

Advanced Programming Using the Spark Core API

Technology feeds on itself. Technology makes more technology possible.

Alvin Toffler, American writer and futurist

In This Chapter:

- Introduction to shared variables (broadcast variables and accumulators) in Spark
- Partitioning and repartitioning of Spark RDDs
- Storage options for RDDs
- Caching, distributed persistence, and checkpointing of RDDs

This chapter focuses on the additional programming tools at your disposal with the Spark API, including broadcast variables and accumulators as shared variables across different Workers in a Spark cluster. This chapter also dives into the important topics of Spark partitioning and RDD storage. You will learn about the various storage functions available for program optimization, durability, and process restart and recovery. You will also learn how to use external programs and scripts to process data in Spark RDDs in a Spark-managed lineage. The information in this chapter builds on the Spark API transformations you learned about in [Chapter 4, “Learning Spark Programming Basics,”](#) and gives you the

additional tools required to build efficient end-to-end Spark processing pipelines.

Shared Variables in Spark

The Spark API provides two mechanisms for creating and using shared variables in a Spark cluster (that is, variables that are accessible or mutable by different Workers in the Spark cluster). These mechanisms are called *broadcast variables* and *accumulators*, and we look at them both now.

Broadcast Variables

Broadcast variables are read-only variables set by the Spark Driver program that are made available to the Worker nodes in a Spark cluster, which means they are available to any tasks running on Executors on the Workers. Broadcast variables are read only after being set by the Driver. Broadcast variables are shared across Workers using an efficient peer-to-peer sharing protocol based on BitTorrent; this enables greater scalability than simply pushing variables directly to Executor processes from the Spark Driver. [Figure 5.1](#) demonstrates how broadcast variables are initialized, disseminated among Workers, and accessed by nodes within tasks.

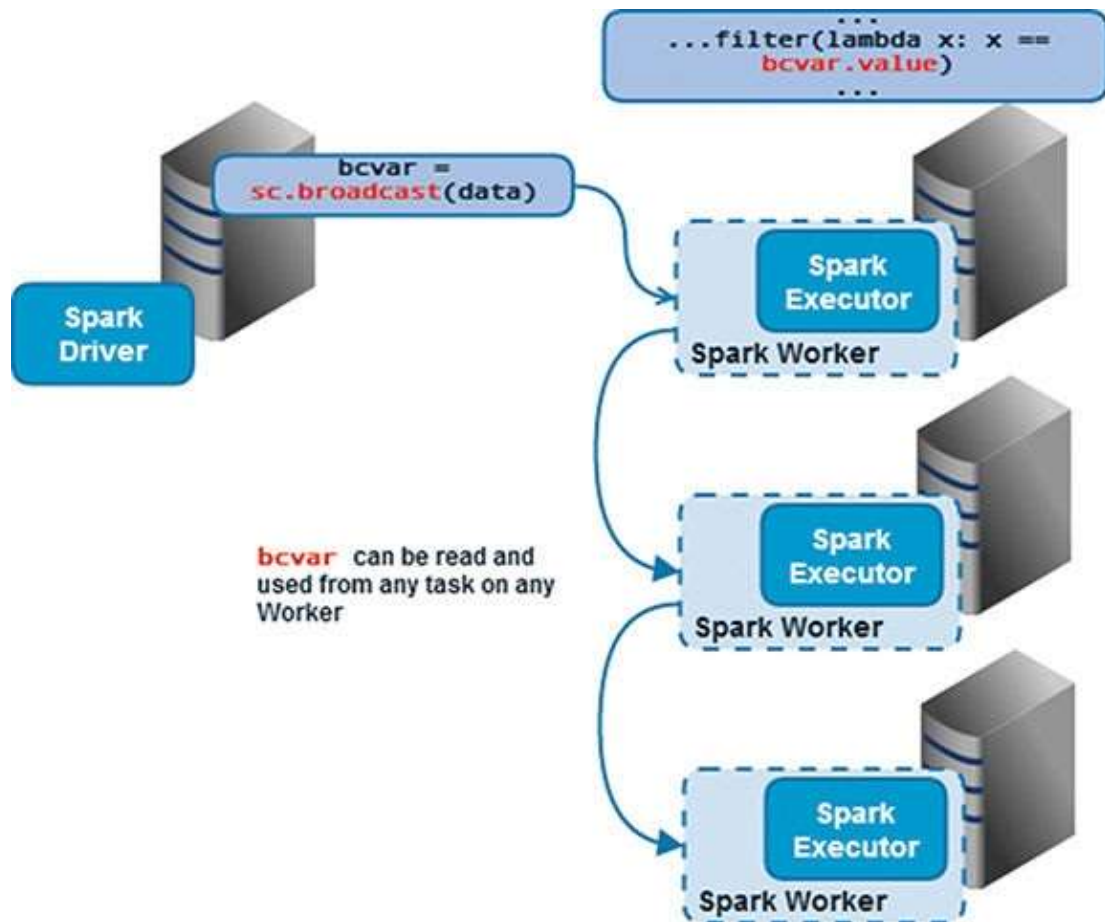


Figure 5.1 Spark broadcast variables.

The “Performance and Scalability of Broadcast in Spark” whitepaper at www.cs.berkeley.edu/~agearh/cs267.sp10/files/mosharaf-spark-bc-report-spring10.pdf documents the BitTorrent broadcast method as well as the other broadcast mechanisms considered for Spark; it’s worth a read.

A broadcast variable is created under a SparkContext and is then accessible as an object in the context of the Spark application. The following sections describe the syntax for creating and accessing broadcast variables.

broadcast ()

Syntax:

```
sc.broadcast(value)
```

The `broadcast ()` method creates an instance of a `Broadcast` object within the specific SparkContext. The value is the object to be serialized and

encapsulated in the `Broadcast` object; this could be any valid Python object. After they're created, these variables are available to all tasks running in the application. [Listing 5.1](#) shows an example of the `broadcast()` method.

Listing 5.1 Initializing a Broadcast Variable by Using the `broadcast()` Function

[Click here to view code image](#)

```
stations = sc.broadcast({'83': 'Mezes Park', '84': 'Ryland Park'})
stations
# returns <pyspark.broadcast.Broadcast object at 0x...>
```

You can also create broadcast variables from the contents of a file, either on a local, network, or distributed filesystem. Consider a file named `stations.csv`, which contains comma-delimited data, as follows:

[Click here to view code image](#)

```
83,Mezes Park,37.491269,-122.236234,15,Redwood City,2/20/2014
84,Ryland Park,37.342725,-121.895617,15,San Jose,4/9/2014
```

[Listing 5.2](#) shows an example of how to create a broadcast variable by using a file.

Listing 5.2 Creating a Broadcast Variable from a File

[Click here to view code image](#)

```
stationsfile = '/opt/spark/data/stations.csv'
stationsdata = dict(map(lambda x: (x[0],x[1]), \
                        map(lambda x: x.split(','), \
                            open(stationsfile))))
stations = sc.broadcast(stationsdata)
stations.value["83"]
# returns 'Mezes Park'
```

[Listing 5.2](#) shows how to create a broadcast variable from a CSV file (`stations.csv`) consisting of a dictionary of key/value pairs, including the station ID and the station name. You can now access this dictionary from within any `map()` or `filter()` RDD operations.

For initialized broadcast variable objects, a number of methods can be called within the SparkContext, as described in the following sections.

value()

Syntax:

```
Broadcast.value()
```

[Listing 5.2](#) demonstrates the use of the `value()` function to return the value from the broadcast variable; in that example, the value is a `dict` (or `map`) that can access values from the map by their keys. The `value()` function can be used within a `lambda` function in a `map()` or `filter()` operation in a Spark program.

unpersist()

Syntax:

```
Broadcast.unpersist(blocking=False)
```

The `unpersist()` method of the `Broadcast` object is used to remove a broadcast variable from memory on all Workers in the cluster where it was present.

The Boolean `blocking` argument specifies whether this operation should block until the variable unpersists from all nodes or whether this can be an asynchronous, non-blocking operation. If you require memory to be released immediately, set this argument to `True`.

An example of the `unpersist()` method is provided in [Listing 5.3](#).

Listing 5.3 The `unpersist()` Method

[Click here to view code image](#)

```
stations = sc.broadcast({'83':'Mezes Park', '84':'Ryland Park'})
stations.value['84']
# returns 'Ryland Park'
stations.unpersist()
# broadcast variable will eventually get evicted from cache
```

There are also several Spark configuration options related to broadcast variables, as described in [Table 5.1](#). Typically, you can leave these at their default settings, but it is useful to know about them.

Table 5.1 Spark Configuration Options Related to Broadcast Variables

Configuration Option	Description
<code>spark.broadcast.compress</code>	Specifies whether to compress broadcast variables before transferring them to Workers. Defaults to <code>True</code> (recommended).
<code>spark.broadcast.factory</code>	Specifies which broadcast implementation to use. Defaults to <code>TorrentBroadcastFactory</code> .
<code>spark.broadcast.blockSize</code>	Specifies the size of each block of the broadcast variable (used by <code>TorrentBroadcastFactory</code>). Defaults to <code>4MB</code> .
<code>spark.broadcast.port</code>	Specifies the port for the Driver's HTTP broadcast server to listen on. Defaults to <code>random</code> .

What are the advantages of broadcast variables? Why are they useful or even required in some cases? As discussed in [Chapter 4](#), it is often necessary to combine two datasets to produce a resultant dataset. This can be achieved in multiple ways.

Consider two associated datasets: `stations` (a relatively small lookup data set) and `status` (a large eventful data source). These two datasets can join on a natural key, `station_id`. You could join the two datasets as RDDs directly in your Spark application, as shown in [Listing 5.4](#).

Listing 5.4 Joining Lookup Data by Using an `RDD join()`

[Click here to view code image](#)

```
status = sc.textFile('file:///opt/spark/data/bike-share/status') \
    .map(lambda x: x.split(',')) \
```

```

        .keyBy(lambda x: x[0])
stations = sc.textFile('file:///opt/spark/data/bike-share/stations') \
        .map(lambda x: x.split(',')) \
        .keyBy(lambda x: x[0])
status.join(stations) \
        .map(lambda x: (x[1][0][3],x[1][1][1],x[1][0][1],x[1][0][2])) \
        .count()
# returns 907200

```

This most likely would result in an expensive shuffle operation.

It would be better to set a table variable in the Driver for `stations`; this will then be available as a runtime variable for Spark tasks implementing `map()` operations, eliminating the requirement for a shuffle (see [Listing 5.5](#)).

Listing 5.5 Joining Lookup Data by Using a Driver Variable

[Click here to view code image](#)

```

stationsfile = '/opt/spark/data/bike-share/stations/stations.csv'
sdata = dict(map(lambda x: (x[0],x[1]), \
        map(lambda x: x.split(','), \
        open(stationsfile))))
status = sc.textFile('file:///opt/spark/data/bike-share/status') \
        .map(lambda x: x.split(',')) \
        .keyBy(lambda x: x[0])
status.map(lambda x: (x[1][3], sdata[x[0]], x[1][1], x[1][2])) \
        .count()
# returns 907200

```

This works and is better in most cases than the first option; however, it lacks scalability. In this case, the variable is part of a closure within the referencing function. This may result in unnecessary and less efficient transfer and duplication of data on the Worker nodes.

The best option would be to initialize a broadcast variable for the smaller `stations` table. This involves using peer-to-peer replication to make the variable available to all Workers, and the single copy is usable by all tasks on all Executors belonging to an application running on the Worker. Then you can use the variable in your `map()` operations, much as in the second option. An

example of this is provided in [Listing 5.6](#).

Listing 5.6 Joining Lookup Data by Using a Broadcast Variable

[Click here to view code image](#)

```
stationsfile = '/opt/spark/data/bike-share/stations/stations.csv'
sdata = dict(map(lambda x: (x[0],x[1]), \
                      map(lambda x: x.split(','), \
                          open(stationsfile))))
stations = sc.broadcast(sdata) status =
sc.textFile('file:///opt/spark/data/bike-share/status') \
    .map(lambda x: x.split(',')) \
    .keyBy(lambda x: x[0])
status.map(lambda x: (x[1][3], stations.value[x[0]], x[1][1], x[1][2])) \
    .count()
# returns 907200
```

As you can see in the scenario just described, using broadcast variables is an efficient method for sharing data at runtime between processes running on different nodes of a Spark cluster. Consider the following points about broadcast variables:

- Using them eliminates the need for a shuffle operation.
- They use an efficient and scalable peer-to-peer distribution mechanism.
- They replicate data once per Worker, as opposed to replicating once per task—which is important as there may be thousands of tasks in a Spark application.
- Many tasks can reuse them multiple times.
- They are serialized objects, so they are efficiently read.

Accumulators

Another type of shared variable in Spark is an *accumulator*. Unlike with broadcast variables, you can update accumulators; more specifically, they are numeric values that be incremented.

Think of accumulators as counters that you can use in a number of ways in Spark programming. Accumulators allow you to aggregate multiple values while your program is running.

Accumulators are set by the Driver and updated by Executors running tasks in the respective SparkContext. The Driver can then read back the final value from the accumulator, typically at the end of the program.

Accumulators update only once per successfully completed task in a Spark application. Worker nodes send the updates to the accumulator back to the Driver, which is the only process that can read the accumulator value.

Accumulators can use integer or float values. [Listing 5.7](#) and [Figure 5.2](#) demonstrate how accumulators are created, updated, and read.

Listing 5.7 Creating and Accessing Accumulators

[Click here to view code image](#)

```
acc = sc.accumulator(0)
def addone(x):
    global acc
    acc += 1
    return x + 1
myrdd=sc.parallelize([1,2,3,4,5])
myrdd.map(lambda x: addone(x)).collect()
# returns [2, 3, 4, 5, 6]
print("records processed: " + str(acc.value))
# returns "records processed: 5"
```

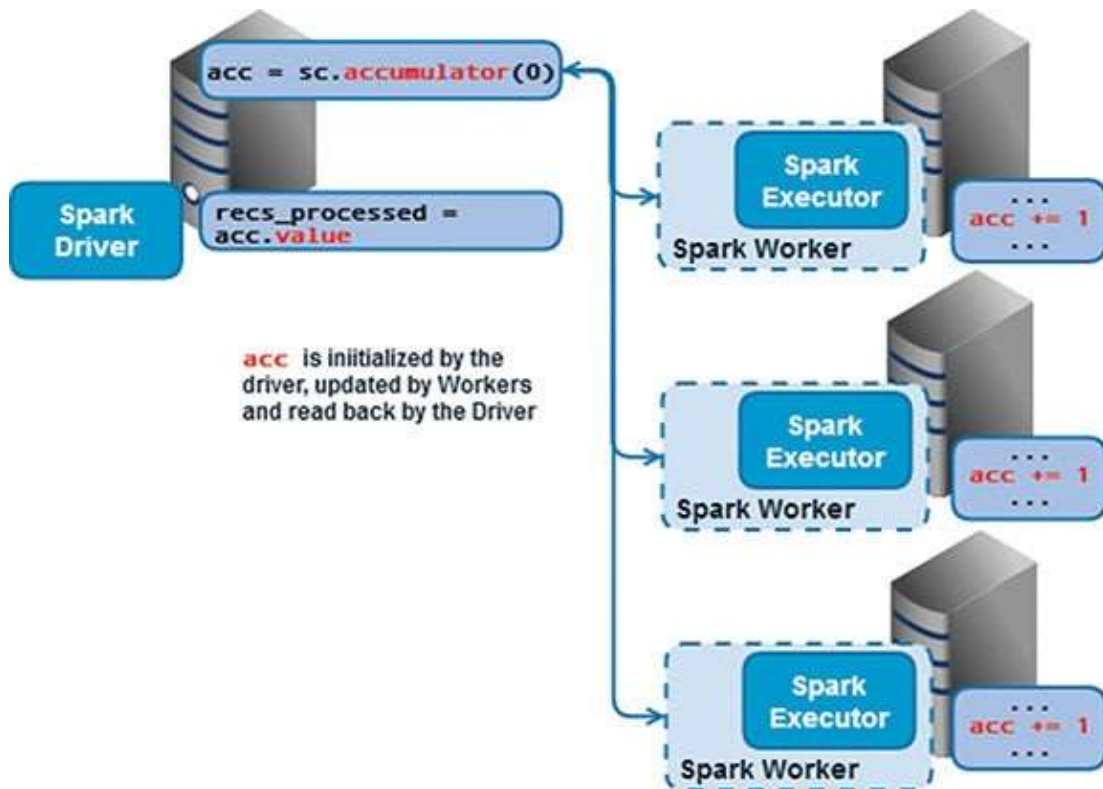


Figure 5.2 Accumulators.

