) are being defined.

For example, we can look at the `foreach` operation. As with the Scala collection method `foreach` , its return type is `Unit`, meaning that this function only proceeds through side effects:

```
1 scala> names.foreach[TAB]
2    def foreach(f: T => Unit): Unit
```

That means, as a consequence, that `foreach` is an action. Its evaluation is therefore triggered at the moment the starting point of any distributed spark program — that is, the launch of the Spark context — is reached.

The Shuffle

In between transformations and actions very often stands a *shuffle stage*. This shuffle stage is characterized by the need for some executors to share information as a requirement to obtain the inputs necessary for computation. This step necessarily requires access to storage and network resources, and includes operations such as `reduce` , `fold` , `aggregate` or `groupBy`. Due to their reliance in secondary storage and communications across the cluster. shuffle stages are I/O heavy and considered expensive. When designing Spark programs, it is advised to reduce the size of data needing “movement” or avoid if possible at all. However, Spark is able to understand when this shuffle needs to happen and does not require to execute one on every other operation, contrarily to MapReduce.

Partitions

An important aspect to understand in the organization of the RDDs is that their distribution across a cluster is organized in *partitions*. *Partitions* represent the distribution unit in the mass of data described by an RDD. They are usually determined by the underlying storage and communicated to Spark by the data ingestion API. For example, when we use the HDFS API to load the `textFile` function we above, the HDFS Name Node will tell Spark the location of the different partitions that comprise the file in question. Once loaded, we can also customize how a distributed collection is cut into partitions, using the `repartition` and `coalesce` operations. Functions that require a data shuffle, usually also provide the option to repartition the data.

The general logic this underlines is that shuffles are a frontier between stages in Spark. Before a shuffle, any number of `map`-like operation — those that are unaware of the location of the data they are operating on — are coalesced into one single stage. The shuffle is the frontier that marks the passage to another stage through a network communication phase. When the data has been reorganized, another set of operations can occur, including transformations as well as actions.

This approach is surprisingly clean to use: it defines operations on one or several RDDs, including their `union` , some `filter` or `map` for example. Those operations get rearranged by Spark to the best of its ability, given the user-provided program. Then Spark notices an operation that requires a shuffle (`reduce` , `aggregate` , `fold` , `groupBy` , `join` and their pair-wise equivalents), and divides operations in two stages that get successfully scheduled and then executed on the cluster.

Those operations that require a shuffle stage, often provide an additional parameter to set the resulting partitions after the operation. This provides the opportunity to optimize the partitioning of the collection without incurring in an extra, separate shuffle operation.

Of course the real science of efficient Spark programs consists in having a sufficiently clear idea of what is going on behind the scenes to be able to express your program as something which will be efficiently and quickly run as fast as possible on your cluster. With that being said, the nice aspect is that Spark is sufficiently expressive to let you do your usually-defined programming in the most natural way possible first, leaving optimizations that would require the user to comply to a particular execution model as a later step — if and when it is necessary.

Debugging RDDs

A great tool to make sense of the state of Resilient Distributed Datasets is the debug information they carry. For that, the method `toDebugString` is a great asset. For example, we can see the difference between map operations and the shuffle operations:

```
scala> names.toDebugString
res5: String =
(2) MapPartitionsRDD[1] at textFile at <console>:24 []
 |  file:/home/huitseeker/names.txt HadoopRDD[0] at textFile at <console>:24 []

scala> names.map( (lastName) => lastName.size ).toDebugString
res6: String =
(2) MapPartitionsRDD[2] at map at <console>:27 []
 |  MapPartitionsRDD[1] at textFile at <console>:24 []
 |  file:/home/huitseeker/names.txt HadoopRDD[0] at textFile at <console>:24 []

scala> names.filter( (name) => name.size > 3).toDebugString
res7: String =
(2) MapPartitionsRDD[3] at filter at <console>:27 []
 |  MapPartitionsRDD[1] at textFile at <console>:24 []
 |  file:/home/huitseeker/names.txt HadoopRDD[0] at textFile at <console>:24 []

scala> names.groupBy( (name) => name.charAt(0) ).toDebugString
res8: String =
(2) ShuffledRDD[5] at groupBy at <console>:27 []
 +-(2) MapPartitionsRDD[4] at groupBy at <console>:27 []
    |  MapPartitionsRDD[1] at textFile at <console>:24 []
    |  file:/home/huitseeker/names.txt HadoopRDD[0] at textFile at <console>:24 []

scala> names.groupBy( (name) => name.charAt(0) ).
                mapValues( l => l.size).toDebugString
res9: String =
(2) MapPartitionsRDD[8] at mapValues at <console>:27 []
 |  ShuffledRDD[7] at groupBy at <console>:27 []
 +-(2) MapPartitionsRDD[6] at groupBy at <console>:27 []
    |  MapPartitionsRDD[1] at textFile at <console>:24 []
    |  file:/home/huitseeker/names.txt HadoopRDD[0] at textFile at <console>:24 []

scala> names.keyBy( (name) => name.charAt(0) ).
         mapValues( (name) => 1 ).reduceByKey( (x, y) => x + y).toDebugString
res10: String =
(2) ShuffledRDD[12] at reduceByKey at <console>:27 []
 +-(2) MapPartitionsRDD[11] at mapValues at <console>:27 []
    |  MapPartitionsRDD[10] at keyBy at <console>:27 []
    |  MapPartitionsRDD[1] at textFile at <console>:24 []
    |  file:/home/huitseeker/names.txt HadoopRDD[0] at textFile at <console>:24 []
```

In this example, `name` is our file read from the local disk through the Hadoop API, which produces one element per line. As the lines of our file contain one last name per line, this is just a name for each element of the RDD. Neither the map nor the filter introduce a shuffle, as we can see. We see that the successive map definitions introduce further transformations, to be grouped by the Spark scheduler at a later stage. However, note we can see the exact content of these transformations. We also see that `groupBy` as well as `reduceByKey` introduce a shuffle, after and before which we can add operations that do not require one. The example culminates in a convoluted way of counting the number of first names in our file by their first letter.

The information that is revealed by the `toDebugString` command is the lineage of an RDD, that we have mentioned throughout this chapter when talking about fault tolerance and the succession of operations that leads to an RDD.

Witnessing caching

Finally, note that `toDebugString` lets us witness the effect of caching. [1] We can indeed look at the previous few lines of our latest example, but with caching thrown in:

```
scala> val namesByFirstletter = names.keyBy( (name) => name.charAt(0) ).cache()
namesByFirstletter: org.apache.spark.rdd.RDD[(Char, String)] =
      MapPartitionsRDD[14] at keyBy at <console>:26

scala> namesByFirstletter.toDebugString
res11: String =
(2) MapPartitionsRDD[14] at keyBy at <console>:26
  [Memory Deserialized 1x Replicated]
 |  MapPartitionsRDD[1] at textFile at <console>:24
   [Memory Deserialized 1x Replicated]
 |  file:/home/huitseeker/names.txt HadoopRDD[0] at textFile at <console>:24
   [Memory Deserialized 1x Replicated]

scala> namesByFirstletter.count()
res12: Long = 88799

scala> namesByFirstletter.toDebugString
res13: String =
(2) MapPartitionsRDD[14] at keyBy at <console>:26
  [Memory Deserialized 1x Replicated]
 |  CachedPartitions: 2; MemorySize: 7.2 MB; ExternalBlockStoreSize: 0.0 B; DiskSize: 0.0 B
 |  MapPartitionsRDD[1] at textFile at <console>:24
   [Memory Deserialized 1x Replicated]
 |  file:/home/huitseeker/names.txt HadoopRDD[0] at textFile at <console>:24
   [Memory Deserialized 1x Replicated]

scala> namesByFirstletter.mapValues( (name) => 1 ).
      reduceByKey( (x, y) => x + y).toDebugString
res14: String =
(2) ShuffledRDD[16] at reduceByKey at <console>:29 []
 +-(2) MapPartitionsRDD[15] at mapValues at <console>:29 []
    |  MapPartitionsRDD[14] at keyBy at <console>:26 []
    |  CachedPartitions: 2; MemorySize: 7.2 MB; ExternalBlockStoreSize: 0.0 B; DiskSize: 0.0 B
    |  MapPartitionsRDD[1] at textFile at <console>:24 []
    |  file:/home/huitseeker/names.txt HadoopRDD[0] at textFile at <console>:24 []
```

Note that caching is actually lazy itself: it only modifies the lineage of an RDD on the first evaluation (here the call to `namesByFirstletter.count()`) and not before. The lineage of dependent RDDs is then modified to take into account that cached information.

Spark Streaming Clusters

The Standalone Spark cluster

Spark has three deployment modes that condition the context in which the distributed computation is executed in the cluster.

Indeed, Spark Streaming requires some pre-existing software to be installed and activated so that it can run on the machines that compose the architecture of the cluster. In particular the master machine will need to run a server for the driver to talk to the cluster, whose role it will be to:

- communicate the work that has to be done on the data,
- ship the closures that compose the program that the user has defined,
- along with the dependence jars that provide the libraries necessary for the master to interpret these closures into an executable program.

The role of that master machine is also, in many cases, to connect the running Spark jobs with the global state of a cluster and to manage the resources that are being used in these machines.

The first cluster mode, and perhaps the simplest, is the Spark standalone cluster mode. It is part of the Spark code base, and as such allows a user to run af Spark cluster without any additional software. It is also the recipient of the most Spark-specific features implemented by the Spark development team. It is aimed at showcasing the performance of Spark when running on real hardware.

However, the Spark standalone cluster makes the assumption that the only program running on the cluster — that is, on the underlying operating system of the master and the executor machines that constitute the cluster.

A consequence of that, is that it does not particularly care for the isolation and limitation of the Spark processes. With the Spark stand alone cluster mode, Spark runs very much in the way old school web server would. There is no supervision involved, and as such not multi-tenancy. In particular, sharing the cluster's resources with another user, running his jobs and yours concurrently in a fair fashion, is not supported out of the box.

Beyond multi-user support, one of the operating consequences of a cluster mode's feature set — and of the Spark Standalone cluster in particular — is that fault tolerance may also have some specific limits. In particular, the user will have to run a hot swappable master (usually using Zookeeper) by itself, to prevent a crash from the master machine vringing down the whole cluster. In case of a crash from the driver machine, the whole Spark job will have to be restarted. [2]

The advantage of the Spark stand alone cluster, however, is that it is the bleeding edge of speed if you want to run a small Spark job for prototyping, testing and benchmarking. It is also at a major advantage if you want to test advanced features that have just been implemented in the new version of Spark.

Yet Another Resource Negotiator (YARN)

However, if you are interested in a more mature — and more robost — way of running Spark in a cluster, you may be interested in the two other cluster modes. The first one is YARN, which stands for Yet Another Resource Negotiator, and in which you will recognize the resource manager of Hadoop from the first chapter.

This resource manager benefits from a long time of testing in the enterprise, indeed the longest of all available Spark clustering modes, since it is oldest system and concerns quite a few enterprise use cases.

YARN is therefore quite mature and interacts particularly well with other programs from the Hadoop world, in particular with a running HDFS server. It allowes running in cluster mode, in which the driver program — specifying the user's commands runs as part of the cluster, so that the cluster's fault tolerance routines can restart it, in case the driver's machine crashes.

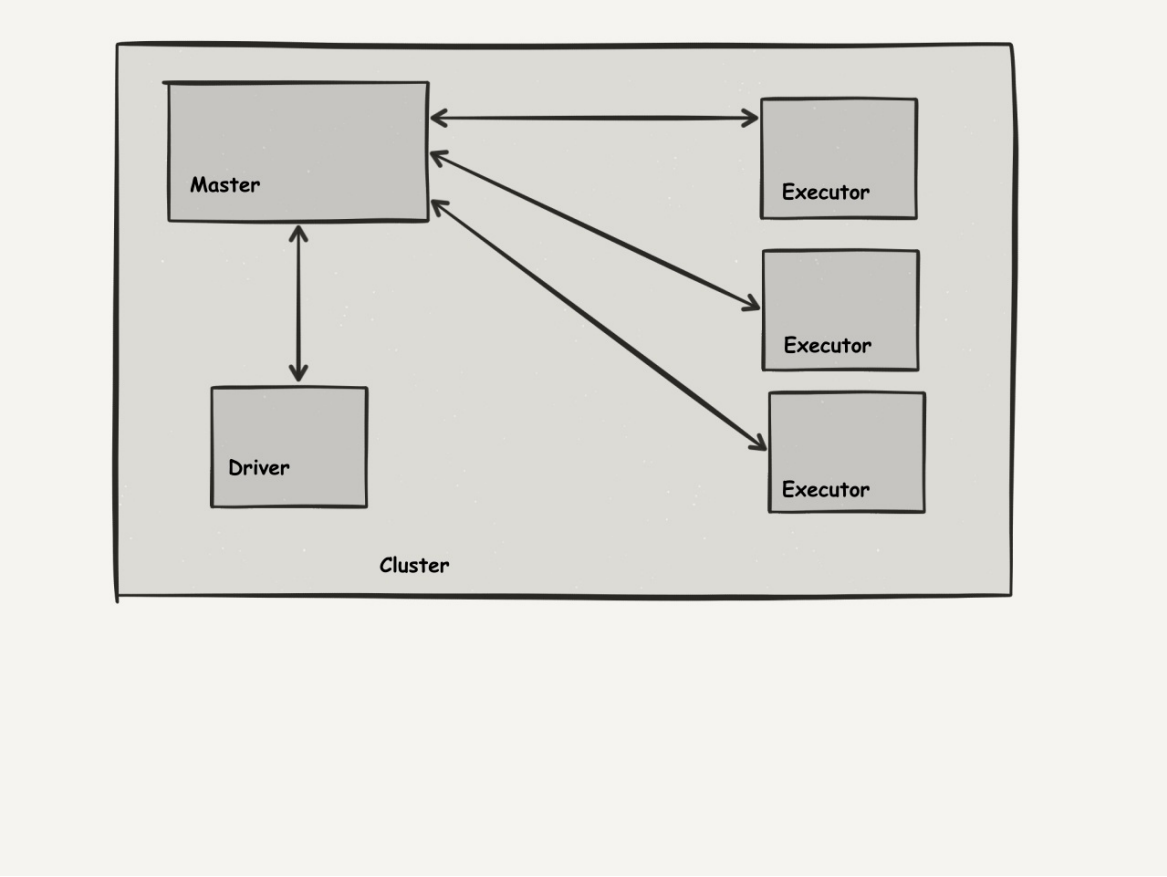


Figure 2-2. Running in cluster mode

The confusingly-named cluster mode is different from the default client mode, in which the Spark job is piloted from the user's personal laptop or workstation. In that later, default mode, if the user closes his or her laptop and goes away, naturally, the driver program, which specifies all the computation to be done, dies without a hope for restart.

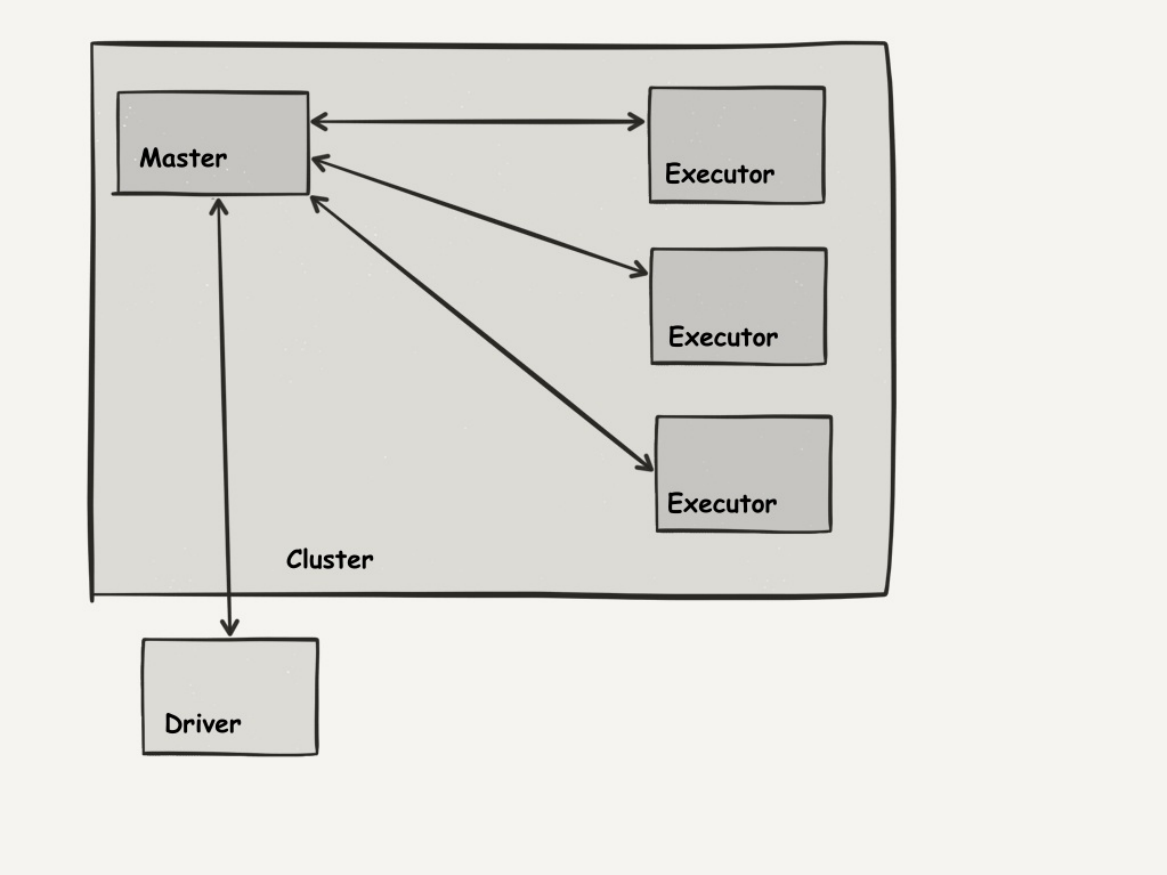


Figure 2-3. Running in client mode

Naturally, as being part of the Hadoop ecosystem, this mode Is limited to running tasks — such as a Spark or MapReduce — that are made to run for it.

In some cases, people might be interested in running a larger variety of distributed tasks on the cluster, despite this variety not including anything that would be related to the world of Hadoop. More importantly, they may want fine-grained control over the performance characteristics of these operations, including CPU as well as memory.

In that case, the major solution would be Mesos, the cluster mode of that was originally developed to be showcased by Spark.

Apache Mesos

Mesos makes use of LXC or lightweight containers, a technology which was added by Google to the Linux kernel to allow better isolation of and control over processes, by adding lightweight visualization techniques that would expose them to a virtual kernel. [3]

Along with containers, Mesos takes advantage of cgroups and namespaces, and leverages them into fine-grained management of resource consumption in terms of CPU and memory (as well as disk and network i/o, with varying levels of integration). Here, Mesos is the component that bridges this control at the single machine level into a global distributed resource management at the cluster level.

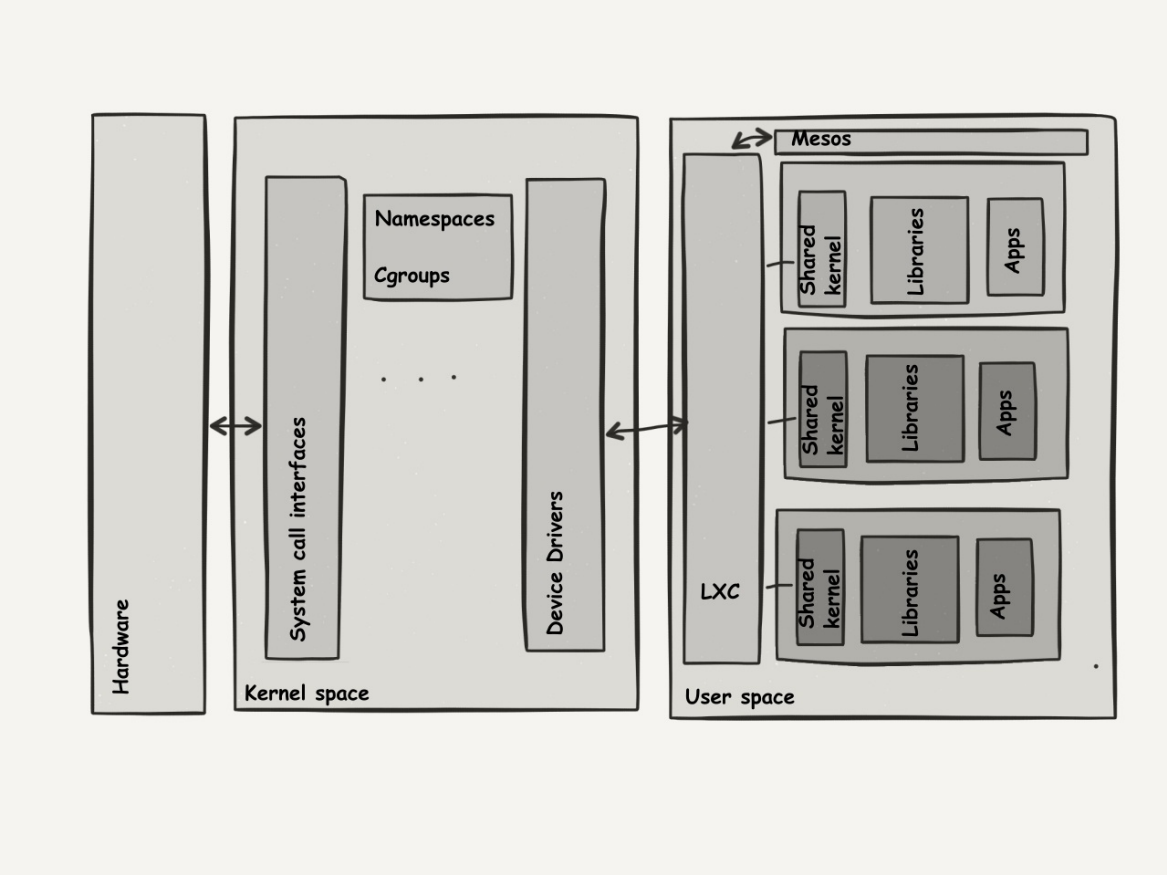


Figure 2-4. Mesos architeccture and LXC Containers

In doing so, Mesos allows the running of of Spark with a variety of other servers that are have auxiliary functions in the whole data processing pipeline. Indeed, those other distributed applicaitons are frameworks in Mesos parlance, meaning that they implement a simple eponymous interface that lets Mesos containerize, monitor and schedule those applications. In particular, we have encountered use cases in which Kafa — a streaming server —, Cassandra — a NoSQL database — would run in parallel with Spark, but still supervised by Mesos, harmonizing the resource consumption in the cluster.

This versatile integration and book keeping for resource sharing makes as one of the most flexible of the ways to run Spark. However, Mesos is still less mature than YARN, and it's coming to the apt with sparkin its interface with Spark specifically, it suffers from a significant feature lag, especially in the domains of security and authentication. YARN, for example, is richer in the domain of offering monitoring and supervision interfaces, and it is better distributed in specific-purpose Linux distributions.

Moreover, Mesos has two particular sub-modes which are terminating a Fusion started with Spark 1.4:

- the default fine-grained mode allocates resources for Spark on an ad hoc basis, but incurs the cost of deploying containers for every Spark task

- and the coarse-grained mode reserves resources on the Mesos cluster in a permanent way, but limits opportunities for resource recycling when SPark is not taxed for resources.

In the future, it seems likely that Mesos will be fused into a coarse-grained mode which will have a dynamic component, allowing Spark to reserve a fixed but low number of executors, a setup to be supplemented, based on need, by dynamically assigned new resources.

Note

Client mode is supported by all Spark cluster modes. As for cluster mode, refer to the table below:

cluster deployment mode	cluster mode (driver runs as a task)
standalone	implemented, but unsupported
YARN	supported
Mesos	In development

Spark Streaming : a delicate deployment

In the context of deploying a Spark Streaming application, it is absolutely vital to consider, in detail, the very specific runtime characteristics of a streaming job.

Streaming jobs are made to run for a longer amount of time. In that context, it is absolutely vital to have a cluster manager able to supervise the job for those extended periods of time — in fact, we'll see that SPark itself has a specific visualization and monitoring interface for Streaming. The fault tolerance capabilities of the cluster also become extremely important.

In particular, the failure of either the master node or the driver node — both of which would be considered improbable in a short-lived batch mode — could be considered a real possibility in a Streaming job, which could run for days. Moreover, it is absolutely essential to be able to do forensics on the resources utilisation of a cluster when running a production streaming job — since as we will see later, Spark Streaming jobs sometimes have unintuitive failures.

Finally, it is often necessary to run another server alongside the Streaming job, in order to reliably ingest the data required by Spark in a proper fashion. That server is often a stream processing server amidst a handful of options that include Apache Flume or Apache Kafka. This server ideally play nice — resource-wise — with the Spark Streaming job in order to run smoothly along the Streaming job's lifetime.

Symmetrically, it is often necessary to run *another* server that would be used downstream of the Spark Streaming job, to receive the results of computation, and perhaps update a visualization server. That kind of architecture now means that at least three servers (ingestion, reception and the Spark Streaming computation itself) will have to be run distributed on the same underlying machines.

Making sure those three applications run without eating each others' resources becomes a serious runtime concern.

Note

Spark also has a local deployment mode — mostly used for testing —, in which it runs jobs in different threads, emulating the behavior of a cluster. However, in that case, there is no real distributed computing, since all ements fo the computaiton run on the user's (driver, single) machine. Therefore we have chosen to leave it aside for the purpose of this comparison.

However, the efficiency of this local deployment mode is not to be underestimated — on top of allowing testing, it allows rapid prototyping, or even the development of an efficient single-machine multi-threaded program. A notable feat, even in the 21st century. That form of development nearly guarantees the possible efficient future deployment of a job in a later deployment.

To learn more about runinng Spark on a cluster

An upcoming reference for Mesos is ???. A main reference for YARN can be found in the latest editions of the reference volume for Hadoop, ???, while a more detailed reference is ???.

Fundamentals of a DStream

A Bulk-synchronous model

In this chapter, we are actually going to see how the theoretical model mentioned in the Big Picture chapter fits into a coherent API. To start with something simple, we will first explain the receiver model, before moving on to other ways of creating a `DStream` in Spark — that model is, after all, still the main way to create a `DStream` in Spark.

Whereas the fundamental abstraction for Spark was the resilient distributed dataset, or RDD, the fundamental abstraction for Spark Streaming is the `DStream`, or distributed stream.

Note

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

It introduces three key concepts:

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

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

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

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

We have mentioned that Spark Streaming would work assuming it could get an infinity of elements. In fact, it consists of a stream, which follows a processing model similar to bulk-synchronous parallelism:

- All the Spark executors on the cluster are assumed to have a synchronous clock, for example synchronized through an NTP server.

- one or several of the executors runs a special Spark job, a Receiver. This receiver is tasked with receiving new elements of the Stream. It receives two clock ticks:

    - the first and less frequent is the *batch interval*. It marks when the Receiver should assemble the data from the stream received since the last clock tick, and produce an RDD for distributed processing on the cluster.

    - the second and most frequent clock tick is called the *block interval*. It signals when elements received from the Stream should be allocated to a block, that is, the portion of the Stream that should be processed by a single executor, for this current interval. A block will make up a partition of the RDD produced at each batch interval.

- during all processing, as is the case with a regular (batch) Spark job, blocks are signaled to the block manager, a component that ensures any block of data put into Spark is replicate according to the configured persistence level, for the purpose of fault tolerance.

- On each batch interval, the RDD which data was received during the previous batch interval becomes available, and is thus scheduled for processing during this batch.

To make the parallel with a bulk-synchronous model, the barrier here would be the arrival of a new RDD at the batch interval. Except this is not real a barrier, since this happens independently of the state of the cluster on the moment of the arrival of the new batch: Spark's Receivers do not wait for the cluster to be finished with receiving data to start on the new batch.

This is not a design fault, but rather a consequence of Spark Streaming trying to do real Stream processing at its most honest: despite having a micro-batching model, Spark Streaming acknowledges that a Stream has no predefined end, and that the system will not be able to do away with receiving data continuously.

We can represent the arrival of this data as follows:

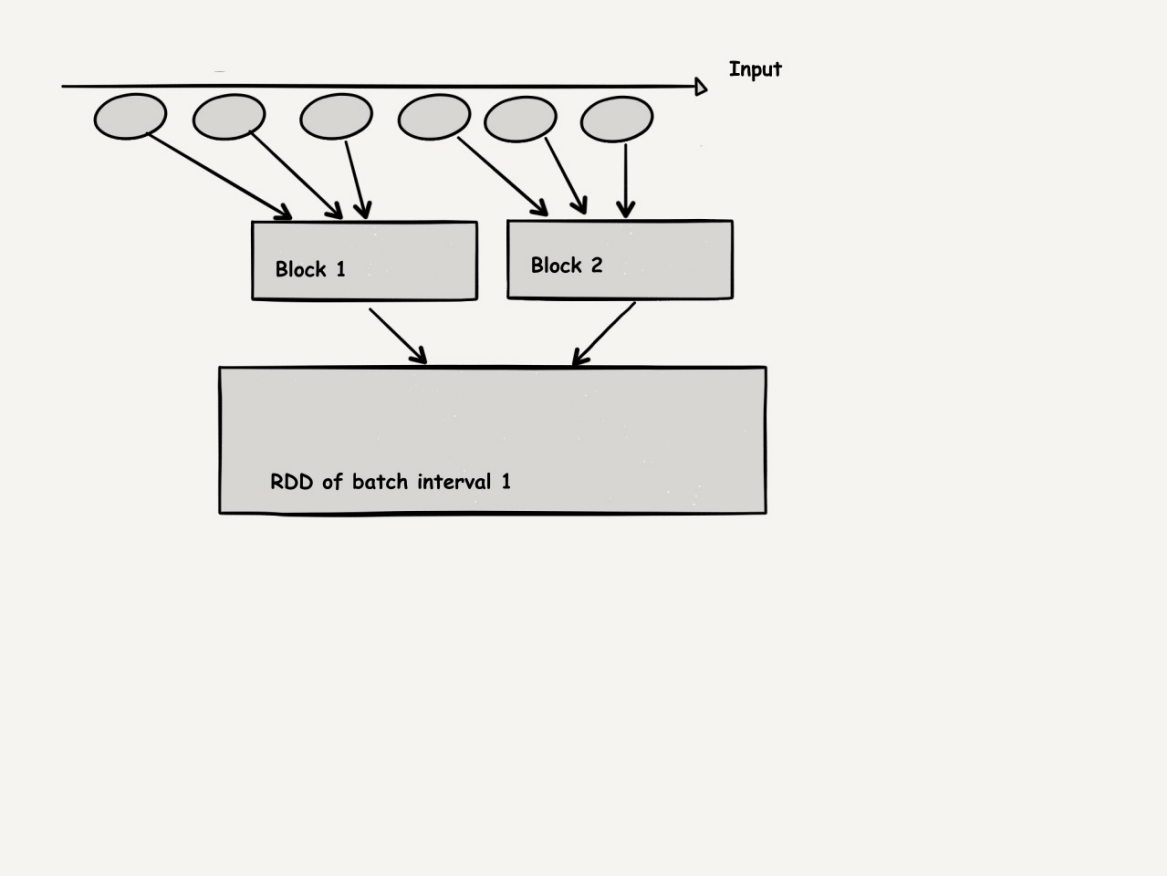


Figure 2-5. The basics of a `DStream` : blocks and Batches

The consequence of this relatively simple model is that the Spark job tasked with receiving data — the Receiver — is a regular job with no special characteristics. As a consequence, if it was to ever crash, it would be restarted on another executor, continuing the ingestion of data without further trouble.

However, this job is a tad special in the sense that it requires the start of other components to schedule the synchronization of scheduling Streaming jobs with RDD reception from the stream of data. As a consequence, Spark Streaming requires another component to fire up at the beginning of processing.

The Spark Streaming Context

The Spark Streaming Context has as a goal to keep tabs on the creation, filling and processing of DStreams in the Spark cluster. As such, it creates jobs based on the Spark RDDs produced at each interval, and keeps track of the `DStream` lineage.

Indeed, just as Spark RDDs can be dependent on other RDDs, and actually are defined by transformation of the previous ones, the work that a user requires to be done on an RDD *may* depend on the previous RDDs of the same stream. That is not a problem as long as the previous RDDs fit in the memory of the cluster, but it begs the question of whether this is a stable form of processing, especially when considering fault tolerance.

To get a clearer idea of this picture, we are going to have a look at how to create a Streaming Context to host our stream. The simplest way is, in the Spark shell, to wrap a streaming context around the Spark context, which in the shell is available under the name `sc`.

```
1 scala> import org.apache.spark.streaming._
2 import org.apache.spark.streaming._
3
4 scala> val ssc = new StreamingContext(sc, Seconds(2))
5 ssc: org.apache.spark.streaming.StreamingContext =
6       org.apache.spark.streaming.StreamingContext@77331101
```

The Spark Streaming Context is in charge of starting the ingestion process, which is deferred until the `start()` method is called on the Spark Streaming Context.

Let's now declare a `DStream`, which will listen for data over an arbitrary local port. A `DStream` on its own does not do anything. In the same way that RDDs need `actions` to materialize a computation, `DStreams` require the declaration of `output operations` in order to trigger the scheduling of the execution of the `DStream` transformations. In this example, we are going to use the `count` transformation, which counts the number of elements received at each batch interval and we will use `print`, an output operation that outputs a small sample of elements in `DStream` at each batch interval.

```
1 scala> val dstream = ssc.socketTextStream("localhost", 8088)
2 dstream: org.apache.spark.streaming.dstream.ReceiverInputDStream[String] =
3       org.apache.spark.streaming.dstream.SocketInputDStream@3042f1b
4
5 scala> val countStream = dstream.count()
6 countStream: org.apache.spark.streaming.dstream.DStream[Long] =
7       org.apache.spark.streaming.dstream.MappedDStream@255b84a9
8
9
10 scala> countStream.print()
```

Now, in a separate console, we can loop through our file of common last names and send it over a local tcp socket, with a small bash script:

```
1 user@localhost$ { while :; do cat names.txt; sleep 0.05; done; } | netcat -l -p 8088
```

This iterates through the file we have and continuously loops over it, sending it infinitely over the TCP socket. We can then start the Spark context and get the results from our computation:

```
1 scala> ssc.start()
2
3 scala> WARN RandomBlockReplicationPolicy: Expecting 1 replicas with only 0 peer/s.
4 WARN BlockManager: Block input-0-1491775414200 replicated to only 0 peer(s) instead of 1 peers
5 ...
6 \-------------------------------------------
7 Time: 1491775416000 ms
8 \-------------------------------------------
9 1086879
10
11 \-------------------------------------------
12 Time: 1491775420000 ms
13 \-------------------------------------------
14 956881
15
16 \-------------------------------------------
17 Time: 1491775422000 ms
18 \-------------------------------------------
19 510846
20
21 \-------------------------------------------
22 Time: 1491775424000 ms
23 \-------------------------------------------
24 0
25
26 \-------------------------------------------
27 Time: 1491775426000 ms
28 \-------------------------------------------
29 932714
30
31 ssc.stop(stopSparkContext = false)
```

Internal Note: This example requires the small sleep in the data production side in order to avoid overwhelming the NetworkReceiver and causing an immediate crash.

Adding a small delay in the script makes it stable. Playing with the delay let us change the throughput of data sent.

```
1 {while :; do cat names.txt; sleep 0.05 ; done } | netcat -l -p 8088
```

For the record:

```
1 17/04/10 00:05:23 ERROR Utils: Uncaught exception in thread driver-heartbeater
2 java.lang.OutOfMemoryError: GC overhead limit exceeded
```

There are a few points to notice:

- The computation is not started until we `start()` the Spark Streaming Context, which is in charge of setting up the data ingestion needed for the computation. It doesn't stop until we call `stop()` on this same context.
- Even when reading data locally in this example, the data ingestion rates are somewhat irregular, and are a function of many hardware factors, including Spark realizing that it cannot replicate the blocks of ingested data on this toy test setup: a single computer.

One key characteristic of the job that we just defined is that it does not depend on history: we do not try to compute, say, a running count of the number of RDDs we have seen, but rather to display the current count for each batch. When we consider fault tolerance, this is an important distinction: in stateful computations such as a running count, restarting the computaiton after a crash would lose the intermediate count unless all the data collected since the beginning of the Spark Context was saved and could be replayed.

To solve this problem, the Spark Streaming Context is also in charge of checkpointing — that is, to save the intermediate state of stateful `DStream` transformations to disk. The configuration simply consists in specifying a checkpoint directory for the Spark Streaming context to write to:

```
1 scala> ssc.checkpoint("/tmp/examplecheckPoint")
```

Depending on the nature of the application, for a given `DStream` it may be important to configure checkpointing to a specific interval. The value of the checkpoint interval must be a multiple of the batch interval otherwise we will get an error when the streaming context starts. Consider this example, where we use a batch interval of 5 seconds and a checkpoint interval of 12 seconds:

```
1 scala> import org.apache.spark.streaming._
2     import org.apache.spark.streaming._
3
4 scala> val ssc = new StreamingContext(sc, Seconds(5))
5     ssc: org.apache.spark.streaming.StreamingContext =
6       org.apache.spark.streaming.StreamingContext@77331102
7
8 scala> ssc.checkpoint("/tmp/examplecheckPoint")
9
10 scala> val dstream = ssc.socketTextStream("localhost", 8088)
11 dstream: org.apache.spark.streaming.dstream.ReceiverInputDStream[String] =
12       org.apache.spark.streaming.dstream.SocketInputDStream@fcae085
13
14 scala> dstream.checkpoint(Seconds(12))
15 res4: org.apache.spark.streaming.dstream.DStream[String] =
16       org.apache.spark.streaming.dstream.SocketInputDStream@fcae085
17
18 scala> dstream.print()
19
20 scala> ssc.start()
21 java.lang.IllegalArgumentException: requirement failed: The checkpoint interval for
22     SocketInputDStream has been set to  12000 ms which not a multiple of its slide time (5000 ms).
Please set it to a multiple of 5000 ms.
```

Checkpointing consists in saving an intermediate result to disk, and therefore has a cost that you may not want to pay at every batch-interval, so that the Spark Streaming documentation itself advises a checkpointing interval of at least 5 times the batch interval. However, this is of course dependent on the amount of memory you may want to put into processing this data and the size of the input data you will put into each batch: If your system is capable of keeping around many batches of data, you may want to set this value to something higher.

```
1 scala> dstream.checkpoint(Seconds(10))
2 res2: org.apache.spark.streaming.dstream.DStream[String] =
3         org.apache.spark.streaming.dstream.SocketInputDStream@293bb4c
```

The default checkpointing interval consists in checkpointing at the first multiple of the batch interval which is at least 10, so that we have done above the equivalent of what the default behavior would provide. Re-launching a Spark Context with that command turned on shows the automatic persistence done by the Streaming Context — in the last part of `toDebugString`, which shows caching information. Lets observe this behavior in a complete example:

```
1 scala> import org.apache.spark.streaming._
2 import org.apache.spark.streaming._
3
4 scala> val ssc = new StreamingContext(sc, Seconds(5))
5 ssc: org.apache.spark.streaming.StreamingContext =
6       org.apache.spark.streaming.StreamingContext@31f5580b
7
8 scala> ssc.checkpoint("/tmp/examplecheckPoint")
9
10 scala> val dstream = ssc.socketTextStream("localhost", 8088)
11 dstream: org.apache.spark.streaming.dstream.ReceiverInputDStream[String] =
```

```
      org.apache.spark.streaming.dstream.SocketInputDStream@b768e4d

scala> dstream.checkpoint(Seconds(30))
res6: org.apache.spark.streaming.dstream.DStream[String] =
      org.apache.spark.streaming.dstream.SocketInputDStream@b768e4d

scala> val display = dstream.foreachRDD{ (rdd, time) =>
     |   println( s"==========\n${rdd.toDebugString}")
     | }
display: Unit = ()

scala> dstream.count().print()

scala> ssc.start()

==========
(25) BlockRDD[1] at socketTextStream at <console>:32
      [Memory Serialized 1x Replicated]
\-------------------------------------------
Time: 1493665330000 ms
\-------------------------------------------
144384

==========
(25) BlockRDD[427] at socketTextStream at <console>:32
      [Memory Serialized 1x Replicated]
\-------------------------------------------
Time: 1493665335000 ms
\-------------------------------------------
144384
```

Representing regular updates to a fixed window of data

Sometimes, however, trying to compute something on a stream of data since the origin of times (as in the running count we evoked above) is not what we are looking for. We would rather have some information for a fixed window of time, longer than a batch, but still relatively fresh.

This allows us to answer questions such as “how many distinct names have I seen in the last hour ?”, by creating a sliding window over the data coming on the wire. The logic of that windowing operation is a logic of grouping over Spark’s batches, and it depends on two quantities:

the window length

> This dictates how many batches, total, there should be in the amount of data the computation is performed on, but using the more intuitive measure of time. Because this works by grouping a certain number of batches, this value should be a multiple of the batch interval.

the slide interval

> This indicates how frequently the window should be refreshed with new elements. It is also a number of batches, set up as a time interval.

Let’s assume that instead of names, we are looking at web site logs in our stream, and that from these logs, we are reconstructing user sessions and computing some statistics on these sessions. We assume that we have a batch interval of 5 minutes minute `new StreamingContext(sc, Minutes(5))`. How should we build our output stream of results ?

- If we want summaries of the last hour of web site traffic, updated every interval, we can use a window of one hour and a sliding interval of 5 minutes. The call would look like `sessions.window(Minutes(60), Minutes(5))`.
- If we want summaries of web site traffic over a particular calendar day, we can use a window of 24 hours, and a slide interval of 24 hours as well. This is the result we will obtain when calling `sessions.window(Minutes(60*24), Minutes(60*24))`.
- If we want summaries of the last day of web site traffic, updated every hour, we can use a window of 24 hours, but this time we will specify a sliding interval of 60 minutes. The call would look like `sessions.window(Minutes(60*24), Minutes(60))`.

Note that the window duration and interval **must be** multiples of the base streaming interval. Otherwise we will encounter an error at the moment the streaming job is started:

```
1 scala> val sessionsByOddWindow = sessions.window(Minutes(21), Minutes(21))
2 scala> ssc.start()
3 java.lang.Exception:
4   The slide duration of windowed `DStream` (1260000 ms)
5   must be a multiple of the slide duration of parent `DStream` (300000 ms)
```

The logic of grouping batches for windowed DStreams can be represented as follows:

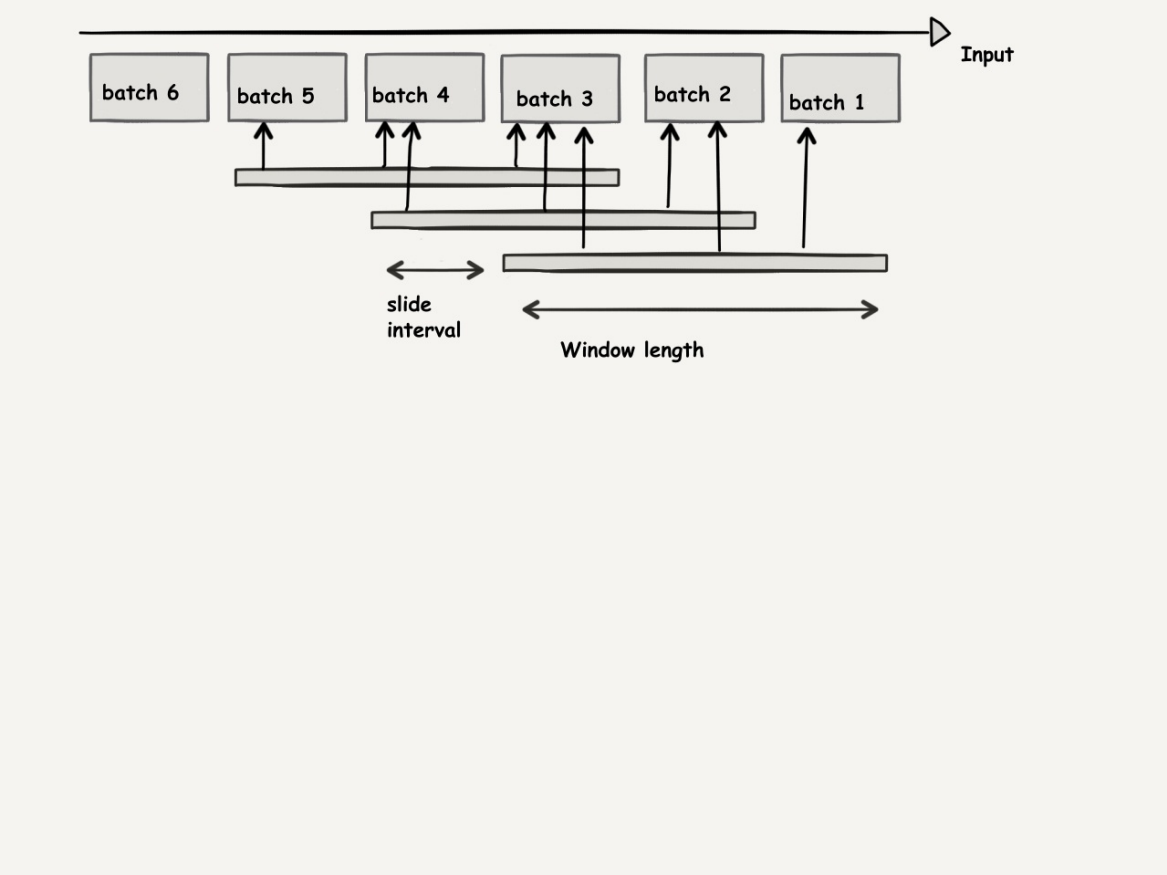


Figure 2-6. Spark Streaming Windowing : slide and window interval

We will come back to how Spark Streaming lets us compute efficiently on windowed streams in the next chapter.

The Receiver Model

Finally, in these examples, we have witnessed the StreamingContext run some background tasks that were not specified by the user:

- checkpointing periodically writes the Stream's summary on the disk
- the Receiver consumes data by having a connection to the data source, and outputting one block on every block interval, that are grouped into an RDD on every batch interval.

The fact that those are *built-in* jobs has consequences, in particular for the resource usage and parallelism accounting of the Receiver. To execute those jobs, the Receiver will consume one core on an executor regardless of the amount of work it has to do.

As a consequence, using a single streaming receiver will result in data ingestion done in sequence by a single core in an executor. We'll see how we can increase data ingestion parallelism using the receiver model in a minute.

