Big Data Analytics with Apache Spark

Nastaran Fatemi

Apache Spark

Throughout this part of the course we will use the **Apache Spark** framework for distributed data-parallel programming.

Spark implements a distributed data parallel model called **Resilient Distributed Datasets (RDDs)**

Distributed Data-Parallel Programming with Spark

So far we have seen:

- Principles of programming in Scala in sequential and non-distributed style.
- A number of useful tools and technologies based on Scala.

Scala also offers parallel programming via "parallel collections", allowing to use collections in a very similar way that we have learned so far but taking advantage of data parallelism, hence better exploiting multi-core/ multi-processor architecture.

Although this aspect will not be treated in the current course.

What we'll see in this course:

- Data parallelism in a *distributed setting*.
- Distributed collections abstraction from Apache Spark as an implementation of this paradigm.

Distribution

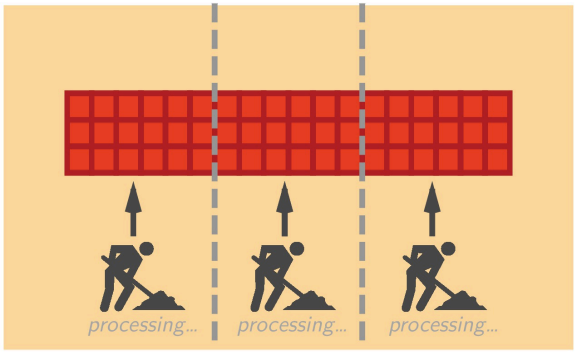
Distribution introduces important concerns beyond what we have to worry about when dealing with parallelism in the shared memory case:

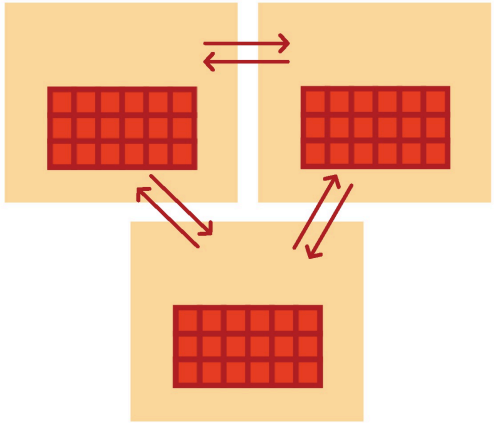
- *Partial failure:* crash failures of a subset of the machines involved in a distributed computation.
- *Latency:* certain operations have a much higher latency than other operations due to network communication.

Latency cannot be masked completely; it will be an important aspect that also impacts the programming model.

Data-Parallel to Distributed Data-Parallel

- What does **distributed** data-parallel look like?





Shared memory case:

Data-parallel programming model.
Data partitioned in memory and operated upon in parallel.

Distributed case:

Data-parallel programming model.
Data partitioned between machines, network in between, operated upon in parallel.

Spark Overview

Goal: Easily work with large scale data in terms of transformations on distributed data

- Traditional distributed computing platforms scale well but have limited API (map/reduce)
- Spark lets you tackle problems too big for a single machine
- Spark has an expressive data focused API which makes writing large scale programs easy

Scala vs Java vs Python

Spark was originally written in Scala, which allows concise function syntax and interactive use

But it also provides Java and Python API for standalone applications and interactive shell.

Resilient Distributed Datasets (RDDs)

RDDs look just like ***immutable*** Scala collections.

Combinators on Scala collections	**Combinators on RDDs:**
map	map
flatMap	flatMap
filter	filter
reduce	reduce
fold	fold
aggregate	aggregate

Combinators on RDDs

While their signatures differ a bit, their semantics (macroscopically) are the same:

```
map[B](f: A => B): List[B] // Scala List
map[B](f: A => B): RDD[B] // Spark RDD

flatMap[B](f: A => TraversableOnce[B]): List[B] // Scala List
flatMap[B](f: A => TraversableOnce[B]): RDD[B] // Spark RDD

filter(pred: A => Boolean): List[A] // Scala List
filter(pred: A => Boolean): RDD[A] // Spark RDD
```