From a programming standpoint, accumulators are very straightforward. The functions related to accumulators in Spark programming, used in [Listing 5.7](#), are documented in the following sections.

accumulator()

Syntax:

```
sc.accumulator(value, accum_param=None)
```

The `accumulator()` method creates an instance of an `Accumulator` object within the specific `SparkContext` and initializes with a given initial value specified by the `value` argument. The `accum_param` argument is used to define custom accumulators, which we discuss next.

value()

Syntax:

```
Accumulator.value()
```

The `value()` method retrieves the accumulator's value. This method can be used only in the Driver program.

Custom Accumulators

Standard accumulators created in a `SparkContext` support primitive numeric datatypes, including `int` and `float`. Custom accumulators can perform aggregate operations on variables of types other than scalar numeric values. Custom accumulators are created using the `AccumulatorParam` helper object. The only requirement is that the operations performed must be associative and commutative, meaning the order and sequence of operation are irrelevant.

A common use of custom accumulators is to accumulate vectors as either lists or dictionaries. Conceptually, the same principle applies in a non-mathematical context to non-numeric operations—for instance, when you create a custom accumulator to concatenate string values.

To use custom accumulators, you need to define a custom class that extends the `AccumulatorParam` class. The class needs to include two specific member functions: `addInPlace()`, used to operate against two objects of the custom accumulators datatype and to return a new value, and `zero()`, which provides a “zero value” for the type—for instance, an empty map for a `map` type.

[Listing 5.8](#) shows an example of a custom accumulator used to sum vectors as a Python dictionary.

Listing 5.8 Custom Accumulators

[Click here to view code image](#)

```
from pyspark import AccumulatorParam
class VectorAccumulatorParam(AccumulatorParam):
    def zero(self, value):
        dict1={}
        for i in range(0,len(value)):
            dict1[i]=0
        return dict1
    def addInPlace(self, val1, val2):
        for i in val1.keys():
            val1[i] += val2[i]
```

```

        return val1
rdd1=sc.parallelize([[0: 0.3, 1: 0.8, 2: 0.4}, {0: 0.2, 1: 0.4, 2:
0.2}])
vector_acc = sc.accumulator({0: 0, 1: 0, 2: 0},
VectorAccumulatorParam())
def mapping_fn(x):
    global vector_acc
    vector_acc += x
# do some other rdd processing...
rdd1.foreach(mapping_fn)
print vector_acc.value
# returns {0: 0.5, 1: 1.2000000000000002, 2: 0.6000000000000001}

```

Uses for Accumulators

Accumulators are typically used for operational purposes, such as for counting the number of records processed or tracking the number of malformed records. You can also use them for notional counts of different types of records; an example would be a count of different response codes discovered during the mapping of log events.

In some cases, as shown in the following exercise, you can use accumulators for processing within an application.

Potential for Erroneous Results in Accumulators

If accumulators are used in transformations, such as when calling accumulators to perform add-in-place operations to calculate results inside a `map()` operation, the results may be erroneous. Stage retries or speculative execution can cause accumulator values to be counted more than once, resulting in incorrect counts. If absolute correctness is required, you should use accumulators only within actions computed by the Spark Driver, such as the `foreach()` action. If you are looking only for notional or indicative counts on very large datasets, then it is okay to update accumulators in transformations. This behavior may change in future releases of Spark; for now, this is a caveat emptor.

Exercise: Using Broadcast Variables and

Accumulators

This exercise shows how to calculate the average word length from the words in the works of Shakespeare text, downloaded in the section “MapReduce and Word Count Exercise” in Chapter 4. In this exercise, you will remove known stop words (“a,” “and,” “or,” “the”) by using a broadcast variable and then compute average word length by using accumulators. Follow these steps:

1. Open a PySpark shell using whatever mode is available to you (Local, YARN Client, or Standalone). Use a single-instance Spark deployment in Local mode for this example:

```
$ pyspark --master local
```
2. Import a list of English stop words (`stop-word-list.csv`) from the book’s S3 bucket using the built-in `urllib2` Python module (Python3) and then convert the data into a Python list by using the `split()` function:

[Click here to view code image](#)

```
import urllib.request
stopwordsurl =
"https://s3.amazonaws.com/sparkusingpython/stopwords/stop-word-
list.csv"
req = urllib.request.Request(stopwordsurl)
with urllib.request.urlopen(req) as response:
    stopwordsdata = response.read().decode("utf-8")
stopwordslist = stopwordsdata.split(",")
```

3. Create a broadcast variable for the `stopwordslist` object:

```
stopwords = sc.broadcast(stopwordslist)
```
4. Initialize accumulators for the cumulative word count and cumulative total length of all words:

[Click here to view code image](#)

```
word_count = sc.accumulator(0)
total_len = sc.accumulator(0.0)
```

Note that you have created `total_len` as a float because you will use it as the numerator in a division operation later, when you want to keep the precision in the result.

5. Create a function to accumulate word count and the total word length:

[Click here to view code image](#)

```
def add_values(word, word_count, total_len):
```

```
word_count += 1
total_len += len(word)
```

6. Create an RDD by loading the Shakespeare text, tokenizing and normalizing all text in the document, and filtering stop words by using the `stopwords` broadcast variable:

[Click here to view code image](#)

```
words = sc.textFile('file:///opt/spark/data/shakespeare.txt') \
    .flatMap(lambda line: line.split()) \
    .map(lambda x: x.lower()) \
    .filter(lambda x: x not in stopwords.value)
```

7. Use the `foreach` action to iterate through the resultant RDD and call your `add_values` function:

[Click here to view code image](#)

```
words.foreach(lambda x: add_values(x, word_count, total_len))
```

8. Calculate the average word length from your accumulator-shared variables and display the final result:

[Click here to view code image](#)

```
avgwordlen = total_len.value/word_count.value
print("Total Number of Words: " + str(word_count.value))
print("Average Word Length: " + str(avgwordlen))
```

This should return 966958 for the total number of words and 3.608722405730135 for the average word length.

7. Now put all the code for this exercise in a file named `average_word_length.py` and execute the program using `spark-submit`. Recall that you need to add the following to the beginning of your script:

[Click here to view code image](#)

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName('Broadcast Variables and Accumulators')
sc = SparkContext(conf=conf)
```

The complete source code for this exercise can be found in the `average-word-length` folder at https://github.com/sparktraining/spark_using_python.

Partitioning Data in Spark

Partitioning is integral to Spark processing in most cases. Effective partitioning can improve application performance by orders of magnitude. Conversely, inefficient partitioning can result in programs failing to complete, producing problems such as Executor-out-of-memory errors for excessively large partitions.

The following sections recap what you already know about RDD partitions and then discuss API methods that can affect partitioning behavior or that can access data within partitions more effectively.

Partitioning Overview

The number of partitions to create from an RDD transformation is usually configurable. There are some default behaviors you should be aware of, however.

Spark creates an RDD partition per block when using HDFS (typically the size of a block in HDFS is 128MB), as in this example:

[Click here to view code image](#)

```
myrdd = sc.textFile("hdfs:///dir/filescontaining10blocks")
myrdd.getNumPartitions()
# returns 10
```

Shuffle operations such as the ByKey operations—`groupByKey()`, `reduceByKey()`—and other operations in which the `numPartitions` value is not supplied as an argument to the method will result in a number of partitions equal to the `spark.default.parallelism` configuration value. Here is an example:

[Click here to view code image](#)

```
# with spark.default.parallelism=4
myrdd = sc.textFile("hdfs:///dir/filescontaining10blocks")
mynewrdd = myrdd.flatMap(lambda x: x.split()) \
    .map(lambda x:(x,1)) \
    .reduceByKey(lambda x, y: x + y)
mynewrdd.getNumPartitions()
# returns 4
```

If the `spark.default.parallelism` configuration parameter is not set,

the number of partitions that a transformation creates will equal the highest number of partitions defined by an upstream RDD in the current RDDs lineage. Here is an example:

[Click here to view code image](#)

```
# with spark.default.parallelism not set
myrdd = sc.textFile("hdfs:///dir/filescontaining10blocks")
mynewrdd = myrdd.flatMap(lambda x: x.split()) \
    .map(lambda x:(x,1)) \
    .reduceByKey(lambda x, y: x + y)
mynewrdd.getNumPartitions()
# returns 10
```

The default partitioner class that Spark uses is `HashPartitioner`; it hashes all keys with a deterministic hashing function and then uses the key hash to create approximately equal buckets. The aim is to disperse data evenly across the specified number of partitions based on the key.

Some Spark transformations, such as the `filter()` transformation, do not allow you to change the partitioning behavior of the resultant RDD. For example, if you applied a `filter()` function to an RDD with four partitions, it would result in a new, filtered RDD with four partitions, using the same partitioning scheme as the original RDD (that is, hash partitioned).

Although the default behavior is normally acceptable, in some circumstances it can lead to inefficiencies. Fortunately, Spark provides several mechanisms to address these potential issues.

Controlling Partitions

How many partitions should an RDD have? There are issues at both ends of the spectrum when it comes to answering this question. Having too few, very large partitions can result in out-of-memory issues on Executors. Having too many small partitions isn't optimal because too many tasks spawn for trivial input sets. A mix of large and small partitions can result in speculative execution occurring needlessly, if this is enabled. *Speculative execution* is a mechanism that a cluster scheduler uses to preempt slow-running processes; if the root cause of the slowness of one or more processes in a Spark application is inefficient partitioning, then speculative execution won't help.

Consider the scenario in [Figure 5.3](#).

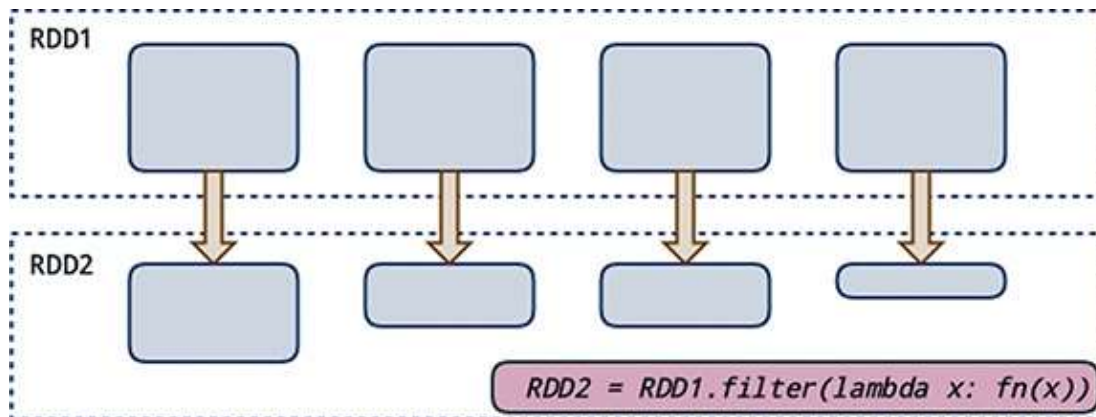


Figure 5.3 Skewed partitions.

The `filter()` operation creates a new partition for every input partition on a one-to-one basis, with only records that meet the filter condition. This can result in some partitions having significantly less data than others, which can lead to bad outcomes, such as data skewing, the potential for speculative execution, and suboptimal performance in subsequent stages.

In such cases, you can use one of the repartitioning methods in the Spark API; these include `partitionBy()`, `coalesce()`, `repartition()`, and `repartitionAndSortWithinPartitions()`, all of which are explained shortly.

These functions take a partitioned input RDD and create a new RDD with n partitions, where n could be more or fewer than the original number of partitions. Take the example from [Figure 5.3](#). In [Figure 5.4](#), a `repartition()` function is applied to consolidate the four unevenly distributed partitions to two “evenly” distributed partitions, using the default `HashPartitioner`.

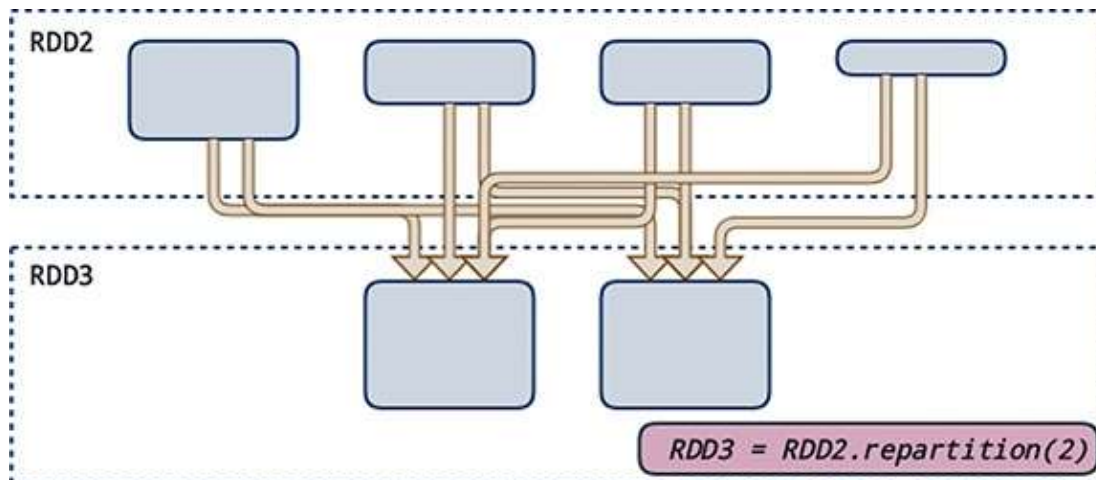


Figure 5.4 The `repartition()` function.

Determining the Optimal Number of Partitions

Often, determining the optimal number of partitions involves experimenting with different values until you find the point of diminishing returns (the point at which each additional partition starts to degrade performance). As a starting point, a simple axiom is to use two times the number of cores in your cluster—that is, two times the aggregate number of cores across all Worker nodes. In addition, as a dataset changes, it is advisable to revisit the number of partitions used.

Repartitioning Functions

The main functions used to repartition RDDs are documented in the following sections.

`partitionBy()`

Syntax:

[Click here to view code image](#)

```
RDD.partitionBy(numPartitions, partitionFunc=portable_hash)
```

The `partitionBy()` method returns a new RDD containing the same data as the input RDD but with the number of partitions specified by the `numPartitions` argument, using the `portable_hash` function (`HashPartitioner`) by default. An example of `partitionBy()` is shown

in [Listing 5.9](#).

Listing 5.9 The `partitionBy()` Function

[Click here to view code image](#)