A frequent beginner's mistake is to attempt to test Spark Streaming by either allocating a single core in local mode (`--master "local[1]"`) or by launching it in a virtual machine with a single core: the consumption of data by the receiver would block any Spark processing on the same machine, resulting in a streaming job that does not progress.

Finally, a constrain of this receiver execution model is the total parallelism you can achieve. The base unit of replication for Spark is a block: any block can be on one or several machines (up to the persistence level indicated in the configuration, so at most two by default), and it is only when a block has reached that persistence level (exactly two) that it can be processed.

It is only when an RDD has every block replicated that it can be taken into account for job scheduling. The consequence is that the units of parallelism are built in the relationship between batch interval and block interval. An RDD contains partitions which are operated on in parallel, as we have mentioned in the first chapter. For an RDD obtained from Spark Streaming, that partitioning corresponds to the number of blocks. So that the level of parallelism we can expect for an RDD obtained for a single receiver is exactly the ratio of batch interval to block interval:

\[{# of partitions} = {batch interval} / {block interval} \]

As usual for Spark, a good rule of thumb is to have enough parallelism for a job to run fast, so that we have around two to three times as many partitions as we have available executors. We will see more in depth how to compute that number of available resources later, as we go into the details of which cluster mode we are using. But let's assume we have managed to compute the number of cores available to Spark for computation. And let's assume we want high parallelism, because we know that we have small tasks with respect to the computing power we have available, so that we don't want to lose too much time with scheduling those short-lived tasks. Say we're therefore going to aim for three times as many tasks as we have available cores. Then we may want to set the following:

$$blockinterval = batchinterval / (3 * sparkcores)$$

Receiver parallelism

A simple way to achieve receiver parallelism is to declare more Dstreams. Each will be attached to its own consumer (and therefore each with its own core consumption) on the cluster. The `DStream` operation `union` allows us to then merge those streams, ensuring we produce a single pipeline of data from our various input streams.

Let's assume we create `DStream`s in parallel an put them in a sequence:

```
1 val inputDstreams: Seq[DStream[(K,V)]] = Seq.fill(parallelism: Int) {
2 ... // the input Stream creation function
3 }
4 val joinedStream  = ssc.union(inputDstreams)
```

In this way we can exploit a Receiver parallelism, here represented by the concurrently-created `Dstream` 's numbering exactly `parallelism` . This has consequences in the number of available cores in our cluster. Let's say that in the previous example we have 9 available cores for Spark, meaning we really have 10 cores, but that one was consumed by the receiver of our `DStream`.

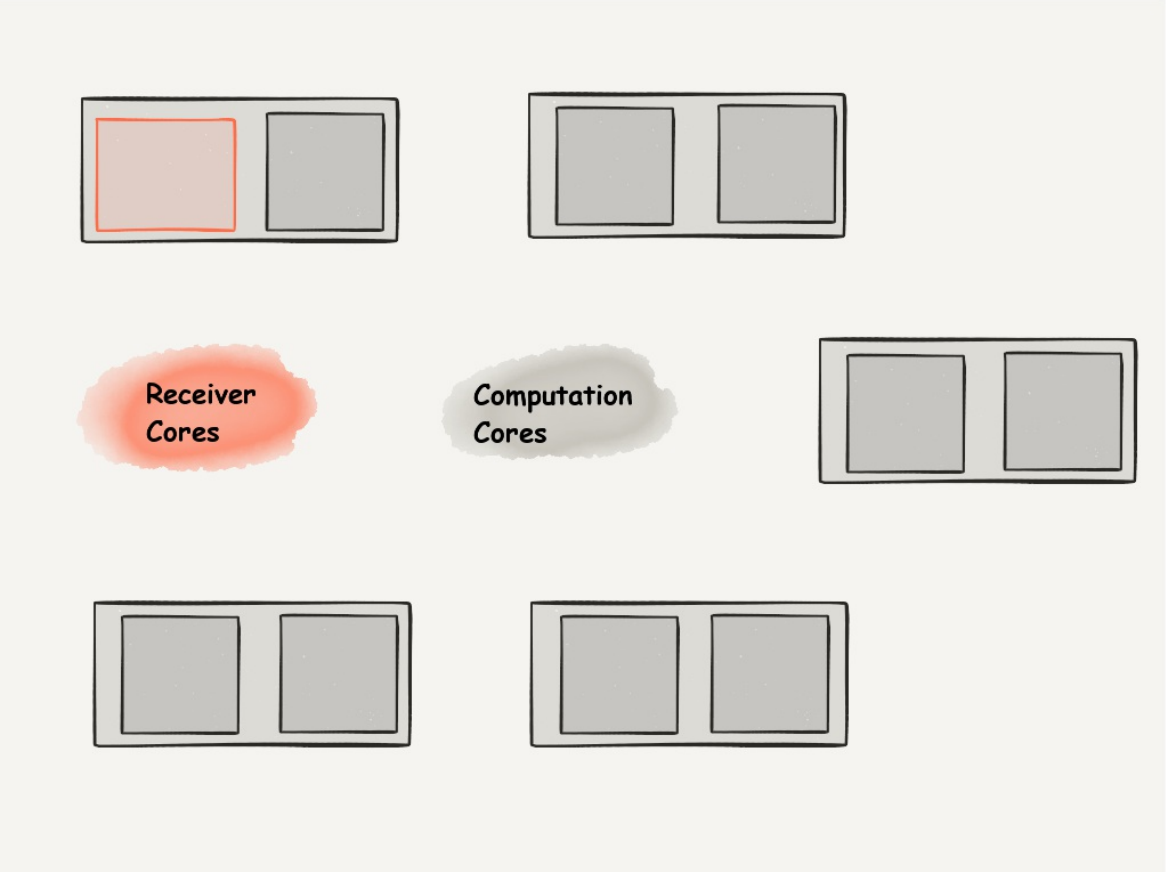


Figure 2-7. Receiver core consumption : with one `DStream`

We now want to ingest data in parallel with four Receivers. Our available cores for computation

will then fall to 6, since we are using two additional cores for parallel ingestion.

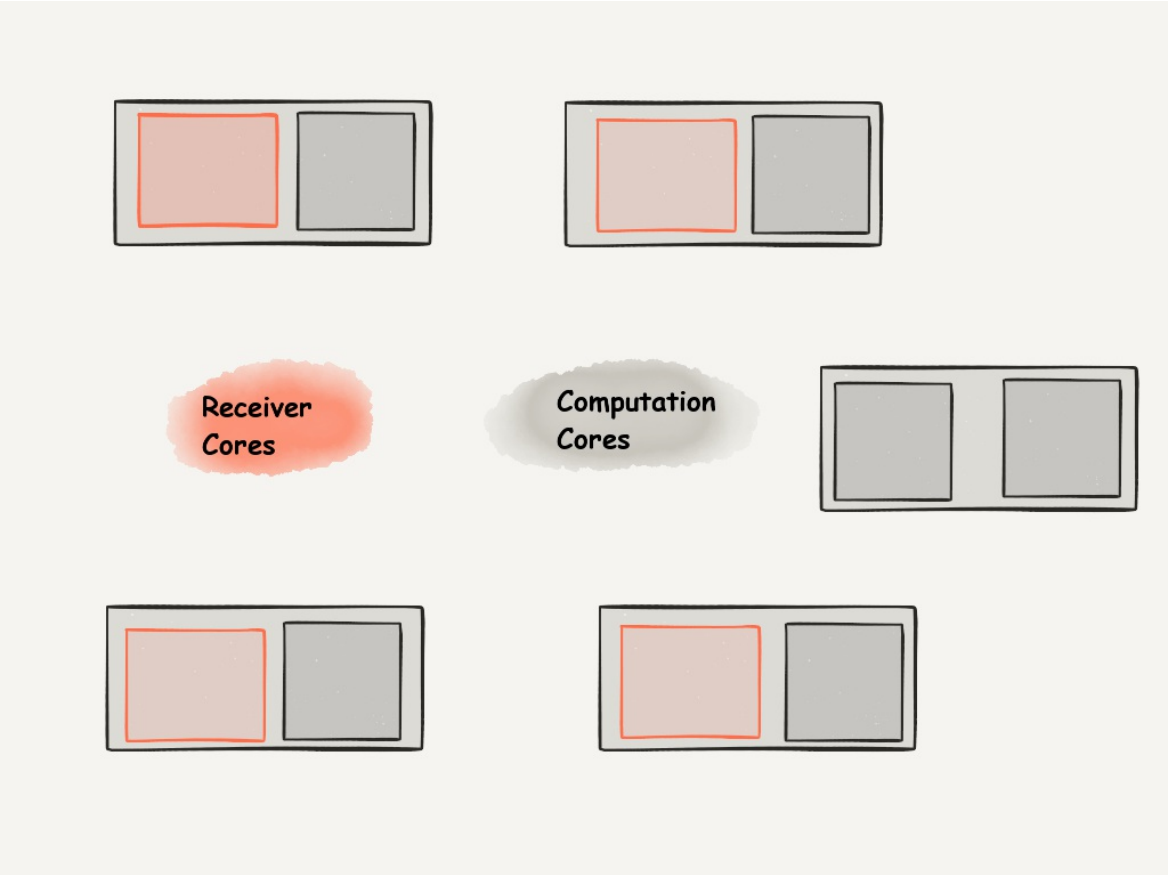


Figure 2-8. Receiver core consumption : with four `DStreams`

The batch interval is fixed by the needs of the analysis, and remains the same. What should the block interval be ? Well, four DStreams ingesting in parallel will necessarily create four times as many blocks per block interval as one single `DStream` would. So, with the same block interval, the number of partitions of the unionized `DStream` will be four times what it was in the original case. So we can't use the same block interval. We should use

$$\text{block interval} = (4 * \text{batch interval}) / (3 * \text{spark cores})$$

Since we want *at least* three partitions, we will round this number *down* to the nearest millisecond.

Generalizing, with an arbitrary set of characteristics, we should use:

\[\text{block interval} = (\text{# of receivers} * \text{batch interval}) / (\text{# of partitions per core} * \text{# of spark cores}) \]

Where the total number of cores used in the system is:

$$ \text{total system cores} = \text{# of receivers} + \text{# of spark cores} $$

Conclusion

So far, we have seen a brief primer on Spark Streaming, along with the fundamentals of how it treats stream processing:

- streams are aggregated data seen over time on a data source. On every block interval, a new partition of data is produced and replicated. In every batch interval (a multiple of the block interval), the resulting data is assembled into an RDD and a job can be scheduled on it.

- scheduling is done by user-defined functions in a script, but can also be the byproduct of some built-in functionality (e.g. checkpointing). The scheduling itself has a fixed core.

- The most common way of creating a Distributed Stream (`DStream`) is the Receiver model, which creates a job connecting to the input source on an executor, consuming one core. Parallelism can be increased by creating several `DStream`s. We should call `streamingContext.union` on the resulting collection of `DStream`s to process the result as a single stream.

- In some stateful computations, where the result depend on previous contents of the stream, it is important to checkpoint the corresponding DStream occasionally so as not to let the lineage for intermediate RDDs grow indefinitely, hindering recovery in case of a fault.

- Finally, Spark can offer a windowed view of a `DStream`, that groups the content of the Stream by a given length of time, possibly refreshed at regular intervals, in which case it becomes a sliding window.

In the next chapter, we'll study how to assemble those basics into a coherent application and deliver our first Spark Streaming application.

Bibliography

- [Das2015] Das, T; Zaharia, M; Wendell, P. *Diving into Apache Spark Streaming's Execution Model* Databricks Engineering Blog. July 30, 2015. URL

- [Greenberg2015] David Greenberg. *Building Applications on Mesos : Leveraging Resilient, Scalable, and Distributed Systems.* O'Reilly Media. 2015 ISBN 1-4919-2652-X

- [Valiant1990] Valiant, L.G. *Bulk-synchronous parallel computers*, Communications of the ACM 33:8, August 199O *http://web.mit.edu/6.976/www/handout/valiant2.pdf*

- [Vavilapalli2013] Vavilapalli et al. *Apache Hadoop YARN: yet another resource negotiator.* SOCC'13, ACM, NY, 2013 doi:10.1145/2523616.2523633

- [White2010] White, T. *Hadoop: The Definitive Guide* O'Reilly Media. 2010. ISBN-13 9781449396893

[1] Note the `cache` command is a special case of the `persist` command, with persistence level `MEMORY_ONLY` . To know more about persistence and persistence levels, consult the Spark documentation

[2] There is a standalone cluster-mode in which the driver runs on as one of the executor machines for standalone, but it is not supported officially.

[3] YARN containers are not containers in the Linux sense. They are processes, which get called containers as a result of the YARN memory allocation process.

Chapter 3. Streaming application design

Starting with an example : Twitter analysis

Until now, we have developed mostly generalities on the functioning of a `DStream` and snippets of Scala code leading to stream usage. Now has come the time to put them together in an application. As we progress further, we will then study how to run this application most efficiently.

The Spark Notebook

Until now, we have created simple examples in the Spark shell. Beyond interactive shells, there is another way of approaching the development of Spark scripts, and that is interactive notebooks.

So-called notebooks are web applications tied to a REPL (Read-Eval-Print Loop) — otherwise known as an interpreter —, They offer the ability to author code in an interactive web-based editor. The code can be immediately executed and the results are displayed back on the page. In contrast with the spark-shell, previoulsy executed commands become part of a single page, which can be read as a document or executed as a program, resulting in an improved user experience. This interactive process creates a fast feedback loop that makes notebooks an ideal environment for learning and exploration. The most famous modern notebook is probably the iPython notebook for the Python programming language, but their legacy is in the notebook interfaces of scientific software such as Maple and Mathematica, that predate their web incarnations.

We are going to use the Spark-Notebook, one open-source notebook application tailored for Apache Spark developed by the founders of Data Fellas, but which receives contributions from a small community of Spark developers. [1].

One of the easiest ways to install and run the Spark Notebook is to go to http://spark-notebook.io/ and download one of the precompiled versions of the notebook. Data Fellas provides builds of the notebook for an amazing matrix of versions of both Scala, Spark and Hadoop, including even Docker images which will work even if you do not have the usual JVM prerequisites installed in advance.

Once you have downloaded a binary build of the Spark notebook, the next steps are simple:

- Extract the file somewhere convenient.
- Open a terminal/command window.
- Change to the root directory of the expanded distribution.
- Execute the command `bin/spark-notebook` (*NIX) or `bin\spark-notebook` (Windows).
- Open your browser to `http://localhost:9000`.

There, you can select the tab clusters and click on the local template to launch a (virtual) Spark cluster in local mode, i.e. on your machine.

You should then be greeted with the following prompt:

Figure 3-1. The prompt of the Spark Notebook

To check that you have indeed managed to get the local cluster you can type the command `sparkContext.getConf.toDebugString`. The result should look somewhat like this:

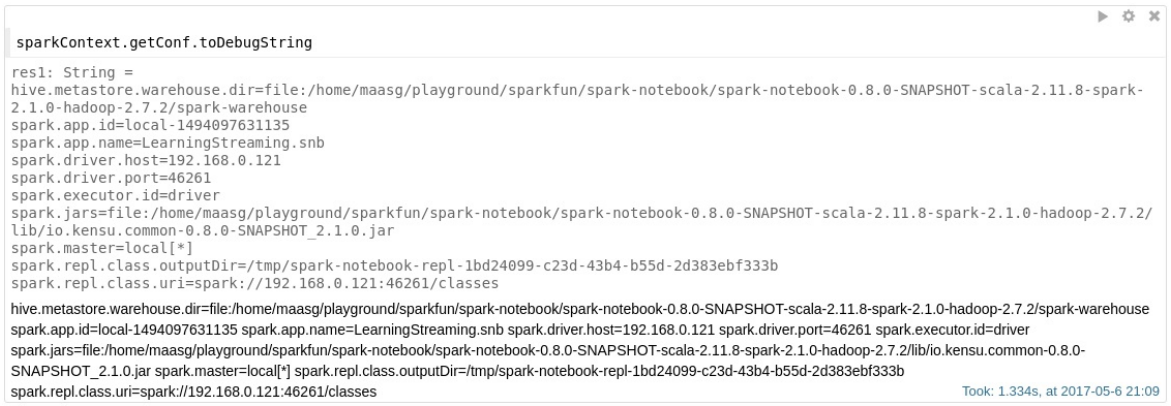


Figure 3-2. The use of the `toDebugString` debugging function

We are going to use the Spark Notebook in our exercises for a few reasons:

- with the interactive notebook, we dramatically shorten the write-compile-submit-execute cycle, resulting in an improved learning experience.
- the Spark Notebook has options for editing the available repositories and jar dependencies in the notebook, which re-launches the Spark session in the background, letting the user go through this procedure seamlessly, without leaving his browser. Since we are going to proceed with a running example that grows in complexity, the notebook should avoid a few tedious restarts.
- having the code in one executable "page" lets us indicate to the reader only the modifications we need to apply to prior code. Since we are going to run Spark Streaming applications, each new execution is going to require that the user restart her cluster. This is again helped by the notebook.

Note

Moreover, the notebook is sparse in its output and does not add any fancy display when not required. While we will display the widgets of the notebook to give the reader an idea of what to expect when following along the examples, when the output is simply textual, we will remain content with displaying it in-line as code blocks.

Creating a Streaming Application

To start with our first Streaming application, we need a skeleton — the "Hello World" of sequential programs.

Any Streaming application in Spark needs to do four things:

- Create a **Spark Streaming Context**
- Define one or several **`DStreamS`** from **Data Sources** or other `DStreamS`
- Define one or more output operations to materialize the results of these `DStream` operations.
- **Start** the Spark Streaming Context

The behavior of the job is defined between the moment that the instance for the Streaming context is defined and the moment it is started. In that sense, the context's manipulation before it starts defines the scaffolding for the streaming application, that will be its behavior and execution for the duration of the streaming application.

It is during this definition phase that all the `DStreams` and their transformations will be defined, "setting in stone" in a manner of speaking, the behavior of the Spark Streaming application. In particular, a `DStream` cannot be added of changed after the Spark Streaming Context is started.

We are going to first create a Streaming Context. As we have seen, in Spark Streaming, the Streaming Context is actor responsible of putting the data ingestion components into place for every initial `DStream` created in the streaming application. A Spark Context is provided at the launch of the notebook application, just as it was in the case of the Spark Shell. To create the Streaming Context, we only require a valid Spark Context and a batch interval as an argument. Therefore, we can use the provided `sparkContext` and an arbitrary streaming interval of our choice.

Creating a Stream

A simple way of creating a stream has been given in ???. For this exercise, we are going to create a more complex example, using one of the libraries that Spark Streaming integrates as a source of data: a Twitter stream.

For this, we are going to need to set a few properties to access the Twitter Streaming API. This API offers a sampled streaming view of the tweets going on around the world, using a subscription based on some keywords or specific users. To gain access to this API we need to authenticate.

This authentication is done using OAuth, a protocol by which the user has to provide credentials, namely a consumer token and secret, to enter in an OAuth conversation with Twitter. Once this is done, the user can enter in a per-application conversation to get specific credentials for a single application. The steps of this negotiation are represented in the following diagram:

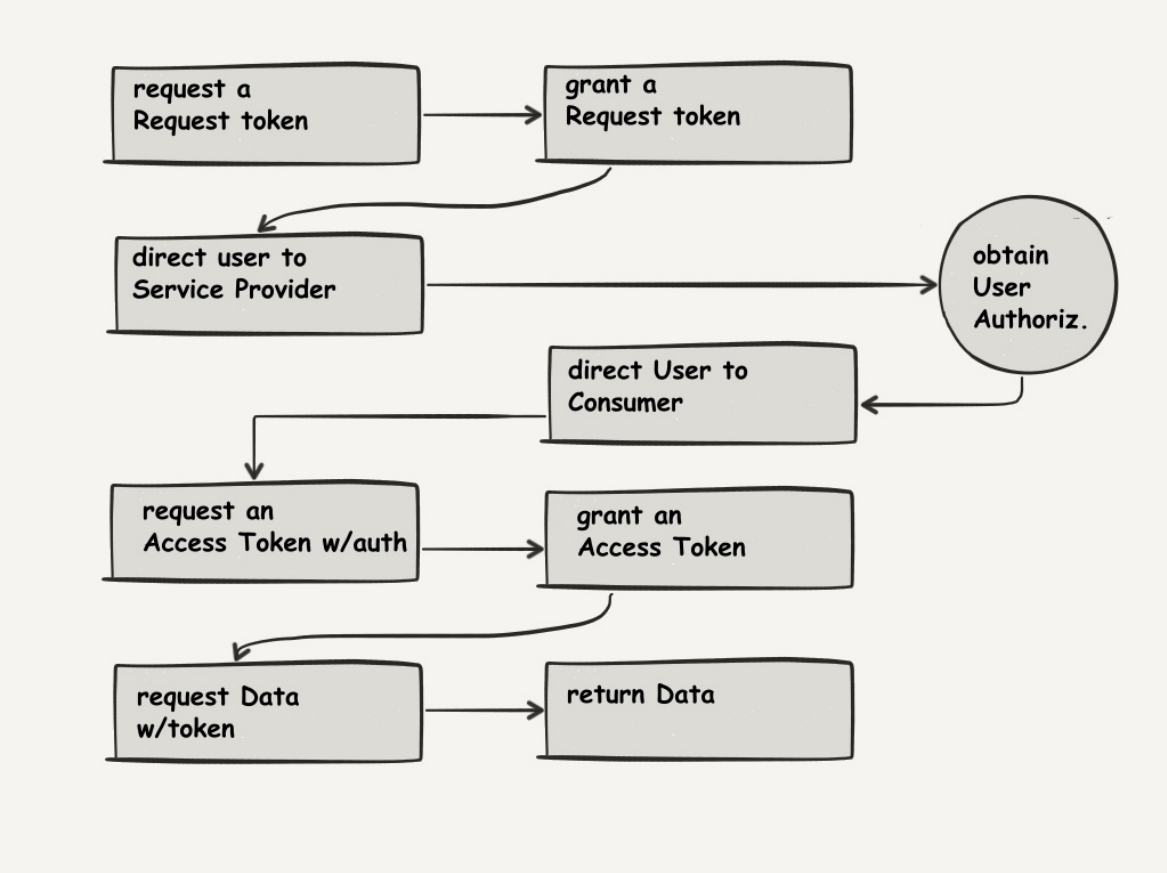


Figure 3-3. The OAuth protocol

You should obtain those credentials now using your own Twitter account at this link:

```
https://apps.twitter.com/
```

To learn more about Twitter's OAuth API, consult the documentation at:

```
https://dev.twitter.com/oauth/overview
```

Note that as a consequence of the back-and forth — the last part of which is application-specific — OAuth requires two sets of tokens, which we will need to pass to the Twitter API as we create a Twitter DStream.

More about getting your Twitter OAuth credentials

You can find a detailed explanation (including screen shots) of how to set up the necessary Twitter OAuth key pairs in the Databricks Advanced Spark Training Resources. Twitter, of course, has their own documentation.

Once you have obtained the two pairs of credentials, we can proceed to work with our notebook.

First, we need to add the packages for Spark Streaming and `spark-streaming-twitter` using the Notebook metadata support. The Spark Notebook metadata is JSON based. We add the extra packages as a JSON array:

Go to `Menu → Edit → Edit Notebook Metadata`

```
"customDeps": [
    "org.apache.spark %% spark-streaming @ 2.1.0",
    "org.apache.bahir %% spark-streaming-twitter % 2.1.0"
  ],
```

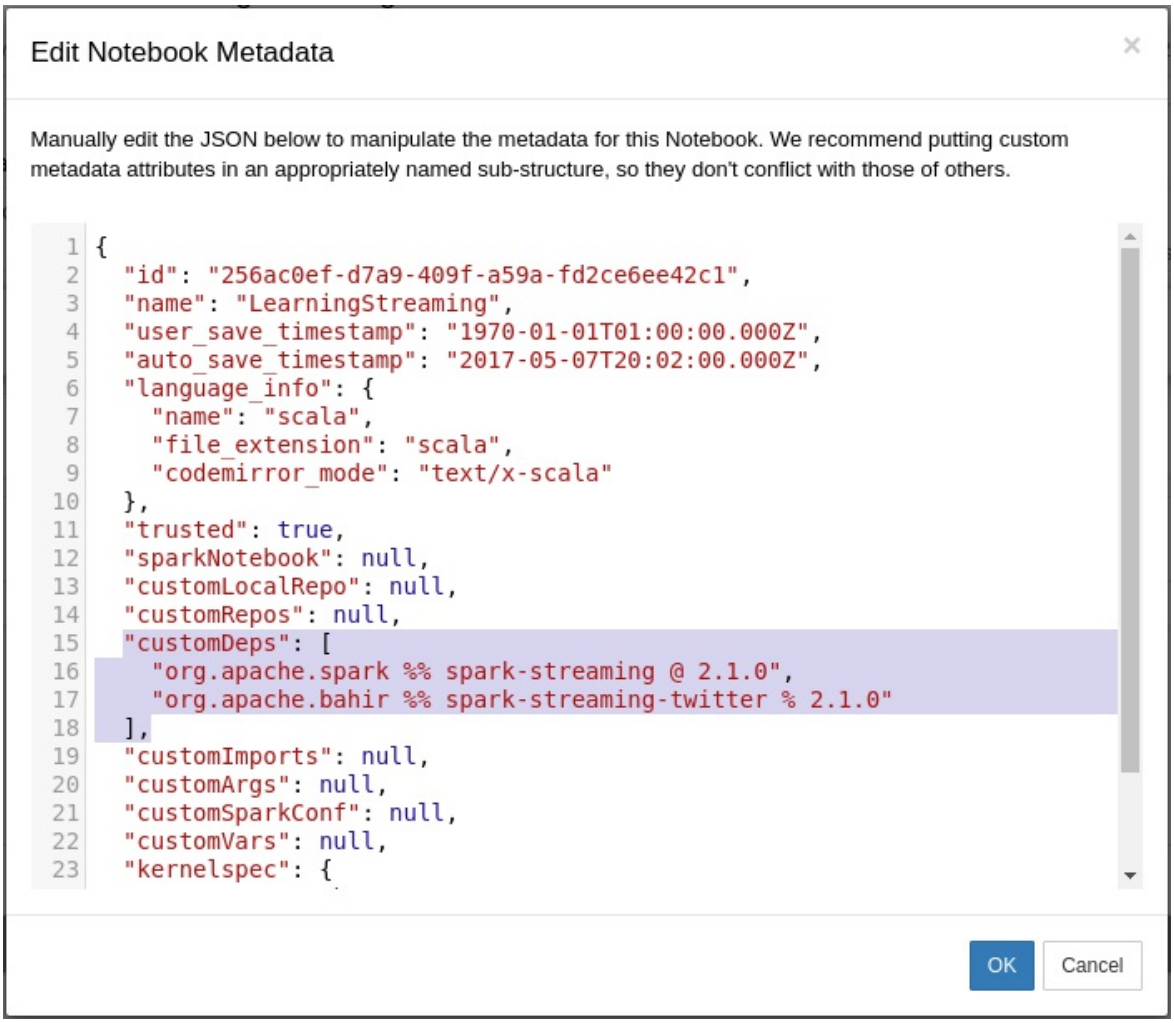


Figure 3-4. Adding Imports in the metadata

After saving the metadata, close the notebook and open it again. This ensures that the new dependencies are loaded. To test that our additional dependencies have become available, we can already issue the imports for the packages we need:

```
1 import org.apache.spark.streaming.{Seconds, StreamingContext}
2 import org.apache.spark.SparkContext._
3 import org.apache.spark.streaming.twitter._
```

Running this cell (`Ctrl-Enter`) should succeed and output a simple acknowledgement of the imports:

[[Successful imports in a cell]] .Successful imports in a cell image::images/SparkNotebookImports.png[]

Tip

The Spark Notebook logs are available on the Javascript console of the browser [`F12` on Chrome].

They can be very helpful when trying to debug configuration and runtime issues.

Note

The package supporting Twitter streaming is hosted at Apache Bahir link:http://bahir.apache.org/ Apache Bahir provides extensions to multiple distributed analytic platforms, extending their reach with a diversity of streaming connectors and SQL data sources. Specifically for Spark Streaming, Apache Bahir hosts the following packages: * Spark `DStream` connector for Akka * Spark `DStream` connector for MQTT * Spark `DStream` connector for Twitter * Spark `DStream` connector for ZeroMQ

Then we prepare the addition of our Twitter credentials:

```
1 def configureTwitterCredentials(consumerKey: String,
2                                   consumerSecret: String,
3                                 accessToken: String,
4                               accessTokenSecret: String) {
5   val configs = Seq("consumerKey" -> consumerKey,
6                     "consumerSecret" -> consumerSecret,
7                     "accessToken" -> accessToken,
8                     "accessTokenSecret" -> accessTokenSecret).toMap
9   val trimmedConfigs = configs.mapValues(_.trim)
10   configs.foreach{ case(key, value) =>
11     require(value.nonEmpty, s"""Error setting authentication - value for  $key  not set""")
12     val fullKey = "twitter4j.oauth." + key
13     System.setProperty(fullKey, value)    }
14   }
15
16 > configureTwitterCredentials: (consumerKey: String, consumerSecret: String,
17     accessToken: String, accessTokenSecret: String)Unit
```

We can then set up our credentials with a correct call of the `configureTwitterCredentials` function, with your personal Twitter credentials. Note that we are here simply setting system properties in the context to the spark driver, that will be picked up by the spark-streaming twitter utilities. Those properties (e.g. `twitter4j.oauth.apiSecret`) are set up with a precise name in a namespace expected by this utility.

We can now start creating a Twitter Stream, which, using the Twitter Streaming public API through twitter4j, creates a filter query. We will filter on the single word “spark”.

Note that for this task, we also need a Spark Streaming Context. The Spark Notebook only provides a configured and started Spark Context at launch. We need to create our Streaming Context programmatically.

```
1 val ssc = new StreamingContext(sparkContext, Seconds(5))
2
3 > ssc: org.apache.spark.streaming.StreamingContext =
org.apache.spark.streaming.StreamingContext@ef1a9b1
```

We choose a batch interval of 5 seconds, to make sure we get enough tweets in every batch to justify distribution.

```
1 val filters = Array("music")
2 val twitterStream = TwitterUtils.createStream(ssc, None, filters)
```

We can then create our stream using the API provided in the `TwitterUtils` module of Spark. The creation takes as argument a spark streaming context, a `twitter4j` authentication, that we leave

here as `None` so that the utility builds it from the system properties that we have set at the call to `configureTwitterCredentials` above, and an array of `String`s, that represent the filters used to parametrize our Twitter search query. We will subscribe for “music” to see what are the current trends are.

Transformations

Once this is done, we will have access to a `twitter4j` object which offers an interface to the the JSON-formatted record representing a tweet. Let's make sure we can print a few elements of the Tweet, such as analyzing its text.

```
1 import StreamingContext._
2 val hashTags = twitterStream.flatMap(status => status.getText.split(
3     " ").filter(_.startsWith("#")))
4
5 val topCounts60 = hashTags.map((_, 1)).reduceByKeyAndWindow(_ + _, Seconds(60))
6                          .map{case (topic, count) => (count, topic)}
7                          .transform(_.sortByKey(false))
```

In this code, we are:

- working on the twitter stream defined above to extract the part of the tweet which contains the actual content, or "status update" in the language of Twitter's user interface. That's the role of the `flatMap` operation, which calls `status.getText` on each tweet of every batch. The output is a `String`
- We then split this status into words and select only those which start with the hash tag character (#).
- Then, in a manner very similar to the word count we evocated above, we attach the number *1* to each such tag, to reflect that each encountered tag is one occurrence. We then sum the hash tags occurrences we have so marked with a specific time-based discipline based on a window of data, such that we return the number of occurrences of each hashtag we meet over the last minutes.
- We then re-key the stream of hash tags with their count to reflect that we are now interested in displaying them by count and sort the corresponding `RDD` in descending order (the `false` argument to `sortByKey`).

Note

Getting into the details of the window operation semantics of our count of hashtags is beyond the scope of this chapter: we'll visit this subject in detail in [Link to Come].

You should note that running this operation per se — for example as a spark-notebook cell — does not trigger any output. If you remember our explanation of transformations, it does not trigger any computation either, as the computation is entirely lazy.

Lazy computation, as already explained in a more general context in the previous chapter, is a computation where the moment where a value is described and the moment where the value is computed are de-coupled. For a lazy computation, the moment when a definition is processed and actually leads to the production of a reduced value is the first moment when that value is actually used or accessed.

The opposite of lazy computation is eager evaluation, the more common paradigm in programming languages today. In that paradigm, all variable definitions trigger an immediate computation at the declaration point.

In the above short example, all the definitions we have written are transformations, meaning they produce `DStream`s from other `DStream`s and as such are entirely lazy.

We will see how to get an output from our definition — namely using an output operation — in a moment.

We should highlight that this snippet presents a mix of `RDD` and `DStream` operations, which deserve particular attention to understand them: At its most basic, the transformations of this example exhibit stateless functional programming: the `RDD`s of Spark are not modified by the user. Every function call in the succession of operations of our example takes as an argument the collection on the left of the dot character, and reconstitutes a different collection as the output.

It is also important to notice whether a function operates on a `DStream` or an `RDD` — an `RDD` operation acts on a collection once, whereas a `DStream` operation acts on data over time, for each `RDD` read over the input stream at each streaming interval.

The following table summarizes the details of this operations.

operation	line	input type	output type	spark category
`flatMap`	2	`DStream[String]`	`DStream[String]`	transformation
`filter`	3	`DStream[String]`	`DStream[String]`	transformation
`map`	5	`DStream[String]`	`DStream[(String, Int)]`	transformation
`reduceByKeyAndWindow`	5	`DStream[(String, Int)]`	`DStream[(String, Int)]`	transformation
`map`	6	`DStream[(String, Int)]`	`DStream[(Int, String)]`	transformation
`transform`	7	`DStream[(Int, String)]`	`DStream[(Int, String)]`	transformation
`sortByKey`	7	`RDD[(Int, String)]`	`RDD[(Int, String)]`	transformation

An intrinsic consequence of the use of these successive operations is that the development of a program with Spark Streaming involves:

- knowing the `DStream` and `RDD` API well,
- being comfortable with expressing these transformations,
- knowing how to use intermediate anonymous functions, since they involve mostly higher order functions.

We'll go in detail over these points later, since they underline central elements in stream processing, but in the meantime, this example lets us notice another point, which is that data flow is often expressed using variables.

The "chain" of successive stream transformations, with each new one leading to a new representation of the data can, of course, be "broken" for readability and code organization reasons. It is often unwieldy to read a dozen successive transformations and understand what they do at each point of the chain, so that intermediary functions assigned to properly named variables are often useful to break down the complexity.

And yet, a dozen transformations are seldom needed in a single stream. Most often, what dictates the breakdown of operations is the reuse of intermediary variables. In our case, `hashtags` and `topCounts60` are not just there to promote the clarity of definitions, but they also let us reuse `hashtags` in another definition, to perhaps reuse a different part of the information found in those tweets.

Actions and Dataflow

Let us now observe our top count. For that, we need an action which lets us display the results of the analysis and trigger computation.

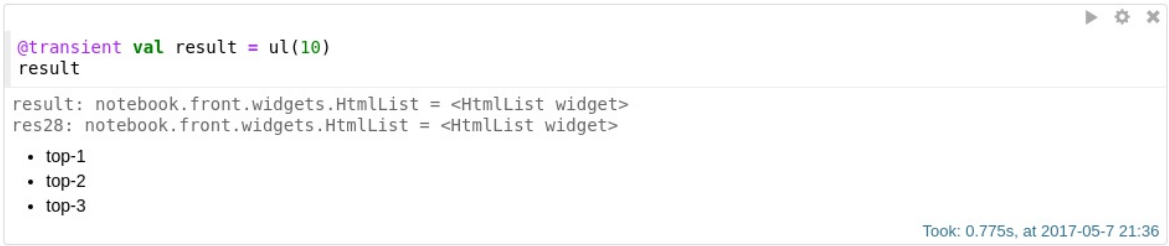
```
1 @transient val result = ul(10)
2 result
```

The `ul` call creates a widget proper to the Spark-notebook, which displays and updates an array of `String`s of a fixed size.

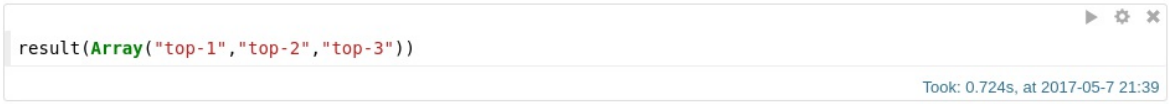
Tip

The `@transient` keyword indicates that this object should not be serialized in a closure. This prevents common Spark serialization errors when running this code.

Finally, we can test the widget behavior by calling `result` on a suitable array. You should witness:



(optional) Use some static test data to test the behaviour of the `Unordered List` widget



Caution

Note that here we are evaluating the variable result in the notebook as an additional statement. This is a convention of the Spark notebook which will not simply display the output element contained in its cells. You may want the display (at evaluation) to occur at a different point than where the assignation of a widget occurs, and if the display was bundled with the definition, it would introduce too much of an inflexible layout, especially when dealing with a cell that defines many widgets.

Note

If you are not following this at home with the Spark notebook, but rather using something akin to the spark-shell, you can simply look into calling the `print` function of the `DStream` type.

You will note, however, that this print function does not display every element of the stream you

call it on. That is because there is potentially infinitely many elements in this Stream, and such a display may be extremely costly in resources if these elements are being read in the Stream at a high throughput.

The `print` function will by default print the first 10 elements of your stream.

Let's now trigger the actual printing of our most popular hashtags. Note that in the snippet below, the call to `result` could perfectly be replaced by a call to `println` — the only consequence is in the position and formatting of the output.

```
1 topCounts60.foreachRDD(rdd => {
2   val topList = rdd.take(10).toList
3   val r = topList.map{case (count, tag) => s"$tag: $count"}
4   result(r)
5 })
```

The result of this execution is the printing of the 10 most popular hashtags in the last minute — note that even if we have batches of five seconds, the `reduceByKeyAndWindow` operation aggregates these results every minute, to display a coherent set of messages — the Twitter stream API is indeed sampled from the real time tweets emitted worldwide, and it may be necessary to wait a little bit to get meaningful results out of this stream.

Expressing a Dataflow

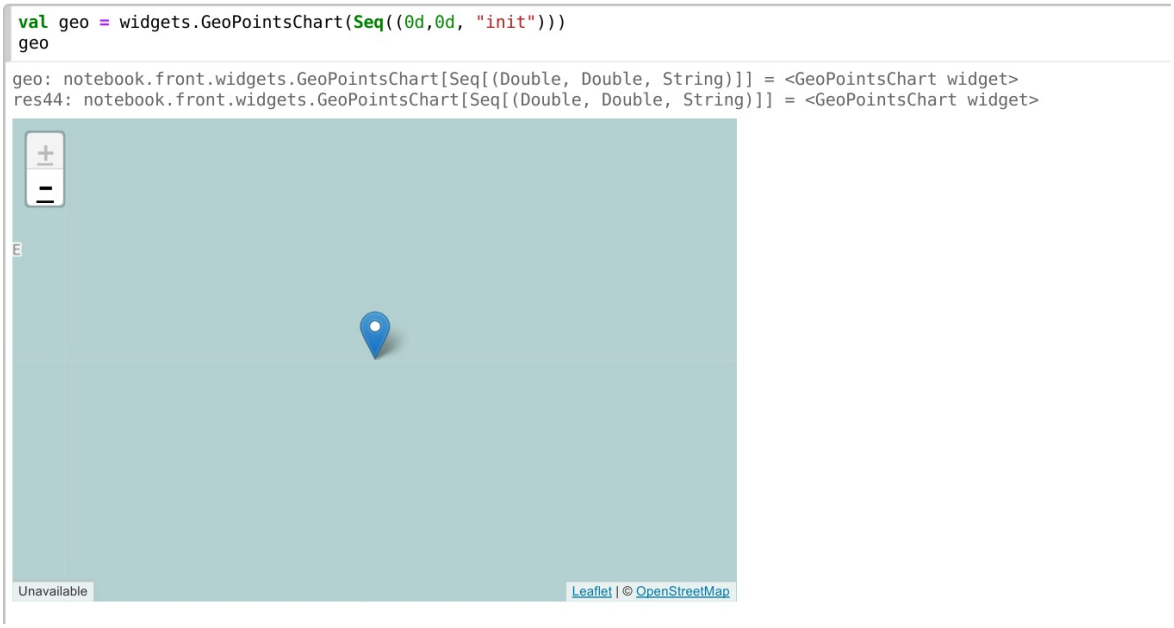
Another visualization that is proper to the Spark notebook is the link with geographic data. In particular, some tweets carry location information, and with the right visualization capabilities, it is possible to plot them on a map as they happen. The Spark notebook makes this easy. Similar functionality could be accessible to any scala or java application interfacing with Spark, given the right libraries and front-end.

```
1 val geo = widgets.GeoPointsChart(Seq((0d,0d, "init")))
2 geo
```

The `GeoPointsChart` helps us display points one by one, as they are fed to the widget in the shape of their WGS84 (modern GPS) coordinates. We can do this, once again, with a call to `foreachRDD`.

```
1 twitterStream .window(Seconds(300), Seconds(15))
2               . filter{ s =>
3                 s.getGeoLocation() != null
4               }
5               .map{s =>
6               (s.getGeoLocation().getLatitude(),
7                s.getGeoLocation().getLongitude(),
8                s.getText())}
9               .foreachRDD{rdd =>
10                geo.applyOn(rdd.take(100))
11               }
```

There is some content to unpack in this example: note how we start by creating a windowed stream — a subject we will come back to in a moment — to get a significant aggregate of our tweet stream. The main point we want to insist on in this example, for now, is that we did not start our Spark context yet, so that the output from the widget is not fixed at any point of emission of a tweet. It is simply in the middle of the ocean, at coordinates (0, 0).



Note also that we did not define our display from the top tweets we had met in the last 300 seconds : we in fact want to display the location of all the tweets of the last minute, whether they had a hashtag or not. To do that, we reused the `twitterStream` defined above. This is perfectly valid since `DStream` elements are *immutable*: operations on a `DStream` do not modify its elements in any way.

As mentioned above, this is possible because forks in the data flow of Spark Streaming are permitted at every point there is a name given to a `DStream` : while the `DStream` definition of `topCounts60` could have directly followed the call to `TwitterUtils.createStream`, it would not have given us access to the `twitterStream`.

Starting the Spark Streaming Context

Until now, we have registered Streams with the Spark Context, including the transformations that operate on them as well as the actions that generate some display. But we haven't triggered any output from the system, since we have not started the Spark Streaming Context yet.

The start of the Streaming Context is what will trigger the jobs to be dispatched on the executors of the cluster, according to what plan of jobs and functions are left to execute. In particular, in the common Receiver model, it is at that stage that the Receiver is made into a job ad dispatched on the cluster.

As a consequence, it is not possible to define more `DStream` elements after the point of your program at which the Spark Streaming Context is started. This makes sense: the starting of the `Receiver` of the corresponding Stream would not be handled correctly if that was the case.

Let us start the context:

```
1 ssc.start()
```

We can now witness the display of the hashtags of the Stream:

We will come back later to the important point of how to stop Spark Streaming once the context is started. For this example using the notebook, we can use `ssc.stop(stopSparkContext = false, stopGracefully = true)`. This stops the streaming job gracefully, and leaves the underlying `sparkContext` active, so that we can further use the notebook. If using `spark-shell`, just send a `SIGTERM` to the shell (using Ctrl-C).

Summary

In this first example, we have thus seen the basic structure of a Spark Streaming application:

- the creation of a Spark Streaming Context, which opens the way to
- the definition of base streams, connecting a Spark Receiver (or similar constructs) to data sources to form a *base stream*
- the definition of transformations, each offering a new, immutable form of processed data. Those new intermediate streams are evaluated lazily, immutable, and their naming hints at the data flow of the program. They can re-define the groupings and the aggregation of results over time, in multiples of the batch interval.
- the definition of actions, which trigger the evaluation of the transformed data streams, in reverse, and possibly output data to visualalizations or database connectors.
- the start of the Spark Steaming Context, which marks the actual start of data processing by creating Receivers and connecting them to data sources.

Windowed Streams

Windowed Streams

As we have hinted at previously, and as we have shown in previous transformations, Spark Streaming offers the capability of building time aggregates of data.

Aggregation is a frequent pattern in stream data processing, reflecting the difference in concerns from the producers of the data (at the input) and the consumers of data (at the output).

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

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

Assume we want to count tweets bearing a given hashtag, to study interest in a given topic. One thing we can implement at first is a running count.

```
1 val runningSums = hashTags.map((hashTag) => (hashTag, 1)).updateStateByKey(
2   (newValues: Seq[Int], partialSum: Option[Int]) => {
3     val currentSum = partialSum.getOrElse(0)
4     Some(currentSum + newValues.sum)
5 })
```

We'll come back in detail to the subject of how `updateStateByKey` works shortly. Suffice to say that we are, at the moment, simply providing a function that is run on the data for every single found hashtag. That function takes the current `partialSum`, if it exists (hence the `Option` type, and new data for this same hashtag. It then sums the count for the fresh data with the `currentSum` of the data encountered up to this batch, and returns.

If you run this example, you can visualize the hashtags in the same fashion we used above:

```
1 runningSums.foreachRDD{rdd =>
2   val randomList = rdd.take(10).toList
3   val r = randomList.map{case (count, tag) => s"$tag: $count"}
4   result(r)
5 }
```

Mandatory checkpointing on Stateful Streams

Note that when starting a Spark Streaming Context for this application, Spark may output an error saying that

> The checkpoint directory has not been set. Please set it by `StreamingContext.checkpoint()`

The reason for this is that the `StateStream` created under the hood by `updateStateByKey` has `RDD`s that inherently each have a dependency on the previous ones, meaning that the only way to reconstruct the chain of partial sums at each batch interval for each hashtag is to replay the entire stream. This does not play well with fault tolerance, as we would need to preserve every record received to be able to recreate the state at some arbitrary point in time. Instead of preserving all records, we save the intermediary result of state to disk. In case one of the executors working on this Stream would crash, it can recover from this intermediate state.

Luckily, the error tells us exactly how to do so, using `ssc.checkpoint("path/to/checkpoint/dir")`. Replace the content of the `String` passed as argument with an accessible directory.

The result of the operation is a display of 10 hashtags encountered so far with their running count. Two things become readily apparent:

- it's perhaps not optimal to keep a count for every single hash tag we encounter. A top-K algorithm allowing us to get an idea of the most popular ones would be a better fit for this task, but it's beyond the scope of this particular example.
- we don't really want to see the total count since the beginning of time for each of the hashtags we have encountered. Studying hash tags is a study in trends: trending topics are only revealed when their usage spikes over a short period of time. As such, we'd like to have an idea of hash tag usage over a bounded window in time.