Combinators on RDDs

While their signatures differ a bit, their semantics (macroscopically) are the same:

```
reduce(op: (A, A) => A): A // Scala List
reduce(op: (A, A) => A): A // Spark RDD


fold(z: A)(op: (A, A) => A): A // Scala List
fold(z: A)(op: (A, A) => A): A // Spark RDD


aggregate[B](z: => B)(seqop: (B, A) => B, combop: (B, B) => B): B //
   Scala
aggregate[B](z: B)(seqop: (B, A) => B, combop: (B, B) => B): B // Spark
   RDD
```

Using RDDs

Using RDDs in Spark feels a lot like normal Scala collections, with the added knowledge that your data is distributed across several machines.

Example: Given, `val encyclopedia: RDD[String]`, say we want to search all of encyclopedia for mentions of HEIG-VD, and count the number of pages that mention HEIG-VD.

```
val result = encyclopedia.filter(page => page.contains("HEIG-VD"))
                         .count()
```

Example: Word Count

The “Hello, World!” of programming with large-scale data.

```
// Create an RDD
// The textFile() method takes an URI for the file (either a local
   path on the machine, or a hdfs://, etc.) and reads it as a
   collection of lines.


val rdd = spark.textFile("hdfs://...")
```

Example: Word Count

The “Hello, World!” of programming with large-scale data.

```
// Create an RDD
val rdd = spark.textFile("hdfs://...")


val count = rdd.flatMap(line => line.split(" ")) // separate lines
                                                 // into words
```

Example: Word Count

The “Hello, World!” of programming with large-scale data.

```
// Create an RDD
val rdd = spark.textFile("hdfs://...")


val count = rdd.flatMap(line => line.split(" ")) // separate lines
                                                 // into words
              .map(word => (word, 1))            // include something
                                                 // to count
```

Example: Word Count

The “Hello, World!” of programming with large-scale data.

```
// Create an RDD
val rdd = spark.textFile("hdfs://...")


val count = rdd.flatMap(line => line.split(" ")) // separate lines
                                                 // into words
              .map(word => (word, 1))            // include something
                                                 // to count
              .reduceByKey(_ + _)                // sum up the 1s in
                                                 // the pairs
```

Transformations and Actions

Collection methods that you have already seen in the Scala standard library are divided into two major groups:

- **Transformers:** Return new collections as results. (Not single values)
  - **Examples:** `map, filter, flatMap, groupBy`

```
map(f: A => B): Traversable[B]
```

- **Accessors:** Return single values as results. (Not collections)
  - **Examples:** `reduce, fold, aggregate`

```
reduce(op: (A, A) => A): A
```

Transformations and Actions

Similarly, Spark defines ***transformations*** and ***actions*** on RDDs.

They seem similar to transformers and accessors, but there are some important differences.

- **Transformations.** Return new **RDDs** as results.

- **Actions.** Compute a result based on an RDD, and either return it or save it to an external storage system (e.g., HDFS).

Transformations and Actions

- Transformations are *lazy*, their result RDD is not immediately computed.
- Actions are *eager*, their result is immediately computed.

Laziness/eagerness is how we can limit network communication using the programming model.

Quiz

- Consider the following simple example:

```
val largeList: List[String] = ...
val wordsRdd = sc.parallelize(largeList)
val lengthsRdd = wordsRdd.map(_.length)
```

What has happened on the cluster at this point?

A. The lengths of the strings in the RDD are calculated on the worker nodes
B. The lengths of the strings in the RDD are calculated on the worker nodes and returned to the master node
C. The sum of the lengths of the strings in the RDD is calculated by the master node
D. Nothing

Example

- Consider the following simple example:

```
val largeList: List[String] = ...
val wordsRdd = sc.parallelize(largeList)
val lengthsRdd = wordsRdd.map(_.size)
val totalChars = lengthsRdd.reduce(_ + _)
```

... we can add an action!

Cluster Topology Matters

If you perform an action on an RDD, on what machine is its result "returned" to?