```
kvrdd = sc.parallelize([(1, 'A'), (2, 'B'), (3, 'C'), (4, 'D')], 4)
kvrdd.getNumPartitions()
# returns 4
kvrdd.partitionBy(2).getNumPartitions()
# returns 2
```

The `partitionBy()` function is also called by other functions, such as `sortByKey()`, which calls `partitionBy()` using `rangePartitioner` instead of the `portable_hash` function. The `rangePartitioner` partitions records sorted by their key into equally sized ranges; this is an alternative to hash partitioning.

The `partitionBy()` transformation is also a useful function for implementing a custom partitioner, such as a function to bucket web logs into monthly partitions. A custom partition function must take a key as input and return a number between zero and the `numPartitions` specified in the `partitionBy()` function and then use that return value to direct elements to their target partition.

`repartition()`

Syntax:

```
RDD.repartition(numPartitions)
```

The `repartition()` method returns a new RDD with the same data as the input RDD, consisting of exactly the number of partitions specified by the `numPartitions` argument. The `repartition()` method may require a shuffle, and, unlike `partitionBy()`, it has no option to change the partitioner or partitioning function. The `repartition()` method also lets you create more partitions in the target RDD than existed in the input RDD. [Listing 5.10](#) shows an example of the `repartition()` function.

Listing 5.10 The `repartition()` Function

[Click here to view code image](#)

```
kvrdd = sc.parallelize([(1, 'A'), (2, 'B'), (3, 'C'), (4, 'D')], 4)
kvrdd.repartition(2).getNumPartitions()
# returns 2
```

coalesce()

Syntax:

```
RDD.coalesce(numPartitions, shuffle=False)
```

The `coalesce()` method returns a new RDD consisting of the number of partitions specified by the `numPartitions` argument. The `coalesce()` method also allows you to control whether the repartitioning triggers a shuffle, using the Boolean `shuffle` argument. The operation `coalesce(n, shuffle=True)` is functionally equivalent to `repartition(n)`.

The `coalesce()` method is an optimized implementation of `repartition()`. Unlike `repartition()`, however, `coalesce()` gives you more control over the shuffle behavior and, in many cases, allows you to avoid data movement. Also, unlike `repartition()`, `coalesce()` only lets you decrease the number of target partitions from the number of partitions in your input RDD.

[Listing 5.11](#) demonstrates the use of the `coalesce()` function with the `shuffle` argument set to `False`.

Listing 5.11 The `coalesce()` Function

[Click here to view code image](#)

```
kvrdd = sc.parallelize([(1, 'A'), (2, 'B'), (3, 'C'), (4, 'D')], 4)
kvrdd.coalesce(2, shuffle=False).getNumPartitions()
# returns 2
```

repartitionAndSortWithinPartitions()

Syntax:

[Click here to view code image](#)

```
RDD.repartitionAndSortWithinPartitions(numPartitions=None,  
partitionFunc=portable_hash,  
ascending=True,  
keyfunc=<lambda function>)
```

The `repartitionAndSortWithinPartitions()` method repartitions the input RDD into the number of partitions directed by the `numPartitions` argument and is partitioned according to the function specified by the `partitionFunc` argument. Within each resulting partition, records are sorted by their keys, as defined by the `keyfunc` argument, in the sort order determined by the `ascending` argument.

The `repartitionAndSortWithinPartitions()` method is commonly used to implement a *secondary sort*. The sorting capability for key/value pair RDDs is normally based on an arbitrary key hash or a range; this becomes more challenging with key/value pairs with composite keys, such as $((k1, k2), v)$. If you wanted to sort on $k1$ first and then within a partition sort the $k2$ values for each $k1$, this would involve a secondary sort.

[Listing 5.12](#) demonstrates the use of the `repartitionAndSortWithinPartitions()` method to perform a secondary sort on a key/value pair RDD with a composite key. The first part of the key is grouped in separate partitions; the second part of the key is then sorted in descending order. Note the use of the `glom()` function to inspect partitions; we discuss this function shortly.

Listing 5.12 The `repartitionAndSortWithinPartitions()` Function

[Click here to view code image](#)

```
kvrdd = sc.parallelize([(1,99), 'A'), ((1,101), 'B'), ((2,99), 'C'),  
((2,101), 'D')], 2)  
kvrdd.glom().collect()  
# returns:  
# [(((1, 99), 'A'), ((1, 101), 'B')), (((2, 99), 'C'), ((2, 101), 'D'))]  
kvrdd2 = kvrdd.repartitionAndSortWithinPartitions( \  
numPartitions=2,  
ascending=False,  
keyfunc=lambda x: x[1])  
kvrdd2.glom().collect()
```

```
# returns:  
# [((1, 101), 'B'), ((1, 99), 'A')], [((2, 101), 'D'), ((2, 99), 'C')]]
```

Partition-Specific or Partition-Aware API Methods

Many of Spark's methods are designed to interact with partitions as atomic units; these include both actions and transformations. Some of the methods are described in the following sections.

foreachPartition()

Syntax:

```
RDD.foreachPartition(func)
```

The `foreachPartition()` method is an action similar to the `foreach()` action, applying a function specified by the `func` argument to each partition of an RDD. [Listing 5.13](#) shows an example of the `foreachPartition()` method.

Listing 5.13 The `foreachPartition()` Action

[Click here to view code image](#)

```
def f(x):  
    for rec in x:  
        print(rec)  
kvrdd = sc.parallelize([((1, 99), 'A'), ((1, 101), 'B'), ((2, 99), 'C'),  
    ((2, 101), 'D')], 2)  
kvrdd.foreachPartition(f)  
# returns:  
# ((1, 99), 'A')  
# ((1, 101), 'B')  
# ((2, 99), 'C')  
# ((2, 101), 'D')
```

Keep in mind that `foreachPartition()` is an action, not a transformation, and it therefore triggers evaluation of the input RDD and its entire lineage. Furthermore, the function results in data going to the Driver, so be mindful of the

final RDD data volumes when running this function.

glom()

Syntax:

```
RDD.glom()
```

The `glom()` method returns an RDD created by coalescing all the elements within each partition into a list. This is useful for inspecting RDD partitions as collated lists; you saw an example of this function in [Listing 5.12](#).

lookup()

Syntax:

```
RDD.lookup(key)
```

The `lookup()` method returns the list of values in an RDD for the key referenced by the `key` argument. If used against an RDD partitioned with a known partitioner, `lookup()` uses the partitioner to narrow its search to only the partitions where the key would be present.

[Listing 5.14](#) shows an example of the `lookup()` method.

Listing 5.14 The lookup() Method

[Click here to view code image](#)

```
kvrdd = sc.parallelize([(1, 'A'), (1, 'B'), (2, 'C'), (2, 'D')], 2)
kvrdd.lookup(1)
# returns ['A', 'B']
```

mapPartitions()

Syntax:

[Click here to view code image](#)

```
RDD.mapPartitions(func, preservesPartitioning=False)
```

The `mapPartitions()` method returns a new RDD by applying a function (the `func` argument) to each partition of this RDD. [Listing 5.15](#) demonstrates using the `mapPartitions()` method to invert the key and value within each

partition.

Listing 5.15 The `mapPartitions()` Function

[Click here to view code image](#)

```
kvrdd = sc.parallelize([(1, 'A'), (1, 'B'), (2, 'C'), (2, 'D')], 2)
def f(iterator): yield [(b, a) for (a, b) in iterator]
kvrdd.mapPartitions(f).collect()
# returns [('A', 1), ('B', 1)], [('C', 2), ('D', 2)]]
```

One of the biggest advantages of the `mapPartitions()` method is that the function referenced is called once per partition as opposed to once per element; this can be particularly beneficial if the function has notable overhead for creation.

Many of Spark's other transformations use the `mapPartitions()` function internally. There is also a related transformation called `mapPartitionsWithIndex()`, which returns functions similarly but tracks the index of the original partition.

RDD Storage Options

Thus far, we have discussed RDDs as distributed immutable collections of objects that reside in memory on cluster Worker nodes. There are, however, other storage options for RDDs that are beneficial for a number of reasons. Before we discuss the various RDD storage levels and then caching and persistence, let's review the concept of RDD lineage.

RDD Lineage Revisited

Recall that Spark plans the execution of a program as a DAG (directed acyclic graph), which is a set of operations separated into stages with stage dependencies. Some operations, such as `map()` operations, can be completely parallelized, and some operations, such as `reduceByKey()`, require a shuffle. This naturally introduces a stage dependency.

The Spark Driver keeps track of every RDD's lineage—that is, the series of transformations performed to yield an RDD or a partition thereof. This enables

every RDD at every stage to be reevaluated in the event of a failure, which provides the resiliency in Resilient Distributed Datasets.

Consider the simple example involving only one stage shown in [Figure 5.5](#).

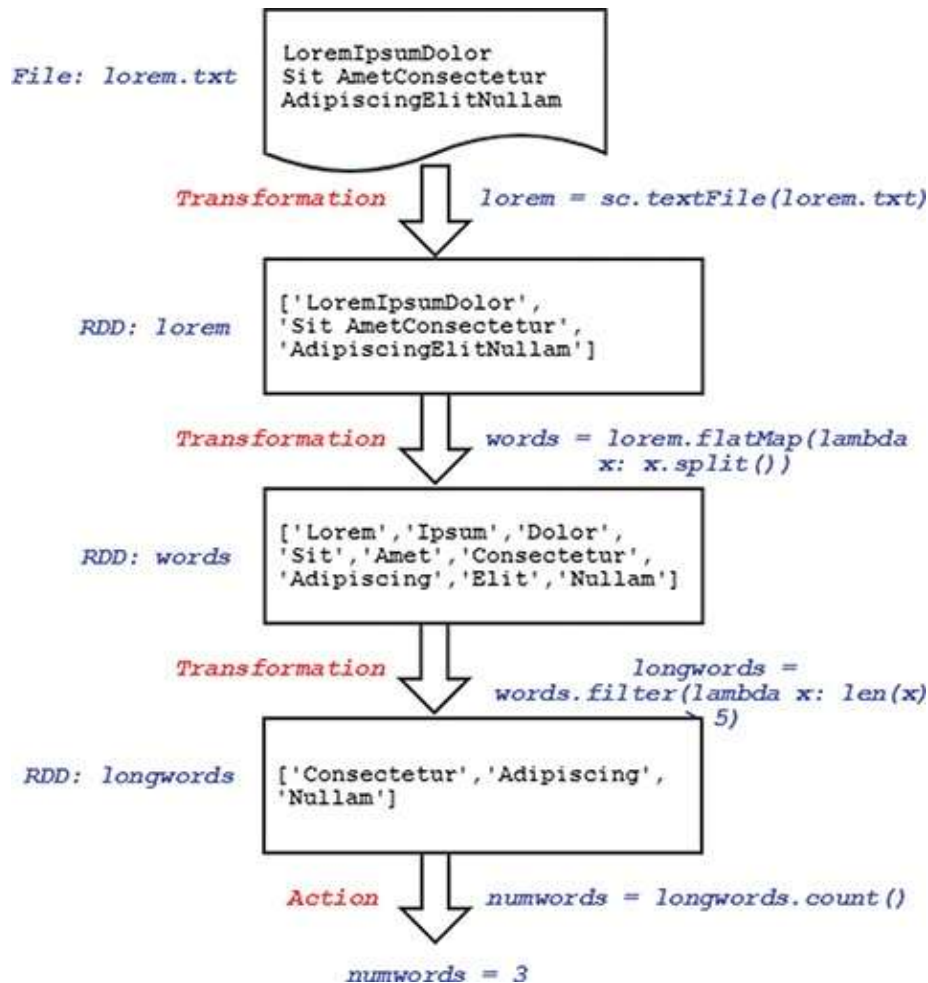


Figure 5.5 RDD lineage.

[Listing 5.16](#) shows a summary of a physical execution plan created by Spark using the `toDebugString()` function.

Listing 5.16 The `toDebugString()` Function

[Click here to view code image](#)

```
>>> print(longwords.toDebugString())
(1) PythonRDD[6] at collect at <stdin>:1 []
| MapPartitionsRDD[1] at textFile at ..[]
```

```
| file://lorem.txt HadoopRDD[0] at textFile at ..[]
```

The action `longwords.count()` forces evaluation of each of the parent RDDs to `longwords`. If this or any other action, such as `longwords.take(1)` or `longwords.collect()`, is called a subsequent time, the entire lineage is reevaluated. In simple cases, with small amounts of data with one or two stages, these reevaluations are not an issue, but in many circumstances, they can be inefficient and impact recovery times in the event of failure.

RDD Storage Options

RDDs are stored in their partitions on various worker nodes in a Spark YARN, Standalone, or Mesos cluster. RDDs have six basic storage levels available, as summarized in [Table 5.2](#).

Table 5.2 **RDD Storage Levels**

Storage Level	Description
MEMORY_ONLY	RDD partitions are stored in memory only. This is the default.
MEMORY_AND_DISK	RDD partitions that do not fit in memory are stored on disk.
MEMORY_ONLY_SER*	RDD partitions are stored as serialized objects in memory. Use this option to save memory, as serialized objects may consume less space than the deserialized equivalent.
MEMORY_AND_DISK_SER*	RDD partitions are stored as serialized objects in memory. Objects that do not fit into memory spill to disk.
DISK_ONLY	RDD partitions are stored on disk only.
OFF_HEAP	RDD partitions are stored as serialized objects in memory. This requires that off-heap memory be enabled. Note that this storage option is for experimental use only.

* These options are relevant for Java or Scala use only. Using the Spark Python API, objects are always serialized using the Pickle library, so it is not necessary to specify serialization.

In addition, there are replicated storage options available with each of the basic storage levels listed in [Table 5.2](#). These replicate each partition to more than one cluster node. Replication of RDDs consumes more space across the cluster but enables tasks to continue to run in the event of a failure without having to wait for lost partitions to reprocess. Although fault tolerance is provided for all Spark RDDs, regardless of their storage level, replicated storage levels provide much faster fault recovery.

Storage-Level Flags

A storage level is implemented as a set of flags that control the RDD storage. There are flags that determine whether to use memory, whether to spill data to disk if it does not fit in memory, whether to store objects in serialized format, and whether to replicate the RDD partitions to multiple nodes. Flags are implemented in the `StorageClass` constructor, as shown in [Listing 5.17](#).

Listing 5.17 `StorageClass` Constructor

[Click here to view code image](#)

```
StorageLevel(useDisk,  
             useMemory,  
             useOffHeap,  
             deserialized,  
             replication=1)
```

The `useDisk`, `useMemory`, `useOffHeap`, and `deserialized` arguments are Boolean values, whereas the `replication` argument is an integer value that defaults to 1. The RDD storage levels listed in [Table 5.2](#) are actually static constants that you can use for common storage levels. Table 5.3 shows these static constants with their respective flags.

Table 5.3 `StorageLevel` Constants and Flags

Constant	<code>useDisk</code>	<code>useMemory</code>	<code>useOffHeap</code>	<code>dese</code>
----------	----------------------	------------------------	-------------------------	-------------------

MEMORY_ONLY	False	True	False	True
MEMORY_AND_DISK	True	True	False	True
MEMORY_ONLY_SER	False	True	False	False
MEMORY_AND_DISK_SER	True	True	False	False
DISK_ONLY	True	False	False	False
MEMORY_ONLY_2	False	True	False	True
MEMORY_AND_DISK_2	True	True	False	True
MEMORY_ONLY_SER_2	False	True	False	False
MEMORY_AND_DISK_SER_2	True	True	False	False
DISK_ONLY_2	True	False	False	False
OFF_HEAP	False	False	True	False

getStorageLevel()

Syntax:

```
RDD.getStorageLevel()
```

The Spark API includes a function called `getStorageLevel()` that you can use to inspect the storage level for an RDD. The `getStorageLevel()` function returns the different storage option flags set for an RDD. The return value in the case of PySpark is an instance of the class `pyspark.StorageLevel`. [Listing 5.18](#) shows how to use the `getStorageLevel()` function.

Listing 5.18 The `getStorageLevel()` Function

[Click here to view code image](#)

```
>>> lorem = sc.textFile('file:///lorem.txt')
>>> lorem.getStorageLevel()
StorageLevel(False, False, False, False, 1)
# get individual flags
>>> lorem_sl = lorem.getStorageLevel()
```

```
>>> lorem_sl.useDisk
False
>>> lorem_sl.useMemory
False
>>> lorem_sl.useOffHeap
False
>>> lorem_sl.deserialized
False
>>> lorem_sl.replication
1
```

Choosing a Storage Level

RDD storage levels enable you to tune Spark jobs and to accommodate large-scale operations that would otherwise not fit into the aggregate memory available across the cluster. In addition, replication options for the available storage levels can reduce recovery times in the event of a task or node failure.

Generally speaking, if an RDD fits into the available memory across the cluster, the default memory-only storage level is sufficient and will provide the best performance.

RDD Caching

A Spark RDD, including all of its parent RDDs, is normally recomputed for each action called in the same session or application. Caching an RDD persists the data in memory; the same routine can then reuse it multiple times when subsequent actions are called, without requiring reevaluation.

Caching does not trigger execution or computation; rather, it is a suggestion. If there is not enough memory available to cache the RDD, it is reevaluated for each lineage triggered by an action. Caching never spills to disk because it only uses memory. The cached RDD persists using the `MEMORY_ONLY` storage level.

Under the appropriate circumstances, caching is a useful tool to increase application performance. [Listing 5.19](#) shows an example of caching with RDDs.

Listing 5.19 Caching RDDs

[Click here to view code image](#)

```
doc = sc.textFile("file:///opt/spark/data/shakespeare.txt")
words = doc.flatMap(lambda x: x.split()) \
    .map(lambda x: (x,1)) \
    .reduceByKey(lambda x, y: x + y)
words.cache()
words.count() # triggers computation
# returns: 33505
words.take(3) # no computation required
# returns: [('Quince', 8), ('Begin', 9), ('Just', 12)]
words.count() # no computation required
# returns: 33505
```

Persisting RDDs

Cached partitions, partitions of an RDD where the `cache()` method ran, are stored in memory on Executor JVMs on Spark Worker nodes. If one of the Worker nodes were to fail or become unavailable, Spark would need to re-create the cached partition from its lineage.

The `persist()` method, introduced in [Chapter 4](#), offers additional storage options, including `MEMORY_AND_DISK`, `DISK_ONLY`, `MEMORY_ONLY_SER`, `MEMORY_AND_DISK_SER`, and `MEMORY_ONLY`, which is the same as the `cache()` method. When using persistence with one of the disk storage options, the persisted partitions are stored as local files on the Worker nodes running Spark Executors for the application. You can use the persisted data on disk to reconstitute partitions lost due to Executor or memory failure.

In addition, `persist()` can use replication to persist the same partition on more than one node. Replication makes reevaluation less likely because more than one node would need to fail or be unavailable to trigger recomputation.

Persistence offers additional durability over caching, while still offering increased performance. It is worth reiterating that Spark RDDs are fault tolerant regardless of persistence and can always be re-created in the event of a failure. Persistence simply expedites this process.

Persistence, like caching, is only a suggestion, and it takes place only after an action is called to trigger evaluation of an RDD. If sufficient resources are not available—for instance, if there is not enough memory available—persistence is not implemented.

You can inspect the persistence state and current storage levels from any RDD at any stage by using the `getStorageLevel()` method, discussed earlier in this chapter.

The methods available for persisting and unpersisting RDDs are documented in the following sections.

`persist()`

Syntax:

[Click here to view code image](#)

```
RDD.persist(storageLevel=StorageLevel.MEMORY_ONLY_SER)
```

The `persist()` method specifies the desired storage level and storage attributes for an RDD. The desired storage options are implemented the first time the RDD is evaluated. If this is not possible—for example, if there is insufficient memory to persist the RDD in memory—Spark reverts to its normal behavior of retaining only required partitions in memory.

The `storageLevel` argument is expressed as either a static constant or a set of storage flags (see the section “RDD Storage Options,” earlier in this chapter). For example, to set a storage level of `MEMORY_AND_DISK_SER_2`, you could use either of the following:

[Click here to view code image](#)

```
myrdd.persist(StorageLevel.MEMORY_AND_DISK_SER_2)
myrdd.persist(StorageLevel(True, True, False, False, 2))
```

The default storage level is `MEMORY_ONLY`.

`unpersist()`

Syntax:

```
RDD.unpersist()
```

The `unpersist()` method “unpersists” the RDD. Use it if you no longer need the RDD to persist. Also, if you want to change the storage options for a persisted RDD, you must unpersist the RDD first. If you attempt to change the storage level of an RDD marked for persistence, you get the exception “Cannot change storage level of an RDD after it was already assigned a level.”

[Listing 5.20](#) shows several examples of persistence.

Listing 5.20 Persisting an RDD

[Click here to view code image](#)

```
doc = sc.textFile("file:///opt/spark/data/shakespeare.txt")
words = doc.flatMap(lambda x: x.split()) \
    .map(lambda x: (x,1)) \
    .reduceByKey(lambda x, y: x + y)
words.persist()
words.count()
# returns: 33505
words.take(3)
# returns: [('Quince', 8), ('Begin', 9), ('Just', 12)]
print(words.toDebugString().decode("utf-8"))
# returns:
# (1) PythonRDD[46] at RDD at PythonRDD.scala:48 [Memory Serialized 1x
# Replicated]
# |      CachedPartitions: 1; MemorySize: 644.8 KB;
# ExternalBlockStoreSize: ...
# | MapPartitionsRDD[45] at mapPartitions at PythonRDD.scala:427 [...]
# | ShuffledRDD[44] at partitionBy at NativeMethodAccessorImpl.java:0
# [...]
# +-(1) PairwiseRDD[43] at reduceByKey at <stdin>:3 [Memory Serialized
# 1x ...]
# | PythonRDD[42] at reduceByKey at <stdin>:3 [Memory Serialized 1x
# Replicated]
# | file:///opt/spark/data/shakespeare.txt MapPartitionsRDD[41] at
# textFile ...
# | file:///opt/spark/data/shakespeare.txt HadoopRDD[40] at
# textFile at ...
```

Note that the `unpersist()` method can also be used to remove an RDD that was cached using the `cache()` method.

Persisted RDDs are also viewable in the Spark application UI via the Storage tab, as shown in [Figures 5.6](#) and [5.7](#).

PySparkShell - Storage x

ec2-13-210-114-32.ap-southeast-2.compute.amazonaws.com:4040/storage/

APACHE **Spark** 2.2.0 Jobs Stages **Storage** Environment Executors SQL PySparkShell application UI

Storage

RDDs

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
PythonRDD	Memory Serialized 1x Replicated	1	100%	644.8 KB	0.0 B

Figure 5.6 Viewing persisted RDDs in the Spark application UI.

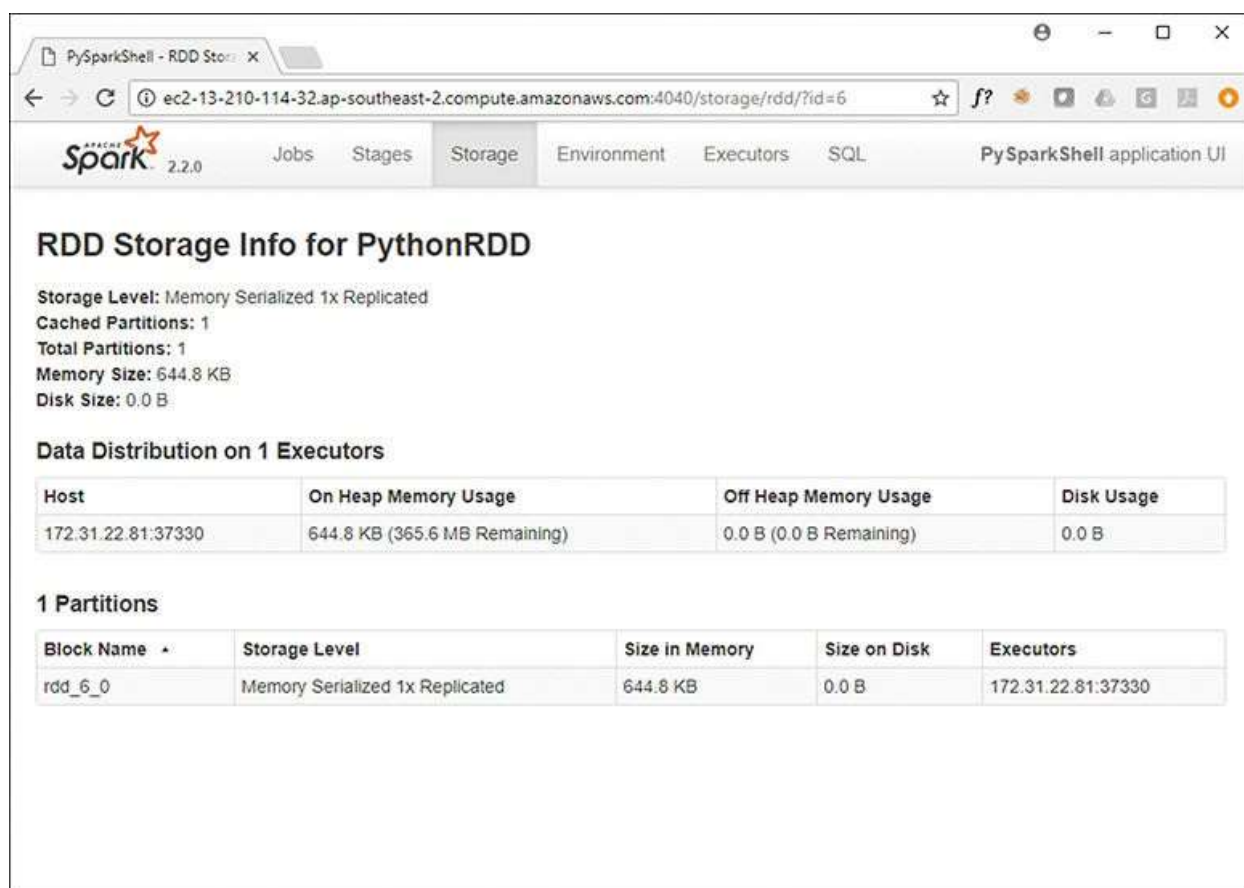


Figure 5.7 Viewing details of a persisted RDD in the Spark application UI.

Choosing When to Persist or Cache RDDs

Caching can improve performance or reduce recovery times. If an RDD is likely to be reused and if sufficient memory is available on Worker nodes in the cluster, it is typically beneficial to cache these RDDs. Iterative algorithms such as those used in machine learning routines are often good candidates for caching.

Caching reduces recovery times in the event of failure because RDDs need to be recomputed only starting from the cached RDDs. However, if you require a higher degree of in-process durability, consider one of the disk-based persistence options or a higher replication level, which increases the likelihood that a persisted replica of an RDD exists somewhere in the Spark cluster.

Checkpointing RDDs

Checkpointing involves saving data to a file. Unlike the disk-based persistence option just discussed, which deletes the persisted RDD data when the Spark Driver program finishes, checkpointed data persists beyond the application.

Checkpointing eliminates the need for Spark to maintain RDD lineage, which can be problematic when the lineage gets long, such as with streaming or iterative processing applications. Long lineage typically leads to long recovery times and the possibility of a stack overflow.

Checkpointing data to a distributed filesystem such as HDFS provides additional storage fault tolerance as well. Checkpointing is expensive, so implement it with some consideration about when you should checkpoint an RDD.

As with the caching and persistence options, checkpointing happens only after an action is called against an RDD to force computation, such as `count()`. Note that checkpointing must be requested before any action is requested against an RDD.

The methods associated with checkpointing are documented in the following sections.

setCheckpointDir()

Syntax:

```
sc.setCheckpointDir(dirName)
```

The `setCheckpointDir()` method sets the directory under which RDDs will be checkpointed. If you are running Spark on a Hadoop cluster, the directory specified by the `dirName` argument must be an HDFS path.

checkpoint()

Syntax:

```
RDD.checkpoint()
```

The `checkpoint()` method marks the RDD for checkpointing. It will be checkpointed upon the first action executed against the RDD, and the files saved to the directory will be configured using the `setCheckpointDir()` method. The `checkpoint()` method must be called before any action is requested against the RDD.

When checkpointing is complete, the complete RDD lineage, including all

references to the RDDs and parent RDDs, are removed.

Specifying the Checkpoint Directory Prior to Running `checkpoint()`

You must specify the checkpoint directory by using the `setCheckpointDir()` method before attempting to checkpoint an RDD; otherwise, you will receive the following error:

[Click here to view code image](#)

```
org.apache.spark.SparkException:  
Checkpoint directory has not been set in the SparkContext
```

The checkpoint directory is valid only for the current `SparkContext`, so you need to execute `setCheckpointDir()` for each separate Spark application. In addition, the checkpoint directory cannot be shared across different Spark applications.

`isCheckpointed()`

Syntax:

```
RDD.isCheckpointed()
```

The `isCheckpointed()` function returns a Boolean response about whether the RDD was checkpointed.

`getCheckpointFile()`

Syntax:

```
RDD.getCheckpointFile()
```

The `getCheckpointFile()` function returns the name of the file to which the RDD was checkpointed.

[Listing 5.21](#) demonstrates the use of checkpointing.

Listing 5.21 Checkpointing RDDs

[Click here to view code image](#)

```
sc.setCheckpointDir('file:///opt/spark/data/checkpoint')  
doc = sc.textFile("file:///opt/spark/data/shakespeare.txt")
```

```
words = doc.flatMap(lambda x: x.split()) \
    .map(lambda x: (x,1)) \
    .reduceByKey(lambda x, y: x + y)
words.checkpoint()
words.count()
# returns: 33505
words.isCheckpointed()
# returns: True
words.getCheckpointFile()
# returns:
# 'file:/opt/spark/data/checkpoint/df6370eb-7b5f-4611-99a8-
# bacb576c2ea1/rdd-15'
```

Exercise: Checkpointing RDDs

This exercise shows the impact that checkpointing can have on an iterative routine. Use any installation of Spark for this exercise and follow these steps:

1. For this exercise, you will run a script in non-interactive mode and need to suppress informational log messages, so perform the following steps:
 - a. Make a copy of the default `log4j.properties` template file, as follows:

[Click here to view code image](#)

```
cd /opt/spark/conf
cp log4j.properties.template log4j.properties.erroronly
```

- b. Use a text editor (such as Vi or Nano) to open the newly created `log4j.properties.erroronly` file and locate the following line:

```
log4j.rootCategory=INFO, console
```

- c. Change the line to the following:

```
log4j.rootCategory=ERROR, console
```

Save the file.

2. Create a new script called `looping_test.py`, and copy and paste the code below into the file:

[Click here to view code image](#)