We can do this with tumbling windows:

```
1 val tumblingSums = hashTags.window(Seconds(60)).map(hashTag => (hashTag,1))
2                            .reduceByKey(_ + _)
```

In the `window` operation, before the `map` and `reduceByKey` steps — that we now know do the a simple counting — we are reprogramming the segmentation of our `DStream` into `RDD`s. The original stream, `hashtags`, follows a strict segmentation according to the *batch interval*: one `RDD` per batch.

In this case, we are configuring the new `DStream`, `hasTags.window(Seconds(60))` to contain one `RDD` per 60 seconds. Every time the clock ticks 60 seconds, a new `RDD` is created on the cluster's resources, bearing no special relation with the previous elements of the same windowed DStream. In that sense, the window is *tumbling*: every `RDD` is 100% composed of new "fresh" elements read on the wire.

Caution

Since the creation of a windowed stream is obtained by coalescing the information of several `RDD`s of the original stream into a single one for the windowed stream, the window interval *must be* a multiple of the batch interval.

Naturally, any window length that's a multiple of the initial `batch interval` can be passed as an argument. Hence, this kind of grouped stream allows the user to ask questions of the last minute, 15 minutes, or hour of data — more precisely, of the k-th interval of the windowed streaming computation runtime. One disadvantage of this approach is apparent when the result of a stream computation is monitored: the `RDD` is actually produced and processed when the wall clock goes past the aggregate's boundary: to get the aggregated result for the previous hour, you have to wait for at least an hour of data to come in.

Caution

The window interval is aligned with the start of the streaming application. For example, given a 30-minute window over a `DStream` with a `batch interval` of 2 minutes, if the streaming job starts at 10:11, the windowed intervals will be computed at 10:41, 11:11, 11:41, 12:11, 12:41,…

However, this is not always what is required in some business cases: while the information of "what were the most popular hashTags during a famous sporting event" is interesting for forensics or future predictions, it is not the kind of question one usually asks during that sporting match. Neither is a tumbling window the relevant time frame when detecting anomalies: in that case, the aggregates are usually necessary because the values being observed have frequent, but often small and therefore meaningless fluctuations. The price of a stock or the temperature of a component have small fluctuations that should not be observed individually. The actual trends become visible by observing a series of recent events.

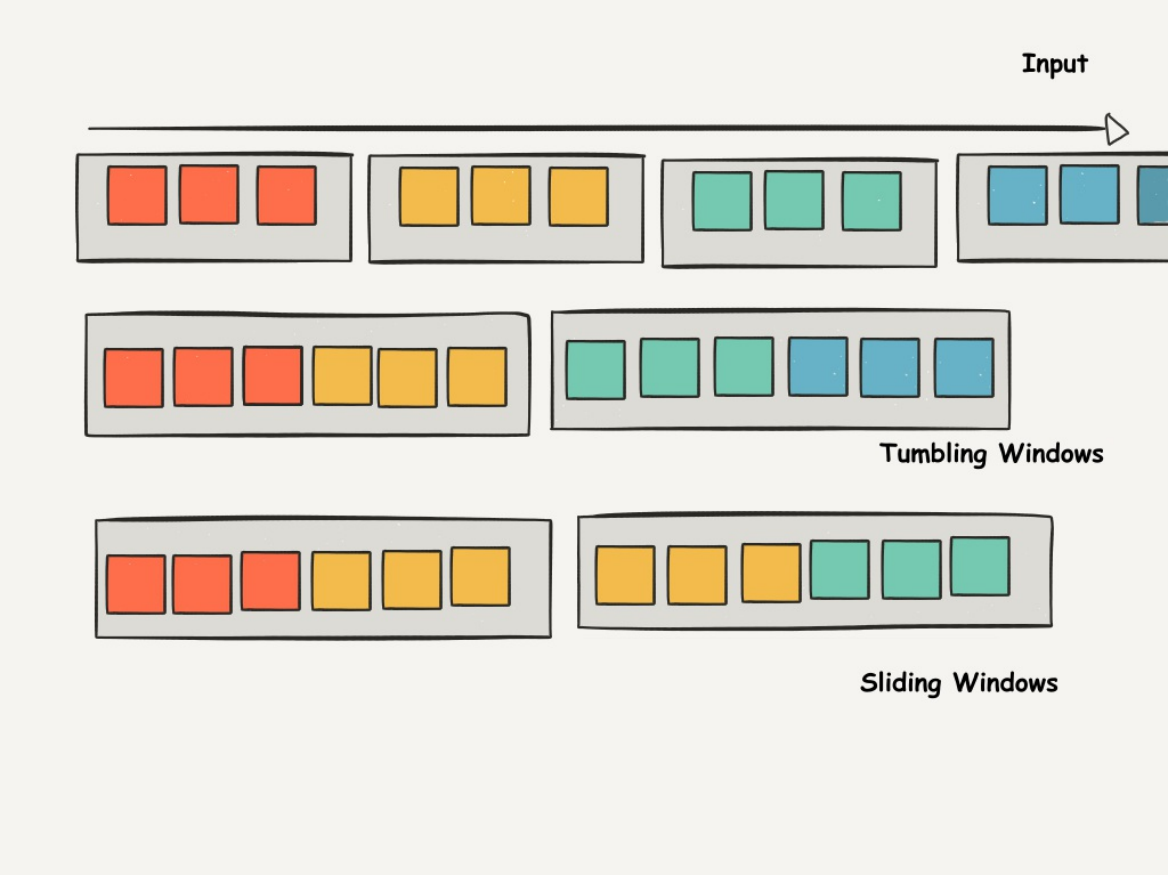


Figure 3-5. Tumbling and Sliding Windows

It is often useful to look at a different type of aggregate that presents data over a relatively large period, while keeping part of it fresh: sliding windows.

```
1 val slidingSums = hashTags.window(Seconds(60), Seconds(10))
2                           .map(hashTag => (hashTag, 1))
3                           .reduceByKey(_ + _)
```

In this example, we are describing a different type of `DStream` to compute the most frequent hashtags on: this time, a new `RDD` is produced every 10 seconds. In this alternative of the `window` function, the first argument, named `windowDuration` determines the length of the window, while the second argument, called `slideDuration` dictates how often we want to observe a new window of data.

Coming back to the example, the resulting windowed DStream will contain RDDs that presents the result of the computation over the latest 60 seconds of data, produced every 10 seconds.

This sliding view makes it possible to implement a monitoring application, and it is not infrequent to see the data produced by such a stream be sent to a dashboard after processing. In that case, naturally, the refresh rate of the dashboard is indexed on the slide interval.

Caution

Again, since the RDDs of the output function are obtained by coalescing the input RDDs of the original DStream, it is clear that the slide interval need to be a multiple of the batch interval, and that the window interval needs to be a multiple of the slide interval.

For example, with a batch interval of 5 seconds, and a base DStream called d , the expression d.window(33, 9) is incorrect (the slide interval is not a multiple of the batch interval), and should be replaced by d.window(30, 10) . The expression d.window(Seconds(40), Seconds(25)) is equally invalid, despite the slide interval being a multiple of the batch interval, because the window interval must be a multiple of the slide interval. In this case, d.window(Seconds(50), Seconds(25)) is a correct sliding interval definition.

In a nutshell, the batch interval can be seen as an “indivisible atom” of the time interval of windowed DStreams.

Finally, note that the sliding window needs to be smaller than the window length for the computation to make sense. Spark will output runtime errors if one of these constraints is not respected.

The trivial case of a sliding window is that one in which the slide interval is exactly the window length. In this case, you’ll notice that the stream is the equivalent to the “tumbling windows” case presented above, in which the window function has only the windowDuration argument. This corresponds precisely to the semantics implemented in the window function.

A word on changing the batch interval

One could wonder, looking at the simple tumbling windows, why it would be necessary to use windowed streams in tumbling mode rather than simply increase the batch interval: after all, if a user wants the data aggregated per minute, isn't that exactly what the batch interval is made for? There are several counterarguments to this approach:

multiple aggregation needs

> it happens sometimes that users want to see data computed upon in different increments, which would require extracting two particular frequencies of looking at the data. In that case, because the batch interval is indivisible and the source of the aggregation for other windows, it is best set to the smallest of the necessary latencies. In mathematical terms, if we want multiple windows of our data, say of durations *x,y and z*, we want to set our `batch interval` to the *greatest common divisor* (gcd) of them: `gcd(x,y,z)`

safety and locality

> The batch interval is not only the source of the division of a `DStream` into `RDD`s. It also plays a role in how its data is replicated and moved across the network. If, for example, we have a cluster of eight machines, each with four cores, since we may want to have on the order of two partitions per core, we will want to set the block interval so that there are 32 block intervals per batch. It is the block interval that determines the clock tick at which data received in Spark is considered for replication by the block manager. Hence, when the batch interval grows, the block interval, which should be configured to a fraction of the batch interval, may grow as well, and would make the system more fragile to a crash that would compromise data ingestion (e.g. if a receiver machine dies). For example, with a large batch interval of one hour, and a block interval of 2 minutes, the potential data loss in case of a receiver crash is of maximum two minutes, which depending on the data frequency, may be inappropriate.

In the case where we would want an aggregation of one hour, a tumbling window of one hour, based on a source `DStream` with batch intervals of five minutes, gives a block interval a little bit under ten seconds, which would make for a lesser potential data loss. In sum, keeping a batch interval of a reasonable size increases the resiliency of the cluster's setup.

Slicing your Stream

Finally, note Spark's `DStream`s also have a selection function called `slice` , which returns the particular subpart of a `DStream` which was included between two bounds. You can specify the bounds using a beginning and end `Time` , which corresponds to Spark's `org.apache.spark.streaming.Time` or as an `org.apache.spark.streaming.Interval` . Both are simple reimplementation of time arithmetic using milliseconds as the base unit — leaving the user with a decent expressivity.

Spark will produce sliced `DStream`s by letting through the elements that carry the correct time stamp. Note also that `slice` produces as many `RDD`s as there are batch intervals between the two boundaries of the `DStream`.

What if your slice specification does not fit neatly into batches ?

The timing of an `RDD` 's original batch and the output of the slice may not exactly match if the beginning and end time are not aligned with the original `DStream` 's batch interval *ticks*. Any change in timings will be reflected in logs at the `INFO` level:

```
1 INFO Slicing from [fromTime] to [toTime]
2   (aligned to [alignedFromTime] and [alignedToTime])
```

Other Data Sources and Connectors

While we have seen how to operate based on a simple data source such as a socket, it is often very useful, in a distributed context, to be able to create and analyze streams in a way that is not as low-level as creating your own socket communications.

Moreover, distributed setups often use multiple machines precisely because they want to obtain more reliability in such a setting, being less dependent on the failure of any single piece of equipment. A consequence of this has been the need to route messages created by a data source, sensors, or an application, through an to several machines — a complexity mostly resolved nowadays with message-passing systems, and stream processing engines, of which we will examine a few, with the view of having them connect to Spark Streaming.

Spark Streaming has a number of connectors to different data sources, that let it interface with data producing servers at large.

Apache Kafka

Apache Kafka is a top-level Apache project that originated at LinkedIn and was open-sourced in early 2011. It is mostly implemented in Scala, with some parts of it in Java, and it consists in distributed high throughput message passing system, that has the goal to be a unified platform for handling the larger, real-time data feeds a large company might need. Some of its design goals included a high-volume of event feeds, the ability to create new derived feeds from the combination of the inputs flowing inside the system, the support for data backlogs or queues, allowing to handle periodic congestion from offline systems, and a low-latency delivery to handle the traditional messaging use cases. One more point of focus on the design of Apache Kafka was also the fault-tolerance in the presence of machine failure.

An important distinction with what existed in other systems at the time, it aimed to be a pull-based system. This means that Kafka has servers called brokers that store events and only deliver them when the client connects to them and tries to pull data. That client, in that case, is called a consumer. We will come back to the subject of how to connect Spark Streaming to Apache Kafka in the next chapter, in the development of stable configurations to Spark Streaming.

However, it is important to have an idea of a few concepts in the organization of Apache Kafka before we see the most basic code that allows to interface Spark Streaming with an Apache Kafka data source. Kafka stores data in topics and those topics are a cemented meaning, which means that while topics may not always be strictly distinct in a particular topology — the same event could be found in several —, topics have a precise meaning as to what should be in them. An orthogonal but important concept which is the configurable number of partitions for a topic. The number of partitions for that topic is important for preference considerations because it is an upper bound on the consumer parallels. If the topic has N partitions, then an application or consumer can only consume this topic with a maximum of N threads in power.[2]

To create a Stream reading data from Kafka, you need a few elements of configuration, but before all, you need a running Kafka server. Thankfully, this is easy to set up.

We are going to set up a local Kafka server, and make it read text messages. You can download a new Kafka version from *http://kafka.apache.org/downloads.html*

```
1 $ tar -xzf kafka_2.11-0.10.0.1.tgz
2 $ cd kafka_2.11-0.10.0.1
```

The Kafka distribution contains all that is needed to start a single-node version of a Kafka server and the Zookeeper server that Kafka depends on.

```
1 kafka_2.11-0.10.0.1$ bin/zookeeper-server-start.sh  config/zookeeper.properties
2 [2016-09-12 09:46:29,942] INFO Reading configuration from: config/zookeeper.properties
3   (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
4 [2016-09-12 09:46:29,944] INFO autopurge.snapRetainCount set to 3
5   (org.apache.zookeeper.server.DatadirCleanupManager)
6 [2016-09-12 09:46:29,944] INFO autopurge.purgeInterval set to 0
7   (org.apache.zookeeper.server.DatadirCleanupManager)
8 [2016-09-12 09:46:29,944] INFO Purge task is not scheduled.
9   (org.apache.zookeeper.server.DatadirCleanupManager)
10 ...
```

Once the Zookeper server is started, Kafka will be able to find a server to which it can register to avoid creating single points of failure when consuming data. This registration to Zookeeper will make more sense when we actually read data later, using this Zookeeper address, which in a more normal setup would consist of a list of IPs, or quorum. Naturally, in a single-machine test run, avoiding single points of failure is a purely fictive exercise, but the concern explains the tight integration of Kafka with Zookeeper. We can now start Kafka proper :

```
1 kafka_2.11-0.10.0.1$ bin/kafka-server-start.sh  config/server.properties
2 [2016-09-12 09:47:00,567] INFO KafkaConfig values:
3         advertised.host.name = null
4         metric.reporters = []
5         quota.producer.default = 9223372036854775807
6         offsets.topic.num.partitions = 50
7 ...
```

Kafka instances are usually classified in three types.

Producers

> consume the data feed and send it to Kafka for distribution to consumers.

Consumers

> applications that subscribe to topics; for example, a custom application or any of the products listed at the bottom of this post.

Brokers

> workers that take data from the producers and send it to the consumers. They handle replication as well.

We just started a Kafka Broker, which has the advantage to deal both with the “reception” and “emission” side.

We can then use the console producer to generate random numbers sent to Kafka:

```
1 kafka_2.11-0.10.0.1$ while true; do echo $RANDOM | bin/kafka-console-producer.sh \
2     --broker-list localhost:9092 --topic random > /dev/null
```

This command sends random numbers to the kafka broker we just started.

We can witness them with the command-line consumer:

```
1 kafka_2.11-0.10.0.1» bin/kafka-console-consumer.sh --zookeeper localhost:2181\
2                           --from-beginning --topic random
3 9767
4 29045
5 19616
6 20694
7 4867
8 15978
9 596
10 24977
11 8088
12 8140
13 5029
14 15908
15 25005
16 5417
```

```
17 10910
18 ...
```

We can then start a simple job in the Spark notebook to ingest these numbers. In the notebook, to import the Kafka connector for Spark Streming, we need to add one package to our classpath:

```
1 :dp org.apache.spark % spark-streaming-kafka_2.10 % 1.6.2
```

Beware executing this cell will restart your context, so make it the first of your notebook. If you are trying this with the Spark shell, the import is just as easy:

```
1 spark-1.6.0-bin-hadoop2.6» bin/spark-shell --master "local[*]" \
2   --packages org.apache.spark:spark-streaming-kafka_2.10:1.6.2
```

We can then run a notebook that creates a Kafka Stream and prints its contents:

```
1 import org.apache.spark.streaming.kafka._
2 val kafkaStream =
3   KafkaUtils.createStream(streamingContext, "localhost:2181", "0", Map("random" -> 1))
4
5 val result = ul(10)
6 result
7
8 kafkaStream.foreachRDD{ (rdd) =>
9   rdd.foreach{(x) => result(Seq(s"${x._1}:${x._2}"))}
10 }
11 streamingContext.start()
```

If using the spark-shell, the processing inside the `foreachRDD` would look like `(x) => println(s"${x._1}:${x._2}")` Here if you insert a new cell at each linebreak, you'll see the unnumbered list being updated with a sequence similar to this:

```
null:42
null:3802983
null:10009
...
```

In creating a Kafka Stream, we can see a few arguments being passed to our factory. The first is the Spark context : as we've seen before, the context needs to register every `DStream` . Then we need to connect to the Zookeeper Quorum of our Kafka server, which explains a port specification that is different from the port range we would expect out of a Kafka server. In a real setup, this single machine specification (here localhost, since our server runs on our single local machine) would be a list. Finally, we provide a string identifying our consumer group.

This consumer group is here because the Spark Receiver-based kafka stream uses the high level API. Using this API, when a new process connects with the same consumer group name, Kafka will add that processes' threads to the set of threads available to consume the Topic and trigger a *re-balance*. Remember that each Kafka *topic* is a label on a stream marking a determined content that is parallelized with an internal number of *parititons*. In this rebalance, Kafka will assign the available partitions to available threads, possibly moving a partition to another process.

Finally, we assign a topic map, which maps topics to the number of threads that should be used to consume it. Ideally, this could be the number of partitions available in Kafka for the topic, but note that these threads will only be created on the receiver machine. Whether or not you want to create as many threads there as there are available partitions for your topic is not obvious.

Let's assume you have a topic *random* readable on your Kafka server, with 10 partitions. You

want to read this topic as fast as possible, using parallelism.

- if you read it with one stream configured to launch 10 threads, you need 10 cores availableon your Receiver, which, let's recall, happens to be just another executor. Note that the number of cores per executor may be constrained by your cluster manager. In that sense, you may have trouble finding executors that have so many cores.

- if you read it with five distinct `DStream`s, each of which is configured to read using 2 threads, you need only two cores available per executor.

In sum, the tuning of parallelism when consuming from Kafka can be a touchy subject, one that we'll come back to in the next chapter.

Finally, we will note that Kafka has as a particularity: the ability to act as a buffer between producer and consumers, when configured to do replays with some level of retention. In this particular configuration, Kafka requires a consumer when it is pulling data to keep track of offsets that it is consuming and to report offsets that it has successfully received. This configuration mode lets Kafka, on demand, replay data that a consumer might not have fully stored, allowing the consumer to rely on Kafka for some level of reliability. We will come back to this topic in the next chapter and we will see that this is extremely efficient for the purpose of deploying stable Spark Streaming configurations.

Apache Flume

Let us now focus on Apache Flume. Apache Flume is an open-source Apache top level project that was more closely developed in conjunction with the Hadoop project. As such, it is more tightly interfaced and tied to the Yarn and Hadoop constellation of projects. It is another distributing message passing system that is reliable and available as a service, but that has as a distinctive component that it is push oriented, meaning that it has as a goal to feed data into such a system as HDFS, or another Hadoop dependent, including some streaming ones.

Apache Flume organizes the rules in its deployments in four distinct concepts that closely match the extract, transform, and load logic. The first of those concepts is a source, which is a component that weeds or extracts data at a particular configurable location. This can be, for example, the reading and production of records from a database, file systems, sockets, or logs created by something such as a web application. That source creates chunks of data known as events that are delivered to channels. Channels are a staging area that defines semantically where the data that corresponds to a particular semantic set should fall, and how it should be manipulated further by Flume as it is being processed. It decouples the sources from the destination of the data by introducing this high-level semantic abstraction. Now, those channels could be in memory, could be file-backed, could be backed by JDBC in a search, be more or less reliable. You can see the Flume documentation for more details on the permanence and the storage of the content of channels.

Flume has an additional component that allows you to modify the content of a channel which is called an interceptor. It lets you perform some transformations and chaining of chain transformations on particular channels. Finally, the whole point of Flume is to deliver data to a sink, a component which loads data after it has been aggregated, processed, and organized to Flink. The intended receivers or consumers of those systems could be Kafka, Avro, or in a particular context, Spark Streaming. We're going to see how to create such a system in the stream that is fed as a Flume Sink in Apache Spark Streaming.

The first step towards testing flume involves downloading at the URL:
http://www.apache.org/dyn/closer.cgi/flume/1.6.0/apache-flume-1.6.0-bin.tar.gz

You can then use `tar -xvf apache-flume-1.6.0-bin.tar.gz` to uncompress the archive and set the variable `FLUME_HOME` to the place where you uncompressed said archive.

```
1 FLUME_HOME="/path/to/apache-flume-1.6.0-bin"
2 export FLUME_HOME
```

Once this is done, we need to configure a Flume server for the typical application of our test : reading the system's logs on a systemd-based system.

We can write a Flume configuration in `flume.conf`, to which we will point the server on startup:

```
# the components on this agent an logical names of source, channel and sink
a1.sources = src-1
a1.sinks = snk-1
a1.channels = ch-1
```

```
# our source uses the systemcl journal reporter to provide log messages:
a1.sources.src-1.type = exec
a1.sources.src-1.command = journalctl -ef
a1.sources.src-1.channels = ch-1

# our sink
a1.sinks.snk-1.type = avro
a1.sinks.snk-1.hostname = localhost
a1.sinks.snk-1.port = 44000

# characteristics of the 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
```

Using the exec source in Flume

The Flume source is, in our toy example, configured using the `exec` source. Beware about using this in production ! The documentation of Flume has this to say about the subject:

> The problem with `ExecSource` and other asynchronous sources is that the source can not guarantee that if there is a failure to put the event into the Channel the client knows about it. In such cases, the data will be lost. As a for instance, one of the most commonly requested features is the `tail -f [file]-like` use case where an application writes to a log file on disk and Flume tails the file, sending each line as an event. While this is possible, there's an obvious problem; what happens if the channel fills up and Flume can't send an event? Flume has no way of indicating to the application writing the log file that it needs to retain the log or that the event hasn't been sent, for some reason. If this doesn't make sense, you need only know this: Your application can never guarantee data has been received when using a unidirectional asynchronous interface such as `ExecSource` ! As an extension of this warning - and to be completely clear - there is absolutely zero guarantee of event delivery when using this source. For stronger reliability guarantees, consider the Spooling Directory Source or direct integration with Flume via the SDK

This configuration file creates a source and a sink on the local agent, and connects both using a channel. The source is the log reader of `systemctl` a log manager that has now mostly replaced the syslog daemon of old in Linux systems. The sink of our server will be a receptor for avro files on port 44000. Note this reception will exert itself in push style. Flume also has a pull mode that we will exert in the next chapter.

```
    ./bin/flume-ng agent --conf conf --conf-file conf/flume.conf
--name a1 &
```

Once you have installed Flume, the first task for consuming events from Flume consists in making sure that:

- all Spark dependencies for consuming events from Flume agents are in the classpath of our Spark driver
- and that they are also available to Spark executors, which will be executing our Spark application

Thankfully in our case, this simply involves starting Spark on reading avro messages — though we wil notice in the next chapter that this can become more involved.

We can then fire up our notebook and create a stream for this source:

```
1 import org.apache.spark.SparkConf
2 import org.apache.spark.SparkContext
3 import org.apache.spark.storage.StorageLevel
4 import org.apache.spark.streaming.Seconds
5 import org.apache.spark.streaming.StreamingContext
6 import org.apache.spark.streaming.flume.FlumeUtils
7
8 val ssc = new StreamingContext(conf, Seconds(10))
```

```
9 ssc.checkpoint(checkpointDir)
10
11 val statefulCount = (values: Seq[Int], state: Option[Int]) =>
12   Some(values.sum + state.getOrElse(0))
13
14 FlumeUtils.createStream(ssc, "localhost", 44000, StorageLevel.MEMORY_ONLY_SER_2)
15        // We separate timestamp and message
16        .map(rec => new String(rec.event.getBody().array()).split("\[.*?\]:"))
17        // source process name is the key
18        .map(rec => ((rec(0).split(" ")(4)), 1))
19       .updateStateByKey(statefulCount)
20       .repartition(1)
21       .transform(rdd => rdd.sortByKey(ascending = false))
22       .saveAsTextFiles(outputPath)
23
24     ssc.start()
25     ssc.awaitTermination()
```

This snippet gives us the count of the number of messages per process providing a hint on which processes produce hte greatest number over time. Easy manipulations of this simple program would let us for example benchmark the difference between the event time and the wall clock, measuring how late the events we are dealing with are.

In this case, we have managed to plug together a source and a sink, resulting in a funcitonal Flume stream.

Kinesis

Amazon Kinesis is another streaming extract, transform, and load processor created by Amazon. It has the advantage, while still being distributed and highly available, to interface very well with the rest of the Amazon ecosystem. As such, it works seamlessly for serving data to all available US services such as S3 Simple Storage, or Elasticsearch, for example. Spark Streaming has basic regard for Amazon Kinesis streams and we will see how to develop a stream that has the availability of matrix reporting and real-time data analytics, and log data, for example, being streamed from Amazon Services. This is especially relevant if you are running a Spark Streaming configuration on a cloud system that is fed by Amazon hardware.

Apache Bahir

A few of these connectors have gained some significant importance, to the point where their code has been separated from the Spark code base, under a difference Apache project called Apache Bahir.

These include the stream connectors for Akka Streams, MQTT, Twitter, and ZeroMQ. Prior to version 2.0, those connectors were part of the Spark code base.

How to write a quick stream generator for testing :

`SocketStream` , `FileStream` , `QueueStream`

Testing a Streaming Application by feeding it an artificial stream is extremely useful for integration testing. You may have unit-tested modular components of your program in isolation, but how do you know that everything runs when put together ?

However, test streams are interesting for more than that. When you wirte an integration test, you are basically feeding synthetic data, or at least an known dataset to your Spark Streaming Application. But more interestingly, the same facilities in Spark Streaming allow you to feed it static data, and in particular historical data.

Hence the methods we are going to see below are perfectly applicable to replay “batch data” or data at rest, into an applicaiton that built for Streaming.

There are several ways of creating an artificial stream. We have seen at the beginning of this section the use of a `SocketStream` for connecting to a particular server. However, you can also create an artificial server on localhost, which will serve the data to the receiver. If you are running in local mode (that is, not on any real cluster, but testing things locally), you can pipe a file repeatedly to the local port 8088, to be then read by Spark:

```
1 while :; do cat mytext.txt; done | netcat -l -p 8088
```

Naturally, if you are running this on a real server, you will want to create a real server, which is visible on some port from all the executors.

The second simplest way to feed an artificial stream to Spark Streaming is through a `FileStream`:

```
1 import org.apache.spark.streaming.{StreamingContext, Seconds}
2
3
4 val ssc = new StreamingContext(sparkContext, Seconds(5))
5
6 val fStream = ssc.textFileStream(dataDirectory)
```

In this case, Spark is going to monitor the `dataDirectory` on the system (this directory has to be readable from executors for this to be usable in a distributed context) to find some new files on every batch interval, and feed their contents into the new `RDD` for this new batch.

To feed a particular file to a Spark File Stream, you can simply use a script to move the files of interest in to the directory monitored by your Stream, at the rythm of the batch interval.

The stream shown above expects simple text strings to be present in these files, which is fine to read logs for example, but there is a more generic `FileStream` that includes the use of various key and value types, in the Spakr code base. You can find more details in the relevant section of the Spark documentation.

Another source of data you can use as a Stream is a `QueueStream` . In this case, no monitoring of the

file system is performed by Spark. Rather, Spark polls a particular Scala queue, that contains `RDD` object supposed to represent the various micro-batches of data ocurrring in the data source.

```
import org.apache.spark.rdd.RDD
val lines = scala.collection.mutable.Queue[RDD[Double]]()
val d = ssc.queueStream(lines)
```

Note that at this stage, the `lines` queue does not contain any element. Indeed, the polling orchestrated by Spark does not start until the Spark Context is itself started.

Here, we would like to feed our queue with synthetic `RDDs` of `Double`s, emulating something close to the normalized readings of a sensor, in order to test our application. We would like to produce artificial `RDD` values, and one classical way of doing so in Spark is using a `RandomRDD`.

```
1 import org.apache.spark.mllib.random.RandomRDDs._
2 def u(): RDD[Double] = normalRDD(ssc.sparkContext, 100000L, 10)
```

This creates an `RDD` of 100 000 random values, distributed evenly in 10 partitions (more details in the documentation). This `RDD` can then simply be fed to the `Queue` which backs the queue stream. With a batch interval of 5 seconds, we can envision the following:

```
1 for (i <- 1 to 50) {
2   lines.synchronized {
3     lines += u()
4   }
5   Thread.sleep(4)
6 }
```

Naturally, the synchronized statement present above is there to deal with the fact that, even in a local environment, the driver thread populating the queue and the Spark Streaming context reading it are accessing the queue concurrently — but more clever concurrency handing will be appropriate if this goes beyond an integration test situation.

The Lambda Architecture

So far, we have seen:

- how to create a streaming application defining one or several streams,
- how to choose the time semantics of these streams,
- and how to have those streams fed with data coming from various connectors.

Now we turn to another aspect of the topology of streaming applications, which is how to interact with data at rest, and how to "mingle" with batch processing.

We can envision several ways in which stream processing interacts with batch :

- *code reuse*, which is a light subject in the context Spark, since as we have seen before, it is particularily easy to call functions that transform RDD s, even if developed with batch processing in mind, in a streaming application. But this reuse would not significantly change the nature of our streaming application.
- *data reuse*, where a streaming application feeds itself from a feature or data source prepared, at regular intervals, from a batch processing job. This is a frequent pattern: for example, some international applications have to deal time conversions, and a frequent pitfall is that daylight saving rules change on a more frequent basis than expected. In this case, it is good to be thinking of this data as a new dependent source that our streaming application feed itself off. We will revisit this notion in ???
- *mixed processing*, where the applicaiton itself is understood to have both a batch and a streaming component during its lifetime. This pattern does happen relatively frequently, out of a joined will to manage both the precision of insights provided by an application, and as a way to deal with the versioning and the evolution of the application itself.

This later option is what we are about to discuss now.

> For many stream processing applications, comparing "present" with "past" is a common task. Thus, a stream processing system must also provide for careful management of stored state. For example, in on-line data mining applications (such as detecting credit card or other transactional fraud), identifying whether an activity is "unusual" requires, by definition, gathering the usual activity patterns over time, summarizing them as a "signature", and comparing them to the present activity in real time. To realize this task, both historical and live data need to be integrated within the same application for comparison.
>
> The 8 Requirements of Real-Time Stream Processing, *Stonebreaker et al.*

The evolution of ideas, rather than products

We are about to embark upon a description of the Lambda architecture, which is a term coined in 2011 to speak about the concept of mixed processing, as we have mentioned above. Why speak about concepts in a book which is essentially about Spark Streaming ?

The field of large-scale data processing is often, as with most technological fields, obsessed by the new platforms and tools that deal with their use case. And producing a fair comparison between those new trends can sometimes be daunting to the architect.

We would contend that the comparison between particular *implementations* and *tools* is not as important as *general trends*, which reveal, in the abstract, progress in the discipline of analyzing data. In other words, the proper use of a NoSQL database is more important than Cassandra or MongoDB, and understanding the impact of in-memory processing is sometimes more important than comparing Apache Flink with Apache Spark. Whereas history shows the use of particular tools inevitably becomes obsolete, the relevance of architectural insight perdures — in fact, one would argue that this is the main legacy of MapReduce in the field of distributed processing today and tomorrow.

With that view, it seems necessary to give a bit of context on the Lambda Architecture. The Lambda Architecture was introduced as such by Nathan Marz in a blog post, How to beat the CAP Theorem[3]. It proceededs from the idea that beyond the precision of the analysis which we all care about, two additional, novel points need to be emphasized :

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

A classical but difficult example

Let’s look at it in the context of an example. Let’s assume you have a business operating in a particular region, say Europe, where most of your customers reside. You also have a partner with a fleet of container ships in a faraway location of the world where those customers want to import merchandise from, say Australia. You can rent passage for a container, for a fixed price, whatever the contents of the container, and your partner has one ship leaving Australia per day, at a fixed hour (5 PM your time). You continually receive customer orders for batches of merchandise, which have a known volume, a fixed latest delivery date, and generate a known revenue. Your objective is to maximize the revenue you generate for a fixed period of time (i.e. by month).

This description is a “realist” version of the well-known weighted knapsack problem, also known as bin-packing : if you have a number of objects of an unknown size, how can you pack them in as few fixed-size bins as possible ? It is a very hard problem:

- the decision version of the problem (is it possible to ship a profitable container — with a revenue superior to the shipping cost —, without going over the maximum volume, possible at all ?) is NP-complete. This means that any algorithm that would either:
    - not work optimally on some month (i.e. couldn’t work with any sequence of orders)
    - or incur prohibitive (exponential) computing costs. [4]

In brief : generality, speed — choose one out of two. There is a pseudo-polynomial algorithm which means that we know how to deal well (in a reasonable amount of time) with orders including any kind of value (volume, price), but it is still true that any generic algorithm (working on whichever month) will still have trouble (have a humongous running time) if it faces many orders for merchandise. - the optimization problem, the one that you would face in real life is NP-hard. This means, informally, that it is at least as hard as the hardest problems in the NP class, such as the one described above. It is expected to be seriously harder to solve than the one above.

Note that those properties do not mean there is no way of finding the optimal way to ship merchandise for any list of orders. It just means that running such an algorithm may take an unavoidably long time in some instances.

Batch processing and a program's life time

A first approach in doing the data analysis is to run an (exponential) algorithm to find the optimal shipment on the open orders at the beginning of the day, wait for the algorithm to complete, and send the result to Australia for shipment as soon as the computation is over. The customers know at the beginning of the day after they submitted their order whether you are going to ship their merchandise, and having run the computation, you know whether you are going to make money this day.

One issue, however, is that if your algorithm does not find an answer in time, you will not ship any merchandise that day. And the complexity properties above mean you're not sure you will be able to find the optimal solution in time for the day's shipment hour.

So you run another algorithm in parallel, a *polynomial* one, based on a heuristic, hoping that it will help you solve the knapsack problem in time. The solution it gives you is guaranteed to come before the shipment's hour, but there is no guarantee of optimality. If you are ready to accept some variability in your profit margin, this may be an interesting way to proceed, and probably a more profitable one than only shipping on the days where you find an optimal way to pack your containers.

But now that you are running a heuristic, you are probably going to try to optimize it incrementally, depending on knowledge you have of your customers, taking into account say, a prediction of their next moves based on repeat orders, for example. The problem with this, however, is *rolling back*. Naturally, saving different versions of your program through version control solves the issue of how to actually effect a roll back, but how do you know you should ? How do you track your performance ? The new version of your algorithm may perform superbly well on one given day, but what guarantees it will keep doing so for a month ? You could compute the two results in parallel for the next month (shipping according to the best heuristic of the two), but that would require either doubling your computing power or starving your old heuristic of half of its computation time.

One other way of answering this question is to answer a simpler one: would it have given better results if it had been running over the previous year ? To compare new code with the old, you need the replayability of historical data, and this is one of the first insights of the Lambda Architecture. The raw data is an invaluable practical asset for building a data analytics solution, irrespective of whether it uses machine learning or not. In some cases, that historical data may not help exploring the data or interactively better understand the field of possible inputs, but it at least helps understanding how we perform in that space compared to our historical performance.

So, you are now running *two batch algorithms* :

- one exponential exhaustive search for the optimal solution
- one polynomial heuristic tuned to give a decent answer by shipping time.

You compare between iterations of the second algorithm using historical replay.

A Streaming improvement

So let's assume you now store historical data. At 9AM you collect the outstanding customer orders, and at 5PM your best program of the day (polynomial heuristic or exponential exhaustive optimizer) outputs the result of the shipment. Between 5PM and 9AM you use your computational power to evaluate other potential heuristics based on historical data.

This is where streaming comes in: customers as well as shrewd engineers in your team notice that you are computing how to compose a good container shipment based on customer orders opened during the *previous day* - or at best, before 9AM on the same day. This lets you have customers know at most in 32 hours whether you'll take on their shipment [5]. Some customers like to be advised by a predictable time but others are miffed at the necessity of keeping their order open for you during up to 32 hours (they may want to ship with one of your competitors). Moreover, what if a new customer comes in with a superb but urgent order, that exactly fills up a container with high-value goods, at precisely 09:01 ? You may miss out on this optimal one-off just because he can't wait for 32 hours, until you finish your batch computation cycles.

So your engineers advise you to go for an incremental algorithm. Such an algorithm differs from a batch algorithm in that it does not know a special set of orders to optimize for in advance: it receives each new order as it comes, and is susceptible to answering with a proposed solution at any given point in time. In the case of our knapsack problem, the algorithm maintains an "as good as possible" container, along with a little bit of state, and tries to decide whether it's possible to optimize this "best" container, given a single new order.

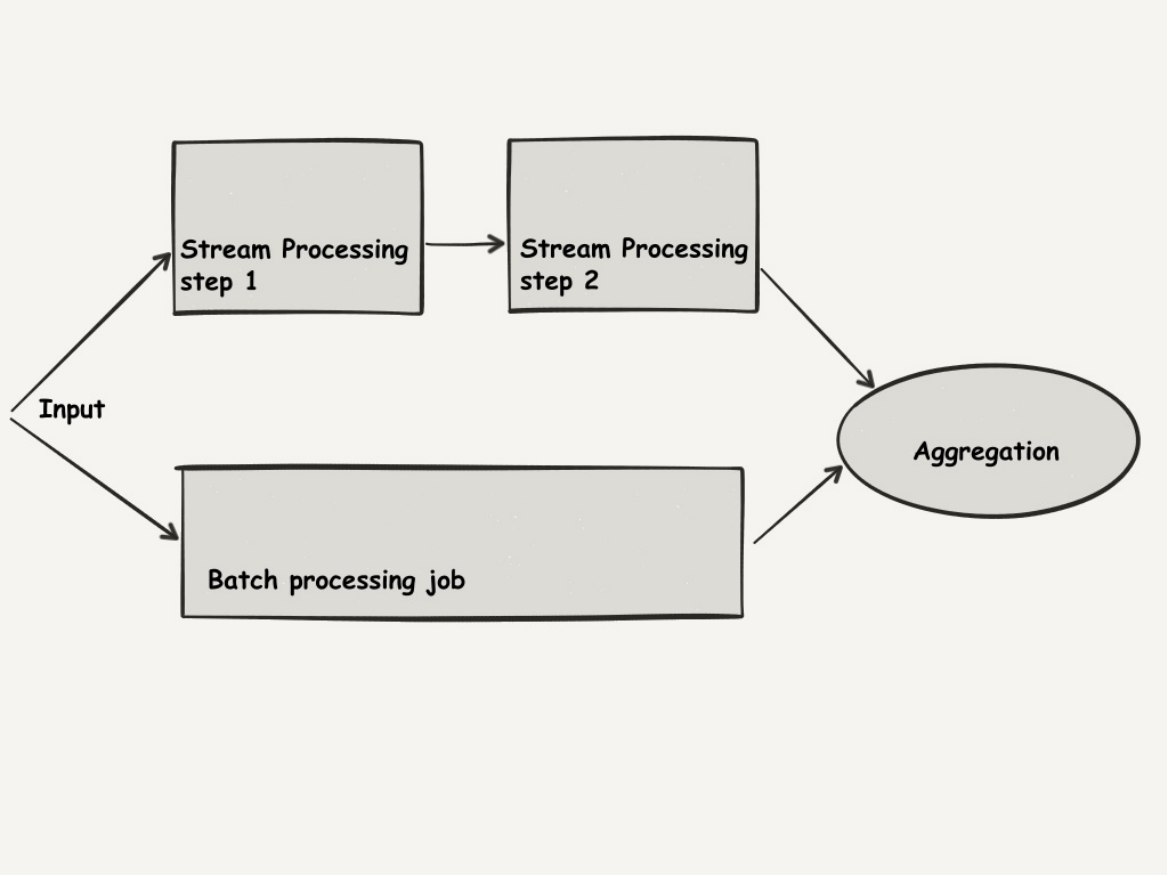


Figure 3-6. The Lambda Architecture

The main advantage of this approach is that you do not have to enforce the cutoff date of 9:00 AM. You can feed order after order, as they come, to your algorithm, and just ask it to output its current best answer as the shipment's hour comes up. More interestingly, this does not preclude the replayability of your solution: if you have a way to store the raw data on say, a distributed file system or in a database, it's very easy for a streaming server (such as Kafka) to emulate a streaming server based on this historical data. This lets you compare new versions of your incremental algorithm easily, or roll back to a new version: you just test them with historical streaming replays.

It is for this continuous flow of results, for the opportunity to explore the computation space with various incremental algorithms, or various parameterizations of the algorithms that Jay Kreps suggested this approach dubbed **the Kappa architecture** (in ???). However, it is not without drawbacks.

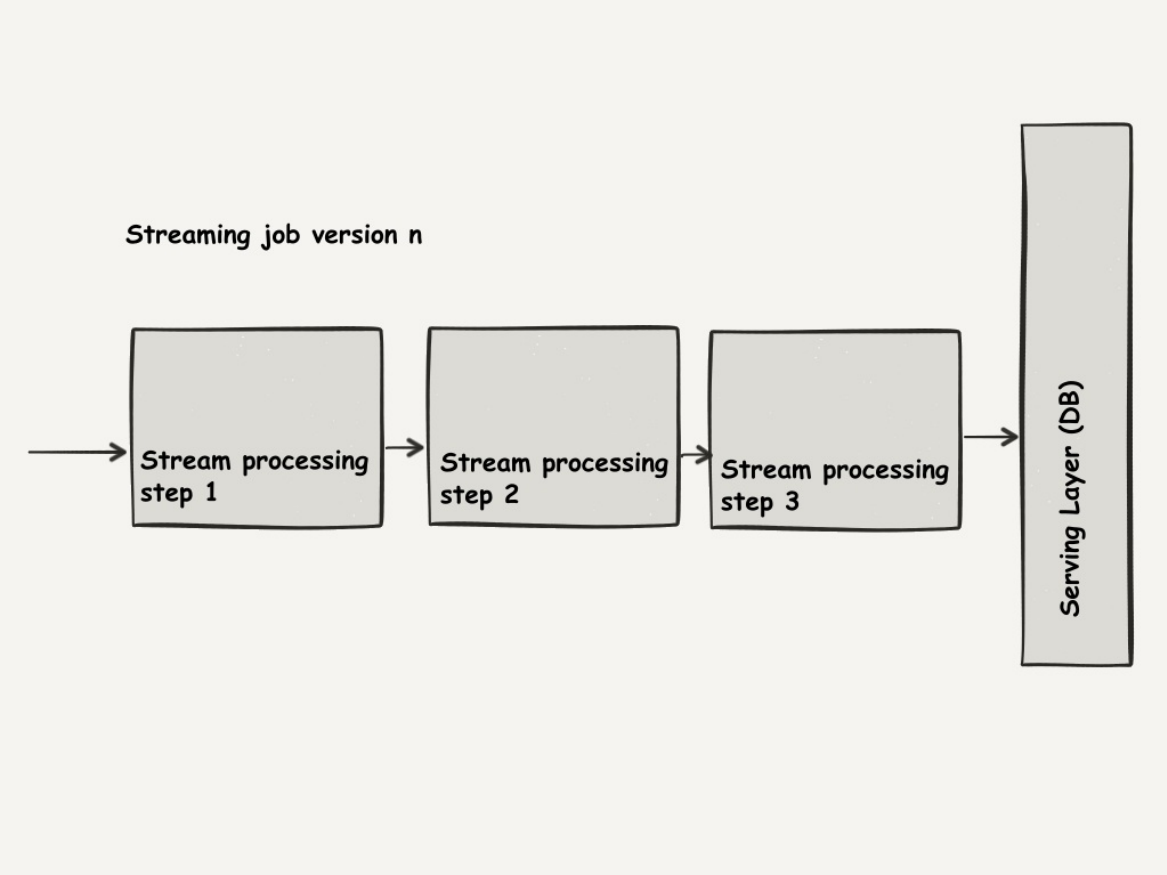


Figure 3-7. The Kappa Architecture

A fundamental difficulty: back to the Lambda architecture ?

On the drawback side, you may have to tell customers whether or not you are going to fulfill their order after an imprecise amount of time (depending on the exact appetite of your incremental algorithm for past state) but this can easily be given a fixed limit. At that cutoff time, you simply expire an order if it is still open.

Another issue, which is more important yet, is that the input space (the orders) can grow arbitrarily large during that extended amount of time, which makes the runtime of your incremental algorithm difficult to predict. Indeed, a single new order may require re-computation of a few possible solutions, and this time may preclude taking into account the new orders at the rate at which they are coming.

But the most important issue of all is the unpredictability of the quality of the result. An incremental algorithm's quality is measured with respect to that of a batch algorithm (the one that would know all of the orders that would come in until the shipment's very departure date, 4:59 PM), by its competitivity. An incremental algorithm is called (α, γ) competitive if there exists positive constants α and γ such that:

$$v_i \leq \alpha v_b + \gamma$$

Where v_b denotes the cost of the batch algorithm, and v_i that of the incremental one. Thus, an α-competitive algorithm A has cost no worse than α times that of the optimal offline algorithm plus some γ initial advantage given to the optimal algorithm based on the problem setup.

One issue here is that the competitivity bound for the knapsack problem can be arbitrarily small (???, Theorem 3.24). In other terms, *there is no limit to how bad an incremental algorithm can be compared to the batch algorithm* — that would have known what the orders at the end of the day are.

We see here that even beyond the case where there may be no algorithm available for solving a problem (in which case there is little risk of implementing a streaming-only solution by mistake), the availability of streaming algorithms does not always guarantee the quality of the solutions, especially in a domain where evaluating quality is difficult.

In particular, comparing incremental, streaming algorithms against each other may be useless, since in some cases, there may be a qualitative leap between the class of batch algorithms and the class of incremental algorithms. In that case, the fact that the streaming algorithm processes based on fresher data means can easily be compensated by the loss in quality it may create compared to a batch algorithm.