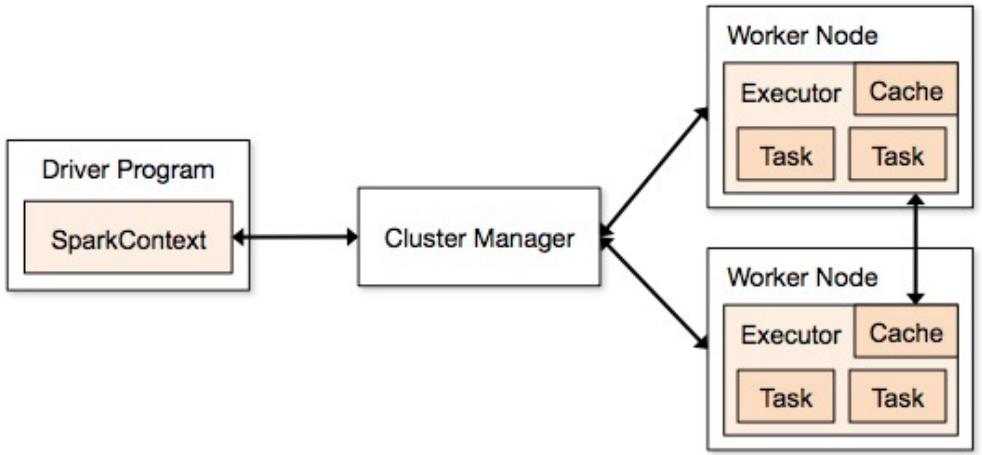
- **Example:**

```
val people: RDD[Person] = ...
val first10 = people.take(10)
```

Where will the Array[Person] representing first10 end up?

Execution of Spark Programs

A Spark application is run using a set of processes on a cluster. All these processes are coordinated by the ***driver program***.



1.The driver program runs the Spark application, which creates a SparkContext upon start-up.

2.The SparkContext connects to a cluster manager (e.g., Mesos/YARN) which allocates resources.

3.Spark acquires ***executors*** on nodes in the cluster, which are processes that run computations and store data for your application.

4.Next, driver program sends your application code to the executors.

5.Finally, SparkContext sends ***tasks*** for the executors to run.

Cluster Topology Matters

If you perform an action on an RDD, on what machine is its result "returned" to?

Example

```
val people: RDD[Person] = ...
val first10 = people.take(10)
```

Where will the Array[Person] representing first10 end up?
The driver program.

In general, executing an action involves communication between worker nodes and the node running the driver program.

Benefits of Laziness for Large-Scale Data

Spark computes RDDs the first time they are used in an action.

This helps when processing large amounts of data.

Example

```
val lastYearsLogs: RDD[String] = ...
val firstLogsWithErrors =
        lastYearsLogs.filter(_.contains("ERROR")).take(10)
```

In the above example, lazy execution of `filter` *makes a big difference.*

The execution of `filter` is deferred until the `take` action is applied.

Spark leverages this by analyzing and optimizing the **chain of operations** before executing it.

Spark will not compute intermediate RDDs. Instead, as soon as 10 elements of the filtered RDD have been computed, `firstLogsWithErrors` is done. At this point Spark stops working, saving time and space computing elements of the unused result of filter.

Benefits of Laziness for Large-Scale Data

Spark uses **lazy evaluation** to reduce the number of passes it has to take over our data by **grouping operations together**.

In systems like Hadoop MapReduce, developers often have to spend a lot of time considering how to group together operations to minimize the number of MapReduce passes.

In Spark, there is no substantial benefit to writing a single complex map instead of chaining together many simple operations. Thus, users are free to organize their program into smaller, more manageable operations.

Caching and Persistence

By default, RDDs are recomputed each time you run an action on them. This can be expensive (in time) if you need to traverse a dataset more than once.

Example

```
val lastYearsLogs: RDD[String] = ...
val logsWithErrors =
   lastYearsLogs.filter(_.contains("ERROR")).persist()
val firstLogsWithErrors = logsWithErrors.take(10)
```

Here, we *cache* logsWithErrors in memory.