```
import sys
```

```

from pyspark import SparkConf, SparkContext
sc = SparkContext()
sc.setCheckpointDir("file:///tmp/checkpointdir")
rddofints = sc.parallelize([1,2,3,4,5,6,7,8,9,10])
try:
    # this will create a very long lineage for rddofints
    for i in range(1000):
        rddofints = rddofints.map(lambda x: x+1)
        if i % 10 == 0:
            print("Looped " + str(i) + " times")
            #rddofints.checkpoint()
            rddofints.count()
except Exception as e:
    print("Exception : " + str(e))
    print("RDD Debug String : ")
    print(rddofints.toDebugString())
    sys.exit()
print("RDD Debug String : ")
print(rddofints.toDebugString())

```

3. Execute the `looping_test.py` script by using `spark-submit` and your custom `log4j.properties` file, as follows:

[Click here to view code image](#)

```

$ spark-submit \
--master local \
--driver-java-options \
"-Dlog4j.configuration=log4j.properties.erroronly" \
looping_test.py

```

After a certain number of iterations, you should see an exception like this:

[Click here to view code image](#)

```

PicklingError: Could not pickle object as excessively deep recursion
required.

```

4. Open the `looping_test.py` file again with a text editor and uncomment the following line:

[Click here to view code image](#)

```

#rddofints.checkpoint()

```

So the file should now read:

```

...
print("Looped " + str(i) + " times")

```

```
rddofints.checkpoint()  
rddofints.count()  
...
```

5. Execute the script again, using `spark-submit`, as shown in step 3. You should now see that all 1,000 iterations have completed, thanks to the periodic checkpointing of the RDD. Furthermore, note the debug string printed after the routine:

[Click here to view code image](#)

```
(1) PythonRDD[301] at RDD at PythonRDD.scala:43 []  
| PythonRDD[298] at RDD at PythonRDD.scala:43 []  
| ReliableCheckpointRDD[300] at count at ...
```

Checkpointing, caching, and persistence are useful functions in Spark programming. They can not only improve performance but, in some cases, as you have just seen, can mean the difference between a program completing successfully or not.

Find the complete source code for this exercise in the `checkpointing` folder at https://github.com/sparktraining/spark_using_python.

Processing RDDs with External Programs

Spark provides a mechanism to run functions (transformations) using languages other than those native to Spark (Scala, Python, and Java). You can also use Ruby, Perl, or Bash, among others. The languages do not need to be scripting languages, either; you can use Spark with C or FORTRAN, for example.

There are different reasons for wanting to do use languages other than those native to Spark, such as wanting to use in your Spark programs some existing code libraries that are not in Python, Scala, or Java without having to rewrite them in a native Spark language.

Using external programs with Spark is achieved through the `pipe()` function.

Possible Issues with External Processes in Spark

Use the `pipe()` function carefully because piped commands may fork excessive amounts of RAM. Because the forked subprocesses created by the `pipe()` function are out of Spark's resource management scope, they may also cause performance degradation for other tasks running on Worker

nodes.

pipe()

Syntax:

[Click here to view code image](#)

```
RDD.pipe(command, env=None, checkCode=False)
```

The `pipe()` method returns an RDD created by “piping” elements through a forked external process specified by the `command` argument. The `env` argument is a `dict` of environment variables that defaults to `None`. The `checkCode` parameter specifies whether to check the return value of the shell command.

The script or program you supply as the `command` argument needs to read from `STDIN` and write its output to `STDOUT`.

Consider the Perl script saved as `parsefixedwidth.pl` in [Listing 5.22](#); it is used to parse fixed-width output data, a common file format with extracts from mainframes and legacy systems. To make this script executable, you need to use the following:

```
chmod +x parsefixedwidth.pl.
```

Listing 5.22 Sample External Transformation Program (parsefixedwidth.pl)

[Click here to view code image](#)

```
#!/usr/bin/env perl
my $format = 'A6 A8 A20 A2 A5';
while (<>) {
    chomp;
    my( $custid, $orderid, $date,
        $city, $state, $zip) =
        unpack( $format, $_ );
    print "$custid\t$orderid\t$date\t$city\t$state\t$zip";
}
```

[Listing 5.23](#) demonstrates the use of the `pipe()` command to run the `parsefixedwidth.pl` script from [Listing 5.22](#).

Listing 5.23 The `pipe()` Function

[Click here to view code image](#)

```
sc.addFile("/home/ubuntu/parsefixedwidth.pl")
fixed_width = sc.parallelize(['3840961028752220160317Hayward
CA94541'])
piped = fixed_width.pipe("parsefixedwidth.pl") \
.map(lambda x: x.split('\t'))
piped.collect()
# returns [['384096', '10287522', '20160317', 'Hayward', 'CA', '94541']]
```

The `addFile()` operation is required because you need to distribute the `parsefixedwidth.pl` Perl script to all Worker nodes participating in the cluster prior to running the `pipe()` transformation.

Note that you also need to ensure that the interpreter or host program (in this case, Perl) exists in the path of all Worker nodes. The complete source code for this example is in the `using-external-programs` folder at https://github.com/sparktraining/spark_using_python.

Data Sampling with Spark

When using Spark for development and discovery, you may need to sample data in RDDs before running a process across the entirety of an input dataset or datasets. The Spark API includes several functions to sample RDDs and produce new RDDs from the sampled data. These sample functions include transformations that return new RDDs and actions that return data to the Spark Driver program. The following sections look at a couple sampling transformations and actions that Spark provides.

`sample()`

Syntax:

[Click here to view code image](#)

```
RDD.sample(withReplacement, fraction, seed=None)
```

The `sample()` transformation creates a sampled subset RDD from an original RDD, based on a percentage of the overall dataset.

The `withReplacement` argument is a Boolean value that specifies whether elements in an RDD can be sampled multiple times.

The `fraction` argument is a double value between 0 and 1 that represents the probability an element will be chosen. Effectively, this represents the approximate percentage of the dataset you wish to return to the resultant sampled RDD. Note that if you specify a value larger than 1 for this argument, it defaults to 1 anyway.

The optional `seed` argument is an integer representing a seed for the random number generator used to determine whether to include an element in the return RDD.

[Listing 5.24](#) shows an example of the `sample()` transformation used to create approximately a 10% subset of web log events from a corpus of web logs.

Listing 5.24 Sampling Data Using the `sample()` Function

[Click here to view code image](#)

```
doc = sc.textFile("file:///opt/spark/data/shakespeare.txt")
doc.count()
# returns: 129107
sampled_doc = doc.sample(False, 0.1, seed=None)
sampled_doc.count()
# returns: 12879 (approximately 10% of the original RDD)
```

There is also a similar `sampleByKey()` function that operates on a key/value pair RDD.

`takeSample()`

Syntax:

[Click here to view code image](#)

```
RDD.takeSample(withReplacement, num, seed=None)
```

The `takeSample()` action returns a random list of values (elements or records) from the sampled RDD.

The `num` argument is the number of randomly selected records to be returned.

The `withReplacement` and `seed` arguments behave similarly to the

`sample()` function just described.

[Listing 5.25](#) shows an example of the `takeSample()` action.

Listing 5.25 Using the `takeSample()` Function

[Click here to view code image](#)

```
dataset = sc.parallelize([1,2,3,4,5,6,7,8,9,10])
dataset.takeSample(False, 3)
# returns [6, 7, 5] (your results may vary!)
```

Understanding Spark Application and Cluster Configuration

Practically everything in Spark is configurable, and everything that is configurable typically has a default setting. This section takes a closer look at configuration for Spark applications and clusters, focusing specifically on the settings and concepts you need to be aware of as a Spark engineer or developer.

Spark Environment Variables

Spark environment variables are set by the `spark-env.sh` script located in the `$SPARK_HOME/conf` directory. The variables set Spark daemon behavior and configuration, and they set environment-level application configuration settings, such as which Spark Master an application should use. The `spark-env.sh` script is read by the following:

- Spark Standalone Master and Worker daemons upon startup
- Spark applications, using `spark-submit`

[Listing 5.26](#) provides some examples of settings for some common environment variables; these could be set in your `spark-env.sh` file or as environment variables in your shell prior to running an interactive Spark process such as `pyspark` or `spark-shell`.

Listing 5.26 Spark Environment Variables

[Click here to view code image](#)

```
export SPARK_HOME=${SPARK_HOME:-/usr/lib/spark}
export SPARK_LOG_DIR=${SPARK_LOG_DIR:-/var/log/spark}
export HADOOP_HOME=${HADOOP_HOME:-/usr/lib/hadoop}
export HADOOP_CONF_DIR=${HADOOP_CONF_DIR:-/etc/hadoop/conf}
export HIVE_CONF_DIR=${HIVE_CONF_DIR:-/etc/hive/conf}
export STANDALONE_SPARK_MASTER_HOST=sparkmaster.local
export SPARK_MASTER_PORT=7077
export SPARK_MASTER_IP=$STANDALONE_SPARK_MASTER_HOST
export SPARK_MASTER_WEBUI_PORT=8080
export SPARK_WORKER_DIR=${SPARK_WORKER_DIR:-/var/run/spark/work}
export SPARK_WORKER_PORT=7078
export SPARK_WORKER_WEBUI_PORT=8081
export SPARK_DAEMON_JAVA_OPTS="-XX:OnOutOfMemoryError='kill -9 %p'"
```

The following sections take a look at some of the most common Spark environment variables and their use.

Cluster Manager Independent Variables

Some of the environment variables that are independent of the cluster manager used are described in [Table 5.4](#).

Table 5.4 **Cluster Manager Independent Variables**

Environment Variable	Description
SPARK_HOME	The root of the Spark installation directory (for example, <code>/opt/spark</code> or <code>/usr/lib/spark</code>). You should always set this variable, especially if you have multiple versions of Spark installed on a system. Failing to set this variable is a common cause of issues when running Spark applications.
JAVA_HOME	The location where Java is installed.
PYSPARK_PYTHON	The Python binary executable to use for PySpark in both the Driver and Workers. If not specified, the default Python installation is

used (resolved by the `which python` command). This should definitely be set if you have more than one version of Python on any Driver or Worker instances.

PYSPARK_DRIVER_PYTHON	The Python binary executable to use for PySpark in the Driver only; defaults to the value defined for PYSPARK_PYTHON.
SPARKR_DRIVER_R	The R binary executable to use for the SparkR shell; the default is R.

Hadoop-Related Environment Variables

The variables described in [Table 5.5](#) are required for Spark applications that need access to HDFS from any deployment mode, YARN if running in YARN Client or YARN Cluster mode, and objects in HCatalog or Hive.

Table 5.5 **Hadoop-Related Environment Variables**

Environment Variable	Description
HADOOP_CONF_DIR or YARN_CONF_DIR	The location of the Hadoop configuration files (typically <code>/etc/hadoop/conf</code>). Spark uses this to locate the default filesystem, usually the URI of the HDFS NameNode, and the address of the YARN ResourceManager. Either of these environment variables can be set, but typically, HADOOP_CONF_DIR is preferred.
HADOOP_HOME	The location where Hadoop is installed. Spark uses this to locate the Hadoop configuration files.
HIVE_CONF_DIR	The location of the Hive configuration files. Spark uses this to locate the Hive metastore and other Hive properties when instantiating a HiveContext object. There are also environment variables specific to HiveServer2, such as HIVE_SERVER2_THRIFT_BIND_HOST and HIVE_SERVER2_THRIFT_PORT. Typically, just

setting `HADOOP_CONF_DIR` is sufficient because Spark can infer the other properties relative to this.

YARN-Specific Environment Variables

The environment variables described in [Table 5.6](#) are specific to Spark applications running on a YARN cluster, either in Cluster or Client deployment mode.

Table 5.6 **YARN-Specific Environment Variables**

Environment Variable	Description
<code>SPARK_EXECUTOR_INSTANCES</code>	The number of Executor processes to start in the YARN cluster; defaults to <code>2</code> .
<code>SPARK_EXECUTOR_CORES</code>	The number of CPU cores allocated to each Executor; defaults to <code>1</code> .
<code>SPARK_EXECUTOR_MEMORY</code>	The amount of memory allocated to each Executor; defaults to <code>1GB</code> .
<code>SPARK_DRIVER_MEMORY</code>	The amount of memory allocated to Driver processes when running in Cluster deployment mode; defaults to <code>1GB</code> .
<code>SPARK_YARN_APP_NAME</code>	The name of your application. This displays in the YARN ResourceManager UI; defaults to <code>Spark</code> .
<code>SPARK_YARN_QUEUE</code>	The named YARN queue to which applications are submitted by default; defaults to <code>default</code> . Can also be set by a <code>spark-submit</code> argument. This determines allocation of resources and scheduling priority.
<code>SPARK_YARN_DIST_FILES</code> or <code>SPARK_YARN_DIST_ARCHIVES</code>	A comma-separated list of files of archives to be distributed with the job. Executors can then reference these files at runtime.

As previously mentioned, you must set the `HADOOP_CONF_DIR` environment

variable when deploying a Spark application on YARN.

Cluster Application Deployment Mode Environment Variables

The variables listed in [Table 5.7](#) are used for applications submitted in Cluster mode—that is, applications using the Standalone or YARN cluster managers submitted with the `--deploy-mode cluster` option to `spark-submit`. In the case of YARN, this property can combine with the `master` argument as `--master yarn-cluster`. These variables are read by Executor and Driver processes running on Workers in the cluster (Spark Workers or YARN NodeManagers).

Table 5.7 Cluster Application Deployment Mode Environment Variables

Environment Variable	Description
SPARK_LOCAL_IP	The IP address of the machine for binding Spark processes.
SPARK_PUBLIC_DNS	The hostname the Spark Driver uses to advertise to other hosts.
SPARK_CLASSPATH	The default classpath for Spark. This is important if you are importing additional classes not packaged with Spark that you will refer to at runtime.
SPARK_LOCAL_DIRS	The directories to use on the system for RDD storage and shuffled data.

When running an interactive Spark session (using `pyspark` or `spark-shell`), the `spark-env.sh` file is not read, and the environment variables in the current user environment (if set) are used.

Many Spark environment variables have equivalent configuration properties that you can set in a number of additional ways; we discuss this shortly.

Spark Standalone Daemon Environment Variables

The environment variables shown in [Table 5.8](#) are read by daemons—Masters and Workers—in a Spark Standalone cluster.

Table 5.8 Spark Standalone Daemon Environment Variables

Environment Variable	Description
SPARK_MASTER_IP	The hostname or IP address of the host running the Spark Master process. This should be set on all nodes of the Spark cluster and on any client hosts that will be submitting applications.
SPARK_MASTER_PORT and SPARK_MASTER_WEBUI_PORT	The ports used for IPC communication and the Master web UI, respectively. If not specified, the defaults 7077 and 8080 are used.
SPARK_MASTER_OPTS and SPARK_WORKER_OPTS	Additional Java options supplied to the JVM hosting the Spark Master or Spark Worker processes. If used, the value should be in the standard form -Dx=y. Alternatively, you can set the SPARK_DAEMON_JAVA_OPTS environment variable, which applies to all Spark daemons running on the system.
SPARK_DAEMON_MEMORY	The amount of memory to allocate to the Master, Worker, and HistoryServer processes; defaults to 1GB.
SPARK_WORKER_INSTANCES	The number of Worker processes per slave node; defaults to 1.
SPARK_WORKER_CORES	The number of CPU cores for the Spark Worker process used by Executors on the system.
SPARK_WORKER_MEMORY	The amount of total memory Workers have to grant to Executors.
SPARK_WORKER_PORT and SPARK_WORKER_WEBUI_PORT	The ports used for IPC communication and the Worker web UI, respectively. If not specified, the defaults of 8081 for the web UI and a random port for the Worker port are used.
SPARK_WORKER_DIR	Sets the working directory for Worker

processes.

Spark Configuration Properties

Spark configuration properties are typically set on a node, such as a Master or Worker node, or an application by the Driver host submitting the application. They often have a more restricted scope—such as for the life of an application—than their equivalent environment variables, and they take higher precedence than environment variables.

There are numerous Spark configuration properties related to different operational aspects; some of the most common ones are described in [Table 5.9](#).

Table 5.9 Common Spark Configuration Properties

Property	Description
<code>spark.master</code>	The address of the Spark Master (for example, <code>spark://<masterhost>:7077</code> for a Standalone cluster). If the value is <code>yarn</code> , the Hadoop configuration files are read to locate the YARN ResourceManager. There is no default value for this property.
<code>spark.driver.memory</code>	The amount of memory allocated to the Driver; defaults to 1GB.
<code>spark.executor.memory</code>	The amount of memory to use per Executor process; defaults to 1GB.
<code>spark.executor.cores</code>	The number of cores to use on each Executor. In Standalone mode, this property defaults to using all available cores on the Worker node. Setting this

	property to a value less than the available number of cores enables multiple concurrent Executor processes to spawn. In YARN mode, this property defaults to 1 core per Executor.
<code>spark.driver.extraJavaOptions</code> and <code>spark.executor.extraJavaOptions</code>	Additional Java options supplied to the JVM hosting the Spark Driver or Executor processes. If used, the value should be in the standard form <code>-Dx=y</code> .
<code>spark.driver.extraClassPath</code> and <code>spark.executor.extraClassPath</code>	Additional classpath entries for the Driver and Executor processes if you require additional classes that are not packaged with Spark to be imported.
<code>spark.dynamicAllocation.enabled</code> and <code>spark.shuffle.service.enabled</code>	Properties that are used together to modify the default scheduling behavior in Spark. (Dynamic allocation is discussed later in this chapter.)

Setting Spark Configuration Properties

Spark configuration properties are set through the `$SPARK_HOME/conf/spark-defaults.conf` file, read by Spark applications and daemons upon startup. [Listing 5.27](#) shows an excerpt from a typical `spark-defaults.conf` file.

Listing 5.27 Spark Configuration Properties in the `spark-defaults.conf` File

[Click here to view code image](#)

<code>spark.master</code>	<code>yarn</code>
<code>spark.eventLog.enabled</code>	<code>true</code>

<code>spark.eventLog.dir</code>	<code>hdfs:///var/log/spark/apps</code>
<code>spark.history.fs.logDirectory</code>	<code>hdfs:///var/log/spark/apps</code>
<code>spark.executor.memory</code>	<code>2176M</code>
<code>spark.executor.cores</code>	<code>4</code>

Spark configuration properties can also be set programmatically in your Driver code by using the `SparkConf` object, as shown in [Listing 5.28](#).

Listing 5.28 Setting Spark Configuration Properties Programmatically

[Click here to view code image](#)