Some (streaming algorithms) are reasonably efficient, but do not guarantee that the model

> learned will be similar to the one obtained by learning on the same data in batch mode. They are highly sensitive to example ordering, potentially never recovering from an unfavorable set of early examples. Others produce the same model as the batch version, but at a high cost in efficiency, often to the point of being slower than the batch algorithm.
>
> Mining High-speed Data Stream, *Domingos*

How do we know ? Well, the only way to effectively do is to at least implement, test and run a batch algorithm, even if it is not the one we eventually make shipment decisions on. What the theorem above tells us is that it may eventually be the one based on which we will wish we had made decisions on.

Caution

Note this is not a one-size fits all argument against the Kappa architecture, or streaming engines. The present few paragraphs have as a goal to show some of the limits of streaming-only algorithmics. There are certainly problems for which a streaming-only solution would not reach those limits. Determining which architecture to use in each case is still left to the wisdom of the architect.

So the Kappa architecture, by focusing on replayable streams, may, for some problems, be depriving itself of the availability of much better batch solutions. To solve this issue, Nathan Marz had proposed, way earlier, something more complicated, known as the Lambda architecture:

- have one batch layer, that computes the optimal solution based on all known orders at a fixed point (9:00 AM).
- that computation may be slow, and it may be inefficient for updating users in a near-instant fashion. But if and when it finishes before the shipment date, you can use the remaining hours to run a streaming algorithm on the orders that have come in since the 9:00 AM cutoff date.

The double data flow (one into the batch computation, one into streaming), inspired the name of the Lambda architecture. It is indeed more complex than the all-streaming-with-replay Kappa architecture, but it does guarantee that you have a set fall back point if your streaming algorithm enters a massively suboptimal region of the solution space : the batch result you compute on a daily basis.

Its drawback, as well identified by Jay Kreps, is that it requires two different implementations of your algorithm. One works in batch, the other in streaming mode. At the time of Jay Kreps' writing no existing solution proposed maintaining two parallel code bases as efficiently as Spark Streaming does. We will see that with Spark, the practical usability and ease of maintenance of this double implementation is excellent.

Saving Streams

Stream Output and other operations

Spark Streaming has, as we'll see later, a number of operations inherited from the ones available on `RDD`s. But one additional feature of Spark Streaming is that it offers save operations that directly operate on a DStream.

print

> The basic print function that exists on `RDD` also has variant on streams. As with its `RDD`-based sibling, it by default operates on the first 10 elements of each batch of the `DStream` , and takes an optional integer argument to print more or less than the default number of elements. It is very useful for debugging, and very simple to use for that purpose, since it operates on the driver.

saveAsHadoopFiles

> This function saves each `RDD` of a `DStream` in a Hadoop file. Because o its tight integration with the MapReduce APIs, which assume data is composed of key-value pairs, this funciton is only available on `PairDStream`s.

Note

`saveAsHadoopFiles` is using an old version of the Hadoop file writing API, and it is preferable to use the newer `saveAsNewAPIHadoopFile` instead, unless you are using a version of Hadoop prior to 0.20.

saveAsTextFiles

> This function lets you save the data encountered in a particular `RDD` in a text file situated on the local system which does the processing, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.

saveAsObjectFiles

> This function does the same as the above, but trying to serialize the object instead of calling its `toString` method. Note that this serialization may prove to be much more space-efficient, att the expense of readability of the produced files.

Custom output operations : `foreachRDD`

Beyond those output operations, that are quite explicit in their dealing with the general data contained in a `DStream` , Spark also provides the very generic operation called `foreachRDD` that as its name indicates, takes a function which has an `RDD` of the corresponding `DStream` as argument, and outputs `Unit` , the empty type in Scala.

This operation is very powerful, and reflects the majority of the more complex, non-test output

operations in practice in a mature Spark Streaming pipeline, so that it's worth lingering on it for a little while.

As we have seen, `foreachRDD` takes an argument of type `RDD[T] => Unit` , a function which has the particularity of acting through side effects, since it cannot have a return value. This puts it squarely in the range of output operations, and will as for any of these trigger the evaluation of the `DStream` it is being called on - which may itself be obtained through several transformations.

But one important peculiarity of this function is that it in the usage of `foreachRDD` there are two repetitions, one implicit, and one explicit. The implicit repetition in `foreachRDD` is that the execution of the function will be repeated over time. `foreachRDD` ,as its name indicates, operates over each `RDD` of the `DStream` it is called on. And since this `DStream` is regulated with a batch interval, it will produce such an `RDD` on each interval, on which the function passed as an argument will then be executed.

The second repetition, which is usually more explicit, is to be found in the lines of the code of the function itself. Moreover, it is often a repetition on the elements of the `RDD` itself. Indeed, usually, an `RDD` is processed at the output stage with a few operations that will get executed to make sure that the relevant information is sent to an output sink - for example on the standard output for one particular machine, to a database, or to any other system that is able to receive some information extracted for interesting elements of the `RDD`.

Within the argument to `foreachRDD` , this final `RDD` needs to be operated on in an explicit manner. And as a consequence, this involves selecting, filtering, or aggregating some of the elements of this `RDD` . After this operation, the user is faces with the choice between collecting all the elements of the `RDD` on the driver machine or writing them on an ouput location from where there are, that is, distributed among the spark executors of the cluster.

```
1 dstream.foreachRDD { rdd => // implicit repetition over time
2   val connection = createNewConnection()  // executed at the driver
3   rdd.foreach { record => // explicit repetition over elements
4     connection.send(record) // executed at the worker
5   }
6 }
```

A few considerations then apply:

- if the data in play is larger than what can be comfortably stored within the memory of the driver, keeping the data in distributed partitions across the executors may be the only solution. With that being said, we will see int he rest of the chapter that writing an `RDD` of such a humongous size - probably commensurate with the size of the input `DStream` - is an anti-pattern.

- if the data can be confortably retrieved on the driver, this output pattern is more susceptible to a failure of the driver. Spark will recover from a driver failure (if the driver program runs in cluster mode) but the recomputation of all the precious output elements may be costly, especially if the `RDD` being collected has not been put into cache. The advantage, however, is simplicity : the output operation can be written as if saving the contents of a Scala collection to an output destination.

- if on the other hand, the output destination can take several simultaneous connections - on

the order of the number of executors in the cluster - it may be more robust and just as simple to write from the partitions of the RDD, using a function such as foreachRDD.

The consequence of that is that there very often is a second repetition over the elements of the RDD , from within the function passed as an argument to foreachRDD . Note however code that occurs in the foreachRDD argument but which is **not** operating on the RDD is **not** distributed, but run on the driver node. This is, in fact, logical : one of the precepts of Spark is that the central distribution data structure that creates jobs and tasks, as we have seen numerous times until now, is the RDD.

As a consequence, the following pattern emerges as a code smell or as a beginner tell. Usually in non-trivial operations, a software developer wants to connect to a database, to a streaming sink, or to some kind of entity over the network, to which the results of the analysis contained in some elements of the upward stream will be sent. That connection is being created as a first thing to do in the foreachRDD and is usually the first line of the function that he writes as an argument to for each RDD . That gets executed on the driver (no RDD is involved in creating a connection), so that the driver now holds a reference to a new connection to the database server.

Then operating on the output RDD , doing a foreach on every element, the software developer uses the previously created connection to operate on the elements of the RDD . The problem with this approach is that this second part, supposed to occur on every element of the RDD , is actually run on the Spark executors. For it to succeed, Spark would need to be able to serialize the database connection that has been created on the driver.

The issue is then that this connection is not serializable. The next step approach is to create a connection to the database server on every executor. It's a simple fix : the developer simply moves the line creating the connection to the executor within the foreach brackets.

```
1 dstream.foreachRDD { rdd =>
2   rdd.foreach { record =>
3     val connection = createNewConnection()
4     connection.send(record)
5     connection.close()
6   }
7 }
```

The issue there is that the connection is put by the software developer in the foreach , explicit repetition of the RDD . The connection is therefore created as many time as there are elements to report on for the current batch. While it could possibly acceptable for every executor of a cluster to have one connection to the local database, it is not acceptable to have as many connections in one executor as that executor holds elements of the RDD . Let's assume we have a stream of 1 million elements per batch, of which we will need to report 0.1%. We will then have a total of 1000 elements to report on. If we have 10 executors, and if we assume that both data and the propotion of output elements are equally shared among executors, each executor will need to report on 100 elements. If each executor opens a connection to our output server, it will need to be able to withstand 10 simultaneous connections. If we create one connection per element, it will have to withstand 1000 connections.

This is traumatic because this plethora of connections is not only taxing for the output server, but it is a wasteful use of resources on executors. An executor can reuse a connection to send several records, one after the other, to a database server. The correct pattern therefore is to make sure that a singleton object contains a factoring method or connection pool that allows every single

executor to get a reference to the unique connection being created on an executor and this is what we're going to show in the example below.

```
1 import com.sksamuel.elastic4s.{ElasticClient, ElasticsearchClientUri}
2 import org.elasticsearch.common.settings.ImmutableSettings
3
4 object ESConnection extends Serializable {
5
6   private[this] var elasticHosts: Option[List[String]] = None
7   private[this] var elasticTransportPort: Option[Int] = None
8   private[this] var elasticClusterName: Option[String] = None
9   def setClusterName(s: String): Unit = {
10     if (!elasticClient.isDefined) elasticClusterName = Some(s)
11   }
12   def setTransportPort(i: Int): Unit = {
13     if (!elasticClient.isDefined) elasticTransportPort = Some(i)
14   }
15   def setHosts(s: List[String]): Unit = {
16     if (!elasticClient.isDefined) elasticHosts = Some(s)
17   }
18
19   def elasticClient: Option[ElasticClient] =
20     (elasticHosts, elasticTransportPort, elasticClusterName) match {
21       case (Some(hosts), Some(port), Some(name)) =>
22         Some(getOrCreateClient(hosts, port, name))
23       case _ =>
24         None
25     }
26
27   private[this] var client: Option[ElasticClient] = None
28   private def createClient(elasticHosts: List[String],
29                            elasticTransportPort: Int,
30                            elasticClusterName: String): ElasticClient = {
31     val uriHosts = elasticHosts.map(host => s"""${host}:${elasticTransportPort}""")
32     val elasticURI =
33       ElasticsearchClientUri(s"""elasticsearch://${uriHosts.mkString(",")}""")
34     val settings =
35       ImmutableSettings
36         .settingsBuilder()
37         .put("cluster.name", elasticClusterName)
38         .build()
39     val eClient = ElasticClient.remote(settings, elasticURI)
40     client = Some(eClient)
41     eClient
42   }
43   private def getOrCreateClient(hosts: List[String],
44                                 transport: Int,
45                                 name: String): ElasticClient = {
46     client.getOrElse(createClient(hosts, transport, name))
47   }
48
49 }
```

This example has as a goal to create a connection to an ElasticSearch Server, in order to push to a particular index. It can be reused inside a `foreachRDD` command, in the following way:

```
1 ESConnection.setHosts(elasticHosts)
2 ESConnection.setClusterName(elasticClusterName)
3 ESConnection.setTransportPort(elasticTransportPort)
4
5 val esClient: Option[ElasticClient] = ESConnection.elasticClient
6 esClient.foreach{ (client) => client.execute {
7   index.
8   into(s"$elasticIndex").
9   id(s"$timeStamp").
10   doc(speeds)}.await
11 }
```

Note that Spark already has a command to save an `RDD` to ElasticSearch: `rdd.saveToES` . But this built-in command, while convenient, is not very flexible as to the index and content that we save. Our tool lets us be more flexible.

In this example, we create an ElasticSearch client in a Scala singleton object, which guarantees us that only one instance of such an object will exist in an executor. We access the connection, if initialized, with a cache mechanism implemented in a manner similar to the lazy variables of Scala. The use of `Option` (a collection type that contains zero or one elements) lets us deal with the case where the `elasticClient` object has not been defined for lack of connection parameters. In sum, any call to `ESConnection.elasticClient` , if it returns, will return the same connection on any given machine.

This simple example is powered by one of the simpler features of Scala - singleton objects - and by the fact the elasticSearch client already deals with some of the more troublesome aspects of connection reuse. However, the subject of connection pooling has already been adressed in Java libraries, and in particular, it is also possible to use `ObjectPool` class of the Apache Commons `pool2` package to deal with the necessary object pooling from the Java world.

Note

The `ESConnection` object is declared as `Serializable` in the example above, because in the case where Spark uses a state stream (the category of `DStream`s that keep some internal state between batches, such as the ones created by the `updateStateByKey` or `mapWithState`), it does enforces the serialization of every single operation, including the whole contents of the `foreachRDD` operation. There are excellent reasons for this check, that we will evoke in Chapter 4, but there are some parts of the content of a `foreachRDD`operation to which it need not apply. Creating a more lenient serializability check is a task in progress.

Of course, there are various connectors that exist to help with saving `RDD` content directly on a particular sink. We can cite :

- the ES-Haadoop connector, which lets Spark write an RDD to an ElasticSearch Index *https://github.com/elastic/elasticsearch-hadoop*
- the Reacive Influx-Spark connector that lets Spakr write an RDD of metrics to influxDB *https://github.com/pygmalios/reactiveinflux-spark*
- the Cassandra connector *https://github.com/datastax/spark-cassandra-connector*

The common point between all these connectors is that they write an `RDD`, using said `RDD` as a primary block for the parallelization of the write operation. You'll find more connectors on Spark-packages (*https://spark-packages.org/*).

A word on content selection

Working with a stream can be confusing at times: at a very high level, stream processing frameworks only take care of parallelizing the computational load — in clever ways, for sure, but that's their only scope.

It might mean you need an additional storage layer to store the results in order to be able to query these streams. While the current state of the computation is contained in the stream processing engine, there is usually no way to access or query this information. Depending on the amount of data you are processing, this might mean you need a pretty large and high performance storage back end.

Often, however, it pays to spend a bit of time thinking about the data flow and storage pattern of a streaming application as a whole rather than just paying attention to the scalability of your storage solution. Frequent patterns in stream processing include:

n- applications that write about as much data as they receive, otherwise known as Extract, Transform, and Load jobs. Those applications have as a goal to prepare and merge data sources in a consistent form, often adding structure, and intending the results to be themselves consumed by other applications. In that case, saving all the results of the operation in a provisioning load may be extremely costly. In fact, the input data is copied as many times as there are ETL (Extract, Transform & Load) jobs. Better architectures often involve feeding back the output in another messaging system (such as Kafka), without passing through a storage system other than caching. That way, rather than working with a highly structured pile of data, you can envision working with a "tap" of highly structured data.

- applications that write a "compressed", or "summarized" form of the data, collecting key indicators on the stream of input. Examples include counters, or time-based aggregates of the input data. In this particular case, a common pattern is that the serving layer for these aggregates is often not a disk, but fall into two categories: SQL like query languages (Drill, Vertica, parstream, streambase), or functional collections (Scalding, Summingbird).

- applications that run structured aggregations from iterative algorithms. Most supervised learning algorithms, for example, have a highly organized iterative structure, going through rounds of an error optimization procedure, before outputting result parameters which get tested. This requires a global comparison, which needs to ship parameters across the parallelize workers owning a part of the computation. Other distinct algorithms, such as Page Rank (when computed with a random walk) share that parameter sharing. Those applications often do not need to be explicitly queried. Their production usage is often real-time itself Instead, they are often plugged through other parts of an application pipeline, as is the case with monitoring, Real-Time Bidding (RTB), or more generally predictive computation. They often feed results to an alerting system, or to a visualization engine. But in this case, developers often see it as valuable to have a log of the running application accessible to debug which decisions it reached and results of computations which led to these. Rather, this use case is a good scenario for indexed logs search engines, such as Splunk, for example. It is possible to structure the results of the application into

such structured indicators that they can be fed in a database, but since the use of these outputs is mostly about historical debugging and forensics, most teams give less attention to that structuring, and use at most the loose specification afforded by NoSQL databases (such as Redis).

Reasons for saving a stream and scaling into real-time

Notice that in the patterns above, we rarely if ever employ a hard drive or a database to save the results of a streaming application. In the cases where it's appropriate, it's often for purposes other than the main production delivery of the application. This is not a coincidence, though there certainly are cases in which hard drives and databases are appropriate delivery mechanisms.

The reason to be wary in front of media used to represent "data at rest" is that they hide one of the fundamentals of stream processing, on which we will circle around several times in this book: data never stops coming, with a stream. As a consequence, when considering storage, it is important to also consider the consequences of this fundamental principle. In particular:

- Most distributed file systems and databases have a cost of increment, which means that it takes longer to append data to a file or a table when the file (resp. table) is already big than when it is freshly created. But since data input for a streaming application is always coming in, it is inevitable that such a destination will grow large as results keep piling up. Since a streaming application must execute in a bounded amount of time to keep its promise to operate in real-time (or near real-time), the time to save the results of the analysis is itself bounded and cannot be allowed to grow indefinitely. [6] That is not to say that there is no way to get around that restriction of data storage. In fact, a crude first solution is to change the destination to which results are written to a new file (or table) every time the size of this destination becomes too large. But the aim of this point is to highlight that a storage solution must be envisioned in a more comprehensive fashion that for planning to just write the output of a single, one-input-to-one-output batch application.

- Most real-time applications do not interact very well with stale data. For many of the practical applications seen in the usage of streaming applications, it is unclear whether using very old data for training machine learning models or basing a computation upon is more harmful than good. For example, fraud and anomaly detection are subject to trends which form and disappear in waves: flagging as fraudulous a case just because it looks like a very old fraud might appear a less optimal use of resources than focusing on the most recent trends. Risk estimation and pricing in financial markets might prefer to avail itself of the fresh condition of markets rather than analyze data over long tie periods that might have become irrelevant because of, e.g. a change in regulation. Naturally, there is an appropriate time scale for which keeping the results of old analyses — as well as, possibly, saving input data — may make sense, depending on the exact use case. But the idea there is that every sector will have its "peremption": a data at which old data and old results have stopped being relevant to the analysis at hand. Good domain knowledge helps in determining which exact time frame this corresponds to, and the best use of resources then consists in tailoring storage solutions to this boundary. In concrete terms, it means at least that the compaction facilities of NoSQL databases — which often make the choice of never executing in-place updates, when they are distributed [7] — should be used effectively. When those features become problematic because of their run time, the good architect will instead turn to data structures that include a form of exponential weight

decay, integrating peremption as a feature.

As a consequence, Stream saving should be considered with some care given not only to the scalability of the storage solution for a given time frame analysis, but also as a whole, taking into account:

- the cost of storing data and its evolution during the life cycle of the application,
- the usefulness of having results stored, and their relevance for future analyses or stake holders.

How to Save Streams with DataFrames

In saving Streams with Apache Spark, we usually want to reflect the stream of results that stems from a computation, whether complex or simple. However, the code for saving the result of a Stream has to be the same for all batches results received over time, and is usually executed, as we have seen above, in a `foreachRDD` statement. As a consequence, it's important to envision a data sink that can receive those results on a continuous basis.

Another aspect is that the results of a computation are usually much more structured than the entry data that is treated by Spark. Indeed, analytics is often about adding successive levels of refinement: while the input may be a simple log, it's often useful to treat the output as more regular.

Hence, Spark Streaming's output is often envisioned jointly with Dataframes. The pattern here is not to save the whole stream, for the reasons we extensively treated about. They are about the careful elaboration of a `DataFrame`object with a few records, that summarizes all the information gathered and computed in the last batch in a highly structured form.

One example is then to output this `DataFrame` as an addition to a database server, or more simply to a parquet record. Indeed, several of the column-oriented datastores allow appending, which can be useful to keep a particular structured record of results durig the lifetime of our application.

Saving a DataFrame is generally implemented in a `foreachRDD`` command. In the folowing, we write a DataFrame to parquet:

```
1 dstream.foreachRDD((rdd: RDD[(String, Int, String)], time: Time) => {
2   val df = rdd.toDF("field1", "field2", "field3").na.drop()
3
4   df.write.mode(SaveMode.Append).format("parquet").save(outputPath)
5 })
```

Modules exist to write `DataFrames` to various sinks. Of note we will cite the `spark-csv` package, maintained by DataBricks, the creators of Spark.

```
https://github.com/databricks/spark-csv
```

Spark can, of course, write a dataframe to a file, and avail itself of numerous other connectors to be found in Spark-packages (*https://spark-packages.org/*).

Caution

Saving in append mode can sometimes, such as with the Parquet format, cost more on each iteration, where the point of append has to be organised in an ever larger set of metadata. This increasing time to save on every `RDD` means that the processing time for a batch will eventually grow beyond the batch interval. To solve this, an easy workaround is to change the output destination on a regular basis:

```
1 dstream.foreachRDD((rdd: RDD[(String, Int, String)], time: Time) => {
2   val df = rdd.toDF("field1", "field2", "field3")
3               .na
```

```
4               .drop()
5   def appendix: String =
6     ((time.milliseconds - timeOrigin) / (3600 * 1000)).toString
7
8   df.write
9     .mode(SaveMode.Append)
10     .format("parquet")
11     .save(s"${outputPath}-H${appendix}")
12 })
```

Bibliography

- [Narkhede2016] Neha Narkhede, Gwen Shapira, Todd. *Kafka : The Definitive Guide.* O'Reilly Media, 2016. ISBN 1-4919-3616-9

- [Stonebreaker2005] M Stonebraker, U Çetintemel, S Zdonik, *The 8 Requirements of Real-Time Stream Processing.* ACM SIGMOD Record (34)4:42-47. URL

- [Kleppmann2015] Martin Kleppmann, *A critique of the CAP theorem* Technical Report, arXiv:1509.05393, Sep 2015 URL

- [Kreps2014] Jay Kreps, *Questioning the Lambda Architecture.* O'Reilly Radar July 2, 2014. URL

- [Kuthan2016] Kuthan, M. *Long-running Spark Streaming Jobs on a YARN Cluster* Marcin Kuthan Blog. September 30, 2016. URL

- [Marz2011] Nathan Marz. *How to beat the CAP theorem.* Blog post, October 13, 2011. URL

- [Marz2015] Nathan Marz and Jamez Warren. *Big Data: Principles and best practices of scalable realtime data systems.* Manning Publications, May 2015. ISBN 1-6172-9034-3

- [Noll2014] Michael Noll. *Integrating Kafka and Spark: and Spark Streaming: Code Examples and State of the Game.* Personal blog. Oct 1st, 2014. URL

- [Sharp2007] Alexa Megan Sharp. *Incremental algorithms: solving problems in a changing world.* PhD thesis, Cornell University, Aug., 2007 URL

- [Shapira2014] Gwen Shapira. *Building The Lambda Architecture with Spark Streaming.* Cloudera Engineering Blog. Aug 29, 2014 URL

[1] Another notable application Is the Apache Zepplin project, which has a slightly more generic focus. Finally, Jupyter is the backend-agnostic project that is the descendent of iPython notebooks, and can be used with Apache Spark as well

[2] You can go beyond this limitation by using the concept of consumer groups in Kafka, but this seen from the point of view of Spark Streaming involves running several parallel applications and at the application level, it is sometimes difficult to merge data sources into one single Spark Streaming processing engine. For those reasons, we will simply find a situation and bypass the running of multiple consumer groups in parallel.

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

[4] if P≠NP, which is highly probable

[5] in the worst case, a customer filed his order at 9:01 AM the previous day, and you answer him on the next day's shipment, at 4:59PM

[6] In fact, the author was surprised when he witnessed that the only operation that made one of his production jobs fail — and that after many hours of running predictably well — was that a summary of the ultimate results was regularly appended to a parquet file. The appending operation was searching for a point of insertion that was taking longer and longer to find as this parquet file grew, eventually making the application unstable.

[7] This choice is related to the limits inherent to distributed computing, selecting only some of the classical CRUD operations. See also <ref>

Chapter 4. Creating robust deployments

Using `spark-submit`

In the last chapter, we have studied how to create an interactive applicaiton using the Spark Notebook. To get a clearer idea of how Streaming applications are really used in production, we are now going to focus on doing this with Spark's `spark-submit` script. Note that the stages used in this are important notions that will be reused in the application.

Let's consider again our Twitter streaming application from [Link to Come], that counted the most frequent hashtags in tweets.

```
1 package learning.streaming.demo
2
3 import org.apache.spark.streaming.{Seconds, StreamingContext}
4 import org.apache.spark.SparkContext._
5 import StreamingContext._
6 import org.apache.spark.streaming.twitter._
7
8 /**
9  * Collect at least the specified number of tweets into json text files.
10  */
11 object BestHashTags {
12
13   def configureTwitterCredentials(apiKey: String,
14                                   apiSecret: String,
15                                   accessToken: String,
16                                   accessTokenSecret: String) {
17     val configs = Seq("apiKey" -> apiKey,
18                       "apiSecret" -> apiSecret,
19                       "accessToken" -> accessToken,
20                       "accessTokenSecret" -> accessTokenSecret).toMap
21     configs.foreach{ case(key, value) =>
22                      if (value.trim.isEmpty) {
23                        throw new Exception(s"""Error setting authentication
24                                               |- value for  $key  not set""".stripMargin('|'))
25                      }
26                      val fullKey = "twitter4j.oauth." + key.replace("api", "consumer")
27                      System.setProperty(fullKey, value.trim)    }
28   }
29
30
31   def main(args: Array[String]) {
32     // Process program arguments and set properties
33     if (args.length < 4) {
34       System.err.println(
35         s"Usage: ${this.getClass.getSimpleName}" +
36         "<intervalInSeconds> <apiKey> <apiSecret> <accessToken> <accessTokenSecret>"
37       )
38       System.exit(1)
39     }
40
41     val Array(intervalSeconds, apiKey, apiSecret, accessToken, accessTokenSecret) = args
42
43     println("Initializing Streaming Spark Context...")
44     val conf = new SparkConf().setAppName(this.getClass.getSimpleName)
45     val sc = new SparkContext(conf)
46     val ssc = new StreamingContext(sc, Seconds(intervalSecs))
47
48     val filters = Array( "spark" )
49     val tweetStream = TwitterUtils.createStream(ssc, None, filters)
50
51     val hashTags = twitterStream.flatMap(status => status.getText.split(
52       " ").filter(_.startsWith("#")))
53
54     val topCounts60 = hashTags.map((_, 1)).reduceByKeyAndWindow(_ + _, Seconds(60))
55       .map{case (topic, count) => (count, topic)}
56       .transform(_.sortByKey(false))
57
58     topCounts60.foreachRDD(rdd => {
```

```
59       val topList = rdd.take(10).toList
60       val r = topList.map{case (count, tag) => s"$tag: $count"}
61       println(r)
62     })
63
64     ssc.start()
65     ssc.awaitTermination()
66   }
67 }
```

Note how we are creating a singleton object with a `main(args : Array[String])` method, which in Scala creates a runnable application. This is the class we want to pass as an argument to `spark-submit`.

Thinking about reliability in Spark Streaming: Closures and Function-Passing Style

We have already mentioned the concept of a *closure*: the pair formed by a function and its arguments, along with the variable definitions necessary for its evaluation. It is conceptually what Spark is meant to package (serialize) and distribute across a cluster of executors.

The key to performance in this behavior is in avoiding to move around arguments of a closure as much as possible, since arguments are, in the context of big data and distributed computing, hard to move across the network. But since this involves separate fallible computation units, usually separated by a computer network, making sure that closures reach their execution environment safely and reliably, and report the result of their execution faithfully is actually a non-trivial task. Hence, Spark has several measures aimed at helping with fault tolerance.

To understand at what level fault tolerance operates in Spark Streaming, it's useful to go through a reminder of the nomenclature of some basic concepts in Spark. These notions have already been addressed and should by now be familiar to you.

- the user application in Spark Streaming is composed of user-specified *function calls* operating on RDDs or DStreams, categorized as *actions* and *transformations*.
- the user program may undergo adjustments that coalesce some of the specified calls, the most approachable and understandable of which is map-fusion [1].
- these remaining operations are then grouped into *stages*, at the boundaries of which a shuffle occurs.
- once these *stages* are defined, what internal actions Spark should take is clear. Indeed, at this stage, a set of interdependent *jobs* is defined. The dependence relation can be many-to-many, which means this data flow and dependency information is represented as a graph.
- jobs can then, depending on where their source data is on the cluster, be cut into *tasks*, crossing the conceptual boundary between distributed and single-machine computing : a task is the name for the local, executor-bound part of a job.

Spark aims to make sure that all of these steps are safe from harm, and to recover quickly in the case of any incident occurring in any stage of this process. This concern is reflected in fault-tolerance facilities that closely mirror the notions above.

Spark’s Reliability primitives

First, Spark saves the user-provided data flow of the application in a directed acyclic graph (DAG), a data structure that lets Spark represent the inter-dependency between some steps of the computation. That DAG is stored on the driver machine.

Second, Spark offers the possibility to store the data of RDDs at a configurable storage level, which lets users persist the results of an intermediate stage of computation. However, since the user operations in Spark are lazy and grouped into conglomerates of operations, there is no particular reason for Spark to store the result of every function call. However, at the points at which an intermediate state in computation needs to be materialized and passed on — such as a shuffle boundary — Spark will use the replication instructions inherent in the storage level setting.

Caution

It is a point of confusion for some that Spark may not have materialized the results of the data for every operation in a chain of user calls consisting of map-like operations (map, filter, etc) in Spark. Because map operations are lazy, Spark knows without computation where the results of a particular map will be used — and if this “where” is not a shuffle operation but a map-like operation, it will jump to directly computing the chain of the maps rather than saving an intermediary result. Of course, the user can use a call to `cache()` or `persist()` after any RDD operation, explicitly materializing the corresponding in-memory or according to the storage level. Note that Spark understands dependencies accurately, but does not implement the book-keeping necessary to figure out that if a piece of data (an intermediary RDD) is used in two, independent chains of operations, computing this data should — in most cases — be done once and saved while switching from one chain of operations to the other one that depends on it. Indeed, some policies as to what data should be put in cache could lead to particularly adverse results, so that this book-keeping is effectively delegated to the user. We’ll see an example where inadvertent caching could be very detrimental to available memory when addressing the subject of caching in Spark Streaming specifically.

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

We will start with the Receiver model, which is the most straightforward and generic way of dealing with Streaming.

Spark’s Fault Tolerance Guarantees

Spark already has a few fault tolerance facilities that we can relate to. For starters, if a task fails, whether or not the underlying executor it was running on crashes, it can be re-launched on another executor. [2] But this is an oversimplification of the actual endeavor of running this task again.

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

Level	Uses Disk	Uses Memory	Uses Off-heap storage	Object (deserialized)	# of replicated copies
`NONE`					1
`DISK_ONLY`	X				1
`DISK_ONLY_2`	X				2
`MEMORY_ONLY`		X		X	1
`MEMORY_ONLY_2`		X		X	2
`MEMORY_ONLY_SER`		X			1
`MEMORY_ONLY_SER_2`		X			2
`MEMORY_AND_DISK`	X	X		X	1
`MEMORY_AND_DISK_2`	X	X		X	2
`MEMORY_AND_DISK_SER`	X	X			1

MEMORY_AND_DISK_SER_2	X	X		2
OFF_HEAP			X	1

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

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

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

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

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

When this failure occurs, it always means some roll-back of the data, possibly up to a significant number of operations if an intermediate cache or persist operation does not offer a close-by check point.

On the occasion of a shuffle, if one executor dies, it blocks the whole shuffle operation, since it may contain arbitrarily important data, that many — if not all — other executors could depend on to complete their operation. For example, when doing a word count, we first transform words — arbitrarily distributed through a cluster — into words with a local count, in a map operation. The shuffle then adds the local counts in a sorted fashion, so that the intermediate local counts for a specific word are sent to a single downstream executor.

If the upstream map operation dies, the local counts for every word in its documents disappear with it. The re-computation of the counts of this particular dead mapper may involve re-processing from scratch, up to and including reading anew the HDFS-stored document these local counts came from. In sum, the last point before a shuffle marks the end of a possibly long computation, and it may be interesting to make this particular boundary more resilient to faults — or more precisely, able to recover from faults more quickly.

The External shuffle service

As a consequence, Spark 1.3 has introduced the *shuffle service*, which lets users work on map data that is persisted and distributed through the cluster with a good locality, but more importantly through a server that is not a Spark task. It's an external file exchange service in Java, which has no dependency on Spark, and is made to be a much-longer running service that a Spark executor. This additional service attaches as a separate process in all cluster modes of Spark, and simply offers a data file exchange for executors to transmit data reliably, right before a shuffle. It is highly optimized through the use of a netty back-end, to allow a very low overhead in transmitting data. This way, an executor can shut down after the execution of its map task, as soon as the shuffle service has a copy of its data. And because data transfers are faster, this transfer time is also highly reduced, reducing the vulnerable time in which any executor could face an issue.

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

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

But more insidious than this is the fact that the driver program can be lost out of convenience rather than unplanned incidents. Indeed, far from a fault tolerance condition about the reliability of the analyst's personal computer, what limits the run time of the driver program is often simply the ability of the user to keep his laptop open and running, without going out of WiFi range to catch a train home.

Cluster-mode deployment

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

Caution

The cluster deployment mode, in Spark parlance, has a very different meaning from the appellation of cluster mode, but the close names often lead to confusion. The deployment mode is one of two options: in client mode, the driver is launched in the same process as the client that submits the application. In cluster mode, however, the driver is launched from one of the Worker processes inside the cluster, and the client process exits as soon as it fulfills its responsibility of submitting the application without waiting for the application to finish. Choosing a cluster mode, however, says something about the framework and software platform that runs on the cluster and hosts Spark computations (currently, Mesos, YARN and Standalone are the cluster modes of Spark).

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

Checkpointing

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

We will come back to the subject of checkpointing later, since it has a particular meaning in Spark Streaming or GraphX: reducing the lineage graph for data in situations where this lineage may become too long to allow an efficient recovery. In the meantime, there is still one component of any Spark cluster whose potential failure we have not yet addressed: the master node.

A hot-swappable master through Zookeeper

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

This tour of Spark-core’s fault tolerance and high-availability modes should have given us an idea of the main primitives and facilities offered by Spark and of their defaults. Note that none of this is so far specific to Spark Streaming, but that all these lessons apply to Spark Streaming in that they are required to deliver long-running, fault-tolerant and yet performant application. Note also that these facilities reflect different concerns in the frequency of faults for a particular cluster: while facilities as a spare master node kept up-to-date through Zookeeper are really about avoiding a single point of failure in the design of a Spark distributed application, the Spark shuffle service is here to avoid any problems with the end of a long string of map-like applications ending in failure because the executor serving these results is about to die — a much more frequent occurrence. The first is about dealing with every possible condition, the second is more about ensuring smooth performance and efficient recovery.

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

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

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

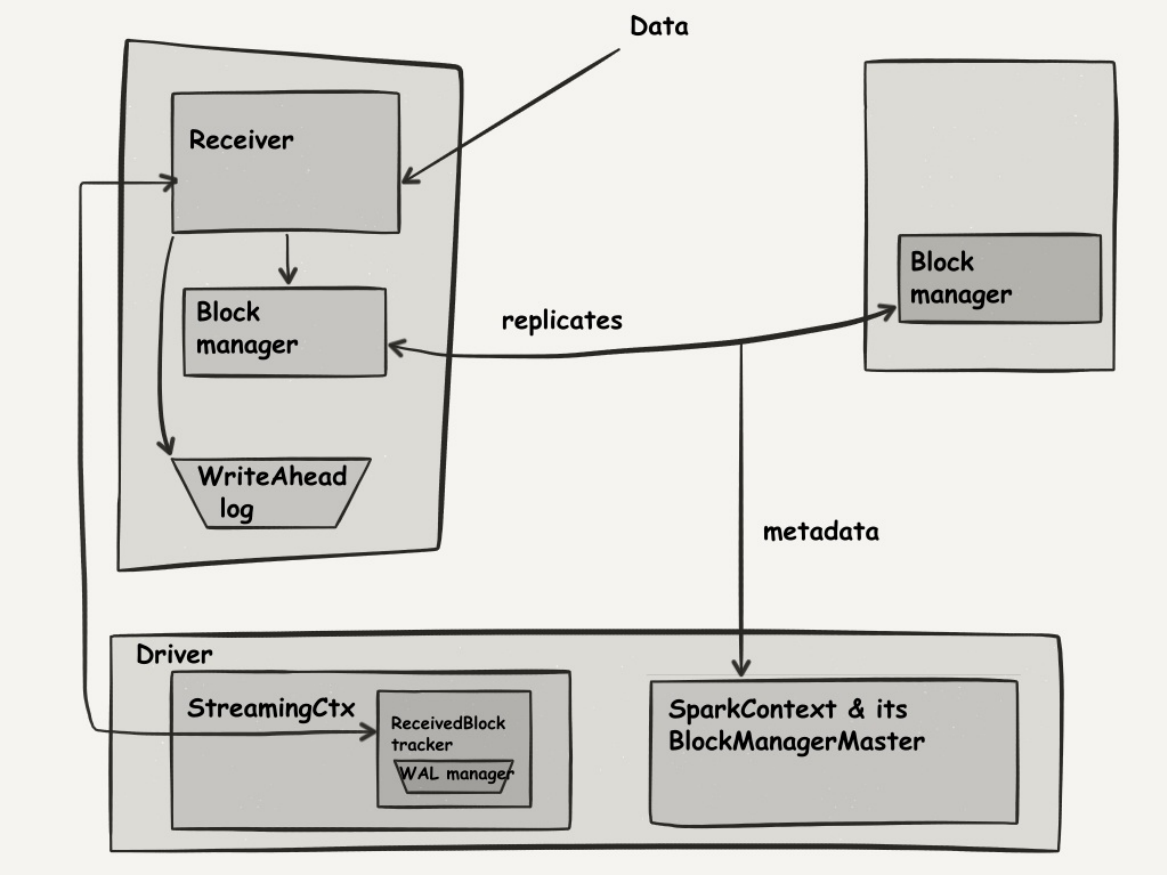


Figure 4-1. The Data flow of Spark's Receiver

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

On each block interval tick (as measured on the executor that is running the Receiver), this

machine groups the data received for the previous block interval into a block, which is then registered with the Block Manager, also present in the bookkeeping of the Spark Context, on the driver. This can initiate replication of the data that this block represents, to ensure that the source data in Spark Streaming is replicated the number of times indicated by the storage level.

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

Synchronizing Executor clocks

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

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

As a consequence, when accessing a single data source, it has become a standard best practice to not create a single `DStream` for this data source. If it is at all possible to serve this data in parallel, users concerned with locality often create several Receivers that share a connection pool with the data source. Spark Streaming will in turn create several Receivers, which will share the load of ingesting this data. The `union` function in the `DStream` API then lets Spark consider these several DStreams be one single merged stream, allowing the user to proceed as if this was a single entry point all along. One good thing about having the Receiver run as a job on the cluster is that it can be re-launched if and when the machine on which the Receiver runs — or the Receiver task itself — dies. In that case, data ingestion will start again on another executor. In the case of a Receiver failure, however is what happens with the data that was received until then? Because the Receiver replicates blocks of data as it is receiving them, the replications, modulo a small replication delay will still be available at the next batch interval to be part of the following RDD. But what of the data that was not yet part of a block? The data that was received from the block interval right before the Receiver's failure and said failure was only present on the Receiver, and because it was not part of a block, it could not yet be replicated.

Spark Streaming's Zero Data Loss guarantees

To solve this problem, Spark introduced a high-availability option called the WriteAhead log. Spark can optionally write all the data it receives — as it's receiving it and in a blocking fashion — to a log file stored in a safe location such as HDFS[1]. Since Spark Streaming takes this Write-Ahead Log into account during the recovery of the driver, this means that a surviving log guarantees that no data has been lost during a Receiver failure. This Zero-data loss result is obtained, however, at the very high performace cost of writing to HDFS in a blocking fashion. As a consequence, few users really need to implement it. But this setting can be particularly important for some applications where the loss of even a little data could be catastrophic, such as anomaly detection. NOTE: [1] Note that the assumption, made several times in this book, that a distributed file system is a safe place to write data to requires the probability of failure of HDFS data nodes to be independent from that of failure of Spark executors. If this happens not to be the case, for example if HDFS nodes and Spark nodes are colocated on the same hardware, one needs to check the latency and locality of write actions from Spark to HDFS. It would be too bad to try to back up some data from a Spark node to an HDFS data node, for the hardware that runs them both to fail during the backup.

Finally, one important point that is glossed over in this schema is that the Receiver interface also has the option of connecting to a data source that delivers a collection (think Array) of data pieces. This is particularly relevant in some de-serialization uses, for example. In this case, the Receiver does not go through a block interval wait to deal with the segmentation of data into partitions, but instead considers the whole collection reflects the segmentation of the data into blocks, and creates one block for each element of the collection. This operation is demanding on the part of the Producer of data, since it requires it to be producing blocks at the ratio of the block interval to batch interval to function reliably (delivering the correct number of blocks on every batch). But some have found it can provide superior performance, provided an implementation that is able to quickly make many blocks available for serialization.

Caution

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

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

Name	Settings to activate	Goal
The Write-Ahead log	`spark.streaming.receiver.writeAheadLog.enable`	Zero data loss in data ingestion
The Shuffle service	`spark.shuffle.service.enabled`, `spark.shuffle.service.port`	Efficient exchange of shuffle files, recovery from late mapper failure
High availability of the master (with Zookeeper)	`spark.deploy.recoveryMode`, `spark.deploy.recoveryDirectory`, `spark.deploy.zookeeper.url`	Avoiding a single point of failure at master
Checkpointing	call `sparkContext.setCheckpointDir(checkpointDirectory: String)`, create context w/`StreamingContext.getOrCreate(checkPointDirectory)`	Limiting RDD Lineage
Persist & Storage Level	call `myRDD.persist(StorageLevel.CHOOSE_A_STORAGE_LEVEL)`	Avoiding prohibitively long recovery
Cluster deployment mode	pass `--depoy-mode cluster` to spark submit, consider `--supervise`	Recovery from driver failure

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

Note that, however, the usage of fault tolerance measures in Spark should not aim at being exhaustive. Setting up a cluster in pure fault tolerance mode, with all the above settings activated, will only serve to lower the performance of the cluster, and increase the complexity of maintaining it. A better approach is to set up fault tolerance facilities as needed, evaluating if the resiliency guarantee is worth the performance trade-off in each case.

Cluster managers and driver restart

Running Spark can be done in one of four cluster managers. Three of these cluster managers offer the possibility to run Spark in a “real” distributed Spark context. The other is the local mode, which emulates the behavior of executors in the distributed context using threading. It is mostly used for testing purposes and for quick prototyping, leveraging the availability of a similar environment as the distributed one.

The first and foremost cluster manager is called Spark Standalone. It is the one that is bundled with Spark, in the sense that its jars come with the distribution, and that its source code lives in the Spark repository. It offers in general the best performance, since this is where the Spark developers prototype, and then develop new features. For instance, though not publicized, the cluster deployment mode was offered in the Spark standalone cluster first. Its particular property is that it assumes a single-tenant deployment, that is that the only version of Spark (or, for that matter, analogous service) running on the cluster is itself. Its purpose is not to keep a particular awareness of other things running on the cluster than its own Spark processes.

YARN is the resource manager of Hadoop, starting with version 2.0. It runs its processes (confusingly called containers, despite having no relationship to the Linux kernel’s container technology) in a JVM, and as such is able to schedule, distribute and oversee mainly this sort of processes. It is a mature piece of software, equipped with all the bells and whistles implied by a corporate environment. In particular, at the time of this writing, YARN is the only cluster manager which is capable of configuring strict authentication of every job, along with per-field data access control.

Mesos is a cluster manager that was born at Berkeley University in 2009 and became a top level Apache Project in 2013 after two years in incubation. It works with other schedulers, offering a way to host entire competing systems for job running (Spark, Storm, MapReduce …) on top of it, while scheduling their jobs efficiently. In sum, Mesos calls itself a meta-scheduler for frameworks — where each framework is an application that solves a specific use case, and consists of at least one master and one executor.

Note

An ill-known fact is that Spark was created as a demonstration framework for Mesos — Both projects then took off in their own direction.

Mesos is based on Linux container technology, a capability of modern Linux kernels to offer lightweight virtualization — for lack of a better term — and resource accounting for these “containerized” applications. By taking this single-machine accounting to a global understanding of resource utilization across a fully distributed environment, Mesos offers a sort of operating system for the data center, allowing for multi-framework, multi-tenant use cases.

Comparing cluster managers

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

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

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

Note

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

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

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

YARN	`sparkContext.checkpoint("hdfs://myDirectory")` and persist the `DStream` strategically	set the shuffle service in the YARN configuration	Launch several receivers and union them,
Mesos	`sparkContext.checkpoint("hdfs://myDirectory")` and persist the `DStream` strategically	configure and launch the shuffle service on each executor	Launch several receivers and union them,

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

- if the long-running Spark Streaming application starts losing nodes — for example if the application faces serious trouble in a multi-tenant cluster, it may become progressively instable. Indeed, we have already mentioned that the executors Spark Streaming has available allow it to keep a job stable only if it keeps the execution time for each given RDD below the batch interval, on average. A consequence of that is that jobs that lose executors, since those executors never reconnect to the application (especially in cluster modes that assign a set of executors once and for all, such as YARN and Mesos) often go even more unstable because they start overwhelming the job completion capabilities — and therefore soon, the memory — of their executors.

- if the long-running Spark execution has a bad executor-to-node distribution. This is pretty much limited to the Mesos coarse deployment mode, which has strong constraints on having a fixed set of executors negotiated at the launch of the application. Nonetheless, if for example all the Receivers get concentrated on a single physical machine, these containers end up competing for network resources.

The consequence of the importance of a set of executors delivered at the beginning of the application is that making this set of executors less fixed has great benefits in terms of failure recovery as well as redistribution of resources. As of Spark 1.5, all cluster modes support dynamic allocation, a mode that allows Spark to incorporate new executors being made available to the Spark cluster manager, when the application requires it and they are assigned by the cluster resource manager.

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

NOTE

> Mesos is a cluster manager that has two distinct modes for launching an application: the coarse mode, negotiates a number of executors for the duration of the application — in

effect creating a mini cluster-within-cluster (this is the default). The other mode is the fine-grained mode, in which resources are requested on a per-task basis. The advantage of this second mode is that it is less dependent on a fixed assignation of executors, since this is being renegotiated on each job, but it has a longer overhead when launching each task. The arrival of dynamic allocation in Spark 1.5 has in effect made the fine-grained mode obsolete, in that allocation works much more efficiently and quickly with this new mode, on top of a minimum allocation of a “base” number of executors in coarse mode.

Job stability: A time budget question

Performance tuning in Spark Streaming can sometimes be complex, but it always starts with the simple equilibrium between the batch interval and the batch processing time. The batch interval, as we've discussed in Chapter 2, is the time at which a new batch of data enters the system of Spark Streaming in the form of an `RDD`. All data that happens to be readable on the source of your `DStream` needs to be inserted in this way for Spark Streaming to have the opportunity to compute on it, and that insertion occurs at synchronous intervals. The batch processing time, on the other hand, represents the amount of time that is necessary for Spark to deal with one unit of data that has been inserted in the system in the past, and to remove it from the queue of data left to be processed.

