Ch 12 Resilient Distributed Datasets (RDDs)

There are two sets of low-level APIs:

- there is one for manipulating **distributed data** (RDDs)
- another for distributing and manipulating distributed **shared variables** (*broadcast variables and accumulators*)

use the lower-level APIs in three situations:

- You need some functionality that you cannot find in the higher-level APIs;
  - for example, if you need very tight control over physical data placement across the cluster.
- You need to maintain some legacy codebase written using RDDs
- You need to do some custom shared variable manipulation.

all Spark workloads compile down to these fundamental primitives

You might need to drop down to these APIs to use some legacy code, implement some custom partitioner, or update and track the value of a variable over the course of a data pipeline's execution

A **SparkContext** is the entry point for low-level API functionality

- You access it through the **SparkSession**, which is the tool you use to perform computation across a Spark cluster.

```
spark.sparkContext
```

RDDs

- RDDs were the primary API in the Spark 1.X series and are still available in 2.X, but they are not as commonly used.
- all Spark code you run, whether DataFrames or Datasets, compiles down to an RDD
- The Spark UI, covered in the next part of the book, also describes job execution in terms of RDDs
- RDD represents an immutable, partitioned collection of records that can be operated on in parallel
- DataFrames have each record as *a structured row containing fields with a known schema*
- RDDs the records are just Java, Scala, or Python objects of the programmer's choosing
- RDDs give you complete control because every record in an RDD is a just a Java or Python **object**
- Store anything you want in these objects, in any format you want.
- Every manipulation and interaction between values **must be defined by hand**
- Must "reinvent the wheel" for any task you are trying to carry out
- Also, optimizations are going to require much more manual work, because Spark does not understand the inner structure of your records as it does with the Structured APIs.
  - Spark's Structured APIs automatically store data in an optimized, compressed binary format, so to achieve the same space-efficiency and performance, you'd also need to implement this type of format inside your objects and all the low-level operations to compute over it

format inside your objects and all the low-level operations to compute over it
    - Optimizations like reordering filters and aggregations that occur automatically in Spark SQL need to be implemented by hand.
- **Use the Spark Structured APIs when possible**
- The RDD API is similar to the Dataset, but RDDs are not stored in, or manipulated with, the structured data engine.
- Easy to convert between RDDs and Datasets, so you can use both APIs to take advantage of each

lots of subclasses of RDD

- these are internal representations that the DataFrame API uses to create optimized physical execution plans
- Usually two types of RDDs:
    - the "generic" RDD type
    - key-value RDD that provides additional functions, such as aggregating by key.
- Each represent a collection of objects, but key-value RDDs have special operations as well as a concept of *custom partitioning by key*

Each RDD is characterized by five main properties:

- list of partitions
- A function for computing each split
- A list of dependencies on other RDDs
- *Optional* - **Partitioner** for key-value RDDs (e.g., *to say that the RDD is hash-partitioned*)
- *Optional* - a **list of preferred locations** on which to compute each split (e.g., *block locations for a Hadoop Distributed File System [HDFS] file*)

Partitioner is probably one of the core reasons why you might want to use RDDs in your code.

- Specifying your own custom Partitioner can give you *significant performance and stability improvements*

RDDs provide **transformations**, which evaluate lazily, and **actions**, which evaluate eagerly, to manipulate data in a distributed fashion.

- However, there is **no concept of "rows"** in RDDs; individual records are just raw Java/Scala/Python objects, and you manipulate those manually instead of tapping into the repository of functions that you have in the structured APIs

Python can lose a substantial amount of performance when using RDDs.

- Running Python RDDs equates to running Python user-defined functions (UDFs) row by row
- serialize the data to the Python process, operate on it in Python, and then serialize it back to the Java Virtual Machine (JVM)
- Causes a high overhead for Python RDD manipulations.
- Recommend building on the Structured APIs in Python and only dropping down to RDDs if absolutely

necessary

Should not manually create RDDs unless you have a very, very specific reason for doing so. They are a much lower-level API that provides a lot of power but also lacks a lot of the optimizations that are available in the Structured APIs.

- For the vast majority of use cases, *DataFrames will be more efficient, more stable, and more expressive* than RDDs.
- Only want to use RDDs is because you need fine-grained control over the physical distribution of data (custom partitioning of data)

Interoperating Between DataFrames, Datasets, and RDD

create simple RDD :

```
spark.range(10).rdd
```

convert this Row-type object to the correct data type or extract values out of it:

```
spark.range(10).toDF("id").rdd.map(lambdarow:row[0])
```

create a DataFrame or Dataset from an RDD with the toDF method on the RDD

To create an RDD from a collection

- you will need to use the **parallelize** method on a SparkContext (within a SparkSession) - This turns a single node collection into a parallel collection.
- creating this parallel collection, you can also explicitly state the number of partitions into which you would like to distribute this array.