```
from pyspark.context import SparkContext
from pyspark.conf import SparkConf
conf = SparkConf()
conf.set("spark.executor.memory", "3g")
sc = SparkContext(conf=conf)
```

There are also several `SparkConf` methods for setting specific common properties. These methods appear in [Listing 5.29](#).

Listing 5.29 Spark Configuration Object Methods

[Click here to view code image](#)

```
from pyspark.context import SparkContext
from pyspark.conf import SparkConf
conf = SparkConf()
conf.setAppName("MySparkApp")
conf.setMaster("yarn")
conf.setSparkHome("/usr/lib/spark")
sc = SparkContext(conf=conf)
```

In most cases, setting Spark configuration properties using arguments to `spark-shell`, `pyspark`, and `spark-submit` is recommended, as setting configuration properties programmatically requires code changes or rebuilding in the case of Scala or Java applications.

Setting configuration properties as arguments to `spark-shell`, `pyspark`, and `spark-submit` is done using specific named arguments for common properties, such as `--executor-memory`. Properties not exposed as named arguments are provided using `--conf PROP=VALUE` to set an arbitrary Spark configuration property or `--properties-file FILE` to load additional arguments from a configuration file. [Listing 5.30](#) provides examples of both methods.

Listing 5.30 Passing Spark Configuration Properties to `spark-submit`

[Click here to view code image](#)

```
# setting config properties using arguments
$SPARK_HOME/bin/spark-submit --executor-memory 1g \
  --conf spark.dynamicAllocation.enabled=true \
  myapp.py # setting config properties using a conf file
$SPARK_HOME/bin/spark-submit \
  --properties-file test.conf \
  myapp.py
```

You can use the `SparkConf.toDebugString()` method to print out the current configuration for a Spark application, as demonstrated in [Listing 5.31](#).

Listing 5.31 Showing the Current Spark Configuration

[Click here to view code image](#)

```
from pyspark.context import SparkContext
from pyspark.conf import SparkConf
conf = SparkConf()
print(conf.toDebugString())
...
spark.app.name=PySparkShell
spark.master=yarn-client
spark.submit.deployMode=client
spark.yarn.isPython=true ...
```

As you can see, there are several ways to pass the same configuration parameter,

including as an environment variable, as a Spark default configuration property, or as a command line argument. [Table 5.10](#) shows just a few of the various ways to set the same property in Spark. Many other properties have analogous settings.

Table 5.10 **Spark Configuration Options**

Argument	Configuration Property	Environment Variable
<code>--master</code>	<code>spark.master</code>	<code>SPARK_MASTER_IP/</code> <code>SPARK_MASTER_PORT</code>
<code>--name</code>	<code>spark.app.name</code>	<code>SPARK_YARN_APP_NAME</code>
<code>--queue</code>	<code>spark.yarn.queue</code>	<code>SPARK_YARN_QUEUE</code>
<code>-- executor- memory</code>	<code>spark.executor.memory</code>	<code>SPARK_EXECUTOR_MEMORY</code>
<code>-- executor- cores</code>	<code>spark.executor.cores</code>	<code>SPARK_EXECUTOR_CORES</code>

Defaults for Environment Variables and Configuration Properties

Looking at the `conf` directory of a fresh Spark deployment, you may notice that by default the `spark-defaults.conf` and `spark-env.sh` files are not implemented. Instead, templates are provided (`spark-defaults.conf.template` and `spark-env.sh.template`). You are encouraged to copy these templates and rename them without the `.template` extension and make the appropriate modifications for your environment.

Spark Configuration Precedence

Configuration properties set directly within an application using a `SparkConf` object take the highest precedence, followed by arguments passed to `spark-submit`, `pyspark`, or `spark-shell`, followed by options set in the

spark-defaults.conf file. Many configuration properties have system default values used in the absence of properties explicitly set through the other means discussed. Figure 5.8 shows the order of precedence for Spark configuration properties.

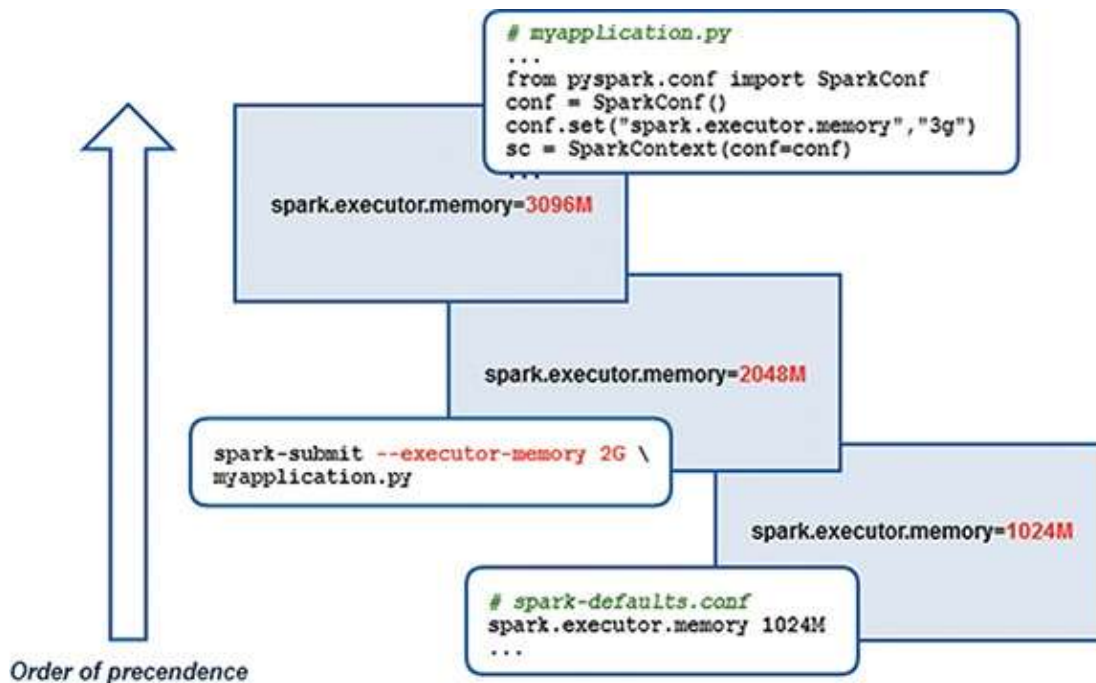


Figure 5.8 Spark configuration precedence.

Configuration Management

Managing configuration is one of the biggest challenges involved in administering a Spark cluster—or any other cluster, for that matter. Often, configuration settings need to be consistent across different hosts, such as different Worker nodes in a Spark cluster. Configuration management and deployment tools such as Puppet and Chef can be useful for managing Spark deployments and their configurations. If you are rolling out and managing Spark as part of a Hadoop deployment using a commercial Hadoop distribution, you can manage Spark configuration by using the Hadoop vendor’s management interface, such as Cloudera Manager for Cloudera installations or Ambari for Hortonworks installations.

In addition, there are other options for configuration management, such as Apache Amaterasu (<http://amaterasu.incubator.apache.org/>), which uses pipelines to build, run, and manage environments as code.

Optimizing Spark

The Spark runtime framework generally does its best to optimize stages and tasks in a Spark application. However, as a developer, you can make many optimizations for notable performance improvements. We discuss some of them in the following sections.

Filter Early, Filter Often

It sounds obvious, but filtering nonrequired records or fields early in your application can have a significant impact on performance. Big Data (particularly event data, log data, or sensor data) is often characterized by a low signal-to-noise ratio. Filtering out noise early saves processing cycles, I/O, and storage in subsequent stages. Use `filter()` transformations to remove unneeded records and `map()` transformations to project only required fields in an RDD. Perform these operations before operations that may invoke a shuffle, such as `reduceByKey()` or `groupByKey()`. Also use them before and after a `join()` operation. These small changes can make the difference between hours and minutes or minutes and seconds.

Optimizing Associative Operations

Associative operations such as `sum()` and `count()` are common requirements when programming in Spark, and you have seen numerous examples of these operations throughout this book. Often on distributed, partitioned datasets, these associative key/value operations may involve shuffling. Typically, `join()`, `cogroup()`, and transformations that have `By` or `ByKey` in their name, such as `groupByKey()` or `reduceByKey()`, can involve shuffling. This is not necessarily a bad thing because it is often required.

However, if you need to perform a shuffle with the ultimate objective of performing an associative operation—counting occurrences of a key, for instance—different approaches that can provide very different performance outcomes. The best example of this is the difference between using `groupByKey()` and using `reduceByKey()` to perform a `sum()` or `count()` operation. Both operations can achieve the same result. However, if you group by a key on a partitioned or distributed dataset solely for the purposes of aggregating values for each key, using `reduceByKey()` is generally a better approach.

`reduceByKey()` combines values for each key prior to any required shuffle operation, thereby reducing the amount of data sent over the network and also reducing the computation and memory requirements for tasks in the next stage. Consider the two code examples in [Listing 5.32](#). Both provide the same result.

Listing 5.32 Associative Operations in Spark

[Click here to view code image](#)