At any given time, we would want Spark Streaming to be always computing on fresh data. And if processing is on schedule, this data would be the `RDD` that was created during the very last batch interval. Sadly, if our Spark Streaming cluster is late, it's the data that was created during a prior batch interval. If our cluster is early with its processing, the cluster can sit idle waiting for data ingestion, something we can address with either tuning parallelism (see "The Receiver model vs. reliable receivers") or dynamic allocation (see [Link to Come]).

While a cluster sitting idle is a waste of resources, the more concerning situation is a cluster that is late in processing, something that can create instabilities in the whole Spark deployment. For the rest of this part, we will focus on the problem of making sure that our cluster can keep up with data ingestion.

Batch interval and processing delay

A strong constraint with Spark Streaming is that data insertion does not stop. The insertion of data occurs at regular intervals and there are no facilities within Spark to turn it off arbitrarily. Hence, if the job queue is not empty by the time that the new batch interval starts, and new data is inserted into the system, Spark needs to finish processing the prior jobs before getting to the new data that is just entering the queue.

With only one job running at a time, it is easy to see that: - if the batch processing time is only temporarily greater than the batch interval, but in general, Spark is able to process a batch in less than a batch interval, then Spark Streaming will eventually catch up and empty the job (RDD) queue. - if, on the other hand, the lateness is systemic and on average the cluster takes more than a batch interval to process an RDD, then Spark Streaming will keep accepting on average more data than it can remove from its memory management on every batch interval. Eventually, the cluster will run out of resources and crash.

We need there to consider what happens when that accumulation of excess data occurs for a stable amount of time. By default, RDDs that represent the data fed into the system are put into the memory of the cluster's machines. Within that memory, the origin data — a *source* RDD — requires a replication , meaning that as the data is fed into the system, a second copy is created for fault tolerance, progressively, on every block interval. As a consequence, for a temporary amount of time, and until the data of this RDD is processed, that data is present in two copies in the memory of executors of the system. In the receiver model, since the data is always present in one copy on the receiver, this machine bears most of the memory pressure.

Eventually, if we add too much data into a spot in the system, we end up overflowing the memory of a few executors. In the receiver model this may well be the receiver executor that happens to crash with an `OutOfMemoryError`. What happens next is that another machine on the cluster will then be designated as the new receiver and will start receiving new data. Because some of the blocks that were in the memory of that receiver have now been lost due to the crash, they will now only be present in the cluster in one single copy, meaning that this will trigger a re-duplication of this data before processing that data can occur. So the existing executors in the cluster will pick up the prior memory pressure — there is no inherent relief from data lost during the crash. A few executors will be busy copying the data and one new executor will be accepting data once again. But remember that if prior the crash our cluster included N executors, it is now composed of N minus 1 executors, and it is potentially slower in processing the same rhythm of data ingestion — not to mention that most executors are now busy with data replication instead of processing as usual. The batch processing times we observed before the crash can now only be higher, and in particular higher than the batch interval.

In conclusion, having a batch processing time that is, on average, higher than the batch interval has the potential from creating cascading crashes throughout your cluster. It is therefore extremely important to maintain Spark's equilibrium in considering the batch interval as a time budget for all the things we may want to do during the normal functioning of the cluster.

Note

The constraint that only one job can execute at a given time can be lifted, by setting `spark.streaming.concurrent.jobs` to a value greater than one in your Spark configuration. However, this can be risky in that it can create a competition for resources and can we make harder to debug whether there are sufficient resources in the system to process the ingested data fast enough. We've seen in this very chapter that the way Spark Streaming deals with stragglers (tasks that execute slower on one given executor than can be expected from the average of other executors) is speculative execution, and speculative execution can only be used if there are available executors to start them on. With concurrent jobs, we could enter a situation in which a straggler would block completion of a job on a specific executor. If this is, for example, an issue of the underlying machine, concurrent job processing would prohibit any mitigation whereas speculative execution would be effective.

Going deeper : scheduling delay and processing delay

Many factors that can have an influence on the batch processing time. Of course, the first and foremost constraint is the analysis that is to be performed on the data — the logic of the job itself. The running time of that computation may or may not depend on the size of the data, and may or may not depend on the values present in the data.

This purely computational time is accounted for under the name of *processing delay*, the difference between the time elapsed running the job and the time elapsed setting it up.

Scheduling delay, on the other hand, accounts for the time necessary in taking the job definition (often a *closure*, as we have seen earlier in this chapter), serializing it, and sending it to any executor that will need to execute it. Naturally, this distribution of tasks implies some overhead — time that is not spent computing — so that it is wise to not decompose our workload in too many small jobs, and to tune the parallelism so that it is commensurate with the number of executors on our cluster. Finally, the *scheduling delay* also accounts for job lateness, if our Spark Streaming cluster has accumulated jobs on its queue. It is formally defined as the time between the entrance of the job (RDD) in the job queue, and the moment Spakr Streaming actually starts computation.

But there are other factors that, perhaps counter-intuitively, can contribute to the batch processing time, in particular checkpointing. Checkpointing is a safeguard that is particularly necessary in the processing of stateful streams, to deal with the possibility of data loss, and the recovering from it. It uses the storage of intermediate computation values on the disk so that in the event of a failure, data that depends on values seen in the stream since the very beginning of processing do not have to be re-computed from the data source, but only from the time of the last checkpoint. The checkpointing operation is structurally programmed by Spark as a periodic job, and as such, the time making the checkpoint is actually considered as part of the *processing delay*, not the *scheduling delay*.

Caution

The usual checkpointing on a stateful stream — where checkpoints are usually significant, both in terms of semantics and in the size of the safeguarded data — can take an amount of time much larger than a batch interval. Checkpointing durations on the order of ten batch intervals are not unheard of. As a consequence, when making sure that the average batch processing time is less than the batch interval, it’s necessary to take checkpointing into account. The contribution of checkpointing to the average batch processing time is

\frac{checkpointing delay}{batch interval} * {checkpointing duration}

This should be added to the average computation time observed during a non-checkpointing job to have an idea of the real batch processing time. Alternatively, another way to proceed is to compute how much time we have left in our budget (the difference between batch interval and batch processing time) without checkpointing and tune the checkpointing interval in function:

{checkpointing delay} \geq {checkpointing duration} / ({batch interval} - {batch processing time}^*)

Where * marks the measure of the batch processing time without checkpointing.

Finally, if all those factors have been taken into account and you are still witnessing spikes in the processing delay of your jobs, another aspect that we really need to pay attention to is the changing conditions on the cluster.

For example, looking at the HDFS file system, if we happen to co-locate such a system on our cluster, has concurrency bugs in its older versions that constrain concurrent disk writes [3]. Therefore, we may be running a cluster at a very stable rate, while simultaneously, a different job — that may not even be Spark-related — can require a heavy use of the disk. This can affect: - data ingestion in the reliable receiver model, when using a Write-Ahead Log, - checkpointing time, - actions of our stream processing that involve saving data to the disk.

To alleviate this issue of external impacts on our job through disk usage, we could: - use a distributed in-memory cache such as Alluxio [4], - reduce disk pressure by saving structured, small data in a NoSQL database rather than on files, or - avoid co-locating more disk-intensive applications with Spark than strictly necessary.

Disk access is only one of the possible bottlenecks that could affect our job through resource-sharing with the cluster. Another possibility can be network starvation or more generally, the existence of workloads that can not be monitored and scheduled through our resource manager.

Fixed-rate throttling

We have enumerated all the possible causes of instability in a job, including: - unreasonable complexity (super-linear or above), - processing time spikes caused by frequently-occurring values, - Spark jobs that are difficult to serialize coupled with eager parallelism (“many tiny tasks”), - not budgeting for checkpointing duration, - resource contention due to other jobs on the same hardware.

To address these issues, we can employ a specialized strategy, which deals with the root causes one by one, as we have addressed above. However, we can also go for a general strategy that improves performance across our cluster.

The strategy that is mentioned frequently is to lengthen the batch interval. While this will help with some parallelism issues, it is not in general a solution to the instability issues mentioned above. Indeed, if we raise a batch interval from one minutes to five minutes, then we only have to serialize the tasks which are the components of our job once every five minutes instead of once every minute — a five-fold reduction. Nonetheless, the batches of our stream will represent five minutes worth of data seen “over the wire” instead of one, and since most of the instability issues above are caused by an inadequation of our resources to the throughput of our stream, the batch interval changes little to this imbalance. More importantly, the batch interval that we seek to implement is often of high semantic value in our analysis — if only because, as we have seen and as we will explore in depth, it constrains the windowing and sliding intervals we can create on an aggregated stream. Changing these analysis semantics to accommodate processing constraints should only be envisioned as a last resort.

A more cogent strategy consists in: - reducing general inefficiencies, such as using a fast serialization library, something we will touch upon on [Link to Come]. We can also accelerate disk writing speeds by replacing our distributed filesystem with an in-memory cache, such as Alluxio. - adding more resources to our cluster, letting us distribute the stream on more executors by correspondingly augmenting the number of partitions that we use through e.g. block interval tuning (see ??? for the Receiver model).

But if getting more resources is absolutely impossible, we will have to look at reducing the number of data elements that we have to deal with. And to do this, there are a few tools that we have at our disposal. We will, in the fifth chapter, spend some time on the notion of random sampling to have an idea of whether we can accept dealing with a randomly-chosen portion of the elements of our stream in the cases when we face more data than we can handle. However, because we are dealing with a notion of stream that is based on micro-batching, random sampling reduces our stream by the same percentage on every batch. If it didn’t, it would disfavor the large batches by picking a smaller proportion of these elements, making them weight unfairly less in the overall analysis of the stream. Hence, sampling is only a solution that can bring a stream that has *on average* a throughput too large to handle to a tractable one. What if, on the other hand, we are faced with a stream that is on average something we can manipulate without sampling but has occasional spikes?

This is a plausible case because spikes of varied nature can occur on a stream. We have seen

above that the job queue of Spark can serve as a cache or dampener, allowing us to deal with a large amount of incoming data for a temporary few batches. And if we consider the case of a retail website, this may well cover the analysis of say, a 5pm spike in traffic on a particular Monday, where a lot of shoppers go online to buy the latest flagship product of a mobile phone maker. While the Spark cluster that performs our website's log analysis may have the resources to face this temporary spike in traffic, however, such a spike can be much larger, and may represent something of a different magnitude. We will see the example of a Black Friday event on the Walmart retail website where the spike lasted for an entire day ([Link to Come]).

In that case, if we are not dealing with a business-critical system, it may be acceptable to drop some of the data seen on the wire, and to tie this drop to our resources, the method we need to adopt is throttling, in number of elements per second.

Since version 1.3, Spark includes, a fixed-rate throttling that allows Spark to accept a maximum number of elements. This can be set by adding `spark.streaming.receiver.maxRate` to a value in elements per second in your Spark configuration. Note that this limitation is enforced at block creation, and simply refuses to read any more elements from the data source if the throttle limit has been reached.

Note that this behavior does not, in and of itself, include any signaling — Spark will just let a limited amount of elements "in" its memory through block creation, and pick up the reading of new elements on the next block interval. This has consequences on the system that is feeding data into Spark.

- If this is a pull-based system, such as in Flume and others, the input system could compute the number of elements read and manage the overflow data in a custom fashion,

- if the input system is more prosaically a buffer (file buffer, TCP buffer), it will overflow after a few block intervals (since our stream has a large throughput than the throttle) and will periodically be flushed (deleted) when this happens.

As a consequence, throttled ingestion in Spark can exhibit some "jitter" in the elements read, where Spark reads every element until an underlying TCP or file buffer, used as a queue for "late" elements reaches capacity, and is flushed as a whole. The effect of this is that the input stream is separated in large intervals of processed elements interspersed with "holes" (dropped elements) of a regular size (e.g. one TCP buffer).

Backpressure

Why backpressure

The queue-based system we have described with fixed-rate throttling has the disadvantage that it makes it obscure for our whole pipeline to understand where inefficiencies lie. Indeed, we have considered a *data source* (e.g. a TCP socket) which consists in *reading data from an external server* (e.g. an HTTP server), into a *local system-level queue* (a TCP buffer), before Spark feeds this data in an *application-level buffer* (Spark Streaming's RDDs). Unless we use a listener tied to our Spark Streaming receiver, it is very difficult to detect and diagnose that our system is congested, and if so, where the congestion occurs. Indeed, the external server could perhaps decide, if it was aware that our Spark Streaming cluster is congested, to react on that signal, and use its own approach to either delay or select the incoming elements to Spark. More importantly, it could make the congestion information flow back up the stream to the data producers it depends on, calling every part of the pipeline to be aware of and help with the congestion. It would also allow any monitoring system to have a better view of how and where congestion happens in our system helping with resource management and tuning.

The *upstream-flowing*, *quantified signal* about congestion is called *backpressure*. This is a continuous signaling that is quantified that explicitly says how many elements the system in question — here, our Spark Streaming cluster — can be expected to process at this specific instant. Back-pressure signaling is at an advantage with respect to throttling, because it is set up as a dynamical signal that varies in function to the influx of elements and the state of the queue in Spark. As such, it does not affect the system if there is no congestion, and it does not require tuning of an arbitrary limit, avoiding the associated risks in misconfiguration (under-used resources if the limit is too restrictive, overflow if the limit is too permissive).

This approach is available in Spark since version 1.5 and can, in a nutshell, provide dynamic throttling.

Dynamic throttling

Dynamic throttling is in Spark regulated by default with a *Proportional-Integral-Derivative* (P.I.D.) controller, which observes an error signal as the difference between the latest ingestion rate, observed on a batch interval in terms of number of elements per second, and the processing rate, which is the number of elements that have been processed per second. We could consider this error as the imbalance between the number of elements coming in, and the number of elements coming out of Spark at the current instant (with an “instant” rounded to a full batch interval).

The PID controller then aims at regulating the number of ingested elements on the *next* batch interval, by taking into account: - a proportional term (the error at this instant) - an integral or “historic” term (the sum of all errors in the past, here the number of unprocessed elements lying in queue) - and a derivative or “speed” term (the rate at which the number of elements has been diminishing in the past).

The PID then attempts to compute an ideal number depending on these three factors.

Backpressure-based dynamic throttling in Spark can be turned on by setting `spark.streaming.backpressure.enabled` to `true` in your spark configuration. Another variable `spark.streaming.backpressure.initialRate` dictates the number of elements per second the throttling should initially expect. You should set it slightly above your best estimate of the throughput of your stream to allow the algorithm to “warm up”.

Note

The approach of focusing on back-pressure to deal with congestion in a pipelined system is inspired by the Reactive Streams specification, an implementation-agnostic API intended to realize a manifesto on the advantages of this approach, backed by numerous industry players with a stake in stream processing, including Netflix, Lightbend and Twitter.

Tuning the backpressure PID

PID tuning is a well-established but vast subject, the scope of which is beyond this book chapter, but the Spark Streaming user should have an intuition of what this is used for. The proportional term helps with dealing with the current snapshot of the error, the integral term helps the system to deal with the accumulated error until now, and the derivative term helps the systems either avoid overshooting in the case where it is correcting too fast, or under-correction in case we face a brutal spike in the throughput of stream elements.

Each of the terms of the PID has a weight factor attached to it. Each factor is comprised between 0 and 1, as befits a classical implementation of PIDs. The parameters to be set in your Spark configuration are:

```
1 spark.streaming.backpressure.pid.proportional
2 spark.streaming.backpressure.pid.integral
3 spark.streaming.backpressure.pid.derived
```

By default, Spark implements a proportional-integral controller, with a proportional weight of 1 and an integral weight of 0.2 (and a derivative weight of 0). This offers a sensible default in Spark Streaming applications where the stream throughput varies relatively slowly with respect to the batch interval, and is easier to understand : Spark aims to ingest no more that the last rate of processing allowed, with a “buffer” for processing one-fifth of the late elements on each batch. note however that if you are faced with a fast-changing stream with an irregular throughput, you may consider having a non-zero derivative term.

Tip

The PID estimator is not the only rate estimator that can be implemented in Spark. It is in fact an implementation of the `RateEstimator` trait and the particular implementation can be swapped by setting the value of `spark.streaming.backpressure.rateEstimator` to your class name. Remember you will need to include the class in quesiton in the Spark classpath, for example through the `--jars` argument to `spark-submit`.

The `RateEstimator` trait is a serializable trait that requires a single method:

```
1   def compute(
2       time: Long,
3       elements: Long,
4       processingDelay: Long,
5       schedulingDelay: Long): Option[Double]
6 }
```

This function should return an estimate of the number of records the stream attached to this `RateEstimator` should ingest per second, given an update on the size and completion times of the latest batch. You should feel free to contribute an alternative implementation.

Note

Throttling in Spark, dynamic or not, is expressed in the `InputDStream` classes which include

`ReceiverInputDStream` for the receiver model, and `DirectKafkaInputDStream` for the Kafka direct receiver. These implementations currently both have a simple way of dealing with excess elements: they are not read from the input source (`ReceiverInputDStream`) or dropped after reading (`DirectKafkaInputDStream`) [5].

But it would of course be reasonable to propose several possible alternative implementations based on the backpressure signal received at the `InputDStream`. We could imagine policies such as taking the first, largest, smallest elements, or a random sample.

Sadly the `rateController: RateController` member of these classes is `protected[streaming]`, but this member has a `getLatestRate` function that lets the Stream implementation receive the relevant limit at any instant. Any implementation of a custom DStream could thus take inspiration from the existing to help dealing with congestion in a better way.

Fault tolerance in Spark Streaming

To understand the semantics of fault recovery in Spark streaming, after a failure in a distributed Spark cluster, it is important to understand the *consequences* of fault recovery on an application. In particular, that drives us to delve on the central but abstract concepts of at least once, at most once, and exactly once processing.

At least once processing

> It is the idea that if I am receiving a message for processing by a streaming application, then the result of that computation will eventually return, irreverently of failures that could occur under a particular failure mode.

At most, once processing

> is another guarantee that says that if a single message's processing by a streaming application produces a result, then the result tied to his message will not be emitted twice.

Exactly once processing

> is the combination of the two above requirements. That is, if you are trying to process a message, that computation returns once and exactly once in the lifetime of the application per specific input.

The desirability any of those properties is highly dependent on what you are trying to achieve in your application. For example:

- say we are trying to implement a process in which *message repetition* is costly, such as signaling confirmations of actions that will flow to an interactive user, or updating a dashboard with news of a uncritical nature — in which repeating news is going to irritate or distract the end-user. Therefore we would be interested in at most once processing.

- Inversely, if the cost is in *message loss*, we care about at least once processing. Think for example fo the case of anomaly detection, where we want to be certain that a user has received the result of a computation.

In most cases, we would like to have the best of both worlds, and are hence interested in the naturally intuitive idea of exactly once processing. That is, Spark Streaming should always return, and never returns twice the same result obtained from a given set of inputs. However, since Spark Streaming puts this in the context of distributed computing, the one to one mapping of inputs to outputs is hard to do in practice.

This is for fundamental reasons, linked to the related problem of distributed consensus. When we look at Spark Streaming's processing guarantees, we would like to have Spar Streaming's processors (executors, and driver) agree on multiple intermediary values in the progressive computation of a result for a given set of inputs. This is an instance of the distributed consensus

problem, which consists in having several processing systems agree on a particular value through communication with the others. Strong theoretical feasibility results (??? ???) show that to not only reach this agreement, but to also reach it in the case of failure of some of the involved processes, we will have to invest in a high degree of coordination, paying a high price in latency and communication. All distributed systems make trade-offs in the amount of performance they are ready to sacrifice to attain these fault tolerance properties, and most of these tradeoffs express themselves using the framework of state replication (???).

Note

For a high-level, whirlwind tour of fault-tolerance goals of different stream-processing engines, a reference is ???. Note however that this is a very fast-changing field, and that the particularities of the guarantees offered by such a system are often fraught with idiosyncracies that a high-level review cannot represent faithfully.

And Spark uses replication, together with serialization of its state (while often involving cooperation with other distributed systems, including Hadoop and Zookeeper), to provide its particular fault tolerance guarantees, which we are going to discuss below. However, one important nuance to understand is that, in part because those guarantees are costly to meet, they are only offered for the set of returned values of Spark, and not for the side effects of computation, that might occur an unpredictable times along the way to agreeing on a return value.

Spark mentions in its documentation the warning that side-effecting output may suffer from at least once semantics, but there is no guarantee that they will be performed exactly once. The reason is that when an executor is crashing, then the last task running on the executor in question will restart, either on the same machine if that machine recovers fast, or on another machine. replication will provide its input data and the meta data indicated in the lineage graph.

More to the point, the task on which the executor that crashed has metadata referring to how to retrieve the data on which that executor was operating. Since that data is replicated, a simple rescheduling of the job that was running on that executor is sufficient to create a retrieval of the data. Separately, the block manager that lives in the Spark context will be in charge of noticing that the number of copies of that RDD is potentially inferior to the number of copies required out of the job, and in the case where one copy of the input data is still remaining on the cluster, then the job will be immediately rescheduled at it's very last moment. However, if the RDD was not replicated, then there is a good chance that no copy of the input data of the RDD exists. There is probably still some sort data through which it is possible to recompute the RDD on which the executor crashed. That is by computing it's parent RDDs. If the parent RDDs themselves do not exist, then there is probably a new way of recomputing those through scheduling other jobs.

That transitive scheduling of jobs, recreating data that was lost at the failure of an executor, is obtained by walking in the lineage graphs of RDDs and comparing it with the book-keeping of the block manager to figure out if the input data required for the particular computations in question is present somewhere in the cluster. Therefore, the failure of an executor may trigger a re-computation of a particular RDD and not only that of this RDD, but the one of its parents as well. A literary number of re-computations depending on the exact replication properties of your application may involve quite an amount of processing power. More interestingly, it will rerun

through the code and the closures that the user wrote to specify the content of those RDDs. As a result, that re-execution will trigger once again the side effects that may be part of this execution. In particular, side effects such as writing to files, or outputting lines in a log, or outputting print out in the driver's console, will be re-executed again.

Spark has no inherent way of preventing those side effects from occurring twice, because the closures that are sent by the user at the application level are opaque to spark, which does not know how to differentiate within a closure between what is strictly needed for producing the output value, that is the part of the `RDD` that you are computing, and the pure side effects. That is the effects that have nothing to do with returning an output.

Note

Note, an interesting aspect of this is the guarantees for execution for accumulators. Accumulators are a particular value in Spark, on which you can find more documentation at the proper page. However, they present an interesting aspect in their guarantees for fault deliverance. Accumulators are fundamentally cells or constructs that receive updates in a monadic way during computation of both transformations and actions. The result of those transformations and actions has the update of the accumulator as a side effect. Spark therefore has a relatively subtle model in its managing of accumulators with failures.

First, in transformations, *Spark guarantees that you will get at least once execution of an accumulator*. In fact, if an accumulator is updated in the processing of a very specific `RDD`, you have the guarantee that the accumulator will be updated exactly as many times as it is necessary to recompute the closure in which the accumulator updates its presence. However, for the output operation, Spark guarantees that the accumulator will be executed exactly once. That accumulator update seems to contradict what we have said with the failure of an executor and the effect of the way Spark manages side effects. In fact it is not, it is due to specific treatment of accumulator updates in output operations. Intentionally, those accumulator updates are delayed to the end of an output operation in which the output operation has succeeded by the time that the accumulator update exists. In that particular case, we can therefore notice that, `foreachRDD`, in the context of a Spark `DStream`, will not create several updates of an accumulator.

Or at least we can assume that for it to create several updates of the same accumulator, will need a very specific failure mode.

There is a difference between side effect operations and results of RDDs. In particular if we contain an `RDD` that sums for example, the count of words present in a document, the word count that we have seen before and that is now the bread and butter or the basic example of distributed computing. Word counts in our particular case proceed from a `map` and `reduce` operation that produces an `RDD` as a result and is a sequence of transformations. Those transformations have the guarantee that their result will consist of data taken into account exactly once. Therefore, as far as the precise computing of the output of your data is, that is the type of the elements of your `RDD`, there is no fear of computation possibly executing several times. If, as in the figure below, you have a word count as part of your computation and if aggregating word counts, for say the letter A at the very last stage, consists of adding the participation of words starting with A. If that participation fails, that means you have a failure at the shuffle stage, the re-execution of that computation will simply reproduce the source data from which that computation proceeded, and

re-execute the relevant part of the computation.

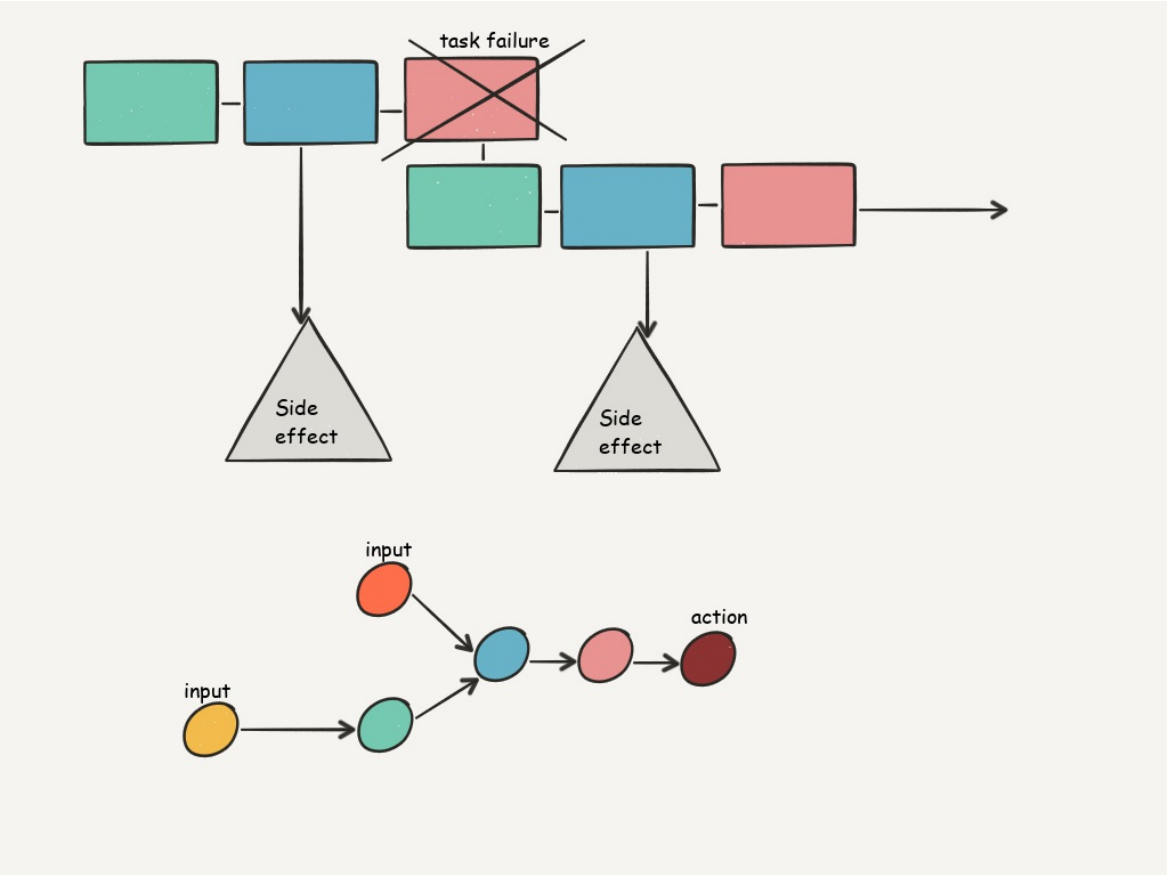


Figure 4-2. A side-effecting computation with reexecution

That is recompute the sum for the partition for the letter A. Here the letter A stands for the simplification of the range of the partition on which you compute your sum. In other terms, at the partition level, when the dependencies of an RDD or the executor that is computing the output RDD itself are going down, then the update of the mental abstract informal array that we consider as storing the results of the sums for the letter A, those updates are not affected until the input data has been reconstructed, and until the executor is able to compute the total sum for the result. Whether the sum for the particular let word aggravate is computed once, twice or thrice because of executor failure, does not impact the total number of words counted as starting with the letter A. The result of this for Spark streaming is two-fold.

Planning for side effect stutter in transformations

First, we should be considerate of the fact that in map transformations and in transform transformations upon a `Dstream`, we are in fact executing a transformation that could be repeated. That transformation that we should write as being sparse in side effects. Or, if side effects there are, should contain only side effects that are here to inform us about the meta information of how our job is running.

For example, if you want to have a counter of how many times the job is running, an accumulator is appropriate because it only guarantees in the context of the map or transform, and at least once guarantee, and if our `RDD` partition requires a re-execution, then the accumulator update will happen again, and our counter will updated to show not the semantics of our data set, but the number of times that our data set was ran through. Hence, this log information, while valuable, should be considered for what it is and not for information about the semantics of the processed data. On the other hand, and this is the second insight that we can use for Spark streaming, the output operations that we might be sending at the end of our processing, trough a `DStream`, to say fine if we want to save information, or a database, should be atomic. In the sense, when are updating data through a `RDD`, we want that particular updae to only execute once. Therefore it is wise to do a good segmentation between the computing of the data that we want to report on, say on a visualization board the basal file, and the real update of that data.

The update in question only has at least once semantics, therefore the use of indexes to have a particular ID of the data that we are updating at a particular time, should be used to make sure that we not produce twice a result though the side effect of the falter runs guarantees of Spark. Finally, what about the guarantees of exactly once processing for, this time, the data provider? Indeed, if a receiver shuts down, Spark streaming will reconnect to the data source by relaunching the receiver job. As we will see in a few lines, the receiver model in Spark is highly criticisable from the point of view of reliability. It guarantees that the driver may restart in the case of a failure, but it does not guarantee retrieval of the data that has come over the wire after the restart of that receiver. In fact, since the arrival over the wire occurs in real-time, it is relatively difficult to guarantee something for a process reading that input data that has fundamentally died before replication. This is the reason behind upgrades of the pipeline of the Spark streaming receiver that we are going to see in a few paragraphs.

However, we can already get an idea of the semantics of data receiving if we abstract from that receiver and if we only consider memory and data as it has entered into Spark's memory. That is, let's look at what happens after a receiver. To get at most one semantics, we need to do a number of things. First, Spark streaming has a process called *speculation*, which is necessary to guarantee that stragglers in data arrival do not create backlog and do not stop the processing of data indefinitely. However that means that in the executor which is particularly slow with the respect to the average executors as measured during the same task, will create another speculative execution and create multiple copies of the same task running on the same data as it has entered into Spark's memory. That repetition is one source of replication of the outputs of Spark and will need transactional memory management to make sure that the multiple messages that Spark produces on the output side will not impact the last application on which we want an at most once guarantee.

A second example is that we need to set the number of times that Spark will attempt to repeat a particular task in the case of a failure to one. That setting is `spark.task.maxfailures` and that it will make the job die as soon as a task fails. That removes a second source of possible replication of messages. Finally, we would like to make sure that when the application restarts we want to make sure that the application requests from the input source data that has never been seen so far, to make sure that the input source is reliable and able to replay some of its log of messages, does not replay messages that have already been seen. So three sources of replication exists in Spark. The failure recovery first, the speculative execution second, and as a third, the possible replaying of a number of messages depending on the exact input source that is being used in the application.

Idempotent side effects for exactly once processing

Second, if we want to guarantee at least once semantics, options here are to make sure that the max failures of the task is larger than one — to make sure that the failure of an executor does not fail the job. We can also make sure that our stream is check-pointed, to make sure that computation that depends on anything in the past context of Spark is actually retrieved. And finally, the input stream should be queried with the ask that if the application is restarted, it should restart its log of messages at the latest point where Spark hadn't seen any prior data.

On the other hand to get exactly once semantics we have to take into account the possibility that Spark will face those replications. Spark has in its very design the tools that are necessary to ensure at least once semantics, and we've been going over them in the last few paragraphs. We will continue to delve into the specifics of that reliability in this chapter. However, we have to take into account that replay of actions might create a replay of output operations. Hence, exactly once in Spark streaming is guaranteed by downgrading or more specifically specializing the at least once guarantees that are provided at the framework level to exactly once semantics that need to be provided at the application level.

Hence, if you want messaging guarantees in your application, it is your role to make then. First example is implementing this with your output operations, with idempotent rights. The idea of idempotent rights is to make sure that in a session you enter the output of your computation into a construct, a database, a file, where you check that the operation that you have done has not been performed before. For example, by assigning a key to each output operation. In the case of a failure, the output action can be retried without any problems. Checkpoint in the stream insures that offset ranges — or more exactly how much data Spark streaming had at the time of the failure — can be recomputed by Spark if your input source is able to replay some of the data. It will therefore restart processing at exactly the right point and since you are checking during the output operation that you are not performing the same operation twice and that you are not updating again, in the case where you have indeed performed that already, then you have guaranteed that you have at most once semantics on a system that already guarantee at least once semantics, and since we have the union of both of those guarantees, we have exactly once semantics.

A second way of performing the same idea is through guaranteeing transactional rights in your output operations. For data stores that support transactions, saving exactly which data you are outputting for is something that will guarantee you the correct semantics for your application. Finally, in conclusion, notice that exactly once, at most once, and at least once semantics are always referred to with respect to a failure model. The details of how we resist subtle failures in either the input source or output sync for all streaming operations have not been seen in detail in this relatively brief overview. You should consider examining those scenarios carefully for your application if you depend crucially on exactly once semantics. But note however, that good results in distributed consensus problems show us that failures with unlimited time bounds might in the end impact your very ability at an abstract level of performing perfectly with respect to those failure modes. As a result, distributed computing is often a trade-off, one in which the chance of each failure mode should be examined carefully and therefore create a response to those possible failures that is proper with respect to how often you expect to met those adverse

conditions.

Checkpointing and its importance

RDD checkpointing in stream construction.

RDD checkpointing is a subject that merits particular attention when dealing with Spark Streaming. The act of checkpointing consists simply in persisting the information necessary to cut the lineage of a particular computation implemented with Spark Streaming. The idea here is that Spark Streaming implements computation that depends possibly on a number of prior RDDs, including prior source data — that is, input RDDs. As we have seen above, this constitutes the lineage of the computation. However, that lineage can grow relatively long in cases where stateful computation or a window of computation is used.

For example the reasoning to compute the length of the maximal lineage obtained out of a windowed computation, for example, is to consider what kind of data we would need to recover, if we were to perform the whole computation due to partial data loss in the worst case. If I have a window of length N we would need, at worst, to recompute the data based on the latest possible RDD of the window which could have been lost.

In the case of a window, whether tumbling or sliding, this is an RDD which is as late as the number of RDDs computed in the window. If a windowed computation of length 5, that is your window is 5 times the batch interval, in that case, you will need to recover data from 5 RDDs. That is, all data from the last 5 batch intervals.

As a consequence, you will have a lineage of length at most 5. By the time you have finished the windowed computation, you will start a new windowed computation of which the earlies RDD will be then 5 batch intervals ago, as you will have progressed by one more batch interval.

But the use of checkpointing jumps from convenient to necessary when the lineage of your computation may be arbitrarily long. In the case of windowed computation, tthe length of the lineage may be long, but not arbitrarily so. The length of your lineage is determined by the number of times that your window contains your batch interval — that is, the length of your window. It thus can be relatively long, In fact, especially if you have a batch interval of say 1 minute, with a windowed computation of say 1 hour. But you might be even more interested in the case where the length of your state dependence on a prior RDD can be as far back as the beginning of the run-time of your application, which is basically the case for stateful streams.

Checkpointing and stateful streams

Two functions in the Spark Streaming API let you define stateful streams which grow a lineage that is arbitrarily long. The first is `updateStateByKey` and the second is `mapWithState`, both functions which we are going to dwell on much more extensively in the next chapter.

However, we can as early as now consider that they grow a very long lineage, which means that Spark will force you to activate checkpointing as soon as you use one of those stateful functions. The reason for that is that, if you imagine the easiest possible case of a stateful function, which is

to add numbers contained in an `RDD` that is produced at each batch interval, reading integers over the wire of an arbitrary input source. To reconstitute the sum of integers since the beginning of the application, in case you lose your intermediary total through machine failure, you will need to save and retrieve all the data that you have seen since the beginning of the application.

As a consequence, Spark forces you to save intermediary points in the computation. In that case, the partial sum of `RDD`s seen so far. For that reason, Spark will require that you set the checkpointing directory — which happens to be in Spark Streaming the same directory which will be used by the Write Ahead log. However, checkpointing is not only configured by setting a directory. You can actually manually call the checkpoint function on a `DStream` to force the interval for the periodic writing of the intermediary state of that `DStream` to disk, in the checkpointing directory.

The directory is set by calling `ssc.checkpoint(checkpointDir)`, where `ssc` is your Spark Streaming context, and `checkpointDir` the path to your checkpointing directory, as a `String`.

Note

To completely benefit from the use of checkpointing, the user is advised to armor against driver failure as well. In this case, the Streaming Context should be created using `StreamingContext.getOrCreate(checkpointDirectory)`, which is the way to recall the state of an application when it is under supervision itself.

To get a full picture of recovery, the best is to consult the example in the Spark distribution,

`org.apache.spark.examples.streaming.RecoverableWordCount`

Checkpointing's cost

This is of a particular concern in Spark Streaming since we consider our batch interval as a computation budget. Since the writing of a potentially *large* state to a disk can be expensive, especially if the disk that is backing this application is relatively slow, such as is often the case in the usage of HDFS as a backing store for the checkpointing operation. Note that this is the most common case of checkpointing : HDFS is reliable and magnetic drives offer replicated storage at a manageable cost.

Checkpointing should hence operate on a reliable data sink, so that in the case of a failure Spark Streaming, it will be able to recover the state of the stream rapidly, by reading the data from that reliable storage. However, considering that writing to HDFS can be slow, we will have to face the fact that checkpointing will periodically require more run time, possibly even more run time than the batch interval. And as we have explained previously, having a batch processing time larger than the abtch interval can be problematic.

When to checkpoint

When does checkpointing occur ? Well, checkpointing occurs on the first batch in which more time has passed since the last checkpoint than the checkpoint interval.

Note

The default checkpoint interval is defined as 10 seconds in Spark Streaming, and unless you specify otherwise, which means that Spark Streaming will save the partial computation that is given as an output by your stream to the checkpointing directory, on every batch which start time is a multiple of the batch interval is larger than 10 seconds.

Note that If you have a relatively large batch interval, notice that this will create a checkpoint on every batch.

If you work with batches superior than a few seconds, you will be interested in specifying the checkpoint interval manually, to something a little bit larger than 10 seconds allowing you to perform checkpointing every few batch intervals. A rule of thumb advised in the Spark documentation is to do checkpointing every 5-7 batch intervals as a start.

Setting the checkpoint interval is done per stream, using `dstream.checkpoint(checkpointInterval)`, where `dstream` is a `DStream` object (which makes sense, because the use of checkpoitning depends on your lineage length, which is per-stream), and `checkpointInterval` is a `org.apache.spark.streaming.Duration`.

But more importantly, the relevant design point is that the stability of your job should not be impacted by the checkpointing operation, which means that since this depends on your ability to write relatively quickly to a reliable storage, you will want to observe the writing operation as it happens in your job. For that, a simple piece of advice is to start with the rule of thumb of 5-7 batches before checkpointing, but also to set a performant non-default library, and to benchmark the job afterwards.

Checkpoint Tuning

You will be able, thanks to the Spark user interface, to measure on average how much more time you will need for a batch interval computation that includes checkpointing, compared to the batch processing time that you observe in RDDs that do not require checkpointing. Let's say that your batch processing time is about 30 seconds for a batch interval that is 1 minute. This is a relatively favorable case : at every batch interval, you in fact spend only 30 seconds computing, and you have 30 seconds of empty time during which your system is sitting idle, while receiving data.

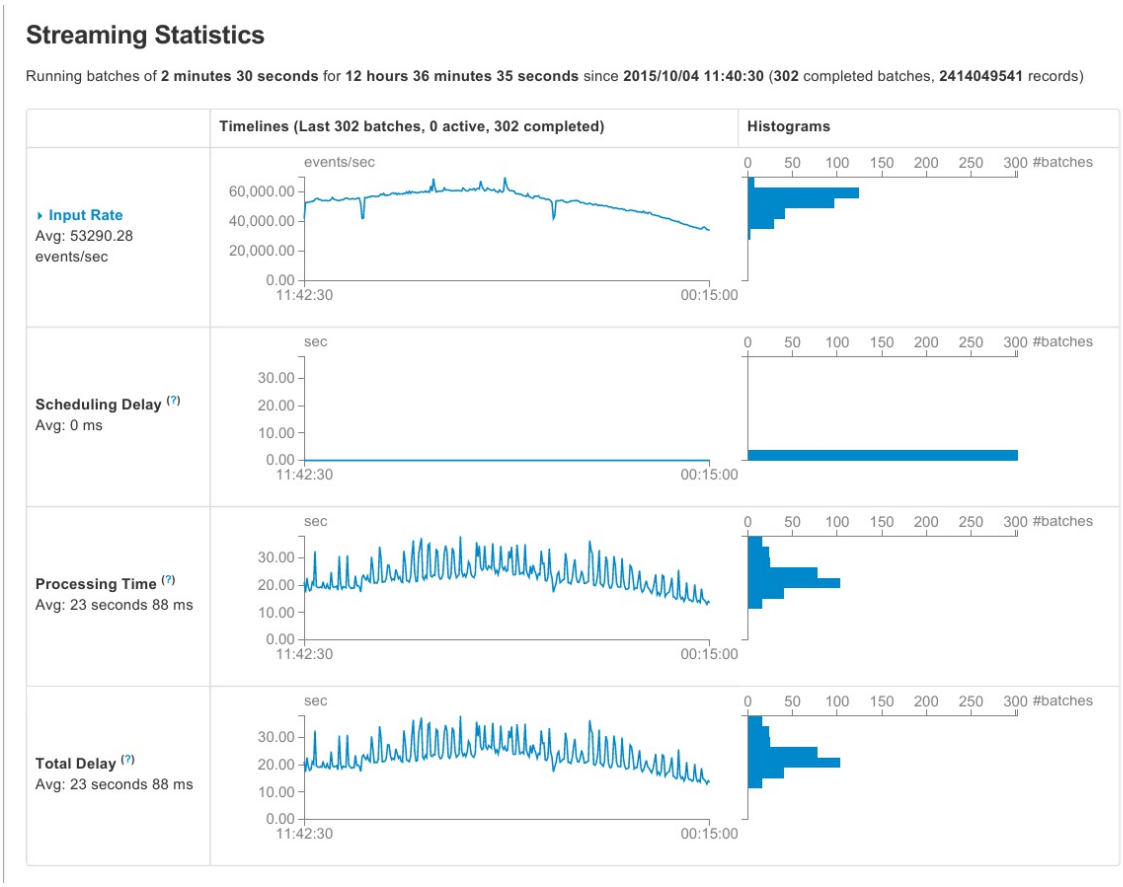
Now, every 5 minutes, you decide to checkpoint your source because, let's say, you are using something like a stateful computation. If you decided to do that checkpointing every 5 minutes, and you measure that your checkpointing batches require 4 minutes of actual batch processing time, you can conclude that you will need about 3.5 minutes to write to disk, considering that during that same batch you have also expanded 30 seconds in computation. It means that in this case, you will require 4 batches to checkpoint once again. Why is that?

This is because when you spend 3.5 minutes actually writing your files to disk, you are in fact still receiving data when those 3.5 minutes elapse. In those 3.5 minutes, you have received 3.5 new batches that you did not have time to process, since your system was blocked waiting for the checkpoint to end. As a consequence, you now have 3.5 — that is 4 full batches of data that are

stored on your system — that you have to process to catch up and reach stability again. Now, you have a computation time on a normal batch of 30 seconds, which means that you will be able to catch up by 1 new batch by every batch interval of 1 minute, so that in 4 batches you will be caight up with the receiveed data. You will be able to checkpoint again at the fifth batch interval. At a checkpointing interval of 5 batches, you are in fact just at the stability limit of the system.

That is, of course, if the state encoded in your stateful stream does reflects a size which is more or less constant with respect to the amount of data you have received from your input source over time. The relationship between the size of your state and the amount of data you have received over time might be more complex, and actually dependent on your particular application and computation, which is why it is often very useful to proceed experimentally with checkpointing length, with the idea that a larger checkpointing interval will give more time to your cluster to catch up with time lost during checkpointing, and the equally important concern that if you set a checkpoint interval that is too high, you might have a cluster that will have some trouble catching up if you incur some kind of data loss through crashes. In that case, you will indeed require your system to load the checkpoint, and to re-process all the `RDD`s that have been seen since that checkpoint.

Finally, note that any Spark user should consider changing the default checkpoint interval on any `DStream`, since it is set to 10 seconds. That means that on each batch, Spark will checkpoint if the interval elapsed since the last checkpoint is greater than the checkpointing interval. As a consequence, if you have a batch interval larger than 10 seconds, this leads to the following “sawteeth” pattern in processing times, showing checkpointing on every other batch (which is probably too frequent for most applications):



Note

There exists another alternative to this discipline of checkpoint interval tuning, which consists in writing to a persistent directory that accepts data at a very high speed. For that purpose, you can choose to point your checkpoint directory to a path of an HDFS cluster that is backed by very fast hardware storage, such as SSDs. Another alternative is to have this backed by memory with a decent offload to disk when memory is filled, something performed by Alluxio, for example — a project which was initially developed as Tachyon, some module of Apache Spark. This simple method for reducing the checkpointing time and the time lost in checkpointing is often one of the most efficient ways to reach a stable computation under Spark Streaming — that is, if it is affordable, of course.

The Reliable Receiver and the Write-Ahead Log

Let us summarize the fault tolerance primitives of Spark as we have encountered them so far: *Tasks* run on *executors*, distributed across a cluster. When a particular executor fails, those tasks can be *restarted* as a basic function of Apache Spark's scheduling.

We have seen that *the driver* is a particular component of Apache Spark, which receives the user's instructions, by way of a program, and orchestrates the computation across the cluster, based on Apache Spark's framework. When that driver program is made to run on the cluster itself, through the cluster deployment mode, equipped with the correct configuration options, will be restarted in case of a failure.

Finally, the *master* program, which does most of the scheduling and communicates with each executor, the particular fragment of a task that he is required to execute, can be made highly available through supervision using a zookeeper quorum.

Consequently, a well-configured execution of Apache Spark is relatively reliable, and gives us an idea of the resilience of tasks to the real-world failures that can be encountered on a real-world cluster. However, if we focus on the data, the story may be a little bit different.