After firstLogsWithErrors is computed, Spark will store the contents of logsWithErrors for faster access in future operations if we would like to reuse it.

```
val numErrors = logsWithErrors.count() // faster
```

Now, computing the count **on** logsWithErrors **is much faster.**

Other Important RDD Transformations

Beyond the transformer-like combinators you may be familiar with from Scala collections, RDDs introduce a number of other important transformations.

sample	Sample a fraction of the data, with or without replacement, using a given random number generator seed.
union	Return a new dataset that contains the union of the elements in the source dataset and the argument. Pseudo-set operations (duplicates remain).
intersection	Return a new RDD that contains the intersection of elements in the source dataset and the argument. Pseudo-set operations (duplicates remain).

Other Important RDD Transformations (2)

distinct	Return a new dataset that contains the distinct elements of the source dataset.
coalesce	Decrease the number of partitions in the RDD to *numPartitions*. Useful for running operations more efficiently after filtering down a large dataset.
repartition	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.

Other Important RDD Actions

RDDs also contain other important actions which are useful when dealing with distributed data.

collect	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count	Return the number of elements in the dataset.
foreach	Run a function *func* on each element of the dataset. This is usually done for side effects such as interacting with external storage systems.
saveAsTextFile	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.

Reduction operations: reduce, fold and aggregate

Spark provides a subset of the reduction operations of Scala.

In fact, not all reduction operations are parallelizable.

Three of the most important reduction operations in Spark are : `reduce`, `fold`, and `aggregate`.

The most common action on basic RDDs you will likely use is `reduce(func)`, which takes a function that operates on two elements of the type in your RDD and returns a new element of the same type.

Please note that any function `func` you provide, should be commutative and associative.

Example:

```
val a = sc.parallelize(1 to 100)
a.reduce(_ + _)
res1: Int = 5050
```

Reduction operations (2)

`fold (zero)(func)` also takes a function with the same signature as needed for reduce(), but in addition takes a “zero value” to be used for the initial call on each partition.

The zero value you provide should be the identity element for your operation; that is, applying it multiple times with your function should not change the value (e.g., 0 for +, 1 for *, or an empty list for concatenation).

Example

```
val a = sc.parallelize(1 to 100)
a.fold(0)(_ + _)
res2: Int = 5050
```

Reduction operations (3)

Both `fold()` and `reduce()` require that the return type of our result be the same type as that of the elements in the RDD we are operating over.

This works well for operations like `sum`, but sometimes we want to return a different type.

For example, when computing a running average, we need to keep track of both the count so far and the number of elements, which requires us to return a **pair**.

We could work around this by first using `map()` where we transform every element into the element and the number 1, which is the type we want to return, so that the `reduce()` function can work on pairs.

The `aggregate()` function frees us from the constraint of having the return be the same type as the RDD we are working on.

Reduction operations (4)

`Aggregate (zeroValue)(seqOp, combOp)` is similar to `reduce` but it is used to return a different type.

It allows the user to apply **two** different reduce functions to the RDD.

- The first function is applied within each partition to reduce the data within each partition [T] in a local aggregate [U] and takes the form: (U,T) => U. You can see it as a fold and therefore it also requires a zero for that operation. This operation is applied locally to each partition in parallel.
- The second operation takes two values of the result type of the previous operation [U] and combines them into one value. This operation will reduce the partial results of each partition to produce one final result.

Example

```
val a = sc.parallelize(1 to 100)
a.aggregate((0,0))((acc,value) =>(acc._1 + value, acc._2 + 1),
                   (acc1,acc2) =>(acc1._1 + acc2._1, acc1._2 + acc2._2))
res6: (Int, Int) = (5050,100)
```

Now let's use Spark

First download Spark from:
http://spark.apache.org/downloads.html

And then open the archive and goto inside Spark directory and run

```
./bin/run-example SparkPi 10
```

It should compute Pi without any error message.

Spark runs on both Windows and UNIX-like systems (e.g. Linux, Mac OS). It's easy to run locally on one machine - all you need is to have java installed on your system PATH, or the JAVA_HOME environment variable pointing to a Java installation.