Creating two partitions:

```
myCollection = "Spark The Definitive Guide : Big Data Processing Made Simple"\
  .split(" ")
words = spark.sparkContext.parallelize(myCollection, 2)
```

name this RDD to show up in the Spark UI according to a given name:

```
words.setName("myWords")
words.name() # myWords
```

Can create RDDs from data sources or text files

- it's often preferable to use theData Source APIs
- RDDs do not have a notion of "Data Source APIs" like DataFrames do
- define their dependency structures and lists of partitions
- **The Data Source API is almost always a better way to read in data**.
- Can also read data as RDDs using sparkContext

Read a text file line by line:

```
spark.sparkContext.textFile("/some/path/withTextFiles")
```

- Creates an RDD for which each record in the RDD represents a line in that text file or files.
- Alternatively, you can read in data for which each text file should become a single record.

The use case here would be where each file is a file that consists of a large JSON object or some document that you will operate on as an individual:

```
spark.sparkContext.wholeTextFiles("/some/path/withTextFiles")
```

No "helper" methods or functions that you can draw upon to simplify calculations - must define each filter, map functions, aggregation, and any other manipulation that you want as a function

Distinct method call on an RDD removes duplicates from the RDD

```
words.distinct().count()
```

Filtering

- equivalent to creating a SQL-like where clause
- You can look through our records in the RDD and see which ones match some predicate function
- Needs to return a Boolean type to be used as a filter function.
- The input should be whatever your given row is

In this next example, we filter the RDD to keep only the words that begin with the letter "S":

```
def startsWithS(individual):
  return individual.startswith("S")
```

run on each record in the RDD individually:

```
words.filter(lambda word: startsWithS(word)).collect()
```

Mapping

- specify a function that returns the value that you want, given the correct input
- apply that, record by record.

Map the current word to the word, its starting letter, and whether the word begins with "S" - defining our functions completely inline using the relevant lambda syntax:

```
words2 = words.map(lambda word: (word, word[0], word.startswith("S")))
```

filter on this by selecting the relevant Boolean value in a new function:

```
words2.filter(lambda record: record[2]).take(5)
```

flatMap

- each current row should return multiple rows
- might want to take your set of words and flatMap it into a set of characters
- Because each word has multiple characters, you should use flatMap to expand it
- flatMap requires that the ouput of the map function **be an iterable that can be expanded**

```
words.flatMap(lambda word: list(word)).take(5)
```

Sort an RDD you must use the sortBy method

- specify a function to extract a value from the objects in your RDDs
- sort based on that

sorts by word length from longest to shortest:

```
words.sortBy(lambda word: len(word) * -1).take(10)
```

randomly split an RDD into an Array of RDDs by using the **randomSplit** method

- which accepts an **Array of weights** and a **random seed**

```
fiftyFiftySplit=words.randomSplit([0.5,0.5])
```

reduce method to specify a function to “reduce” an RDD of any kind of value to **one value**

- Given a set of numbers, you can reduce this to its sum by specifying a function that takes as input two values and reduces them into one:

```
spark.sparkContext.parallelize(range(1, 21)).reduce(lambda x, y: x + y)
```

Uses function to reduce to single value

- This reducer is a good example because you can get one of two outputs.
- Because the reduce operation on the partitions is not deterministic, you can have either “definitive” or “processing”(both of length 10) as the “left” word.
- This means that sometimes you can end up with one, whereas other times you end up with the other

```
def wordLengthReducer(leftWord, rightWord):
  if len(leftWord) > len(rightWord):
    return leftWord
  else:
```

```
    return rightWord

words.reduce(wordLengthReducer)
```

countApprox

- an approximation of the count method - but it *must execute within a timeout*
- can return incomplete results if it exceeds the timeout
- The **confidence** is th*e probability that the error bounds of the result will contain the true value*
- if countApprox were called repeatedly with confidence 0.9, we would expect 90% of the results to contain the true count
- The confidence must be in the range [0,1], or an exception will be thrown

```
confidence=0.95
timeoutMilliseconds=400
words.countApprox(timeoutMilliseconds,confidence)
```

countApproxDistinct

- based on streamlib's implementation of"*HyperLogLog in Practice: Algorithmic Engineering of a State-of-the-Art Cardinality EstimationAlgorithm*."
- In the first implementation, the argument we pass into the function is the relative accuracy.Smaller values create counters that require more space. The value must be greater than 0.000017:

```
words.countApproxDistinct(0.05)
```

- other implementation you have a bit more control; you specify the relative accuracy based on two parameters:
  - one for "regular" data
  - another for a sparse representation
- The two arguments are p and sp where p is **precision** and sp is **sparse precision**
- The relative accuracy is approximately *1.054 / sqrt(2).*
- Setting a nonzero (sp > p) can reduce the memory consumption and increase accuracy when the cardinality is small. Both values areintegers:

```
words.countApproxDistinct(4,10)
```

countByValue

- counts the number of values in a given RDD
- does so by finally loading the result set into the memory of the driver.
- [illegible]