Indeed, if we focus on data that has been assembled into an RDD, we have seen that the persistence level set by Apache Spark gives us a guarantee. It guarantees that the original data that the computation operates on is replicated across the cluster, and it can be recovered.

In particular, Spark Streaming forces replication of Source RDDs independent of the user-configuration for persistence. This helps fault tolerance, of course. But it also allows a better distribution of blocks across the cluster, in the Receiver model, than if all the data was centered on the Receiver machine. Therefore, for any source, the separation of data into replicated blocks guarantees us that should any particular executor machine crash, at least the original source data would be present on another machine on the cluster, which would allow us to restart the computation from that second copy of the same data.

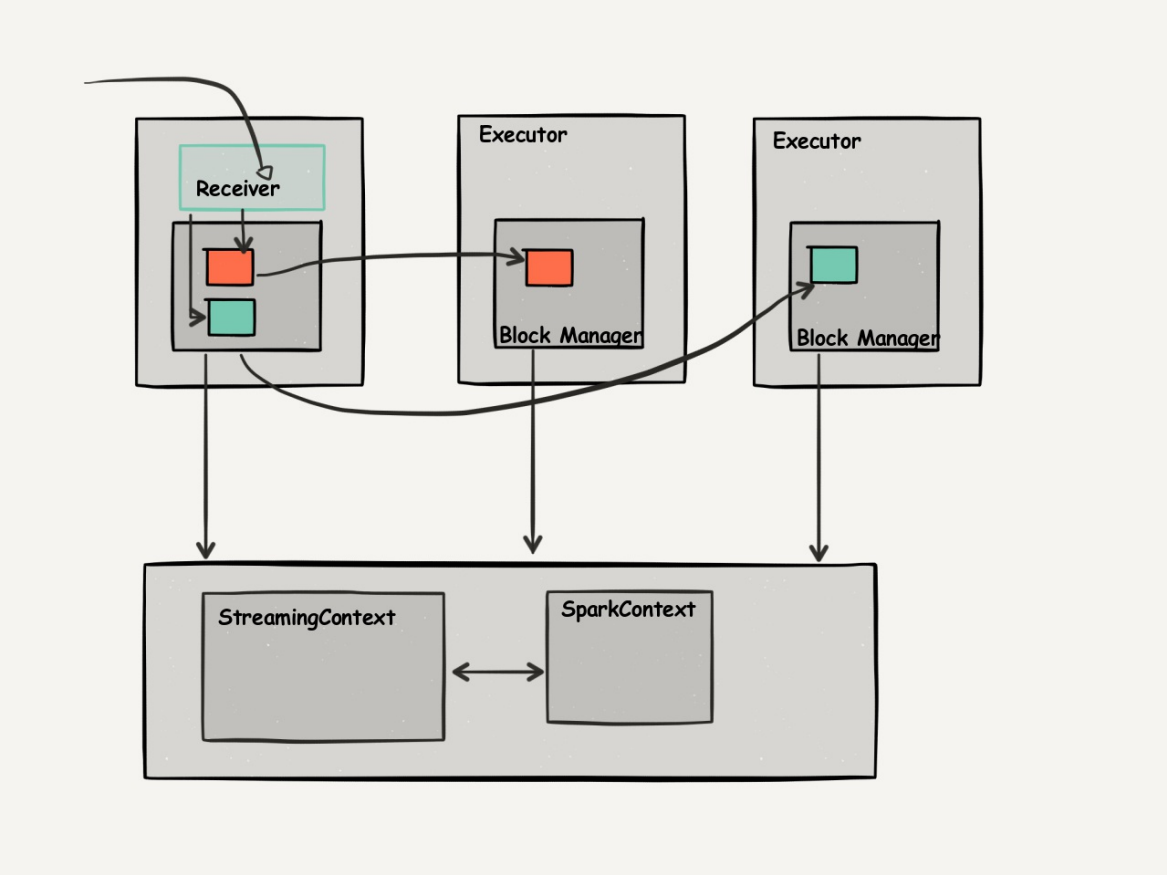


Figure 4-3. Block replication in a Spark RDD sourced from the Receiver

However, looking in detail at how that source RDD is created forces us to focus on a particular component that we have addressed before, which is the *Receiver*. The Receiver is that particular executor, or more precisely, that particular task — running on an arbitrary executor — which consists of ingesting data from outside in a Spark Streaming job.

This receiver is tasked with receiving data, creating blocks out of it, and reporting the creation of those blocks to the *Block Manager* on the driver, which then triggers the replication of those blocks. When a *batch interval* has elapsed, all the blocks that have been replicated thus far are grouped into an RDD. This is the RDD of the distributed stream (DStream), for the current batch.

This is normal functioning that we went over in the previous part of this chapter. However, in the case of a failure of the Receiver, even if we know that this Receiver will have to be restarted, what will happen of its data?

The RDDs that have been created so far are already part of a computation, which depends on replicated blocks. So that should mean that any blocks of those RDDs still exist as a copy of the blocks of the executor the Receiver was running on, those blocks will simply be available elsewhere, and will start replication again if necessary.

However, what happens of the blocks that were not yet integrated into an RDD? What happens

of the data that was not yet integrated into a block? Remember that data is integrated into an RDD on every *batch interval* (say for example every five minutes), and data is grouped into a block, and — therefore made ready for replication — on every *block interval*, (let's say every 20 seconds for the sake of an example).

Note

From that, remember that we can also compute the parallelism level of our RDDs, which has an influence on the run time characteristics and parallelism of any job we can write to that executes on that particular RDD:

In our example above, with a *block interval* of 20 seconds, we can have (5 minutes) divides by (20 seconds), which is 15 blocks during a batch interval.

The main data safety issue, in fact, is for data that has not yet reached the state of RDD. This data is not yet replicated and made safe inside of Spark, and may be at risk for different reasons:

- The main issue is data that has been received in the last few seconds (less than 20) between the last block interval and the crash. That data is not part of a block, and as such, has not started replication.
- Also at risk are data that arrived between the last block interval and the last batch interval: while it is part of a block and may have started replication, it may not have had the opportunity to finish replicating by the time of the crash.
- Some data may also exist in lower-level buffers inside the receiver machine. For example, Spark reads in a buffered fashion in TCP-based receivers, which may imply some significant amount of data could be in this TCP buffer by the time the executor containing the Receiver crashes.
- Naturally, data that was "in flight" during the moment the executor crashed is equally lost.

To correct for the limitations of Apache Spark Streaming in the case of a receiver crash, the developers have introduced two possible solutions. The first is the use of a *reliable receiver* which occurs in the case where the data source is able to replay the input and ingestion of some data on the condition that Spark gives some information on where the data flow has been interrupted. In this case, Apache Spark does not need to own the data safety task in between block intervals, but can offload it to that particular reliable source, provided it reports the data it received and replicated after each block interval. Hence the introduction of a reliable receiver, equipped to connect with some specific reliable sources, and to exchange bookkeeping information with this source. The other alternative is simply to store the data inside of Apache Spark from the moment of reception — that is without waiting for a block interval to elapse and, therefore, for block replication to ensure that data is saved in multiple copies, making it resistant to the loss of the receiver executor. This "double bookkeeping" is called the *write-ahead log*.

- The reliable receiver has mainly been implemented for Kafka — which happens to be a reliable receiver, but it can also be found in some instances of Spark connectors, either in Spark packages or in some of the connectors delivered with Apache Spark streaming.

Apache Flume, for example, is a source that can be made reliable with the proper options.

- The write-ahead log relies on the existence of an HDFS distributed file system on the cluster and lets you configure the spark streaming to right to the particular folder on the HDFS5 system where scratch detail will be registered the moment it enters the Spark streaming system.

Note

The assumption that writing data as it comes to an HDFS file system is more secure, than taking your chances with a receiver machine going down needs to be examined carefully.

- First, HDFS has a replication level and can be set to accept writes asynchronously, without waiting for replication to occur.

- Second, the HDFS cluster may well be co-located with your existing Spark cluster.

Those two properties can be combined to form a configuration in which an executor supporting both the Spark Streaming Receiver and the only current copy of the latest elements on the write-ahead log could go down, taking both elements down with it.

Notice that there are serious performance penalties with that particular way of functioning. In particular, writing to HDFS almost always involves writing to a hard drive peripheral and writing to a hard drive implies a significant performance loss when compared to writing in memory — which can be what happens when a proper storage level has been configured in Spark. Hence, in this no-data-loss configuration, you will need to wait until you are finished writing to a disk before getting access to your data in memory. Why wait? That is because the write-ahead log is synchronous, because its use implies not releasing any data until it has been deemed safe in the case of a crash.

Blocking writing is also implied by the tied coupling of the receiver's data ingestion with the TCP-IP stack that underlies it, involved in the connections to the sources of the Spark Streaming connectors. Indeed, it will be very natural to match the ingestion rate of data — which occurs in a blocking fashion as you write to the HDFS "bottleneck" more or less at a stable rate — with the acknowledging that can occur upstream as you empty the TCP buffer of the Receiver machine.

This particular all-synchronous chain guarantees that synchronous calls writing on the disk can be synchronized with synchronous calls acknowledging the flow of data. It is hard to tune, but in that particular manner, you can be sure that if a Spark streaming receiver crashes and died, it will only have consumed for you from your producer source the data that it has written on the disk, or very nearly so.

Apache Kafka and the DirectKafkaReceiver

The Kafka model and its Receiver

Counting cores and topics

Consuming data from Kafka, as we have seen in the previous chapter, can be relatively simple. Creating a `KafkaInputDStream` using Spark streaming leaves us with two control knobs for the parallelism of wer application that we are going to detail below. That parallelism, naturally dictates the parallelism that we can obtain in Spark Streaming. The `KafkaInputDstream` uses, indeed, the high level Kafka API, which means that we have access to exactly two ways of controlling what gets consumed from Kafka at the same time. Remember, Kafka uses the concept of topics to select which data we should subscribe to and consume and the topic of partitions to put a higher bound on the simultaneous amount of data that can be consumed using Kafka with simultaneous threads or consumers.

From the point of view of an executor, though, if we use `KafkaInputDStream`, we are implementing data reception in the receiver model. So if we want higher parallelism than one machine consuming some data at a time, we need to first think of increasing the number of input `DStream`s. Indeed, Spark will run one receiver (which is one task from the point of view of Spark Streaming) per input `DStream` ,which means that using multiple input `DStream`s will parallelize the read operations across multiple executors and hopefully, across multiple machines.

The other aspect of consuming data from a Spark streaming Receiver is that we can actually control the number of consumer threads per input `DStream` . This specific possibility is given by the input `DStream` parameters in the `KafkaInputDStream` . In case this is set, the same receiver will run multiple threads. That is, read, and read operations will happen in parallel, but on the same machine.

For practical purposes, we can also combine both options, using several threads and several cores, but one important consequence of that is that if we use multiple cores, we have to get an idea of the total number of available cores on wer Spark Streaming executors to correctly manage data ingestion, replication and the execution of tasks in Spark Streaming.

For example, if we have only one core available in one executor and the receiver runs on that executor, the receiver itself will, indeed, require the available core to run. Let’s remember that it is a scheduled task, dealt with as any task we would implement on a `DStream`.

Another aspect to pay attention to is that if we increase the number of cores being used to consume data from Kafka, we may not see a very big speedup at the cost of consuming cores available on wer Spark executor machine. The reason for that is that reading from Kafka is normally more network-bound, that is limited by the number of network interfaces on a machine.

We typically will not increase throughput by running more threads on the same machine, because those threads fundamentally share the same network card. The consequence is that it is usually much more frequent to adapt the number of input `DStream`s that read data from Kafka, rather than control the number of threads that do the same thing on one single machine. Those input `DStream`s

have receivers that, as a task, can be scheduled on different machines, and their union can then be used as a global `DStream` that represents the total amount of data being read out of Kafka. As a consequence of that practice, it has become somewhat of a black art to determine the level of parallelism switchable for a Spark Streaming application that consumes from Kafka using a `KafkaInputDStream`.

But if we remember the simple rules of the batch interval and the block interval, we can actually compute this pretty easily. Remember, if we have only one receiver, that is one machine receiving data, we can simply compute the parallelism as the division of the batch interval divided by the block interval. As a consequence, the number of partitions for one given `DStream` reading from Kafka is simply that number, batch interval divided by block interval.

If, on the other hand, we use several `DStream`s, unionized, to read from Kafka using the `KafkaInputDstream` , we have to multiply that number by the number of consumers, because very simply, at every batch interval, as many `DStream`s as there are consumers will produce a block on each block interval. The global consequence is that if we use k intermediate `DStream`s as consumers to read data from Kafka, the parallelism and the number of partitions that you can expect in the unionized resulting `DStream` is the number of consumers, multiplied by the batch interval, divided by the block interval.

Offsets and Fault tolerance

Let us now focus on the alternative way of creating a `DStream` from Kafka, which is called the direct API. It uses a slightly ifferent API from Kafka or more exactly it uses the transactional features of Kafka to continuously receive data using a high level API, but doing it from every single receiver. In fact, the concept of receiver itself disappears in that new direct API. It eliminates the needs for both, a writer head-log and a receiver for Kafka even in the most paranoid exactly once semantics processing modes of the integration of Apache Kafka and Apache Spark Streaming.

The gist of it is that the driver queries offsets and decides offset ranges for every batch interval from Apache Kafka. Once it receives those offsets — which are indexes inside the data marking the amount of elements that have been received over time — the driver shares them accordinig to the available number of partitions, and launches jobs which retrieve data using specific offset ranges. What the driver at this stage is sending to the Spark Streaming executors is not data. It is simply offset ranges, so that the Spark executor, as the first step of starting its own job, reads specific portions of data inside its own job, using a simple API to connect to Kafka. As a consequence, the parallelism of data ingestion from Apache Kafka is much better. The reason why this was implemented as an extension the writer head-log and high availability semantics for Spark streamings is that originally Spark Streaming wanted to avail itself of the transactional feature of offset-based data indexing and replayability that is provided by Apache Kafka.

Indeed, the executors in that `DirectStream` acknowledge the reception of data for their particular offsets. Apache Kafka then makes sure that every single record is effectively received by Spark Streaming exactly once. Note, however, that the responsibility of checking that all the data that should be received has been put on Apache Kafka, rather than on Apache Spark. As such, it is a specialized usage of the features of a reliable receiver, rather than a new feature of Spark

Streaming. However, it has great benefits in that it makes Spark Streaming's processing pipelines more efficient and easier to use.

In that case, indeed, we find again that the creation of partition is simplified. We do not have to configure the number of Kafka partitions to be consumed per receiver. Each Kafka partition is automatically read in parallel, and each Kafka partition corresponds to an RDD Partition, simplifying the parallelism model. Note, however, that if you have a high number of Kafka partitions, the ingestion task will be separate. If you don't have the number of Spark Streaming executors available to match that number of partitions, the data ingestion task might be slowed down until a Spark Streaming executor is available for the conception of each partition. You will see below an example of the usage of that direct stream API.

```
1 val ssc = new StreamingContext(sparkConf, Seconds(2))
2
3     // Create direct kafka stream with brokers and topics
4     val topicsSet = topics.split(",").toSet
5     val kafkaParams =
6       Map[String, String]("metadata.broker.list" -> brokers)
7     val messages =
8       KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder](
9         ssc, kafkaParams, topicsSet)
```

Other Reliable Sources of Data

The Flume receiver has a reliable variant which takes advantage of Flume's ability to replay parts of certain streams, and is therefore a reliable source as well. However, it functions in pull moe rather than in push mode, and therefore has an impact on the semantics of data ariival.

We have already seen an example of how to create a Flume Stream in [Link to Come], that included Flume's setup. So we'll focus below on the changes required for a reliable setup.

What will be different here is that the pull-based Flume connector requires manipulating a peculiar jar, the **Flume sink** . This new artifact is named `spark-streaming-flume-sink_2.10` (or `_2.11`, depending on your version of Scala) whereas the old module was bearing the artifactId `spark-streaming-flume_2.10` (or `_2.11`).

Counter-intuitively, this module is named from the point of view of Flume, meaning that this module lets **Spark** become a Flume sink, a notion that we already met. Correspondingly, it means that the new configuration file we need to use for this receiver is the following:

```
# the components on this agent an logical names of source, channel and sink
a1.sources = src-1
a1.sinks = snk-1
a1.channels = ch-1

# our source uses the systemcl journal reporter to provide log messages:
a1.sources.src-1.type = exec
a1.sources.src-1.command = journalctl -ef
a1.sources.src-1.channels = ch-1

# our sink
a1.sinks.snk-1.type = org.apache.spark.streaming.flume.sink.SparkSink
a1.sinks.snk-1.hostname = localhost
a1.sinks.snk-1.port = 44000

# characteristics of the 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
```

Once you have edited this configuration, remains the task of starting the Flume agent and letting it know where to find the code that corresponds to these. Flume by default looks for plugins in `$FLUME_HOME/plugins.d/snk-1/lib` for our sink, so this is where you should put the Spark Flume Sink jar and its dependencies — the Scala library jar (`org.scala:scala-library`) and the Apache Commons library (`org.apache.commons:commons-lang3`).

Once this is set up, Flume understands how to connect to a polling Spark cluster, and how to keep tabs on the data that is consumed by it, replaying offsets in case of failure. This is done from the moment that we create a reliable stream in the following fashion:

```
FlumeUtils.createPollingStream(ssc, "localhost", 44000, StorageLevel.MEMORY_ONLY_SER_2)
```

The Flume reliable connector exhibits a simple way to gain tolerance to a worker failure. Note that in a way similar to the Kafka direct connection, this allows every worker to retrieve some of the data, allowing for a better data locality and making the whole cluster less reliant on a single machine.

Parallel consumers

The Receiver model vs. reliable receivers

Accounting for cores and data consumption in both models.

As we have seen in ??? there is a frequent need for creating several `Receiver`-based `DStream`s in Spark Streaming in order to increase throughput. And as we have seen in ???, any `DStream` based on a `Receiver` will create a partition on every block interval, creating as many as the number of times the block interval fits in the batch interval. Moreover, a `union` operation — the frequent way to merge these `DStream`s into one — does not change the number or spread of partitions in the underlying block storage of Spark.

When putting those elements together in the `Receiver` model it means that we are tying together data parallelism (the final number of partitions), locality (where the first copy of each `DStream` block is), and reliability (the size of our block interval, which is the interval during which data loss can occur with non-reliable data producers).

For example, let's assume we have an input Stream that we can not, or can not afford, to read from a reliable data producer or a `DirectStream`. We can also assume that this is a high-throughput Stream, so that we want to read it from five different `Receiver`s. Finally, we are wary of losing any data on this `Stream` so that we feel that with a batch interval of one minute, we have to set the block interval to one second.

We would then create five separate `DStream`s reading the same source, sharing the same batch and block interval since they are created under the same `StreamingContext`. Each `DStream` creates 60 partitions on every minute, and these five `DStream`s, after `union`, merge into a single `Stream` of 300 partitions. Since we are on the `Receiver` model, the first copy of each block of one of the five `DStream`s is present on each of the five receivers, with the other blocks being replicated elsewhere.

This poses a few issues: - *the explosion of the number of partitions* has nefarious consequences on the speed of our analysis. It is a good rule of thumb to have around two partitions per executor, to compensate for small imbalances in processing speeds per executors. With a cluster of 30 machines and a single stream, the block interval of one second seems reasonable. With five unionized `Streams`, we have 10 times as many partitions as we have executors, and will need to repartition to avoid constantly spending time serializing short-lived tasks to send to executors. - `data locality` is acceptable only if Spark allocates the five `Receiver`s in five different machines. In some cases, Spark will allocate several `DStream`s on the same nodes, forcing the `Receiver` to compete with others for resources, and making the machine that hosts them slow to replace in case of a failure. - `element ordering` becomes dependent on the rightful synchronization of clocks in between the `Receiver` machines. Spark assumes that their clocks, dictating batch and block interval ticks, are synchronized to a time server such as `ntpdate`. This assumption is sometimes harder to realize than expected, especially in distant servers, and while a single-stream model only required the `Receiver` and the Spark master to agree on block cutoff times, it is of course more difficult with many `Receiver`s.

Hence it seems apparent these days that the direct approach of the Kafka `DirectReceiver`, is to be preferred: with it, there is no need to create multiple input Kafka streams and union them. With

`directStream`, Spark Streaming will create as many RDD partitions as there are Kafka partitions to consume, which will all read data from Kafka in parallel. So there is a one-to-one mapping between Kafka and RDD partitions, which is easier to understand and tune.

Caution

1. A word on serialization

It is sometimes not possible to use a Kafka `DirectStream`, and the `Receiver` model with its explosion in the number of partitions may sometimes be the only alternative. In this case, it is essential to consider Kryo serialization as well as repartitioning: Spark can also use the Kryo library (version 2) to serialize objects more quickly. Kryo is significantly faster and more compact than Java serialization (often as much as 10x), but does not support all `Serializable` types and requires you to **register** the classes you'll use in the program in advance for best performance.

You can use Kryo for serialization by setting `spark.serializer` to `"org.apache.spark.serializer.KryoSerializer"` in the Spark configuration. Spark automatically includes Kryo serializers for the many commonly-used core Scala classes, from the `AllScalaRegistrar` of the Twitter Chill library.

You can refer to the Kryo section of the documentation for more information.

Bibliography

- [Bhartia2016] Bhartia, R. *Optimize Spark-Streaming to Efficiently Process Amazon Kinesis Streams* February 26, 2016. URL

- [Chintapalli2015] Chintapalli, S; Dagit, D; Evans, B; Farivar, R; Graves, T; Holderbaugh, M; Liu, Z; Musbaum, K; Patil, K; Peng, B; Poulosky, P. *Benchmarking Streaming Computation Engines at Yahoo!* Yahoo! Engineering. December 18, 2015. URL

- [Das2013] Tathagata Das. *Deep Dive With Spark Streaming*. Spark meetup, June 17, 2013. URL

- [Das2014] Tathagata Das; Yuan Zhong. *Adaptive Stream Processing using Dynamic Batch Sizing* 2014 ACM Symposium on Cloud Computing.November 3-5,2014. URL

- [Das2015] Tathagata Das. *Improved Fault Tolerance And Zero Data Loss in Spark Streaming.* Databricks Engineering Blog. January 15, 2015. URL

- [Doley1987] Doley, D; Dwork, C; StockMeyer, L. *On the minimal synchronism needed for distributed consensus*. Journal of the ACM. 34 (1):77-97; 1987; URL

- [Dunner2016] Dünner, C; Parnell, T; Atasu, K; Sifalakis, M; Pozidis, H. *High-Performance Distributed Machine Learning using Apache SPARK* December 2016. URL

- [Fischer1985] Fischer, M. J.; Lynch, N. A.; Paterson, M. S.*Impossibility of distributed consensus with one faulty process*. Journal of the ACM. 32 (2): 374–382. 1985. URL

- [Kestelyn2015] Kestelyn, J. *Exactly-once Spark Streaming from Apache Kafka* Cloudera Engineering Blog. March 16, 2015. URL

- [Koeninger2015] Cody Koeninger, Davies Liu and Tathagata Das. *Improvements to Kafka integration of Spark Streaming.* Databricks Engineering Blog. March 30, 2015 URL

- [Lamport1998] Lamport, Leslie. *The Part-Time Parliament* ACM Transactions on Computer Systems. 16 (2): 133–169. URL

- [Mas2014] Gérard Maas. *Tuning Spark Streaming for Throughput.* Virdata Engineering Blog. December 22, 2014. URL

- [Venkat2015] Venkat, B; Padmanabhan, P; Arokiasamy, A; Uppalapati, R. *Can Spark Streaming survive Chaos Monkey?* The Netflix Tech Blog. March 11, 2015. URL

- [Nasir2016] Nasir, M.A.U. *Fault Tolerance for Stream Processing Engines*. arXiv preprint arXiv:1605.00928, May 2016 URL

- [Zaharia2012] Matei Zaharia, Tathagata Das et al. *Discretized Streams: A Fault-Tolerant Model for Scalable Stream Processing* UCB/EECS-2012-259 URL

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

[2] Note that a task can fail without the underlying executor failing, for example if the task throws an exception or if it reaches one of the memory limits imposed by Spark.

[3] You can refer to HDFS-7489 for an example of one of those subtle concurrency issues.

[4] Alluxio was originally named Tachyon, and was part of the Spark code base, which hints at how complementary its features are to data processing with Spark.

[5] The alternative Kafka Receiver by Dibyendu Bhattacharya improves on this latest aspect.

Chapter 5. Streaming Programming API

In this chapter we will take a detailed look at the Spark Streaming API. After a detailed review of the operations that constitute the `DStream` API we will learn how to interact with Spark SQL and get insights about the measuring and monitoring capabilities that will help us understand the performance characteristics of our Spark Streaming applications.

In the Spark Streaming programming model, we can observe two broad levels of interaction: - Operations that apply to a single element of the stream and, - Operations that apply to the underlying `RDD` of each micro-batch.

As we will learn through this chapter, these two levels correspond to the split of responsibilities in the interaction between the Spark core engine and Spark Streaming. We have seen how `DStream`s or *Discreatized Streams* are a streaming abstraction where the elements of the stream are grouped into *micro batches*. In turn, each *micro-batch* is represented by an `RDD`. At the execution level, the main task of Spark Streaming is to schedule and manage the timely collection and delivery of data blocks to Spark. In turn, the Spark core engine will apply the programmed sequence of operations to the data that constitute our application logic.

Going back to how this is reflected in the API, we will see that there are operators, like the classical `map`, `filter`,… that operate on a single element. These operations follow the same principles of distributed execution and abide to the same serialization constraints as their batch Spark counterparts.

There are also operators, like `transform` and `foreachRDD` that operate on an `RDD` instead of an element. These operators are executed by the Spark Streaming scheduler and the functions provided to them run in the context of the driver. It is within the scope of these operators that we can implement logic that crosses the boundary of micro-batches, like keeping historing or maintaining application-level counters. They also provide access to all the spark execution contexts. Within these operators we can interact with Spark SQL, Spark ML or even manage the lifecycle of the streaming application. These operations are true bridges between the recurrent streaming micro-batch scheduling, the element-level transformations and the Spark runtime context.

Basic Stream transformations

Operations at the `DStream` level in Spark are made at a different level than simple `RDD` operations: whereas `RDD`s represent collections of elements, `DStream`s represent a sequence of RDD`s over time. The `RDD` is the collection that represents the parallelization of the work, whereas the `DStream` is the structure that embodies the sequentiality of the dataflow over time. The operations in the `DStream` API let us manipulate those different encapsulation levels. We divide these operation into those that apply to a single element of the stream and those that apply to the underlying `RDD` of a `DStream`.

Element-centric `DStream` Operations

In general, the element-centric operations on the `DStream` API are an implementation of the homonymous function defined on the `RDD` API.

The most commonly used functions follow:

Note

The signature of each operation has been simplified by removing implicit parameters to make the signature more concise, where applicable.

map

```
map[U](mapFunc: (T) => U): DStream[U]
```

The `map` function on a `DStream` takes a function `T => U` and applies it to every element of a `DStream[T]`, leaving the `RDD` structure unchanged. As in `RDD`s, it is the appropriate choice to make for a massively parallel operation, for which it is of no importance whether its input is in a particular position with respect to the rest of the data.

flatMap

```
flatMap[U](flatMapFunc: (T) => TraversableOnce[U]): DStream[U]
```

`flatmap` is the usual companion of `map`, which instead of returning elements of type `U`, returns the type of a `TraversableOnce` container of `U`. These containers are coalesced in a single one before returning. All Scala collections implement the `TranversableOnce` interface, making them all usable as target types for this function.

The common use cases for `flatMap` are when we want to create zero or more target elements from a single element. In particular, when used in combination with the `Option` type, it can be applied to filter out input records that do not meet a certain criteria.

An important remark is that this version of `flatMap` does not follow the strict monadic definition, which would be: `flatMap[U](flatMapFunc: (T) => DStream[U]): DStream[U]`. This is often cause of confusion to newcomers with a functional programming background.

mapPartitions

```
mapPartitions[U](mapPartFunc: (Iterator[T]) => Iterator[U],
                 preservePartitioning: Boolean = false): DStream[U]
```

This function, like the homonymous function defined on `RDD`, allows us to directly apply a `map` operation on each of the partitions of an `RDD`. The result is a new `DStream[U]` where the elements are mapped. As with the `mapPartitions` call as defined on an `RDD`, this function is useful because it allows us to have executor-specific behavior, i.e. some logic that will *not* be repeated for every element but will rather be executed once for each executor where the data is processed.

One classical example is initializing a random number generator which is then used in the processing of every element of partition, accessible through the `Iterator[T]`. Another useful case is to amortize the creation of expensive resources, such as a server or database connection and reuse such resource to process every input element. An additional advantange is that the initialization code is run directly on the executors, letting us use non-serializable libraries in a distributed computing process.

filter

```
filter(filterFunc: (T) => Boolean): DStream[T]
```

This function selects some elements of the `DStream` according to the predicate passed as an argument. As for `map`, the predicate is checked on every element of the `DStream`. Beware this can generate empty `RDD`s if no element verifying the predicate is received during a particular batch interval.

glom

```
glom(): DStream[Array[T]]
```

This function, like the homonymous function defined on `RDD` allow us to coalesce elements in an array. In fact, as the `glom` call on a `RDD` returns arrays of elements — as many as there are partitions — the `DStream` equivalent returns the result of calling the `glom` function on each of its constituent `RDD`s.

reduce

```
reduce(reduceFunc: (T, T) => T): DStream[T]
```

This is the equivalent of the `reduce` function on an `RDD`. It allows us to aggregate the elements of an `RDD` using the provided aggregation function. `reduce` takes a function of two arguments, the *accumulator* and a new element of the `RDD`, and return the new value for the accumulator. The result of applying `reduce` on a `DStream[T]` is hence a `DStream` of the same `T` type of which at each `batch interval` will contain an `RDD` with only one element: the final value of the accumulator. Note in particular that `reduce` should be used with care: it cannot deal with an empty `RDD` on its own, and in a streaming application, an empty batch of data can always happen, like when data production or ingestion are stalled.

We can summarize these various transformations in the following way, according to what type of action they have on the source `DStream` . In Table 5-0 we can see whether any operation operates on a whole `RDD` rather than element-wise, and whether it has constraints on the output `RDD`.

operation	**effect**	**output `RDD` 's structure**
`map` ,`filter`	element-wise	unchanged (as many elements as original)
`glom`	partition-wise	as many arrays as there are partitions in the original
`mapPartitions`	partition-wise	as many partitions as original `RDD`

`reduce` ,`fold`	aggregating	one element
`flatMap`	element-wise	as many elements as the size of the output *container*

RDD-centric `DStream` Operations

These operations give us direct access to the underlying `RDD` of a `DStream`. But what makes these operations special is that they execute in the context of the Spark driver, and therefore, we can have access to the facilities provided by the Spark Session (or Spark context) as well as to the execution context of the driver program.

transform

```
transform[U](transformFunc: (RDD[T]) ⇒ RDD[U]): DStream[U]
transform[U](transformFunc: (RDD[T], Time) => RDD[U]): DStream[U]
```

`transform` allows us to reuse a transformation function of type `RDD[T] => RDD[U]` and apply it on each constituent `RDD` of a `DStream`. It is often used to leverage some processing written for a batch job — or more simply in another context — to yield a streaming process.

`transform` also has a timed version, with signature `(RDD[T], Time) => RDD[U]`. As we will remark soon for the `foreachRDD` action, this can be very useful for tagging the data in a `DStream` with the time of the batch that this data was part of.

transformWith

```
transformWith[U,V](
  other: DStream[U], transformFunc: (RDD[T], RDD[U]) => RDD[V]
): DStream[V]
transformWith[U,V](
  other: DStream[U], transformFunc: (RDD[T], RDD[U], Time) => RDD[V]
): DStream[V]
```

`transformWith` let's us combine this `DStream` with another `DStream` using an arbitrary transformation function. It can be used to implement custom `join` functions between the two `DStream`s where the `join` function is not based on the key. For example, we could apply a similarity function and combine those elements that are *close enough*. Like `transform`, `transformWith` offers an overload that provides access to the batch time to provide a differentiation of timestamp mechanism to the incoming data.

Counting

Because the contents of a stream are often comprised of data whose cardinality is important, such as counting erros in a log, or hashtags in a tweet, counting is a very frequent operation that has been optimized enough in Spark to warrant a specific API function. Moreover, as streams of data are often organized in strings of key-value pairs, it is frequent that the keys are organized according to each of these predicates.

Spark has several counting functions for a given `DStream`, which are better seen with an example.

Let's assume we have a `DStream` consisting of names keyed by their first letter, and that this `DStream` "repeats" itself: each `RDD`, on each batch interval, consists of **ten** distinct such names per alphabet letter.

```
val namesRDD: RDD[(Char, String)] = ...
val keyedNames: DStream[(Char, String)] =
    ConstantInputDStream(namesRDD, 5s)
```

This would lead to the results shown in Table 5-1:

Table 5-1. Count Operations

operation	return type	result
`keyedNames.count()`	`DStream[Long]`	260
`keyedNames.countByWindow(60s)`	`DStream[Long]`	260
`keyedNames.countByValue()`	`DStream[((Char, String), Long)]`	1 for each of the 260 distinct (1st character, name) pairs
`keyedNames.countByKey()`	`DStream[(Char, Long)]`	10 for each of the 26 letters
`keyednames.countByValueAndWindow(60s)`	`DStream[((Char, String), Long)]`	12 for each of the 260 distinct (1st character, name) pairs

The above operations are all transformations, that is, they always return a `DStream`. There are other transformations, including `repartition` and `union`, but they are more about changing the flow of data through the Spark cluster than the actual content of the data on the stream, and as such we'll only focus on them later.

However, an important point about the execution of these functions is that like all *transformations*, they are lazy, meaning that they only express the intention of the program in

terms of functions applied to the data. These transformations are only materialized when we need to effectively obtain a result.

In the next section, we are going to learn about a special set of operations, called *output operations* in Spark Streaming jargon, that trigger the execution of *transformations* over our data streams.

Output Operations

Output operations play a crucial role in every Spark Streaming application: They are required to trigger the computations over the `DStream` and, at the same time, they provide access to the resulting data. Thus output operations provide the link between the sequence of lazy transformation and the Spark Streaming scheduler.

From the execution model perspective, every output operation declared in the streaming program is attached to the Spark Streaming scheduler in the same order that they were declared in the program. This ordering guarantees a sequencing semantics in which later output operations will be triggered after the previous operation has finished execution.

Every Spark Streaming job must have, at least, an output operation. This is a logical requirement, as otherwise there's no way to materialize transformations or obtain results. This requirement is enforced at runtime, at the moment that the `sparkStreamingContext.start()` operation gets called. A streaming job that does not provide at least one output operation will fail to start with the following error:

```
scala> ssc.start()

17/06/30 12:30:16 ERROR StreamingContext:
    Error starting the context, marking it as stopped
java.lang.IllegalArgumentException:
    requirement failed: No output operations registered, so nothing to execute
```

Output operations, as the name suggests, provide access to the data that results from the computations declared on the `DStream`. We use them to observe the resulting data, save it to disk or feed it to other systems, like storing it in a DB for later querying or directly sending it to an online monitoring system for observation.

In general terms, the function of output operations is to schedule the provided action at the intervals dictated by the *batch interval*. For output operations that take a closure, the code in the closure executes once at each batch interval on the spark driver, not distributed in the cluster!. All output operations return `Unit`, that is, they only execute side-effecting actions and are not composable. A streaming job may have as many output operations as necessary. They are true endpoints for the transformation DAG of `DStream`s

There are few output operations:

`print()`

> Outputs the first elements of a `DStream` to the standard output. When used without arguments, it will print the first ten elements of the `DStream` at every streaming interval, including the timestamp when the operation executed. It's also possible to call `print(num: Int)` with an arbitrary number, to obtain that given maximum of elements at each streaming interval.
>
> Given that the results are only written to the standard output, the practical use of `print` is limited to exploration and debugging of a streaming computation, where we can see on the

console a continuous log of the first elements of a `DStream`.

For example, calling `print()` in the network stream of names that we used in Chapter 2, we see:

```
namesDStream.print()
ssc.start()

-------------------------------------------
Time: 1498753595000 ms
-------------------------------------------
MARSHALL
SMITH
JONES
BROWN
JOHNSON
WILLIAMS
MILLER
TAYLOR
WILSON
DAVIS
...
```

```
saveAsTextFiles(prefix, suffix)
```

Stores the content of the `DStream` as files in the filesystem. The prefix and optional suffix are used to locate and name the file in the target filesystem. One file is generated at each *streaming interval*. The name of each generated file will be `prefix-<timestamp_in_milliseconds>[.suffix]`

```
saveAsObjectFiles(prefix, suffix)
```

Saves a `DStream` of serializable objects to files using standard Java serialization. The file generation dynamic is the same as for `saveAsTextFiles`.

```
saveAsHadoopFiles(prefix, suffix)
```

Saves the `DStream` as Hadoop files. The file generation dynamic is the same as for `saveAsTextFiles`.

1. HDFS Disk Usage and Streaming Files

It is important to highlight that, depending of the data throughput and chosen *streaming interval*, `saveAsxxx` operations often results in many small files, one generated at each interval. This is potentially detrimental when used in combination with filesystems that have large allocation blocks. Such is the case of HDFS, the Hadoop distributed file system, where the default block size is of 128MB. Doing a back-of-the-napkin calculation, a `DStream` with a throughput of 150Kb/second and with a *batch interval* of 10 seconds will generate files of approx 1.5Mb at each interval. After a day, we should have stored about 12.65Gb of data in 8640 files. The actual usage in an HDFS filesystem with a block size of 128Mb will be 8640 x 128Mb, or roughly 1Tb of data. If we take into account a replication factor of 3, which is a typical HDFS setting, then our initial 12.65Gb of data occupy 3TB of raw disk storage. Needless to say, this is a waste of resources that we would like to avoid.

When using `saveAsxxx` output operations, one approach to consider is to make tumbling windows of a suitable size to match the characteristics of the secondary storage. Continuing with the previous example, we could make a `window` of 850 seconds (almost 15 minutes) to create files of

about 127Mb that are suitable for optimized storage in those conditions. As we will discuss in detail in "Dynamic Windows" we should also consider reducing the data before any storage step, although it is understood that this is a use-case dependent requirement.

Finally, note that the new Hadoop API makes use of Hadoop sequence files to alleviate the issues tied to HDFS storage space consumption. This is reflected in the explicit mentioning of `newAPI` in the name of the methods that access these better implementations. To Learn more about Sequence files and their impact on disk usage, see ???.

`foreachRDD(func)`

> foreachRDD is a general purpose output operation that provides access to the underlying `RDD` within the DStream at each streaming interval. It is our workhorse to materialize Spark Streaming results and arguably the most useful native output operation. As such, it deserves its own section below.

`foreachRDD`

`foreachRDD` is the primary method to interact with the data that has been processed through the transformations declared on the `DStream`. `foreachRDD` has two method overloads:

```
foreachRDD(foreachFunc: RDD => Unit)
```

> The function passed as parameter takes an `RDD` and apply side-effecting operations on it.

```
foreachRDD(foreachFunc: (RDD[T], Time) => Unit)
```

> This is an alternative where we also have access to the time when the operation takes place, which can be used to differentiate the data from the time-of-arrival perspective.

Within the closure of the `foreachRDD`, we have access to the two abstraction layers of Spark Streaming:

The Spark Streaming Scheduler

> Functions applied within the `foreachRDD` closure that does not operate on the `RDD` execute locally in the driver program within the scope of the Spark Streaming scheduler. This level is useful for book-keeping of iterations, accessing external web services to enrich the streaming data, local (mutable) variables or the local file system. It's important to remark that at this level we have access to the `SparkContext` and the `SparkSession`, making it possible to interact with other subsystems of Spark, such as *Spark SQL*, `DataFrames` and `Datasets`.

RDD operations

> Operations applied to the `RDD` provided to the closure function will be executed distributedly in the Spark cluster. All usual `RDD`-based operations are allowed in this scope. These operations will follow the typical Spark-core process of serialization and distributed execution in the cluster.

A common pattern is to observe two closures in `foreachRDD`: An outer scope, containing the local operations and an inner closure applied to the `RDD` that executes in the cluster. This duality is often a source of confusion and is best observed with a code example:

Let's consider the following snippet, where we want to sort the incoming data into a set of *alternatives* (any concrete classification). The alternatives change dynamically and are managed by an external service that we access through a web service call. At each batch interval, the external service is consulted for the set of alternatives to consider. For each alternative we create a `formatter`. We use this particular `formatter` to transform the corresponding selected records by applying a distributed `map` transformation. Lastly, we use a `foreachPartition` operation on the filtered `RDD` to get a connection to a database and store the records. The `DB` connection is not serializable, and therefore we need this particular construct on the `RDD` to get a local instance on each executor.

Note

This is a simplified version of an actual Spark Streaming production job that took care of sorting IoT data for devices of many different customers.

```
1 dstream.foreachRDD{rdd =>
2     rdd.cache()
3     val alternatives = restServer.get("/v1/alternatives").toSet
4     alternatives.foreach{alternative =>
5       val filteredRDD = rdd.filter(element => element.kind == alternative)
6       val formatter = new Formatter(alternative)
7       val recordRDD = filteredRDD.map(element => formatter(element))
8       recordRDD.foreachPartition{partition =>
9           val conn = DB.connect(server)
10           partition.foreach(element => conn.insert(alternative, element)
11       }
12     }
13     rdd.unpersist(true)
14 }
```

At first glance, a lot is going on in this `DStream` output operation. Let's break it down in more digestible pieces:

- On *line 1* we declare a `foreachRDD` operation on a `DStream` and use the closure notation `f{x => ... }` to provide an inline implementation.
- We start our implementation on *line 2*, by caching the provided `RDD` because we are going to iterate multiple times over its content. (We discuss caching in detail in ???).
- We access the web service in *line 3*. This happens once at each *batch interval* and the execution takes place on the driver host. This is also important from the networking perspective, as executors might be "enclosed" in a private network while the driver might have firewall access to other services in the infrastructure.
- On *line 4* we declare a loop over the values contained in the received set, letting us iteratively filter over the values contained by the `RDD`.
- *line 5* declares a `filter` transformation over the `RDD`. This operation is lazy and will take place distributedly in the cluster when some action would call for materialization.
- *line 6* we obtain a local reference to a serializable *Formatter* instance that we use in *line 7* to format the records filtered in the previous step.
- at *line 8* we use a `foreachPartition` action on our filtered `RDD`. We need to do this to get a local instance of the DB driver at the executor (*line 10*), to issue parallel insertion of the records into the database (*line 11*).

As we can see, `foreachRDD` is a versatile output operation where we can mix and match local and distributed operations to obtain results of our computations and push data to other systems after it has been processed by the streaming logic.

3rd Party Output Operations

Several 3rd party libraries add Spark Streaming support for specific target systems by using the *pimp my library* pattern in Scala to add output operations to `DStream`s. For example, the Spark-Cassandra Connector by Datastax enables a `saveToCassandra` operation on `DStream`s to directly save the streaming data to a target Apache Cassandra keyspace and table. The spark-kafka-writer by Ben Fradet enables a similar construct `dStream.writeToKafka` to write a `DStream` to a Kafka topic. ElasticSearch provides support for Spark Streaming using the same pattern. The Spark support for ElasticSearch library enriches the `DStream` API with a `saveToEs` call. More information can be found in their Spark integration guide: Elastic Search Spark Integration Guide.

Those library implementations make use of Scala implicits and `foreachRDD` behind the scenes to offer user-friendly high-level APIs towards specific 3rd party systems, saving the users from the -sometimes intricate- details that deal with the multi-level abstractions within the `foreachRDD` explained previously.

Spark SQL and Spark Streaming

So far, we have seen how Spark Streaming can work as a standalone framework to process streams of many sources and produce results that can be sent or stored for further consumption.

Data in isolation has limited value. We often want to combine datasets to explore relationships that only become evident when data from different sources are merged.

In the particular case of streaming data, the data we see at each *batch interval* is merely sample of a potentially infinite dataset. Therefore, to increase the value of the observed data at a given point in time, it's imperative that we have the means to combine it with the knowledge we already have. It might be historical data we have in files or a database, a model we created based on data from the previous day or even earlier streaming data.

One of the key value propositions of Spark Streaming is its seamless interoperability with other Spark frameworks. This synergy among the Spark modules increases the spectrum of data-oriented applications we can create, resulting in applications with a lower complexity than combining arbitrary, and often incompatible libraries by ourselves. This translates in increased development efficiency that in turn improves the business value delivered by the application: faster time to market, timely business insights, …

In this chapter, we are going to explore how we can combine streaming applications with Spark SQL.

Spark SQL

Spark SQL is the Spark module that works with structured data. It implements functions and abstractions classically found in the realms of databases, like a query analyzer, optimizer, and an execution planner, to enable a table-like manipulation of arbitrarily structured data sources on top of the Spark engine.

Spark SQL introduces three important features:

- The use of the SQL query language to represent data operations
- Datasets, a type-safe data processing DSL resembling SQL
- DataFrames, dynamically-typed counterpart of Datasets

For the purpose of this chapter, we assume the reader's familiarity with Spark SQL, `Dataset`s and `DataFrame`s. For a deeper Spark SQL discussion, we refer the reader to ???

With the combination of Spark Streaming and Spark SQL, we get access to the significant data wrangling capabilities of Spark SQL from the context of a streaming job. We could efficiently enrich an incoming stream using reference data loaded from a database through DataFrames. Or apply advanced summary computations using the available SQL functions. We could also write the incoming or resulting data to one of the supported `writer` formats, such as `Parquet`, `ORC` or to an external database through `JDBC`. The possibilities are endless.

Thinking further ahead, structured data in a stream could be seen as an ever-growing table that can be queried at times to extract the information we are interested in. This is the fundamental principle behind Structured Streaming, the successor of Spark Streaming that we will discuss in [Link to Come]

Accessing Spark SQL Functions From Spark Streaming

The most common use case of augmenting Spark Streaming with Spark SQL is to gain access to the query capabilities and write access to the structured data formats supported, such as relational databases, CSV and Parquet files.

Example: Writing Streaming data to Parquet

Our Streaming dataset will consist of sensor information, containing the sensorId, a timestamp, and a value. For the sake of simplicity in this self-contained example, we are going to generate a randomized dataset, using an scenario that simulates a real IoT use case. The timestamp will be the time of execution and each record will be formatted as a string coming from “the field” of comma separated values.

We also add a bit of real-world chaos to the data: Due to weather conditions, some sensors publish corrupt data.