```
rdd.map(lambda x: (x[0],1)) \  
    .groupByKey() \  
    .mapValues(lambda x: sum(x)) \  
    .collect()  
# preferred method  
rdd.map(lambda x: (x[0],1)) \  
    .reduceByKey(lambda x, y: x + y) \  
    .collect()
```

Now consider [Figure 5.9](#), which depicts the `groupByKey()` implementation.

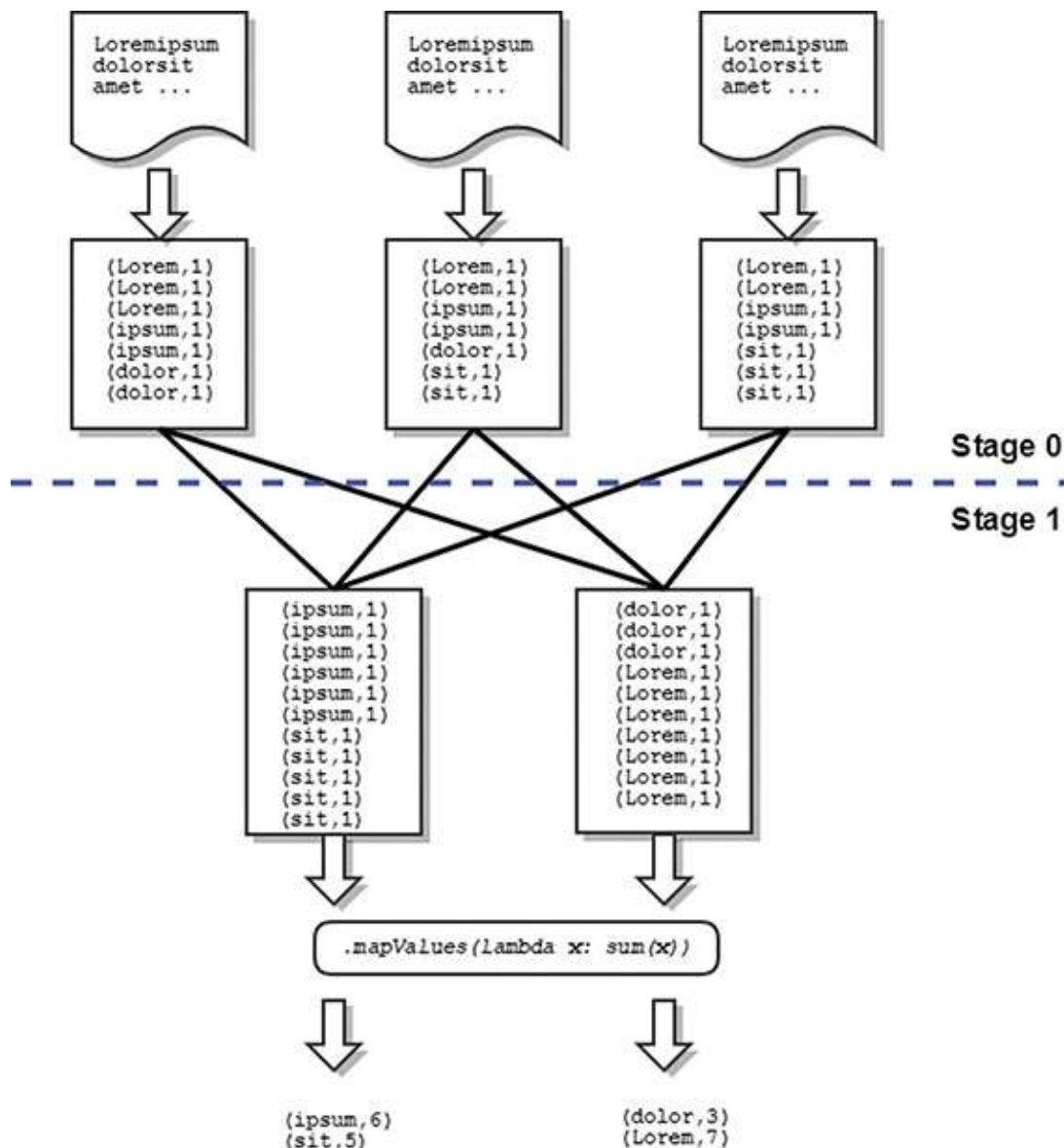


Figure 5.9 `groupByKey()` for an associative operation.

Contrast what you have just seen with [Figure 5.10](#), which shows the functionally equivalent `reduceByKey()` implementation.

As you can see from the preceding figures, `reduceByKey()` combines records locally by key before shuffling the data; this is often referred to as a *combiner* in MapReduce terminology. Combining can result in a dramatic decrease in the amount of data shuffled and thus a corresponding increase in application performance.

Some other alternatives to `groupByKey()` are `combineByKey()`, which

you can use if the inputs and outputs to your reduce function are different, and `foldByKey()`, which performs an associative operation providing a zero value. Additional functions to consider include `treeReduce()`, `treeAggregate()`, and `aggregateByKey()`.

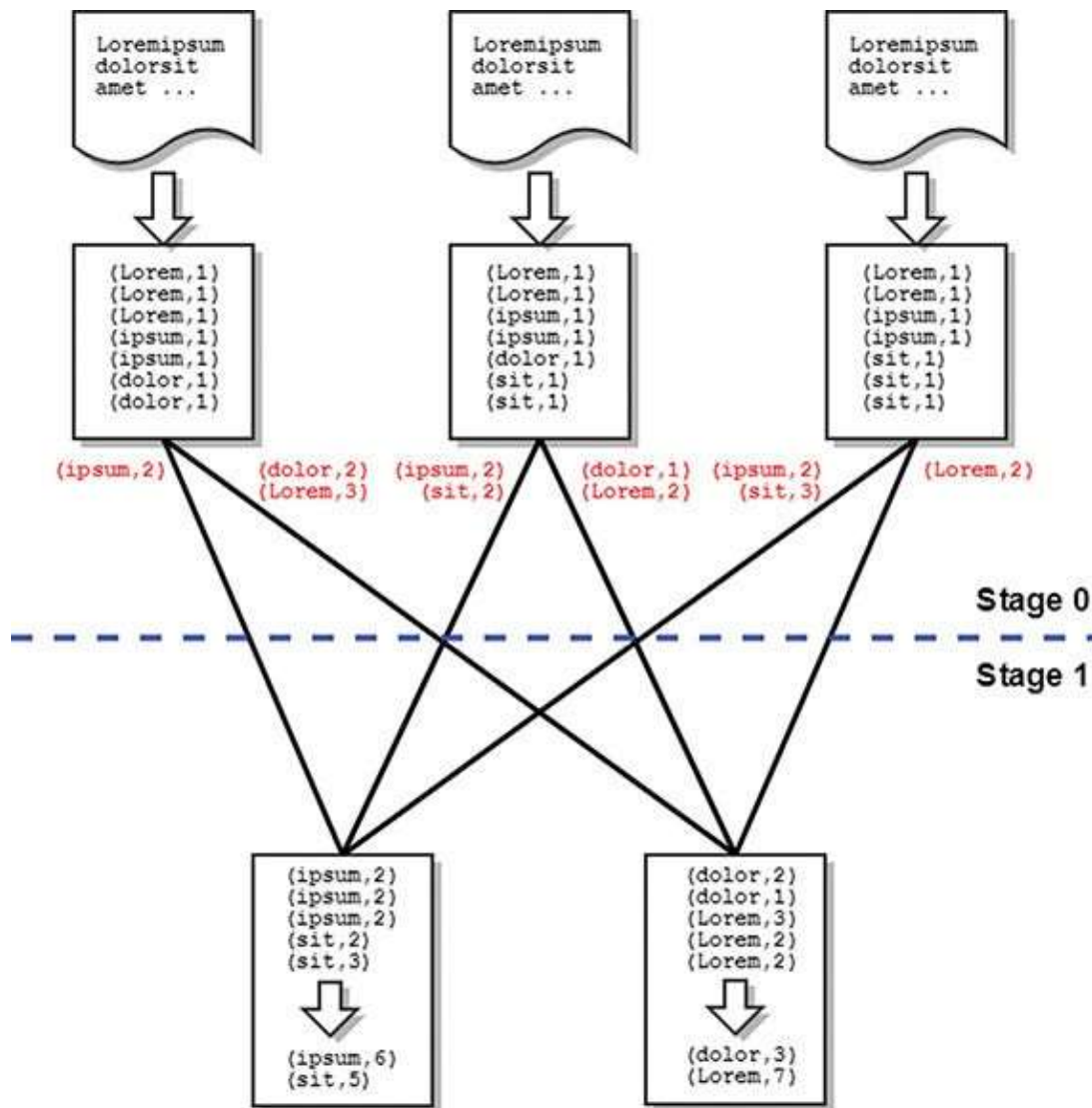


Figure 5.10 `reduceByKey()` for an associative operation.

Understanding the Impact of Functions and Closures

Recall the discussions of functions and closures in [Chapter 1, “Introducing Big Data, Hadoop, and Spark.”](#) Functions are sent to Executors in a Spark cluster, enclosing all bound and free variables. This process enables efficient, shared-

nothing distributed processing. It can also be a potential issue that impacts performance and stability at the same time. It's important to understand this.

A key example of an issue that could arise is passing too much data to a function in a Spark application. This would cause excessive data to be sent to the application Executors at runtime, resulting in excess network I/O, and it could result in memory issues on Spark Workers.

[Listing 5.33](#) shows a fictitious example of declaring a function that encloses a large object and then passing that function to a Spark `map()` transformation.

Listing 5.33 Passing Large Amounts of Data to a Function

[Click here to view code image](#)

```
...
massive_list = [...]
def big_fn(x):
# function enclosing massive_list
...
...
rdd.map(lambda x: big_fn(x)).saveAsTextFile...
# parallelize data which would have otherwise been enclosed
massive_list_rdd = sc.parallelize(massive_list)
rdd.join(massive_list_rdd).saveAsTextFile...
```

A better approach might be to use the broadcast method to create a broadcast variable, as discussed earlier in this chapter; recall that broadcast variables are distributed using an efficient peer-to-peer sharing protocol based on BitTorrent. You could also consider parallelizing larger objects, if possible. This is not meant to discourage you from passing data in functions, but you do need to be aware of how closures operate.

Considerations for Collecting Data

Two useful functions in Spark are `collect()` and `take()`. Recall that these actions trigger evaluation of an RDD, including its entire lineage. When executing `collect()`, all resultant records from the RDD return to the Driver from the Executors on which the final tasks in the lineage are executed. For large

datasets, this can be in gigabytes or terabytes of magnitude. It can create unnecessary network I/O and, in many cases, result in exceptions if there is insufficient memory on the Driver host to store the collected objects.

If you just need to inspect the output data, `take(n)` and `takeSample()` are better options. If the transformation is part of an ETL routine, the best practice is to save the dataset to a filesystem such as HDFS or a database.

The key point here is not to bring too much data back to the Driver if it's not required.

Configuration Parameters for Tuning and Optimizing Applications

In addition to application development optimizations, there are also some systemwide or platform changes that can provide substantial increases to performance and throughput. The following sections look at some of the many configuration settings that can influence performance.

Optimizing Parallelism

A specific configuration parameter that could be beneficial to set at an application level or using `spark-defaults.conf` is the `spark.default.parallelism` setting. This setting specifies the default number of RDD partitions returned by transformations such as `reduceByKey()`, `join()`, and `parallelize()` where the `numPartitions` argument is not supplied. You saw the effect of this configuration parameter earlier in this chapter.

It is often recommended to make the value for this setting *equal to* or *double* the number of cores on each Worker. As with many other settings, you may need to experiment with different values to find the optimal setting for your environment.

Dynamic Allocation

Spark's default runtime behavior is that the Executors requested or provisioned for an application are retained for the life of the application. If an application is long lived, such as a `pyspark` session or Spark Streaming application, this may

not be optimal, particularly if the Executors are idle for long periods of time and other applications are unable to get the resources they require.

With *dynamic allocation*, Executors can be released back to the cluster resource pool if they are idle for a specified period of time. Dynamic allocation is typically implemented as a system setting to help maximize use of system resources.

[Listing 5.34](#) shows the configuration parameters used to enable dynamic allocation.

Listing 5.34 Enabling Spark Dynamic Allocation

[Click here to view code image](#)