Spark runs on Java 7+, Python 2.6+ and R 3.1+. For the Scala API, Spark 1.5.1 uses Scala 2.10. You will need to use a compatible Scala version (2.10.x).

Part1: Interactive Analysis with the Spark Shell

We use Spark shell to learn the framework. Run following command from Spark home directory:

```
./bin/spark-shell
```

We are going to process README.md file of Spark. Run following commands on Spark shell and explain each line and show the results:

```
scala> val textFile = sc.textFile("README.md")

scala> textFile.count

scala> textFile.first

scala> val linesWithSpark = textFile.filter(line =>
                                line.contains("Spark"))

scala> textFile.filter(line => line.contains("Spark")).count
```

Part1: Interactive Analysis with the Spark Shell

We use Spark shell to learn the framework. Run following command from Spark home directory:

```
./bin/spark-shell
```

We are going to process README.md file of Spark. Run following commands on Spark shell and explain each line and show the results:

```
scala> val textFile = sc.textFile("README.md")

scala> textFile.count

scala> textFile.first

scala> val linesWithSpark = textFile.filter(line =>
                                line.contains("Spark"))

scala> textFile.filter(line => line.contains("Spark")).count
```

Part2: Self-Contained Applications

Now we create a very simple Spark application in Scala

```
/* SimpleApp.scala */
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object SimpleApp {
    def main(args: Array[String]) {
        val logFile = "YOUR_SPARK_HOME/README.md" // Should be some
                                                  // file on your system
        val conf = new SparkConf().setAppName("Simple Application")
        val sc = new SparkContext(conf)
        val logData = sc.textFile(logFile, 2).cache()
        val numAs = logData.filter(line =>line.contains("a")).count()
        val numBs = logData.filter(line => line.contains("b")).count()
        println("Lines with a: %s, Lines with b: %s".format(numAs,numBs))
    }
}
```

Part2: Self-Contained Applications

Note that applications should define a `main()` method instead of extending `scala.App`. Subclasses of `scala.App` may not work correctly.

What does this program?

Our application depends on the Spark API, so we'll also include an sbt configuration file, `build.sbt` which explains that Spark is a dependency. This file also adds a repository that Spark depends on:

```
name := "Simple Project"

version := "1.0"

scalaVersion := "2.10.5"

libraryDependencies += "org.apache.spark" %% "spark-core" % "1.5.1"
```

For sbt to work correctly, we'll need to layout `SimpleApp.scala` and `build.sbt` according to the typical directory structure. Once that is in place, we can create a JAR package containing the application's code, then use the spark-submit script to run our program.

```
# Your directory layout should look like this
$ find .
.
./build.sbt
./src
./src/main
./src/main/scala
./src/main/scala/SimpleApp.scala

# Package a jar containing your application
$ activator package
...
[info] Packaging {..}/{..}/target/scala-2.10/simple-project_2.10-1.0.jar

# Use spark-submit to run your application
$ YOUR_SPARK_HOME/bin/spark-submit \
--class "SimpleApp" \
--master local[4] \
target/scala-2.10/simple-project_2.10-1.0.jar
...
```

What is the output of the program?

Exercise

1- Use the Spark's `aggregate` function to calculate the sum of the lengths of the strings in an RDD.

- Write a first function to locally accumulate the size of the strings for each partition, call it `stringSizeCummulator`
- Write a second function to add-up the accumulated sizes, call it `add`.
- Apply the two above functions with the appropriate zero element in the `aggregate` function.

2- Use the Sparks map and reduce functions to write the same function, i.e to calculate the sum of the lengths of the strings in an RDD.

References

The content of this section is partly taken from the slides of the course "Parallel Programming and Data Analysis" by Heather Miller at EPFL.

Other references used here:

- *Learning Spark*
  by Holden Karau, Andy Konwinski, Patrick Wendell & Matei Zaharia.
  O'Reilly, February 2015.
- Spark documentation, available at
  http://spark.apache.org/docs/latest/