We start by defining our data generation functions

```
import scala.util.Random
// 100K sensors in our system
val sensorId: () => Int = () =>  Random.nextInt(100000)
val data: () => Double = () => Random.nextDouble
val timestamp: () => Long = () => System.currentTimeMillis
val recordFunction: () => String = { () =>
  if (Random.nextDouble < 0.9) {
    Seq(sensorId().toString, timestamp(), data()).mkString(",")
  } else {
    "!!~corrupt~^&##$"
  }
}
```

```
> import scala.util.Random
> sensorId: () => Int = <function0>
> data: () => Double = <function0>
> timestamp: () => Long = <function0>
> recordFunction: () => String = <function0>
```

Caution

We use a particular trick that requires a moment of attention. Note how the values above are functions. Instead of creating an RDD of text records, we create an RDD of record-generating functions. Then, each time the RDD is evaluated, the record function will generate a new random record. This way we can simulate a realistic load of random data that delivers a different set on each batch.

```
val sensorDataGenerator = sparkContext.parallelize(1 to 100)
                                      .map(_ => recordFunction)
val sensorData = sensorDataGenerator.map(recordFun => recordFun())
```

```
> sensorDataGenerator: org.apache.spark.rdd.RDD[() => String] =
  MapPartitionsRDD[1] at map at <console>:73
> sensorData: org.apache.spark.rdd.RDD[String] =
```

```
  MapPartitionsRDD[2] at map at <console>:74
```

Let's sample some data

```
sensorData.take(5)

> res3: Array[String] = Array(
                !!~corrupt~^&##$,
                26779,1495395920021,0.13529198017496724,
                74226,1495395920022,0.46164872694412384,
                65930,1495395920022,0.8150752966356496,
                38572,1495395920022,0.5731793018367316
)
```

Create the Streaming Context

```
import org.apache.spark.streaming.StreamingContext
import org.apache.spark.streaming.Seconds

val streamingContext = new StreamingContext(sparkContext, Seconds(2))
```

Our stream source will be a `ConstantInputDStream` fed by a record-generating `RDD`. By combining a `ConstantInputDStream` with the record generating `RDD`, we create a self-generating stream of fresh random data to process in our example. This method makes the example self-contained. It removes the need of an external stream generating process.

```
import org.apache.spark.streaming.dstream.ConstantInputDStream
val rawDStream  = new ConstantInputDStream(streamingContext, sensorData)
```

Provide schema information for our streaming data

> Now that we have a DStream of fresh data processed in a 2-second interval, we can start focusing on the gist of this example. First, we want to define and apply a schema to the data we are receiving. In Scala, we define a schema with a `case class`

```
case class SensorData(sensorId: Int, timestamp: Long, value: Double)

> defined class SensorData
```

Apply the schema to the dstream using the `flatMap` function

Note

We use `flatMap` instead of a `map` because there might be cases when the incoming data is incomplete or corrupted.

If we would use `map`, we would have to provide a resulting value for each transformed record. That is something we cannot do for invalid records. With `flatMap` in combination with `Option`, we can represent valid records as `Some(recordValue)` and invalid records as `None`. By the virtue of `flatMap` the internal `Option` container gets flattend and our resulting stream will only contain valid `recordValue`**s**.

During the parsing of the comma separated records, we not only protect ourselves against missing fields, but also parse the numeric values to their expected types. The surrounding `Try` captures any `NumberFormatException` that might arise from invalid records.

```
import scala.util.Try
val schemaStream = rawDStream.flatMap{record =>
  val fields = record.split(",")
```

```
  if (fields.size == 3) {
      Try {
        SensorData(fields(0).toInt, fields(1).toLong, fields(2).toDouble)
      }.toOption
    } else { None }
  }
```

```
> schemaStream: org.apache.spark.streaming.dstream.DStream[SensorData] =
org.apache.spark.streaming.dstream.FlatMappedDStream@4c0a0f5
```

Saving DataFrames

> With the schema stream in place, we can proceed to transform the underlying RDDs in DataFrames. We do this in the context of the general-purpose action `foreachRDD`. It's impossible to use transformation at this point because a `DStream[DataFrame]` is undefined. This also means that any further operations we would like to apply to the DataFrame (or DataSet) needs to be contained in the scope of the `foreachRDD` closure.

```
import org.apache.spark.sql.SaveMode.Append
schemaStream.foreachRDD{rdd =>
  val df = rdd.toDF()
  df.write.format("parquet").mode(Append).save("/tmp/iotstream.parquet")
}
```

Finally, we start the stream process

```
streamingContext.start()
```

We can then inspect the target directory to see the resulting data streaming into the parquet file.

Now, we should head to the URL `http://<spark-host>:4040` to inspect the Spark Console. We will see that both `SQL` and `Streaming` tabs are present. In particular, the `Streaming` Tab is of our interest.

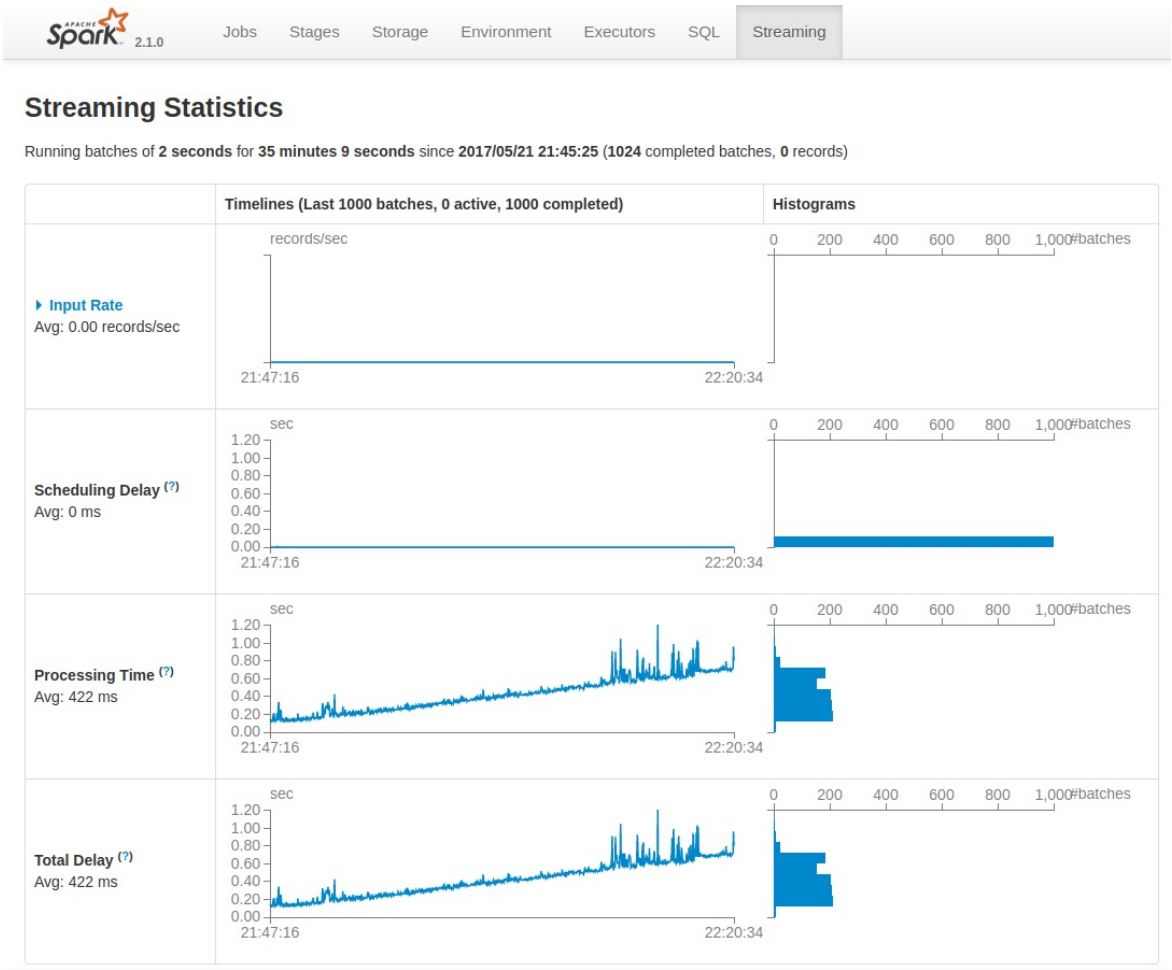


Figure 5-1. Saving a Stream to Parquet

In Figure 5-1, can see how the writing time to the parquet quickly increases with time. Append operations to parquet files get more expensive with time. For long-lived processes like a Spark Streaming job, one way to work around this limitation is to append to a fresh file every so often.

```
def ts: String = ((time.milliseconds - timeOrigin)/(3600 * 1000)).toString
df.write.mode(SaveMode.Append).format("parquet").save(s"${outputPath}-$ts")
```

In the listing ???, we change the suffix of the file every hour (3600 seconds), starting to measure from the moment the streaming job started.

Note

In the Streaming UI, you may notice that the `Input Rate` chart remains flat on zero. This chart collects its information from the `Receiver` implementation. As we are using a `ConstantInputDStream`, there's no actual incoming record count for this chart.

Dealing with Data at Rest

When building streaming applications, a typical question that arises during the design and implementation of streaming applications is how to use existing data, being historical records of the same kind as the streaming data, lookup tables to resolve specific fields or reference data used for comparison purposes, among others.

We can use the capabilities of Spark SQL to load "resting" datasets that can be combined with the incoming streaming data. A key advantage of Spark SQL in this scenario is that the loaded data is structured. This reduces the effort that the streaming developer needs to invest in preparing the data into the right format before performing a join operation.

This example illustrates the use of `join` of a fixed dataset in the form of a `DataFrame` and the streaming data that is consumed continuously by our applicaton.

Using `join` to enrich the input stream

In our previous example, we processed an incoming stream of sensor data, parsed it into valid records and stored it directly into a *Parquet* file. We miss key information about the sensors before we can start processing their data. In particular, we cannot interpret the values if we don't know the type of sensor, its working range and the units of each recorded value. In our IoT application, sensors are initially registered before they are deployed. The registration process captures the information that we are requiring. Luckily for us, this information is available as a CSV file export that we can import in our streaming application, as we can see in Figure 5-2.

Note

For this example we are only going to discuss the relevant differences with the previous program. The complete notebook is available online for the reader's own exploration at *https://github.com/LearningSparkStreaming/notebooks/*

Define some constants with the locations of our data

```
val sensorCount = 100000
val workDir = "/tmp/learningsparkstreaming/"
val referenceFile = "sensor-records.parquet"
val targetFile = "enrichedIoTStream.parquet"
```

Load the reference data from a Parquet file

> We also cache the data to keep it in memory and improve the performance of our streaming application

```
val sensorRef = sparkSession.read.parquet(s"$workDir/$referenceFile")
sensorRef.cache()
```

sensorId	sensorType	unit	minRange	maxRange
25001	"noise"	"dB"	0	200
25002	"luminosity"	"Lux"	0	100000
25003	"noise"	"dB"	0	200
25004	"humidity"	"Rel%"	0	100
25005	"radiation"	"mSv"	0	100000
25006	"temp"	"C"	-100	100
25007	"radiation"	"mSv"	0	100000
25008	"radiation"	"mSv"	0	100000
25009	"temp"	"C"	-100	100
25010	"temp"	"C"	-100	100
25011	"noise"	"dB"	0	200
25012	"pressure"	"kPa"	0	140
25013	"pressure"	"kPa"	0	140
25014	"luminosity"	"Lux"	0	100000
25015	"radiation"	"mSv"	0	100000
25016	"luminosity"	"Lux"	0	100000

Figure 5-2. Sample reference data

Enrich the streaming data

With the schema stream in place, we can proceed to transform the underlying RDDs into DataFrames. This time, we are going to use the reference data to add the specific sensor information. We are also going to de-normalize the recorded value according to the sensor range so that we don't have to repeat that data in the resulting dataset.

As before, we do this in the context of the general-purpose action foreachRDD.

```
val stableSparkSession = sparkSession
import stableSparkSession.implicits._
import org.apache.spark.sql.SaveMode.Append
schemaStream.foreachRDD{ rdd =>
  val sensorDF = rdd.toDF()
  val sensorWithInfo = sensorDF.join(sensorRef, "sensorId")
  val sensorRecords =
    sensorWithInfo.withColumn(
      "dnvalue", $"value"*($"maxRange"-$"minRange")+$"minRange"
    ).drop("value", "maxRange", "minRange")
  sensorRecords.write
               .format("parquet")
               .mode(Append)
               .save(s"$workDir/$targetFile")
}
```

Warning

The seemingly weird construct for the `stableSparkSession` is necessary because, in the SparkNotebook, the `sparkSession` reference is a mutable variable and we cannot issue an import from a non-stable reference.

Inspect the result

We can use the current Spark Session concurrently with the running Spark Streaming job in order to inspect the resulting data, as shown in Figure 5-3

```
val enrichedRecords = sparkSession.read.parquet(s"$workDir/$targetFile")
enrichedRecords
```

```
enrichedRecords
res31: org.apache.spark.sql.DataFrame = [sensorId: int, timestamp:
                                                    1  >>
```

sensorId	timestamp	sensorType	unit	dnvalue
26140	1495402992211	"radiation"	"mSv"	8529.460065196548
84739	1495402992212	"luminosity"	"Lux"	40891.46792556275
82070	1495402992212	"temp"	"C"	14.345990752880283
10484	1495402992212	"noise"	"dB"	91.61852200516864
70760	1495402992212	"radiation"	"mSv"	53155.29779322772
75464	1495402992212	"luminosity"	"Lux"	18450.203608756743
3970	1495402992212	"radiation"	"mSv"	833.5520188078372
52765	1495402992212	"humidity"	"Rel%"	33.70152466216159
99686	1495402992212	"noise"	"dB"	16.095900641350113
64578	1495402992212	"pressure"	"kPa"	81.47635868016002
65802	1495402992213	"radiation"	"mSv"	25745.591428913216

Figure 5-3. Sample enriched records

See the record count evolving

We can issue a `count` on the resulting dataset to see how the streaming process increments the data set. We wait a few moments between the two executions to observe the difference.

```
enrichedRecords.count
>res33: Long = 45135
// ... wait few seconds ...
enrichedRecords.count
>res37: Long = 51167
```

Join Optimizations

Our current solution has a major drawback: it will drop incoming data for unregistered sensors. As we are loading the reference data only once at the start of our process, sensors registered after that moment will be silently dropped. We can improve on this situation by using a different kind of `join` operation. Let's remember that within the `foreachRDD`, we have full access to the capabilities of other Spark libraries. In this particular case, the `join` operation we are using comes from Spark SQL and we can make use of options in that package to enhance our streaming process. In particular, we are going to use an *outer* join to differentiate between the IoT sensor ids that are known to the system from those that are unknown. We can then write the data from the unknown devices to a separate file for later reconciliation.

The rest of the program remains the same except for the `foreachRDD` call, where we add the new logic.

```
schemaStream.foreachRDD{rdd =>
  val sensorDF = rdd.toDF()
  val sensorWithInfo = sensorRef.join(
    broadcast(sensorDF), Seq("sensorId"), "rightouter"
  )
  val unknownSensors = sensorWithInfo.filter($"sensorType".isNull)
  val knownSensors = sensorWithInfo.filter(!$"sensorType".isNull)
  val denormalizedSensorData = knownSensors.withColumn(
    "dnvalue", $"value"*($"maxRange"-$"minRange")+$"minRange"
  )
  val sensorRecords = denormalizedSensorData.drop(
    "value", "maxRange", "minRange"
  )
  sensorRecords.write
               .format("parquet")
               .mode(Append)
               .save(s"$workDir/$targetFile")

  unknownSensors.write
                .format("parquet")
                .mode(Append)
                .save(s"$workDir/$unknownSensorsTargetFile")
}
```

Sharp eyes will notice that we have introduced two changes to the join operation:

```
val sensorWithInfo = sensorRef.join(
  broadcast(sensorDF), Seq("sensorId"), "rightouter"
)
```

The first difference, highlighted in listing ???, is that we changed the order of the join. Instead of joining the incoming data with the reference dataset, we do the opposite. We need that change in direction for a particular reason: We are adding a `broadcast` hint to the join expression to tell Spark to perform a broadcast join.

Broadcast joins, also known as *map-side joins* in Hadoop jargon, are useful when there's a large difference in size between the two datasets participating in the join and one of them is small enough to be sent to the memory of every executor. Instead of performing an I/O heavy shuffle-based join, the small dataset can be used as an in-memory lookup table on each executor in parallel. This results in lower I/O overhead and hence, faster execution time.

In our scenario, we swap the order of the participating datasets, because we know that our reference dataset is much larger than the received device data at each interval. While our reference dataset contains a record for every device known to us, the streaming data contains only a sample of the total population. Therefore, it's performance-wise better to broadcast the data from the stream to execute the broadcast join with the reference data.

The last remark in this line of code is the join direction, given by the join type `rightouter`. This join type will preserve all records from the right side, our incoming sensor data, and add the fields from the left side if the join condition match. In our case, that is a matching sensorId.

We store the results in two files. One with the enriched sensor data for the known `SensorId`s (see Figure 5-3) and another with the raw data of the sensors we don't know at this time (see Figure 5-5).

sensorId	sensorType	unit	timestamp	dnvalue
62882	"windspeed"	"m/s"	1498867506096	23.349344176689513
62931	"humidity"	"%Rh"	1498867506096	24.07863007670845
63047	"brightness"	"lux"	1498867506096	13391.298697905462
64111	"pressure"	"mmHg"	1498867506096	858.4349371259426
64192	"humidity"	"%Rh"	1498867506096	91.01956672907862

Figure 5-4. Sample of enriched records

sensorId	sensorType	unit	minRange	maxRange	timestamp	value
100200					1498867518294	0.4296108409658099
100275					1498867506376	0.3535681531311208
100855					1498867506366	0.6230050066156928
100920					1498867530329	0.30692198074148225
100562					1498867510248	0.05377385469435081

Figure 5-5. Sample of records from unknown devices

For an in-depth discussion of join options and dynamics, we refer the reader to ???

Updating Reference Data

In the previous section we saw how we can load a static dataset to enrich an incoming stream.

Although some datasets are fairly static, like last-year synthetic profile for Smart Grids, calibration data for IoT Sensors or population distribution following a census, oftentimes we will also find that the dataset that we need to combine with our streaming data changes as well. Those changes might be at a different, much slower pace than our streaming application and hence do not need to be considered a stream on their own.

In their digital evolution organization often have a mix of processes of difference cadences. Slow-paced output processes, such as data exports or daily reports are valid inputs of our streaming system that require regular, but not continuous updates.

We are going to explore a Spark Streaming technique to integrate such “slow data” into out “fast data” track.

At its core, Spark Streaming is a high-performance scheduling and coordination application. To ensure data integrity, Spark Streaming schedules the different output operations of a streaming job in sequence. The order of declaration in the application code becomes the execution sequence at runtime.

In [Link to Come] we learnt about the `batch interval` and how it provides the synchronization point between the collection of data and the submission of previously collected data to the Spark engine for further processing. We can hook onto the Spark Streaming scheduler in order to execute Spark operations other than stream processing. In particular, we will use a `ConstantInputDStream` with an empty input as the key building block to schedule our additional operations within the streaming application. We combine the empty DStream with the general-purpose operation `foreachRDD` to have Spark Streaming take care of the regular execution of those additional operations we require in the context of our Spark Streaming application.

Updating Reference Dataset in a Streaming Application

To better understand the dynamics of this technique, we will continue with our running example.

In the previous section we used a reference dataset that contained the description of each sensor known to the system. That data was used to enrich the streaming sensor data with the parameters needed for its further processing. One important limitation of that approach is that, once the streaming application is started, we cannot add, update or remove any sensor present in that list.

In the context of our example, we get an update of that list every hour and we would like that our streaming application would use the new reference data.

Load the reference data from a Parquet file

> As in the previous example, we load the reference data from the Parquet file. We also

cache the data to keep it in memory and improve the performance of our steaming application. The only difference we will observe in this step is that we now use a `variable` instead of a `value` to keep the reference to our reference data. We need to make this reference mutable because we will be updating it with new data as the streaming application runs.

```
var sensorRef: DataFrame = sparkSession.read.parquet(s"$workDir/$referenceFile")
sensorRef.cache()
```

Setup the refreshing mechanism

In order to periodically load the reference data, we are going to *hook* onto the Spark Streaming scheduler. We can only do that in terms of the `batch interval` which serves as the internal tick of the clock for all operations. Hence, we will express the refresh interval as a `window` over the base `batch interval`. In practical terms, every x batches we are going to refresh our reference data.

We use a ConstantInputDStream with an empty RDD. This ensures that, at all times, we have an empty DStream whose only function will be to give us access to the scheduler through the `foreachRDD` function. At each `window` interval, we will update the variable that points to the current `DataFrame`. This is a safe construction as the Spark Streaming scheduler will linearly execute the scheduled operations that are due at each `batch interval`. Therefore, the new data will be available for the upstream operations that make use of it.

We use caching to ensure that the reference dataset is only loaded once over the intervals that it's used in the streaming application. It's also important to `unpersist` the expiring data that was previously cached in order to free up resources in the cluster and ensure that we have a stable system from the perspective of resource consumption.

```
import org.apache.spark.rdd.RDD
val emptyRDD: RDD[Int] = sparkContext.emptyRDD
val refreshDStream  = new ConstantInputDStream(streamingContext, emptyRDD)
val refreshIntervalDStream = refreshDStream.window(Seconds(60), Seconds(60))
refreshIntervalDStream.foreachRDD{ _ =>
  sensorRef.unpersist(false)
  sensorRef = sparkSession.read.parquet(s"$workDir/$referenceFile")
  sensorRef.cache()
}
```

We use a tumbling `window` of 60 Seconds for our refresh process. Loading a large dataset has a cost in time and resources and therefore it has an operational impact. In the illustration Figure 5-6 we can appreciate recurrent spikes in processing time that correspond to the loading of the reference dataset.

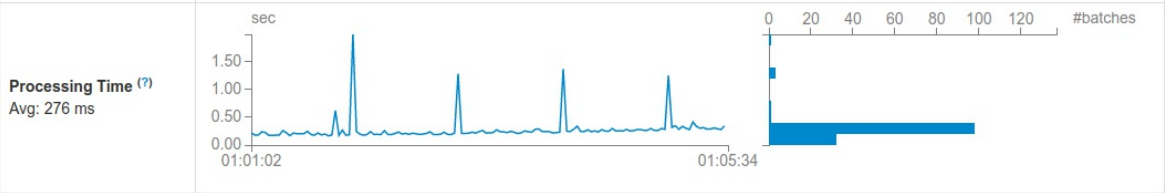


Figure 5-6. Loading reference data has a visible impact on the runtime

By scheduling the load of reference data at a rate much lower than the cadence of the streaming application, the cost is amortized over a relative large number of micro-batches. Nevertheless, this additional load needs to be taken into consideration when planning for cluster resources in order to have a stable execution over time.

Stateful Streaming Computation

So far in this chapter, we have seen how Spark Streaming can work on the incoming data independently of previously received records. In many applications, we are also interested in analysing the evolution of the data that arrives with respect to older data points. We can also be interested in tracking changes generated by the received data points, that is, in building a stateful representation of the data already seen.

Spark Streaming provides several functions that let us build and store knowledge about previously seen data as well as use that knowledge to transform new data.

UpdateStateByKey

Functional programmers are already used to liking functions without statefulness — that is, they return something which is independent from the state of the world outside their function definition, caring only about the value of their input.

However, a function can be stateless, care only about its input, and yet maintain a notion of a managed value along its computation, without breaking any rules about being functional. The idea in this case is that this value is used in the traversal of one or several arguments of the computation, to keep some record along the traversal of the argument's structure.

For example, the `reduce` operation we were talking about earlier in the present chapter was keping one single value updated along the traversal of whichever `RDD` they were given as argument:

```
val tumblingSums = myDStream.reduce {
  case (accum, x) => (accum + x)
}
```

Here the computation of the intermediate sums for each `RDD` along the input `DStream` is made by iterating over the elements of the `RDD`, from left to right, and keeping an accumulator variable updated, an operation specified thanks to the update operation which returns the new value of the accumulator (between brackets).

The idea of this computation by accretion, though, is that it is bounded by the limits of the `RDD` which can be seen as its input: there is no way — through the fold operation alone — to compute a result that would depend on the contents of an input `RDD` and of the ones that would occur before in the stream.

Statefulness at the scale of a stream

Sometimes it is useful to compute some result that depends on the previous elements of the stream that can have occured more than one batch before the current one. Examples include:

- the running sum of all elements of the stream
- the number of occurrences of a specific, marker value
- the highest elements encountered in the stream, given a particular ordering of the elements of the stream

This computation can often be thought of as the result of a big `reduce` operation, that would update some representation of the state of a computation, all along the traversal of the stream.

Spark Streaming offers this operation, but it is not expressed as simply as a sort of `reduce`, because of the need to structure the ouptut of the computation differently. A simple fold outputs when the updates to the accumulator are finished and the input fully traversed. However, a stream is by nature potentially infinite, so that it would not do to have an operation that only gives a result if the stream reaches some sort of end. To represent that more successfully, we'll consider the `updateStateBykey` function.

updateStateByKey

> `updateStateBykey` is an operation that exists only on `DStreams` of key-value pairs, and that takes a state update function as argument.

This state update function should have the type

```
`Seq[T] -> Option[U] -> Option[U]`
```

This type reflects how the update operation takes a set of new values fo type `T`, arrived during the current batch, and an optional state, here represented with type `U`. It the computes and returns a new values for the state if there is one to return.

There is a lot to unpack in this function, and we will see an example in a few lines, but the first thing to notice is that this processing occurs on every batch, per key. A consequence of this is that several values of type `U` (up to one per key) are returned on each batch. Hence, to a stream of input key-value pairs corresponds the output of `updatestateBykey` , itself a stream, but of state snapshots : values of type `U`.

The second thing to know is that there is one value for key encountered during the last `RDD` — so there will be a call of the update state function for each of the keys for which a new value recently came through the stream. The counter-intuitive thing to notice as well is that the state-update function will also be called for each of the values that do not necessarily receive an update on the current batch.

To sum up, the update state function is called, on each batch, on all the keys met since the beginning of processing this stream. On some cases this is on a new key that was never met before — this is the case where the second argument of the update function, the state, is `None`. On other cases, it will be on a key for which no new values have come in this batch — this is the case where the first argument of the update function, the new values, is `Nil`. Finally, there will be cases where both arguments contain actual data, for both list and state: a previously-encountered key that is receiving new arguments. Depending on wheter the use case for the function is more or less dense or sparse data, this will occur more or less often.

updateStateByKey and its limitations

The `updateStateByKey` function that we have described so far allows to do stateful programming using Apache Spark. It allows you to encode, for example, the concept of user sessions — for which no particular batch interval is a clear match to the application at hand. However, there are 2 problems with this approach.

Performance

The first is a performance problem: this `updateStateByKey` function is run on every single key encountered in the framework of the application since the beginning of the run. This is problematic because, even on a data set that is somewhat sparse — provided there is a long tail to the variety of the data and in particular, in the variety of its keys — there is a clear argument that the total amount of data represented in memory grows indefinitely.

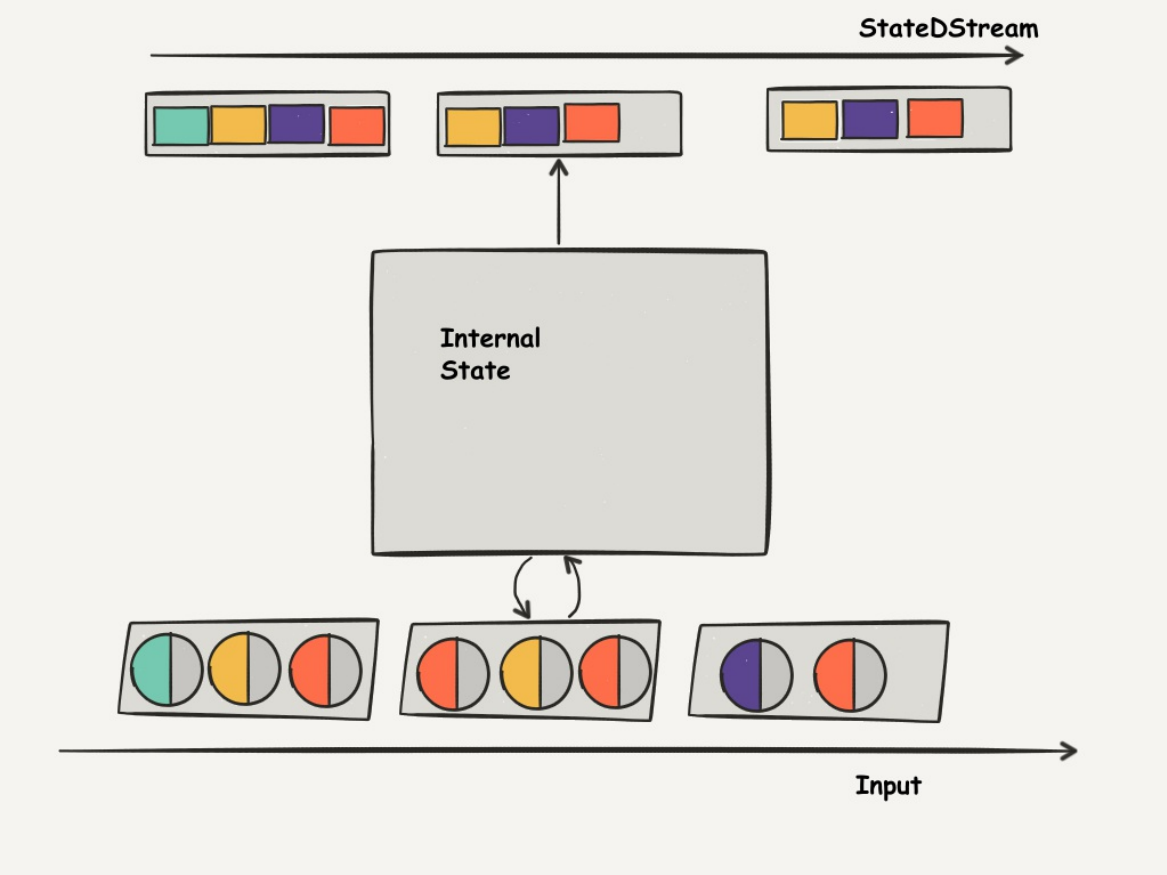


Figure 5-7. The data flow produced by updateStateByKey

This is an issue because the run time necessary to update the state for every single key can be unnecessarily long. For example, if a key or a particular user is seen on a website at the

beginning of the run of the application, what is the relevance of updating the state of that user to signify that we have not seen a session from this particular individual since the beginning of the application, which is since last month? The benefits to the application program are not clear.

Memory Usage

The second issue is that since the state cannot grow indefinitely because of the bounded memory available in a cluster, the programmer has to do the bookkeeping of memory by himself, writing code for every key, to figure out if it is still relevant to keep data in state for that particular element. This is not just a problem of performance, but of manual accounting and memory management.

Indeed, for most stateful computations, dealing with state is a simple operation: either a key is still relevant — such as a user who has visited a website within a certain time frame — or he is not, and the difference is usually decided by time since the last visit. This timeout operation is not provided as a default in `updateStateByKey`. In that particular occasion, we could argue that Apache Spark is — or was — a little bit too low level of an abstraction in the context of stateful programming.

`mapwithState`

The developers of Apache Spark, starting with version 1.5, have thought of those two issues and sought to find a better model for stateful programming using Apache Spark.

That better model was reached first in beta, then in a wider release in the version 1.6 to a function called `mapWithState` that we are going to demonstrate a little bit later, has 2 advantages:

Selective Key Updates

The first is that it does not update every single key on the reception of a new batch of data. It only updates those keys that receive new data for that particular batch. The state for all the other keys is unchanged, which is an advantage because the user does not have to explicitly specify what should happen. However, it is essential that the state kept in memory in the distributed cluster on which our Apache Spark application runs stays limited.

The necessary obstruction here is a default for bookkeeping of the state of the particular keys that are encountered throughout the lifecycle of the application. That timeout is a good default used by many programmers in those bookkeeping operations. We have seen that this pattern is the one most often used by programmers choosing to do stateful programming with Apache Spark. It seems like a best default for the `mapWithState` function. The way this timeout functions is that the state for a particular key is swept out of the state store, provided that the last update for the key in question was longer ago than the specified timeout.

Another shortcoming of the `updateStateByKey` function approach that we have described previously is that the output of that function is always a snapshot of the state for the current batch. This default makes sense, in the sense that it allows a programmer to inspect the maintained state using functions that operate on `DStream`s, distributed streams. You would have to notice that it is particularly problematic to analyze in terms of operational programming needs. Indeed, for a particular user, you might be keeping into state, not only the history of his previous sessions, but the last few events of the last session. The time frame for a session might be around 10 minutes. The time frame for your state might cover the last few visits of that user to your website, let’s say, during the last few hours. In that context where you keep into state more than one session per user, it is difficult to analyze every single user and to figure out what the latest updates for that particular individual’s wear.

Was that latest update particular new session that has begun? Was it a few elements on an already begun session, or was it the closing of the current session because the activity that the user has demonstrated during the last batch, which is none, has shown that he has disengaged, and that his session should now close. This complex information now needs to be kept in the state of your distributed stream rather than in your application. Now, whether this is a good or bad pattern depends on the particular use case. Forcing all this detailed and complex information into one particular `DStream` forces the programmer to manipulate lambda functions to update all the details of his state representation.

On the other hand, the `mapWithState` function allows you to output one particular piece of

information expected from both new values received for a particular key and the pre-existing state for that user on each and every batch interval. In that particular situation, the bookkeeping of the state is kept inside the `mapWithState` function, and the output and the treating of changes to the state is kept as the output of that function, and therefore, is treated by the rest of the Spark application outside of that function. This has 2 advantages. The first is separation of concern between the bookkeeping of the state and whichever actionable data can be extracted from the change in the last few values for a particular key, which is represented as the output. In Figure 5-8 we can see how the data flows when using the `mapWithState` function.

Scoped Value Output

The second advantage is that only users that is keys for which there was an update during the last batch interval, create an output for the map with state function, because usually, in Sparks applications, or in the case of situations where the variants in the frequency of updates between keys is very large, you do not receive an output of the size of your state on every single batch interval. The consequence is that you don't have to scan something of the size of your state. On every batch interval, the function that is operating based on the output of your stateful computation is a function that only has to deal with new information provided by the input data during the last batch interval, because this is usually much less information, though your whole stateful Apache Spark streaming pipeline is, as a whole, much faster. Note, however, that this has implications as to the usability of the `mapWithState` function.

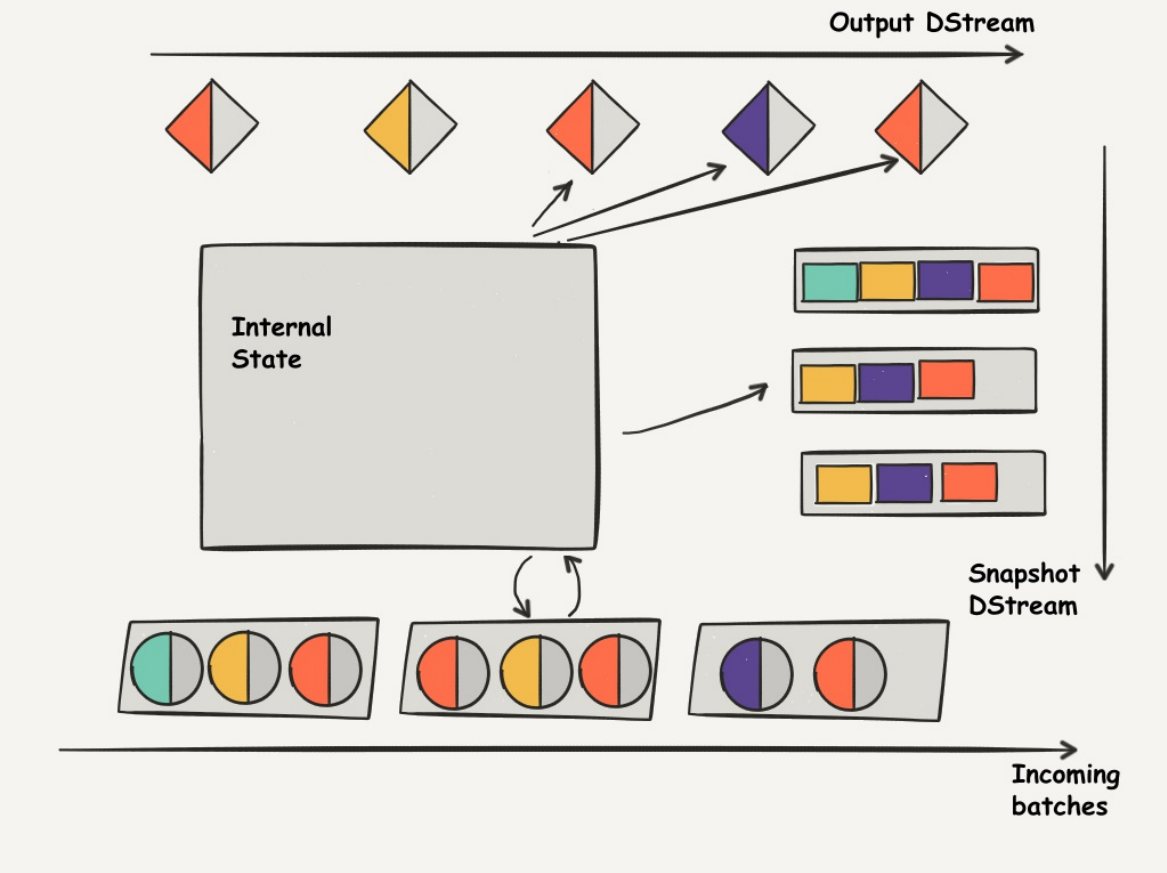


Figure 5-8. The dataflow produced by mapWithState

The implication is that you can only use `mapWithState` function if your criteria for releasing data out of your stateful representation and reducing the amount of consumed memory for a particular key is a timeout. If your application does have to keep all the information for a particular key in memory, potentially indefinitely, then `mapWithState` may not be a good fit for you. One example of such an application is, let's say, a particular stateful filter. That stateful filter would have to deal with a particular configuration if a few bad apples occur on the stream. Those bad apples would be represented by a handful of keys. Those keys would indicate particular events occurring in your input data. Because your particular events are specified by an exterior need that has nothing with the shape of your computation, those externalities imply that you should keep data in memory for the duration of the application, especially if the occurrence of those bad apple events is especially significant for the purpose of your application.

However, note that the particular output of the `mapWithState` function is not a limitation on your application. If you want to see a snapshot of the state for each and every key on each and every batch interval, this is a possible output that has been planned and defined by the Apache Spark developers. Just call the `.snapshots` function on your particular Dstream created by `mapWithState` , that stateful stream will be able to output exactly the same output that you would obtain with the update method described a few paragraphs ago. A consequence of the speed of those updates, and the high performance of `mapWithState` , it allows a new kind of application, and a new kind of semantics into Spark Streaming.

For example, one particularly important application in streaming world is that events often occur on the wire in the input stream, out of their order, with respect to the world clock that witnesses their arrival on the stream. For example, if your stream is supposed to witness the arrival of events that occur every second, you might see the event for second 5 occurring on your stream before even for second 4. The explanation for this can be varied for particular use cases and domain specifics. One simple example of latency might be simply that you encounter a redistribution server that may be, in the past, between your creation of your events and your reception of it.

The treatment of those events is done using the notion of a watermark. A watermark is a number boundary on the limit of time that you're going to allocate for waiting for the arrival of out of order events. The task of your stream, then, is simply to reproduce an order of events that reflects the event time that the timestamp that your events contain, and that indicates to you which precedents you should observe in ordering those events. This assumes that, first, you have an original timestamp for each of those events. It also assumes that you are able to wait up until the watermark arrival of your event for processing on your data to be finished. However, it does not necessarily mean that you need to wait until the arrival of the watermark for the processing on your input event to be finished.

The operation of your stream, that is the core of your computation, can stop as events are being received, and ideally, completes shortly after last few structural events are received. We are going to see an example of this in the next few pages.

Using `mapWithState`

`mapWithState` requires the user to provide a `StateSpec` object, that describes the workings of the state computation. The core piece of this is, naturally, the function that takes in new values for a given key, and both returns an output and updates the state for this key. In fact, the builder object for this `StateSpec` makes this function mandatory.

This `StateSpec` is parameterized by four types: - the key type K, - the value type V, - the state type S, representing the type used to store the state, - the output type U

In its most general form, the `StateSpec` builder `StateSpec.function` requires a `(Time, K, Option[V], State[S]) => Option[U]` argument, or a `(K, Option[V], State[S]) => Option[U]` if you don't need the timestamp of the batch that `mapWithState` comes with.

The `State` type that intervenes in the definition of this function can be seen as a mutable cell with support for timeout. You can query it with `state.exists()` and `state.get()`, or even treat it like an option with `state.getOption()`, you can check if it's timing out with `state.isTimingOut()`, erase it with `state.remove()` or update it with `state.update(newState: S)`.

Let's assume we're monitoring a plant with sensors and we want both the average temperature over the last batch and an extremely simple way of detecting anomalous temperatures. We will define an anomalous temperature to be above 80 degrees:

```
case class Average(count: Int, mean: Float){
  def ingest(value: Float) =
    Average(count + 1, mean + (value - mean) / (count + 1))
}

def trackHighestTemperatures(sensorID: String,
    temperature: Option[Float],
    average: State[Average]): Option[(String, Float)] = {
  val oldMax = average.getOption.getOrElse(Average(0, 0f))
  temperature.foreach{ t => average.update(oldMax.ingest(t)) }
  temperature.map{
    case Some(t) if t >= (80) => Some(sensorID, t)
    case _ => None
  }
}

val highTempStateSpec = StateSpec.function(trackHighestTemperatures)
                                 .timeout(Seconds(3600))
```

In this function, we extract the old maximum value and run both an aggregation of the latest value with the mean and with the threshold value, routing the results to, correspondingly, the state update and our output value. This lets us exploit two streams:

- `temperatureStream.mapWithState(highTempStateSpec)`, which tracks high temperatures as they occur,
- and `temperatureStream.mapWithState(highTempStateSpec).stateSnapshots()`, which tracks the mean temperatures for each sensor.

If a sensor stops emitting for 60 minutes, its state is removed automatically, preventing the state

storage explosion that we feared. Note that we could have used the explicit `remove()` function for this as well.

There is, however, an issue with this : in the first few values of a sensor, we will compare the sensor's value to a low default, which may not be appropriate for each sensor. We might detect temperature spikes reading values that may be perfectly appropriate for this particular sensor, simply because we do not have values for it yet.

In this case, we have the opportunity to provide initial values for our sensor, using `highTempStateSpec.initialState(initialTemps: RDD[(String, Float)])`.

Event-time Stream computation with `mapWithState`

An auxiliary benefit of `mapWithState` is that it lets us efficiently and explicitly store data about the past in its `State` object. This can be very useful in order to accomplish event-time computing.

Indeed, elements seen "on the wire" in a streaming system can arrive out of order, be delayed, or even arrive exceptionally fast with respect to other elements. Because of this, the only way to be sure we are dealing with data elements that have been generated at a very specific time is to timestamp them at the time of generation. A stream of temperatures in which we are trying to detect spikes, as above, could confuse temperature increases and temperature decreases if some events arrive in inverse order, for example.

However, if we aim to process events in order, we need to be able to detect and invert the mis-ordering by reading the timestamps present on data elements seen on our stream. To perform this re-ordering, we need to have an idea of the order of magnitude of the delays (in one direction or another) we can expect to see on our stream. Indeed, without this boundary on the scope of the reordering, we would need to wait infinitely to be able to compute a final result on a particular period of time: we could always receive yet another element that would have been delayed.

To deal with this practically, we are going to define a *watermark*, or the highest duration of time we are going to wait for straggler elements. In the spirit of Spark Streaming's notion of time, it should be a multiple of the batch interval. After this watermark, we are going to "seal" the result of the computation, and consider that elements that are delayed by more than the watermark can be ignored.

Caution

The spontaneous approach to dealing with this misordering could be windowed streams: define a window interval equal to the watermark, and make it slide by exactly one batch, defining a transformation that orders elements by their timestamp.

This is correct in so far as it will result in a correct view of the ordered elements, once past the first watermark interval. However, it requires the user to accept an initial delay equal to the watermark to see the results of computation. It's plausible, however, to see a watermark that's one order of magnitude higher than the batch interval, and such a latency would not be acceptable for a system like Spark Streaming, which already incurs high latency because of its micro-batching approach.

A good event-time streaming solution would allow us to compute based on a provisional view of the events of the stream, and update this result if and when delayed elements arrive.

Let's say we have a notion of a circular buffer, a fixed-size vector of size k, that contains the last k elements it has received.

```
import scala.collection.immutable

object CircularBuffer {
```

```
  def empty[T](): CircularBuffer[T] = immutable.Vector.empty[T]
}

implicit class CircularBuffer[T](v: Vector[T]) extends Serializable {
  val maxSize = 4
  def get(): Vector[T] = v
  def addItem(item : T) : CircularBuffer[T]  =
    v.drop(Math.min(v.size, v.size - maxSize + 1)) :+ item
}
```

This object will keep an inner vector of at least one, and at most `maxSize` of elements, selecting its most recent additions.

Let's now assume that we are tracking the average temperature for the last four batches, assuming a batch interval of 5 milliseconds.

```
import org.apache.spark.streaming.State

def batch(t:Time): Long = (t.milliseconds % 5000)

def trackTempStateFunc(
  batchTime: Time,
  sensorName: String,
  value: Option[(Time, Float)],
  state: State[CB]): Option[(String, Time, Int)] = {

  value.flatMap { (t: Time, temperature: Float) =>
     if ( batch(t) <= batch(batchTime)) { // this element is in the past
      val newState: CB =
        state.getOption.fold(Vector((t, Average(1, temperature))): CB){ c =>
          val (before, hereOrAfter) =
            c.get.partition{case (timeStamp, _) => batch(timeStamp) < batch(t) }
          (hereOrAfter.toList match {
            case (tS, avg: Average) :: tl if (batch(tS) == batch(t)) =>
              (tS, avg.ingest(temperature)) ::tl
            case l@_ => (t, Average(1, temperature)) :: l
          }).toVector.foldLeft(before: CB){ case (cB, item) => cB.addItem(item)}
        }
      state.update(newState) // update the State
      // output the new average temperature for the batch that was updated!
      newState.get.find{ case (tS, avg) => batch(tS) == batch(t) }.map{
        case (ts, i) => (key, ts, i)
      }
    }
    else None // this element is from the future! ignore it.
  }
}
```

In this function, our `State` is a set of four cells containing averages for each of the batches. Here we are using `mapWithState` in the variant that takes the current batch time as an argument. We use the `batch` function in order to make batch comparisons sensible, in that if `t1`, `t2` are within the same batch, then we expect `batch(t1) == batch(t2)`.