```
# enable Dynamic Allocation, which is disabled by default
spark.dynamicAllocation.enabled=True
spark.dynamicAllocation.minExecutors=n
# lower bound for the number of Executors
spark.dynamicAllocation.maxExecutors=n
# upper bound for the number of Executors
spark.dynamicAllocation.executorIdleTimeout=ns
# the time at which an Executor will be removed if it has been idle,
defaults to 60s
```

Avoiding Inefficient Partitioning

Inefficient partitioning is one of the major contributors to suboptimal performance in a distributed Spark processing environment. The following sections take a closer look at some of the common causes for inefficient partitioning.

Small Files Resulting in Too Many Small Partitions

Small partitions, or partitions containing a small amount of data, are inefficient, as they result in many small tasks. Often, the overhead of spawning these tasks is greater than the processing required to execute the tasks.

A `filter()` operation on a partitioned RDD may result in some partitions being much smaller than others. The solution to this problem is to follow the

`filter()` operation with a `repartition()` or `coalesce()` function and specify a number less than the input RDD; this combines small partitions into fewer more appropriately sized partitions.

Recall that the difference between `repartition()` and `coalesce()` is that `repartition()` always shuffles records if required, whereas `coalesce()` accepts a `shuffle` argument that can be set to `False`, avoiding a shuffle. Therefore, `coalesce()` can only reduce the number of partitions, whereas `repartition()` can increase or reduce the number of partitions.

Working with small files in a distributed filesystem results in small, inefficient partitions as well. This is especially true for filesystems such as HDFS, where blocks form the natural boundary for Spark RDD partitions created from a `textFile()` operation, for example. In such cases, a block can only associate with one file object, so a small file results in a small block, which in turn results in a small RDD partition. One option for addressing this issue is to specify the `numPartitions` argument of the `textFile()` function, which specifies how many RDD partitions to create from the input data (see [Figure 5.11](#)).

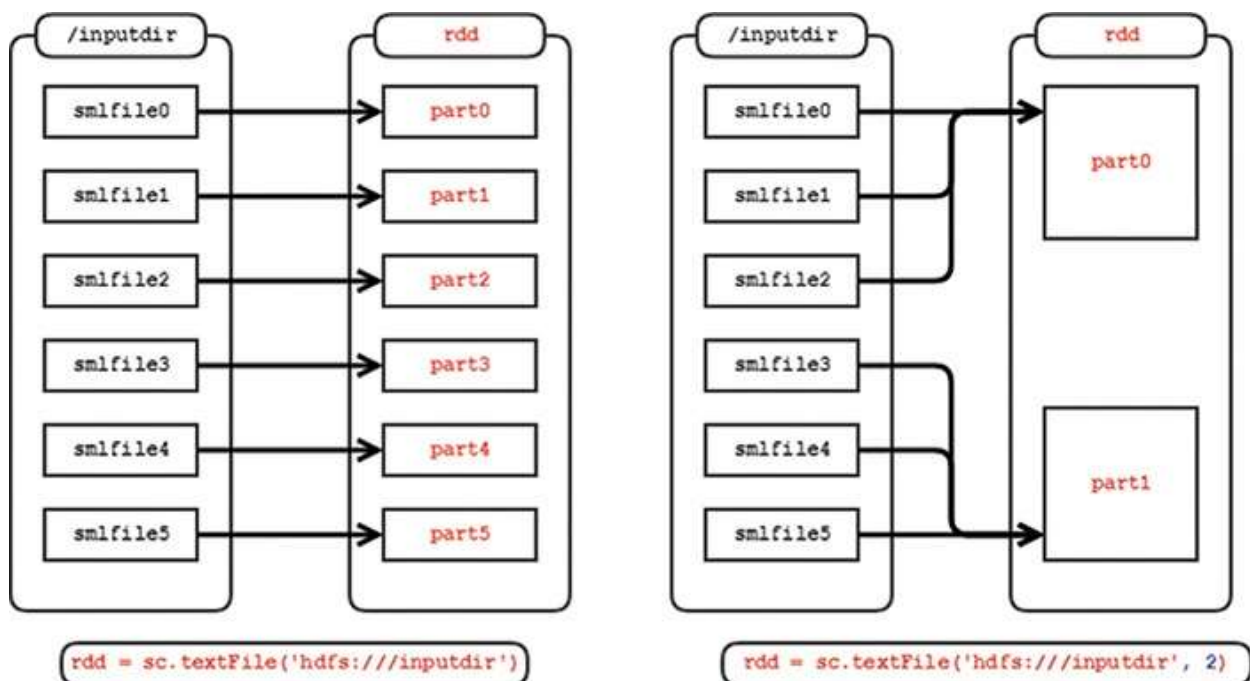


Figure 5.11 Optimizing partitions loaded from small files.

The `spark.default.parallelism` configuration property mentioned in the previous section can also be used to designate the desired number of

partitions for an RDD.

Avoiding Exceptionally Large Partitions

Exceptionally large partitions can cause performance issues. A common reason for large partitions is loading an RDD from one or more large files compressed using an unsplittable compression format such as Gzip.

Because unsplittable compressed files are not indexed and cannot be split (by definition), the entire file must be processed by one Executor. If the uncompressed data size exceeds the memory available to the Executor, the partition may spill to disk, causing performance issues.

Solutions to this problem include the following:

- Avoid using unsplittable compression, if possible.
- Uncompress each file locally (for example, to `/tmp`) before loading the file into an RDD.
- Repartition immediately after the first transformation against the RDD.

Moreover, large partitions can also result from a shuffle operation using a custom partitioner, such as a month partitioner for a corpus of log data where one month is disproportionately larger than the others. In this case, the solution is to use `repartition()` or `coalesce()` after the reduce operation, using a hash partitioner.

Another good practice is to repartition before a large shuffle operation as this can provide a significant performance benefit.

Determining the Right Number or Size of Partitions

Generally, if you have fewer partitions than Executors, some of the Executors will be idle. However, the optimal, or “Goldilocks,” number or size for partitions is often found only by trial and error. A good practice is to make this an input parameter (or parameters) to your program so you can easily experiment with different values and see what works best for your system or your application.

Diagnosing Application Performance Issues

You have seen many application development practices and programming

techniques in this chapter and throughout the book that can provide significant performance improvement. This section provides a simple introduction to identifying potential performance bottlenecks in your application so you can address them.

Using the Application UI to Diagnose Performance Issues

The Spark application UI that you have seen throughout this book is probably the most valuable source of information about application performance. The application UI contains detailed information and metrics about tasks, stages, scheduling, storage, and more to help you diagnose performance issues. Recall from our discussions that the application UI is served on port 4040 (or successive ports if more than one application is running) of the host running the Driver for the application. For YARN clusters, the application UI is available via the ApplicationMaster link in the YARN ResourceManager UI. The following sections take a further look at how you can identify various performance issues using the application UI.

Shuffle and Task Execution Performance

Recall that an application consists of one or more jobs, as a result of an action such as `saveAsTextFile()`, `collect()`, or `count()`. A job consists of one or more stages that consist of one or more tasks. Tasks operate against an RDD partition. The first place to look when diagnosing performance issues is the stage summary from the Stages tab of the application UI. On this tab, you can see the duration of each stage as well as the amount of data shuffled, as shown in Figure 5.12.

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	count at <stdin>:2 +details	2017/09/28 04:30:22	3 s	2/2			20.7 MB	
0	join at <stdin>:1 +details	2017/09/28 04:30:13	10 s	2/2	25.7 MB			20.7 MB

Figure 5.12 Spark application UI stage summary.

By clicking on a stage in the Description column of the Completed Stages table, you can see details for that stage, including the duration and write time for each task in the stage. This is where you may see disparity in the values of different tasks, as shown in Figure 5.13.

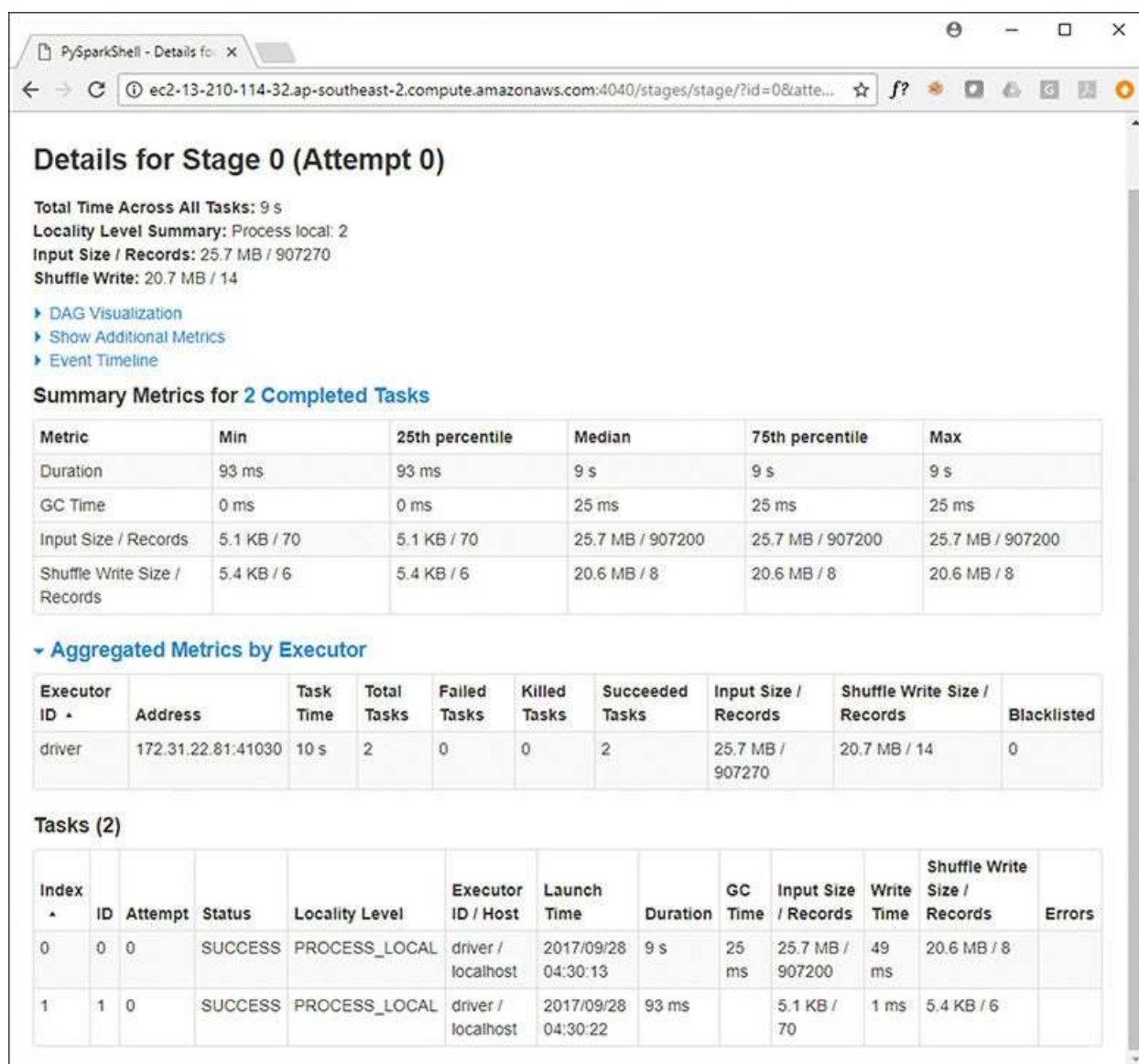


Figure 5.13 Spark application UI stage detail.

The difference in task durations or write times may be an indication of inefficient partitioning, as discussed in the previous section.

Collection Performance

If your program has a collection stage, you can get summary and detailed performance information from the Spark application UI. From the Details page, you can see metrics related to the collection process, including the data size collected, as well as the duration of collection tasks; this is shown in Figure 5.14.

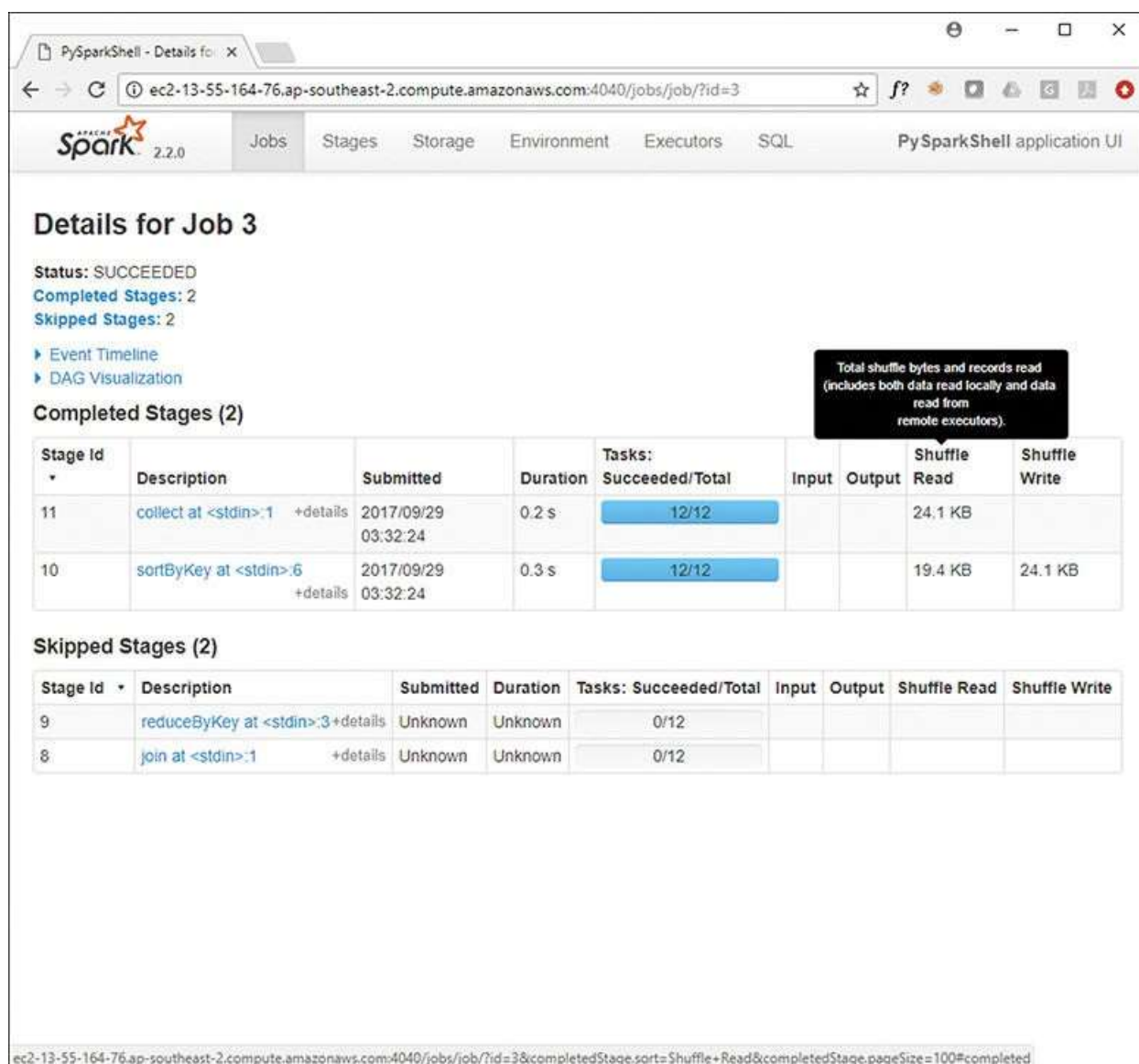
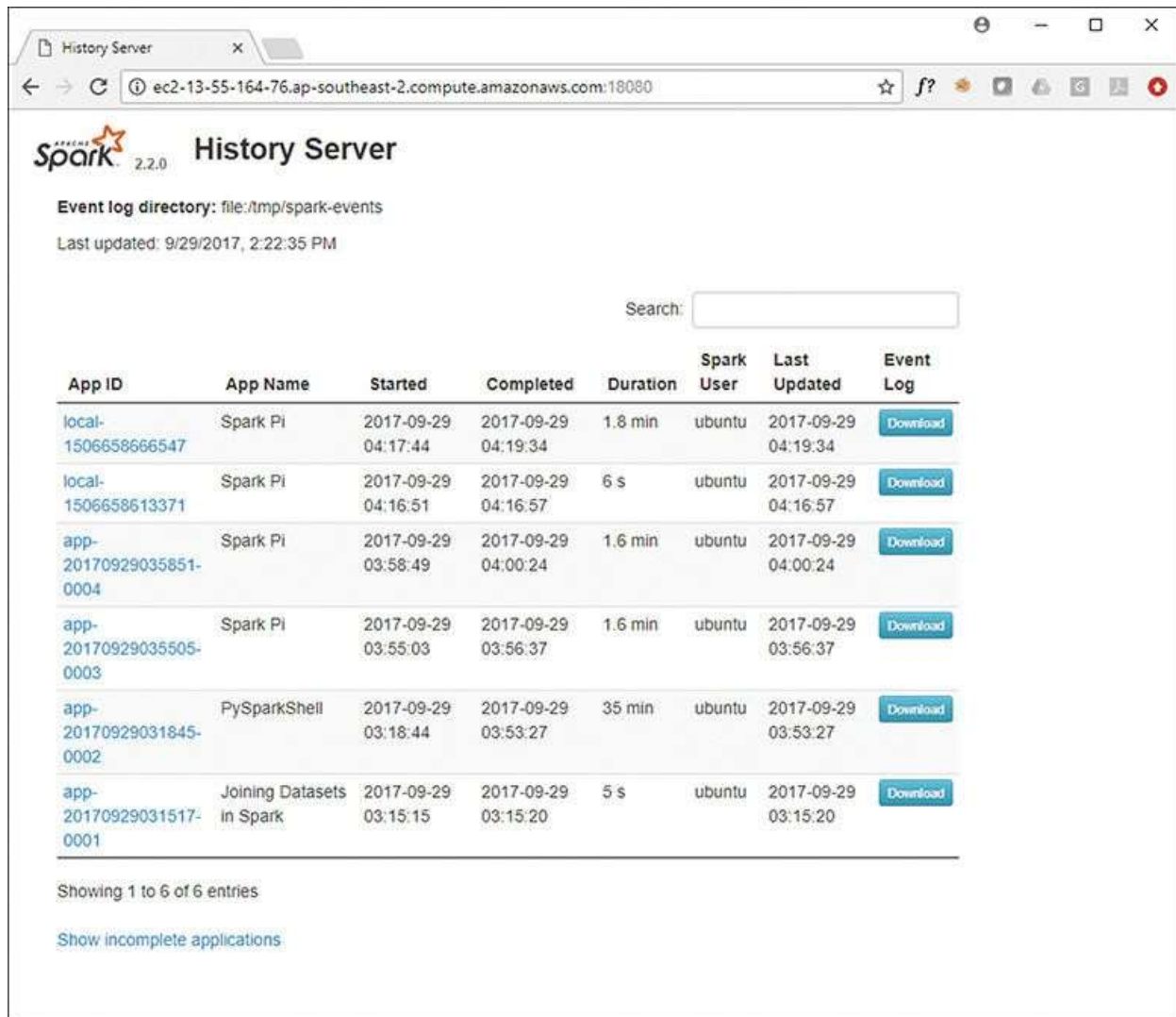


Figure 5.14 Spark application UI stage detail: collection information.

Using the Spark History UI to Diagnose Performance Issues

The application UI (served on port 404x) is available only during an application's lifetime, which makes it handy for diagnosing issues with running applications. It's useful and sometimes necessary to profile the performance of completed applications, successful or otherwise, as well. The *Spark History Server* provides the same information as the application UI for completed applications. Moreover, you can often use completed application information in the Spark History Server as an indicative benchmark for the same applications that are currently running. Figure 5.15 shows an example of the Spark History

Server UI, typically served on port 18080 of the host running this process.



App ID	App Name	Started	Completed	Duration	Spark User	Last Updated	Event Log
local-150665866547	Spark Pi	2017-09-29 04:17:44	2017-09-29 04:19:34	1.8 min	ubuntu	2017-09-29 04:19:34	Download
local-1506658613371	Spark Pi	2017-09-29 04:16:51	2017-09-29 04:16:57	6 s	ubuntu	2017-09-29 04:16:57	Download
app-20170929035851-0004	Spark Pi	2017-09-29 03:58:49	2017-09-29 04:00:24	1.6 min	ubuntu	2017-09-29 04:00:24	Download
app-20170929035505-0003	Spark Pi	2017-09-29 03:55:03	2017-09-29 03:56:37	1.6 min	ubuntu	2017-09-29 03:56:37	Download
app-20170929031845-0002	PySparkShell	2017-09-29 03:18:44	2017-09-29 03:53:27	35 min	ubuntu	2017-09-29 03:53:27	Download
app-20170929031517-0001	Joining Datasets in Spark	2017-09-29 03:15:15	2017-09-29 03:15:20	5 s	ubuntu	2017-09-29 03:15:20	Download

Showing 1 to 6 of 6 entries

[Show incomplete applications](#)

Figure 5.15 Spark History Server.

Summary

This chapter completes our coverage of the Spark core (or RDD) API using Python. This chapter introduces the different shared variables available in the Spark API, including broadcast variables and accumulators, along with their purpose and usage. Broadcast variables are useful for distributing reference information, such as lookup tables, to Workers to avoid expensive “reduce side” joins. Accumulators are useful as general-purpose counters in Spark applications and also can be used to optimize processing. This chapter also discusses RDD

partitioning in much more detail, as well as the methods available for repartitioning RDDs, including `repartition()` and `coalesce()`, as well as functions designed to work on partitions atomically, such as `mapPartitions()`. This chapter also looks at the behavior of partitioning and its influence on performance as well as RDD storage options. You have learned about the effects of checkpointing RDDs, which is especially useful for periodic saving of state for iterative algorithms, where elongated lineage can make recovery very expensive. In addition, you have learned about the `pipe()` function, which you can use with external programs with Spark. Finally, you got a look at how to sample data in Spark and explored some considerations for optimizing Spark programs.