- use this method only *if the resulting map is expected to be small* because the entire thing is loaded into **the driver's memory**
- method makes sense only in a scenario in which either the total number of rows is low or the number of distinct items is low:

```
words.countByValue()
```

countByValueApprox

- same thing as the previous function, but as an approximation
- must execute within the specified timeout (first parameter) (and can return incomplete results if it exceeds the timeout).
- The confidence is the probability that the error bounds of the result will contain the true value.
- That is, if countApprox were called repeatedly with confidence 0.9, we would expect 90% of the results to contain the true count.
- The confidence must be in the range [0,1], or an exception will be thrown

```
words.countByValueApprox(1000,0.95)
```

```
words.first()
```

```
spark.sparkContext.parallelize(1to20).max()
spark.sparkContext.parallelize(1to20).min()
```

take

- derivative methods take a number of values from your RDD
- works by first scanning one partition and then using the results from that partition to estimate the number of additional partitions needed to satisfy the limit
- There are many variations on this function, such as **takeOrdered**, **takeSample**, and **top**
- **takeSample -** specify a fixed-size random sample from your RDD. Y
  - can specify whether this should be done by using **withReplacement**, the **number of values**, as well as the random **seed**
- **top** is effectively the opposite of **takeOrdered** in that it selects the top values according to the implicit ordering

```
words.take(5)
words.takeOrdered(5)
words.top(5)
withReplacement=True
numberToTake=6
randomSeed=100
words.takeSample(withReplacement,numberToTake,randomSeed)
```

Saving files

- means writing to plain-text files
- cannot actually “save” to a datasource in the conventional sense
- You must iterate over the partitions in order to save the contents of each partition to some external database
- This is a low-level approach that reveals the underlying operation that is being performed in the higher-level APIs
- Spark will take each partition, and write that out to the destination.saveAsTextFileTo save to a text file, you just specify a path and optionally a compression codec:

```
words.saveAsTextFile("file:/tmp/bookTitle")
```

sequenceFile is a flat file consisting of binary key–value pairs.

- extensively used in MapReduce as input/output formats
- write to sequenceFiles using the **saveAsObjectFile** method or by explicitly writing key–value pairs

```
words.saveAsObjectFile("/tmp/my/sequenceFilePath")
```

caching RDDs

- Can either cache or persist an RDD
- By default, cache and persist only handle data in memory
- We can name it if we use the **setName** function that we referenced previously in this chapter:

words.cache()

specify a storage level

- as any of the storage levels in the singleton object: **org.apache.spark.storage.StorageLevel**
- which are combinations of memory only; disk only; and separately, off heap

can subsequently query for this storage levels (important relative to persistence)

```
words.getStorageLevel()
```

Checkpointing

- is the act of saving an RDD to disk so that future references to this RDD point to those intermediate partitions on disk
- Rather than recomputing the RDD from its original source
- Similar to caching except that it’s not stored in memory, only disk
- This can be helpful when performing iterative computation, similar to the use cases for caching
- When we reference this RDD, it will derive from the checkpoint instead of the source data.

```
spark.sparkContext.setCheckpointDir("/some/path/for/checkpointing")
words.checkpoint()
```

pipe

- return an RDD created by piping elements to a forked external process
- resulting RDD is computed by executing the given process once per partition
- All elements of each input partition are written to a process's stdin as lines of input separated by a newline
- resulting partition consists of the process's stdout output, with each line of stdout resulting in one element of the output partition
- A process is invoked even for empty partitions

The print behavior can be customized by providing two functions. We can use a simple example and pipe each partition to the command **wc**.

- Each row will be passed in as a new line, so if we perform a line count, we will get the number of lines, one per partition:

```
#In this case, we got five lines per partition
words.pipe("wc -l").collect()
```

mapPartitions

- The previous command revealed that Spark operates on a per-partition basis when it comes to actually executing code
- You also might have noticed earlier that the return signature of a map function on an RDD is actually MapPartitionsRDD
- This is because map is just a row-wise alias for mapPartitions, which makes it possible for you to map an individual partition - represented as an iterator
- That's because physically on the cluster we operate on each partition individually - not a specific row
- Needs a return value to work properly

A simple example creates the value "1" for every partition in our data, and the sum of the following expression will count the number of partitions we have:

```
words.mapPartitions(lambda part: [1]).sum() # 2
```

foreachPartition

- iterates over all the partitions of the data - but the function has no return value
- great for doing something with each partition like writing it out to a database
- **This is how many data source connectors are written**

Can create our own text file source if you want by specifying outputs to the temp directory with a random ID:

```
def indexedFunc(partitionIndex, withinPartIterator):
  return ["partition: {} => {}".format(partitionIndex,
    x) for x in withinPartIterator]
```

```
words.mapPartitionsWithIndex(indexedFunc).collect()
```

glom

- takes every partition in your dataset and converts them to arrays
- can be useful if you're going to collect the data to the driver and want to have an array for each partition
- can cause serious stability issues because if you have large partitions or a large number of partitions
- easy to crash the driver

In the following example, you can see that we get two partitions and each word falls into one partition each:

```
spark.sparkContext.parallelize(["Hello", "World"], 2).glom().collect()
# [['Hello'], ['World']]
```