We start by examining our new value and its event time. If that event time's batch is beyond the current batch time, then there is an error in our wall clock or the event time. For this example we return `None`, but we could log an error as well. If the event is in the past, we have to find which batch it belongs to. For that we use Scala's partition function on the batches of each cell of our Circular State buffer, and separate the elements coming from before our element's batch from those coming from the same batch or after.

We then look at whether there was already an average initialized for the batch of our event, that we should find at the head of the latter list (thanks to our partition). If there is one, we add the new temperature to it, otherwise we make an average out of our single element. Finally, we take the batches from before the current element's time and add all the posterior batches to it in order.

The `CircularBuffer` natively ensures we retain only the latest elements if there are more than our threshold (4).

As a last step, we look up the updated average on the cell we updated with our new element (if there indeed was one … we could have updated a stale element), and we output the new average if so. As a consequence, the `mapWithState` stream we can create on an `RDD` of `(String, (Time, Float))` elements (with the sensor name as a key and the timestamped temperature as a value) updates the up-to-date averages for the last updates we received, from the very first batch.

Naturally, it uses a linear time in processing the content of our `Circular Buffer`, a consequence of the simplicity we wanted to reach through this example. Note however, how we are dealing with a structure that is ordered by timestamp, and how a different data structure, such as a skip list, would let us gain a lot in processing speed and let us make this scalable.

In sum, `mapWithState`, with its powerful state update semantics, it parcimonious timeout semantics, and the versatility brought by `snapshots()`, gives us a powerful tool to represent basic event-time processing in a few lines of Scala.

Dynamic Windows

One of the interesting functions of the `DStream` API is `reduceByWindow`. The intent of `reduceByWindow` is to refer to the opportunity of creating a reduction of a `DStream` that dynamically integrates a windowing function. The reason for this is that towards the end of building a complex pipeline, we often want to see indicators of our data that are inherently something that depends on various notions of time.

Indeed, we expect that a user will want to see, for example, the count of distinct users on our website during the last hour, during the last 15 minutes, and during the last day. Those three pieces of information can all be computed based on counts on a windowed `DStream` which we have already touched upon in Chapter 3, and yet they represent very diverse and significant pieces of information, that may each be vital features if you manage a retail website.

`reduceByWindow`

The API of the `reduceByWindow` function is straightforward. It is a function of a `DStream` — and a function of a `PairDStream` in the case of `reduceByKeyAndWindow`. It takes a *reduction function* and as additional arguments a specification of the window of interest using the same notions as in Chapter 3: the window and slide intervals. The *window duration* is the second argument to the reduce function, and the *slide duration* is the third.

When we discussed windowed streams, we mentioned how the window duration needs to be a multiple of the batch duration. That is, to compute on a particular window, Spark will need to aggregate the values created during a defined number of completed batches, therefore, if you want a window of 15 minutes, it would be compatible with your stream if your batch interval is 3 minutes long, resulting in a 5-batch aggregation. If your batch interval is 4 minutes, a window of 15 minutes is not possible because 15 is not a multiple of 4. You would have to choose between a 12 or a 16-minute aggregation of your stream, corresponding to respectively three and four batches.

An additional fact, often forgotten, is that the window duration needs to be a multiple of the sliding duration.

Invertible Aggregations

The `reduceByWindow` (and `reduceByKeyAndWindow`) function contains an additional fourth argument as an optional parameter. That argument is called *the inverse reduce function*. It only matters if you happen to use an aggregation function that is invertible, meaning you can "take away" an element from the aggregate.

Formally, the inverse function `invReduceFunc` is such that for any accumulator y, invertible element x: `invReduceFunc(reduceFunc(x, y), x) = y`

Behind the scenes, this invertible notion will let Spark simplify the computation of our aggregates by letting Spark, on each new sliding window compute going over the elements of a couple of slide intervals, rather than of the full order of a window.

Let's say, for example, that we aggregate integer counts that we see with a batch interval of 1 minute, over a window of 15 minutes, sliding every minute. If you are not specifying an inverse reduce function, you have to add every new count over 15 minutes for summarizing the data that you have seen on your DStream. We sketch this process in Figure 5-9

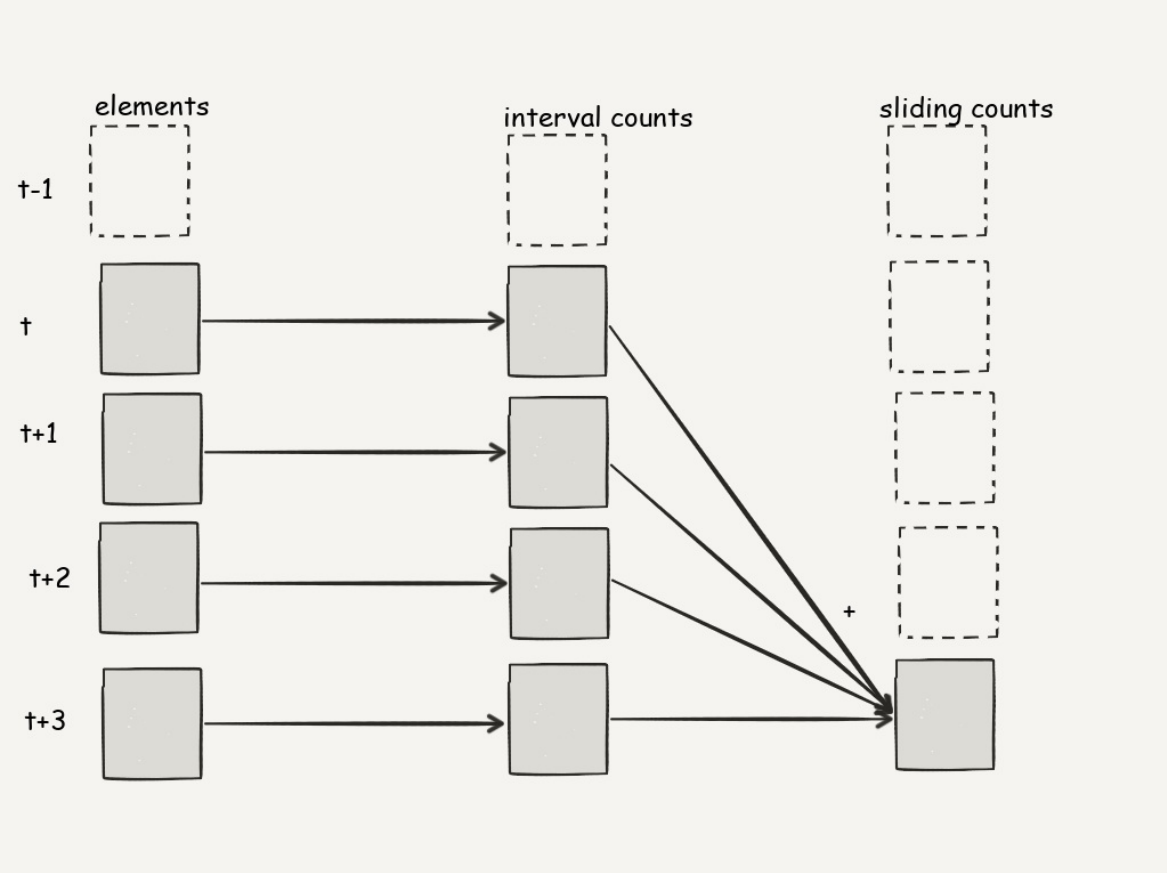


Figure 5-9. The aggregation of a reduceByWindow with a non-invertible function

This works, but if we see 10000 elements in a minute, we are summing 150000 integers, on every minute — and more importantly, we need to store those 15k integers in memory. Can we do better?

We could remember the count over the previous 15 minutes and consider that we have a new minute of data incoming. Taking the count over those previous 15 minutes, we could subtract the count for the oldest minute of that 15-minute aggregate, because a counting (summation) function is invertible. We subtract the count for the oldest minute, resulting in a 14-minute aggregate, to which we just have to add the newest 1 minute of data, giving us the last 15 minutes of data. We can see this process at work in Figure 5-10

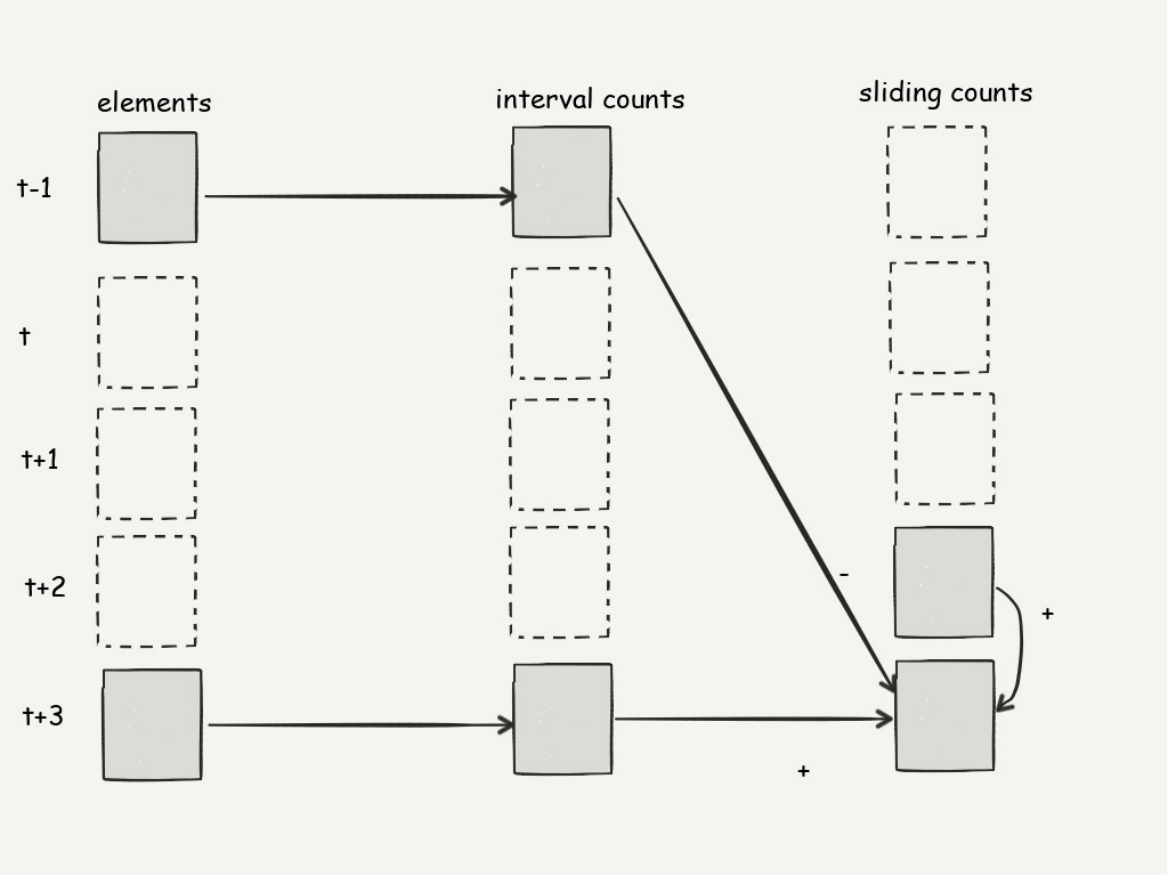


Figure 5-10. The aggregation of a reduceByWindow with an invertible function

The interesting part of this is that we do not have to store 15 thousand integers, but rather just the intermediate counts for every minute — i.e. 15 integers.

As you can see, having an invertible function for reduction can be tremendously useful in the case of windowed aggregations. It is also something that we can make the generation of various DStreams created by `reduceByWindow` inexpensive, giving us a good way to share information and aggregates on the values that we want to consider when analyzing our stream.

Caution

The inverse of the aggregation function is useless for tumbling windows, since every aggregation interval is completely disjoint from the others. It is therefore not worth bothering with this option if you do not use a slide interval!

Caching

Caching in Spark streaming is a feature that — when well manipulated — can significantly speed up the computation performed by your application. This seems to be counter-intuitive since the base RDDs representing the data stored in the input of a computation are actually replicated twice before any job runs on them.

However, along the lifetime of your application, there may be a very long pipeline that takes your computation from those base RDDs to some very refined and structured representations of the data that usually involves a key-values tuple. At the end of the computation performed by your application, you are probably looking at doing some distribution of the output of your computation into various outlets: data stores or databases such as Cassandra, for example. That distribution usually involves looking at the data computed during the previous batch interval and finding which portions of the output data should go where.

A typical use-case for that is to look at the keys in the RDD of the structured output data (the last DStream in your computation), to find exactly where to put the results of your computation outside of Spark, depending on these keys. Another use-case would be to look for only some specific elements on the RDD received during the last batch. Indeed, your RDD may actually by the output of a computation that depends not only on the last batch of data, but on many prior events received since the start of the application. The last step of your pipeline might summarize the state of your system, as we'll see in stateful computation in the next chapter. Looking at that RDD of output structured results, we might be looking for some elements that pass certain criteria, marking an exceptional use-case.

For example, think of anomaly detection. You might compute some metrics or features on values (users or elements that you are monitoring on a routine basis). Some of those features might reveal some problems or that some alerts need to be produced. To output those to an alerting system, you want to find elements that pass some criteria in the RDD of data that you're currently looking at. To do that, you going to *iterate* over the RDD of results. You may also, besides the alerting, want to store some of the rest of the state of your application to, for example to feed a data visualization or a dashboard, informing you on more general characteristics of the system that you are currently surveying.

The point of this thought exercise is to envision that computing on output DStream involves several operations for each and every RDD that composes the final result of your pipeline, despite it being very structured - and probably very reduced in size from the input data. For that purpose, using the cache to store that final RDD before several iterations occur on it is extremely useful. Indeed, when you do several iterations on a particular RDD , like in Spark, while the first cycle takes the same time as the end cached version, each subsequent iteration takes only a fraction of the time. The reason for that is that while the base data of spark streaming is cached in the system, intermediate steps needs to be recovered from that base data along the way, using the potentially very long pipeline defined by your application. Retrieving that elaborated data takes time, in every single iteration that is required to process the data as specified by your application.

```
dstream.foreachRDD{ (rdd)  =>
```

```
  rdd.cache()
  keys.foreach{ (key) =>
    rdd.filter(elem=> key(elem) == key).saveAsFooBar(...)
  }
  rdd.unpersist()
}
```

As a consequence, if your `DStream` or the corresponding `RDD`s is used multiple times, caching it significantly speeds up the process. However, it is very important to not overtax Spark's memory management and assume `RDD`s of a `DStream` will naturally fall out of cache when your `DStream` moves to the next `RDD` - after a batch interval. This means that if you are iterating over a `DStream` (say in a foreach, for example) caching the `RDD` before iterating can yield significant performance benefits. You would the write the iterative part of your application using foreach under-keys and then the filter. And remember, it is very important that at the end of the duration over every single `RDD` of your `DStream` you think of unpersisting the `RDD` to let it fall out of cache.

Otherwise Spark will have to do some relatively clever computation to try to understand which pieces of data it should retain. That particular computation might slow down the results of your application or limit the memory that would be accessible to it.

Measuring and Monitoring

Application monitoring is an integral part of any robust deployment. Monitoring provides insights on the application performance characteristics over time by collecting and processing metrics that quantify different aspects of the application's performance, such as responsiveness, resource usage, and task specific indicators.

Streaming applications have strict requirements regarding response times and throughput. In the case of distributed applications, like Spark, the number of variables that we need to account for during the application's lifetime are multiplied by the complexities of running on a cluster of machines. In the context of a cluster, we need to keep tabs on resource usage, like CPU, memory and secondary storage across different hosts, both from the perspective of each host, as well as a consolidated view of the running application.

To manage this complexity, we need a comprehensive and smart monitoring system. It needs to collect metrics from all the key *moving parts* that participate in the streaming application runtime and, at the same time, it needs to provide them in an understandable and consumable form.

In the case of Spark Streaming, next to the general indicators just discussed, we are particularly concerned with the relationship between the amount of data received, the batch interval chosen for our application and actual execution time of every micro-batch. As discussed in "Fundamentals of a DStream", the relation among these three parameters is key for a stable Spark Streaming job in the long run. To ensure that our job performs within stable boundaries, we need to make performance monitoring an integral part of the development and productization process.

Spark offers several monitoring interfaces that cater for the different stages of that process: - The SparkUI: A web interface that provides key information about the running job - The monitoring API: a set of APIs that can be consumed to obtain metrics programmatically - The metrics subsystem: a pluggable SPI (Service Provider Interface) that allows for tight integration of external monitoring tools into Spark - The internal event bus: A pub/sub Spark event subsystem that subscribers receive events about various different aspects of the application execution on the cluster

In this section we will explore these monitoring interfaces and how they apply to the different phases of the development cycle of a streaming application.

The Streaming UI

The SparkUI is a web application available on the Spark driver node, usually running on port 4040, unless there are other Spark processes running on the same node, in which case, the port used will increase (4041, 4042, …) until one free port is found. This port can also be configured using the configuration key `spark.ui.port`.

What we refer to as the *Streaming UI* is a tab in the Spark UI that becomes active whenever a `StreamingContext` is started as seen in Figure 5-11.

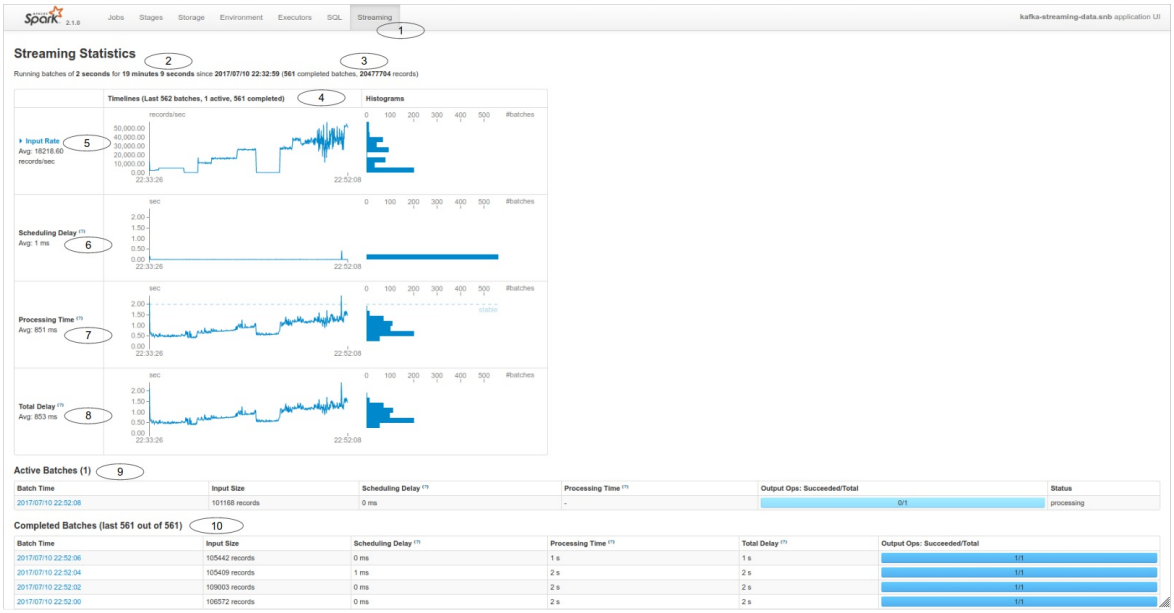


Figure 5-11. Streaming UI

Note

To reproduce this example locally, there are two notebooks provided on the github repository at *https://github.com/LearningSparkStreaming/notebooks* :

kafka-data-generator.snb

> this notebook is used to produce a configurable number of records per second that are sent to a local Kafka topic

kafka-streaming-data.snb

> this notebook consumes from that same topic, joins the data with the reference dataset used earlier in this chapter and writes the result to a local file.

By experimenting with the producer rate, we can observe the behavior of the Streaming UI and

experience how stable and overload situations look like. This is a good exercise to put into practice when moving a streaming application to production, as it helps understanding its performance characteristics and determining the load thresholds where the application performs correctly.

The Streaming UI is comprised of several visual elements that provide an at-a-glance view of the performance of a Spark Streaming job.

1. Streaming UI Tab: This is the Streaming Tab on the Spark UI. Clicking on it opens that Streaming UI

2. Time-based stats: The overall statistics line includes that `batch interval`, the time this application has been up and the start timestamp.

3. Summary of batches and records: Next to the timing information, we find the total number of batches completed and the grand total of records processed.

4. Performance Charts: The title in the table of charts reports what data is used in the charts displayed. The data represented in the charts is preserved in circular buffers. We only see the thousand (1000) most recent data points received.

5. The Input Rate Chart: a timeseries representation of the number of records received at each `batch interval` with a distribution histogram next to it.

6. The Scheduling Delay Chart: This charts reports the difference between the moment when the batch is scheduled and when it's processed.

7. The Processing Time Chart: A timeseries of the amount of time (duration) needed to process each batch.

8. Total Delay Chart: A timeseries of the sum of the scheduling delay and the processing time. It provides a view of the joint execution of Spark Streaming and the Spark core engine.

9. Active Batches: provides a list of the batches currently in the Spark Streaming queue. It shows the batch or batches currently executing as well as the batches of any potential backlog in case of an overload. Ideally, only batches in status *processing* are in this list. This list might appear empty if at the time of loading the Streaming UI, the current batch had finished processing and the next one is not yet due for scheduling.

10. Completed Batches: It shows the last most recent thousand batches processed with a link to the details of that batch.

Understanding Job Performance using the Streaming UI

The four charts that comprise the main screen of the Streaming UI provide a snapshot of the current and most recent performance of our streaming job. By default, the last 1000 processing intervals are presented in the UI, meaning that the period of time we are able to view is *intervals*

x `batch interval`, so for a 2-second interval, we see the metrics of roughly the last half hour (2x1000 = 2000 seconds or 33,33 minutes). The number of remembered intervals can be configured using the `spark.ui.retainedJobs` configuration parameter.

Input Rate Chart

The *Input Rate* chart, on top, gives a view of the input load that the application is sustaining. All charts share a common timeline. We can imagine a vertical line through all of them that would serve as a reference to correlate the the different metrics to the input load, as illustrated in Figure 5-12. The data points of chart lines are clickable and will link to the corresponding job detail line that appears underneath the charts. As we will explore later on, this navigation feature is very helpful to track back the origins of certain behavior that we can observe on the charts.

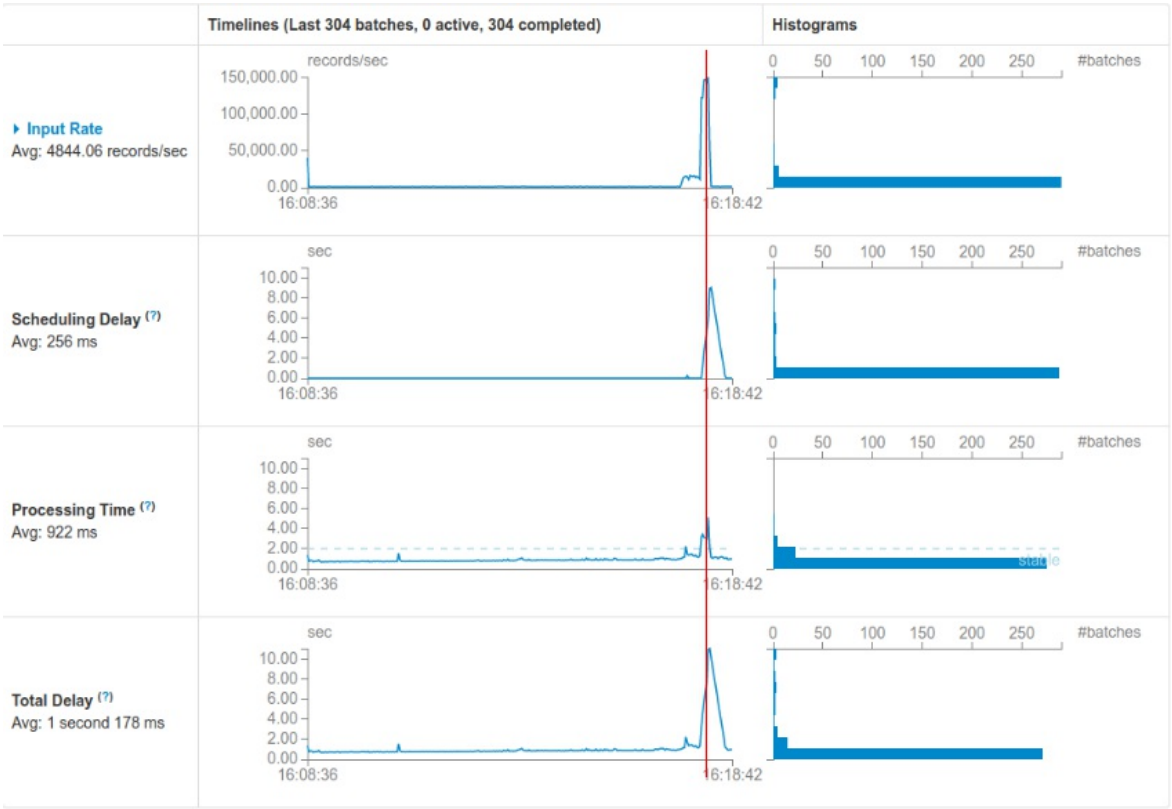


Figure 5-12. Streaming UI - Correlation of metrics

Scheduling Delay Chart

The next chart in the UI is the Scheduling Delay chart. This is a key health indicator: For an application that is running well within its time and resource constraints, this metric will be constantly flat on zero. Small periodic disturbances might point to regular supporting processes such as snapshots.

Window operations that create exceptional load will potentially also affect this metric. It's important to note that the delay will show on the batches that immediately follow a batch that

took longer than the batch interval to complete. These delays will not show a correlation with the input rate. Delays introduced by peaks in input data input will correlate with a high peak in the *Input Rate* with an offset. As we can see in Figure 5-13 the peak in the input rate chart happens earlier than the corresponding increase in the *Scheduling Delay*. As this chart represents a `delay`, we see the effects of the peak in input rate after the system has started to "digest" the data overload.

Figure 5-13. Streaming UI - Scheduling Delay

Processing Time Chart

This chart, shown in Figure 5-14 represents the time that the data processing part of the streaming job takes to execute. This execution takes place on the Spark cluster, so this is the main indicator on how the actual data crunching is performing on the (potentially) distributed environment. An important aspect of this chart is the high watermark line at the level corresponding to the `batch interval` time. Let's quickly recall that the `batch interval` is the time of each microbatch, and also the time we have to process the data that arrived at the previous interval. A processing time below this watermark is considered *stable*. An occasional peak above this line might be acceptable if the job has enough room to recover from it. A job that is constantly above this line will build a backlog using storage resources in memory and/or disk. If the available storage is exhausted, the job will eventually crash. This chart has usually a high correlation with the *Input Rate Chart* as it's common that the execution time of the job is related to the volume of data received on each batch interval.

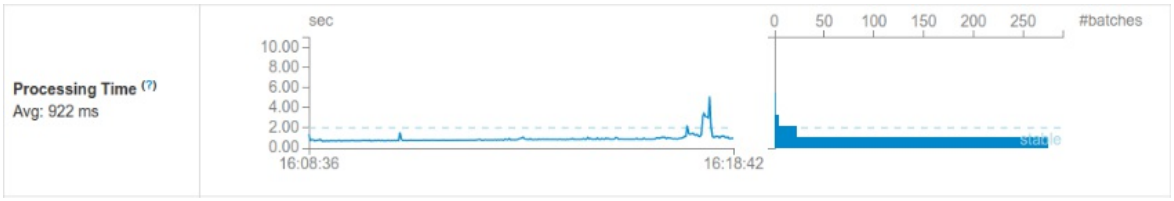


Figure 5-14. Streaming UI - Processing Time

Total Delay Chart

The total delay chart is a graphical representation of the end-to-end latency of the system. Total

delay comprises the length of time that Spark Streaming takes to collect, schedule and submit each microbatch for processing and the amount of time it takes for Spark to apply the job's logic and produce a result. This chart provides a holistic view of the system performance and reveals any delays that may happen during the execution of the job. As for the *Scheduling Delay Chart*, any sustained increase in the `total delay` metric is a reason of concern and might indicate high load or other conditions, such as increasing storage latency, that negatively affect the job's performance. On its

Batch Details

When we scroll below the charts that constitute the main screen of the Streaming UI, we find two tables: *Active Batches* and *Completed Batches*. These tables have columns that correspond to the charts we just studied: `Input Size`, `Scheduling Delay`, `Processing Time` and an `Output Operations` counter. In addition to these fields, *Completed Batches* also shows the `Total Delay` of the corresponding micro batch.

Note

Output Ops relates to the number of `output operations` registered to execute on the micro batch. This refers to the job's code structure and should not be confused with a parallelism indicator. As we recall, *Output Operations*, such as `print()` or `foreachRDD` are the operations that trigger the lazy execution of a DStream.

Active Batches contains information about the micro batches in the Spark Streaming scheduler queue. For a healthy job, this table contains at most one row: the batch currently executing. This entry indicates the number of records contained in the micro batch and any delay before the start of execution. *Processing Time* is unknown until the micro batch has been completed, so this metric is never present on the *Active Batches* table.

When more than one row is present in this table, it indicates that the job has a delay beyond the `batch interval` and new *_micro batches* queue awaiting execution, forming an execution backlog.

Completed Batches

Right after a batch execution completes, its corresponding entry in the *Active Batches* table transitions to the *Completed Batches* table. In this transition, the *Processing Time* field is filled with its execution time and the *Total Delay* is finally known and also included.

Each entry is identified by the timestamp, labeled as *batch time*. This label also provides a link to the details of the corresponding Spark job that provided the execution for this batch.

The link to the batch details deserves further exploration. As explained in earlier chapters, the Spark Streaming model is based on *micro* batches. The *Batch Detail* page provides insights in the execution of each batch, broken down into the different jobs that constitute the batch. We can appreciate its structure in Figure 5-15. A batch is defined as a sequence of output operations that are executed in the same order as they were defined in the application code. Each output operation contains one or more jobs. This page of summarizes this relationship, displays the duration per job and the parallelism level in the task overview.

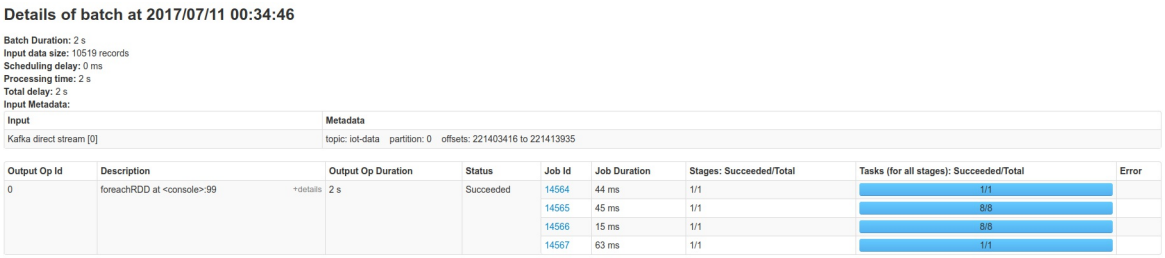


Figure 5-15. Streaming UI - Batch Details

The jobs are listed by `jobId` and provide a link to the job page in the Spark UI. These are *normal* Spark jobs executed by the core engine. By clicking through, we can explore the execution, in terms of stages, assigned executor and execution time statistics.

The Monitoring API

The Streaming UI is an excellent tool for active monitoring. That is, we sit behind the UI and interact with it while the streaming application executes. We can use this method during the initial stages of the development and deployment cycle in order to gain insights of the performance characteristics of our newly developed streaming application. As our application gains maturity and moves to a deployment environment where it runs 24/7, it's clear that we need automated means of monitoring the key performance indicators discussed.

The Spark core engine offers two APIs that allow the integration of existing monitoring tools: - The REST Monitoring API, that offers the UI metrics as JSON encoded REST endpoints - The Metrics subsystem, which allows the development and integration of monitoring plugins

The REST Monitoring API

The Monitoring REST API exposes the job's streaming metrics as a set of HTTP endpoints that deliver the data as JSON-formatted objects that can be consumed by external monitoring and alerting applications. The Monitoring API is available on the Spark driver node, on the same port as the Spark UI, and mounted at the `/api/v1` endpoint. The `/api/v1/applications/:app-id` resource offers information about the application id `app-id` provided. Note that for a running Spark Streaming context, there will be only one current application id. This id must be first queried by calling `/api/v1/applications` in order to construct the specific streaming calls. In the following URLs we will refer to this variable application id as `app-id`

Warning

The Monitoring API introduced support for Spark Streaming only from Spark version 2.2 onwards. For earlier versions of Spark, consider the metrics servlet, further explained in the next section

The monitoring resources corresponding to the running Spark Streaming context are available at: `/api/v1/applications/:app-id/streaming`

Resource	Meaning	Corresponding UI element
`/statistics`	A set of summarized metrics (see below for details)	Streaming UI charts
`/receivers`	A list of all receivers instantiated in this streaming job	Receivers summary on the *Input Rate Chart*
`/receivers/:stream-id`	Details of the receiver indexed by the provided `stream-id`.	Click-open receivers on the *Input Rate Chart*

`/batches`	A list of all the currently kept batches.	List of batches underneath the charts
`/batches/:batch-id/operations`	The output operations	Click-through a batch in the batch list
`/batches/:batch-id/operations/:output-op-id`	Detailed information of the corresponding output operation in the given batch.	Click-through in the operations details under the batch detail page

Note

In the case of a running streaming job, there will be only one application id. This id can be obtained from the `/api/v1/applications` call in order to use all subsequent calls requiring the `app-id`: `/api/v1/applications/:app-id`

From a monitoring perspective, we should pay some additional attention to the `statistics` object. It contains the key performance indicators that we need to monitor in order to ensure a healthy streaming job.

`/api/v1/applications/:app-id/statistics`

Key	Type	Description
startTime	String	timestamp encoded in ISO 8601 format
batchDuration	Number (Long)	batch interval duration in milliseconds
numReceivers,	Number (Long)	count of registered receivers
numActiveReceivers	Number (Long)	count of the currently active receivers
numInactiveReceivers	Number (Long)	count of the currently inactive receivers

numTotalCompletedBatches	Number (Long)	count of the completed batches since the start of this streaming job
numRetainedCompletedBatches	Number (Long)	count of the currently kept batches from which we still store information
numActiveBatches	Number (Long)	count of the batches in the execution queue of the streaming context
numProcessedRecords	Number (Long)	totalized sum of the records processed by the currently running job
numReceivedRecords	Number (Long)	totalized sum of the records received by the currently running job
avgInputRate	Number (Double)	arithmetic mean of the input rate over the last kept batches
avgSchedulingDelay	Number (Double)	arithmetic mean of the scheduling delay over the last kept batches
avgProcessingTime	Number (Double)	arithmetic mean of the processing time over the last kept batches
avgTotalDelay	Number (Double)	arithmetic mean of the total delay over the last kept batches

When integrating a monitoring tool with Spark Streaming we are particularly interested in `avgSchedulingDelay` and making sure that it does not grow over time. Given that the value provided by the API is an average over the last kept batches (1000 by default), setting up alarms on this metric should take into consideration small increasing changes.

The Metrics Subsystem

Available throught the Spark core engine, the Spark metrics subsystem offers a configurable metrics collection and reporting API with a pluggable *sink* interface. Spark comes with several such sinks, including HTTP, JMX, and CSV files. In addition to that, there's a Ganglia sink that needs additional compilation flags due to licensing restrictions.

By default, the HTTP sink is enabled. It's implemented by a servlet that registers an endpoint on the driver host and same port as the Spark UI. The metrics are accessible at the `/metrics/json` endpoint.

In contrast with the monitoring API we discussed in the previous section, the metrics endpoint delivers the most recent values of Spark and Spark Streaming processes. This gives us raw access to performance metrics that can drive remote performance monitoring applications and also enable accurate and timely alerting on abnormalies in a stream process.

The specific metrics for the streaming job are available under the key `<app-id>.driver.<application name>.StreamingMetrics.streaming`. These are:

- `lastCompletedBatch_processingDelay`
- `lastCompletedBatch_processingEndTime`
- `lastCompletedBatch_processingStartTime`
- `lastCompletedBatch_schedulingDelay`
- `lastCompletedBatch_submissionTime`
- `lastCompletedBatch_totalDelay`
- `lastReceivedBatch_processingEndTime`
- `lastReceivedBatch_processingStartTime`
- `lastReceivedBatch_records`
- `lastReceivedBatch_submissionTime`
- `receivers`
- `retainedCompletedBatches`
- `runningBatches`
- `totalCompletedBatches`
- `totalProcessedRecords`
- `totalReceivedRecords`
- `unprocessedBatches`
- `waitingBatches`

Although not readily offered, the `lastCompletedBatch_processingTime` can be obtained by simple arithmetic of the `lastCompletedBatch_processingEndTime - lastCompletedBatch_processingStartTime`

In this API, the key indicator to track for job stability is `lastCompletedBatch_processingDelay`, which we expect to be zero or close to zero and stable over time. A moving average of the last 5-10 values should remove the noise that small delays sometimes introduce and offer a metric we can rely on to trigger alarms or pager calls.

The Streaming Listener

All the metrics interfaces discussed in this chapter have a single source of truth in common: they are all consuming data from an internal Spark notification bus though dedicated `StreamingListener` implementations.

Spark uses several internal notification buses to deliver lifecycle events and metadata about the executing Spark jobs to subscribed clients. This interface is mostly used by internal Spark consumers that offer the data in some processed form. The Spark UI is the most significant example of such interaction.

For those cases where the existing high-level interfaces are not sufficient to fulfill our requirements, it's possible to develop custom listeners and register them to receive events. To create a custom Spark Streaming Listener, we extend the `org.apache.spark.streaming.scheduler.StreamingListener` trait. This trait has a no-op default implementation for all of its call-back methods, so that extending it only requires to override the desired call-backs with our custom metric processing logic.

Note that this internal Spark API is marked as `DeveloperApi`. Hence, its definitions, such as classes and interfaces are subject to change without public notice.

Note

We can explore a custom Streaming Listener implementation in the online resources. The notebook `kafka-streaming-with-listener` extends the Kafka notebook that we used before with a custom notebook listener that delivers all events to a `TableWidget` that can be directly displayed in the notebook.

The `StreamingListener` interface

The `StreamingListener` interface consists of a trait with several call-back methods. Each method is called by the notification process with an instance of a subclass of `StreamingListenerEvent` that contains the relevant information for the call-back method.

In the following overview we will highlight the most interesting parts of those data events.

onStreamingStarted

```
def onStreamingStarted(streamingStarted: StreamingListenerStreamingStarted): Unit
```

This method is called when the streaming job starts. The `StreamingListenerStreamingStarted` instance has a single field, `time`, that contains the timestamp in milliseconds when the streaming job started.

Receiver Events

All call-back methods that relate to the lifecycle of receivers share a common `ReceiverInfo` class that describes the receiver that is being reported. Each event-reporting class, will have a single `receiverInfo: ReceiverInfo` member. The information contained in the attributes of the `ReceiverInfo` class will depend on the relevant receiver information for the reported event.

onReceiverStarted

[source, scala] ---- def onReceiverStarted(receiverStarted: StreamingListenerReceiverStarted): Unit ---- This method is called when a receiver has been started. The `StreamingListenerReceiverStarted` instance contains a `ReceiverInfo` instance that describes the receiver started for this streaming job. Note that this is method is only relevant for the receiver-based streaming model.

onReceiverError

[source, scala] ---- def onReceiverError(receiverError: StreamingListenerReceiverError): Unit ---- This method is called when an existing receiver reports an error. Like for the `onReceiverStarted` call, the provided `StreamingListenerReceiverError` instance contains a `ReceiverInfo` object. Within this `Receiver Info` instance we will find detailed information about the error, such as the error message, and the timestamp of the occurence.

onReceiverStopped

[source, scala] ---- def onReceiverStopped(receiverStopped: StreamingListenerReceiverStopped): Unit ---- This is the counterpart of the `onReceiverStarted` event. It will fire when a receiver has been stopped.

Batch Events

These events relate to the lifecycle of batch processing, from submission to completion. Let's recall that each output operation registered on a `DStream` will lead to an independent job execution. Those jobs are grouped in batches, that are submitted together and in sequence to the Spark core engine for execution. This section of the listener interface will fire events following the lifecycle of submission and execution of batches.

```
As these events will fire following the processing of each micro-batch, their reporting rate is
at least as frequent as the `batch interval` of the related `StreamingContext`.
Following the same implementation pattern as the `receiver` call-back interface, all batch-
related events report a container class with a single `BatchInfo` member. Each `BatchInfo`
instance reported will contain the relevant information corresponding to the reporting call-back.
`BatchInfo` contains also a `Map` of the output operations registered in this batch, represented
by the `OutputOperationInfo` class. This class contains detailed information about the time,
duration and eventual errors for each individual output operation.
It could be used to split the total execution time of a batch into the time taken by the
different operations that lead to individual job execution on the Spark core engine.
```

onBatchSubmitted

[source, scala] ---- def onBatchSubmitted(batchSubmitted: StreamingListenerBatchSubmitted) ---- This method is called when a batch of jobs is

submitted to Spark for processing. The corresponding `BatchInfo` object, reported by the `StreamingListenerBatchSubmitted` contains the timestamp of the batch submission time. At this point, the optional values `processingStartTime` and `processingEndTime` are set to `None` as those values are obviously unknown at this stage of the batch processing cycle.

onBatchStarted

[source, scala] ---- def onBatchStarted(batchStarted: StreamingListenerBatchStarted): Unit ---- This method is called when the processing of a batch has stated. The `BatchInfo` object, embedded in the `StreamingListenerBatchStarted` instance provided will contain a populated `processingStartTime`.

[[onBatchCompleted]]

onBatchCompleted

[source, scala] ---- def onBatchCompleted(batchCompleted: StreamingListenerBatchCompleted): Unit ---- This method is called when the processing of a batch has been completed. The provided `BatchInfo` instance will be fully populated with the overall timing of the batch. The map of `OutputOperationInfo` will also contain the detailed timing information for the execution of each output operation.

Output Operation Events

This section of the `StreamingListener` callback interface provides information at the level of each job execution triggered by the submission of a batch. As there might be many output operations registered into the same batch, this interface might fire at a much higher rate than the `batch interval` of the `StreamingContext`. Any receiving client of the data provided by this interface should be dimensioned to receive such load of events.

```
The events reported by these methods contain a `OutputOperationInfo` instance that provides
further timing details about the _Output Operation_ being reported.
This `OutputOperationInfo` is the same data structure contained by the `outputOperationInfo` of
the `BatchInfo` object we just saw in the batch-related events.
In cases when we are only interested in the timing information of each job but we don't need to
be informed at real time about the execution lifecycle, it shoud be preferred to consult the
event provided by <<onBatchCompleted>>
```

onOutputOperationStarted

[source, scala] ---- def onOutputOperationStarted(outputOperationStarted: StreamingListenerOutputOperationStarted): Unit ---- This method is called with a job stated being processing. Individual jobs can be related back into their corresponding batch, by the `batchTime` attribute.

onOutputOperationCompleted

[source, scala] ---- def onOutputOperationCompleted(outputOperationCompleted: StreamingListenerOutputOperationCompleted): Unit ---- This method is called when the processing of an individual job as completed. Note that there are no call backs to notify individual job failures. The `OutputOperationInfo` instance contains an attribute `failureReason` which is an `Option[String]` In case of a job failure this option will be populated with `Some(error message)`.

`StreamingListener` Registration

Once we have developed our custom `StreamingListener`, we will need to register it in order to consume events. The streaming listener bus is hosted by the `StreamingContext` which exposes a registration call to add custom implementations of the `StreamingListener` trait.

Let's say we implemented a `LogReportingStreamingListener` that forwards all events to a logging framework. ??? shows how to register our custom listener in a `StreamingContext`

```
val streamingContext = new StreamingContext(sc, Seconds(10))
val customLogStreamingReporter = new LogReportingStreamingListener(...)
streamingContext.addStreamingListener(customLogStreamingReporter)
```

Conclusion

In this section we have learned about the different methods to observe and continuously monitor our developed Spark Streaming applications. Given that the performance characteristics of a job are a critical aspect to guarantee its stable deployment to a production environment, performance monitoring is an activity that should be performed from the early phases of the development cycle. The Streaming UI, a tab in the Spark UI is ideal for the interactive monitoring of our streaming application. It provides high-level charts the track the key performance indicators of a streaming job and provides links to detailed views where we can investigate the different execution times related to a batch interval to the level of the individual jobs that make up a batch. This is an active process that leads to actionable insights in the performance characteristics of the streaming job. Once the job is put to a continuous execution, it's clear that we cannot keep human monitoring in a 24/7 basis. Integration with existing monitoring and alerting tools can be achieved by consuming the REST-based monitoring interfaces that report data equivalent to that available in the Streaming UI. For the particular use cases that require a fully customizable solution that can get the finest-grained resolution of execution information at real-time, we can implement a custom `StreamingListener` and register it to the `StreamingContext` of our application.

This broad spectrum of monitoring alternatives ensure that our Spark Streaming application deployments can be deployed and coexist in a production infrastructure with a common monitoring and alerting subsystem, following the internal quality processes of the enterprise.

Bibliography

- [Armbrust2015] Michael Armbrust; Reynold S. Xin; Cheng Lian; Yin Huai; Davies Liu; Joseph K. Bradley; Xiangrui Meng; Tomer Kaftan; Michael J. Franklin; Ali Ghodsi; Matei Zaharia *Spark SQL: Relational Data Processing in Spark*, SIGMOD 2015, URL

- [Das2016] Das, T; Zhu, S. *Faster Stateful Stream Processing in Apache Spark Streaming* Databricks Engineering Blog. February 1, 2016. URL

- [Krishnamurthy2010] Sailesh Krishnamurthy. *Continuous Analytics Over Discontinuous Streams* Sigmod 2010 URL

- [Medasani2017] Guru Medasani; Jordan Hambleton. *Offset Management For Apache Kafka With Apache Spark Streaming.* Cloudera Engineering Blog. June 21,2017. URL

- [Hammer2015] Hammer Lab team, *Monitoring Spark with Graphite and Grafana.* February 27, 2015. Hammer Lab Blog URL

- [White2009] Tom White. *The Small Files Problem* Cloudera Engineering Blog., February 2nd, 2009. URL

- [Karau2017] Holden Karau, Rachel Warren. High Performance Spark, 1st Edition_ O'Reilly Media, June 2017. ISBN 9781491943205