Big Data Frameworks: Spark Practicals

28.03.2017

Eemil Lagerspetz, Mohammad Hoque, Ella Peltonen, Professor Sasu Tarkoma

These slides: https://is.gd/bigdataspark2017

Outline

- Important Numbers
- Practicals
- Idioms and Patterns
- Tips
- Converting Pearson Correlation to Spark
- List of Spark Transformations and Operations (RDD API)

Important Numbers 1/3

- Data size: How many GB? How large files? How many files?
  - Hadoop/Spark prefers ~ 500MB files
- How many tasks? `sc.textFile("data", tasks)`
  - Minimum: 2xCPU cores in cluster
  - Optimal: Each computer's memory is used fully for tasks
  - Maximum: Too large → high overhead
- Spark does not tune this for you – It depends on your job

Important Numbers 2/3

- How many objects? How large are they? How many cores per computer, how many objects does each process?
  - --conf spark.task.cpus=X to control this
- Example VM: 8 cores/tasks, 32GB RAM → Spark has 4GB / core. Too little?
  - Set cpus=2 and Spark will assign 8/2 = 4 tasks to the node at a time.

Important Numbers 3/3

- Does your task cause the data size to grow? How much?
  - Deserialization, data grouping, data multiplication with cartesian() or tasks size possibly doubling with join()
- `rdd.mapPartitions(func)` makes one task handle one file/partition, this is more efficient if your objects are small
  - With this, all the objects of one partition are available at the VM
- `rdd.mapPartitions( part => part.map( x => x*2 ))`
- Results in the same thing as

  `rdd.map( x => x*2 )`
- Tip: mapPartitions returns an Iterable, so you can do filtering too before returning it

Practicals: import SparkContext._

Import org.apache.spark.SparkContext
Import SparkContext._

- The above is needed so that RDDs have groupByKey etc advanced operations
  - This imports Spark's implicit imports, like PairedRDD functions (key-value stuff)
- Transformations of types from DataSet[Row] require also import spark.implicits._

Practicals: Keys and Values

- In Spark, any RDD of type (a, b) is an RDD of keys and values
  - Keys can be used in groupByKey, reduceByKey, etc.
- Example:

```
val data =
    Array( ((0,1), "first"),((1,0), "second"))
val rdd = sc.parallelize(data)
// rdd: RDD[((Int, Int), String)]
```

- Here (Int, Int) is the key
- Key can be any class that implements hashCode, for example, any Tuple (of any length), or any case class
  - Optionally, you can implement Ordering to allow sortByKey

Spark key-value RDD pattern

- Use RDDs of type k -> v to allow reduceByKey, groupByKey, sortByKey, ...
  - Cleaner than rdd.groupBy(_.x)
- Load text data directly to (k, v) RDD:

```scala
case class MovieInfo(title:String, genres: Array[String])
val txt = sc.textFile("moviedata")
val movies = txt.map{ line => line.split("::") match {
  case Array(id, title, genres, _*) =>
   id.toInt -> new MovieInfo(title, genres.split("|"))
  }
 } // movies: RDD[(Int, MovieInfo)]
```

Naming Tuple Fields in Transformations

```scala
something.map{ x =>
  val (a, b, c, d, e) = x
  (a, c, d) }
```

- Allows splitting a tuple to elements on the first line
- Naming: no need to get confused by tuple indices

```scala
something.map{ x =>
  val (a, (b, c, d, e)) = x
  (c, (a, e)) }
```

- Supports nested tuples like above. Can be done with case too:

```scala
something.map{ case (a, (b, c, d, e)) => (c, (a, e)) }
```

Tips: Joining two datasets

```scala
val xrdd = sc.textFile(p).map(x => x.head -> x.tail)
val yrdd = sc.textFile(q).map(y => y.head -> y.tail)
```

- First element is always key, second the data

```scala
val grouped = xrdd.groupWith(yrdd)
// grouped: RDD[(Char, (Iterable[String], Iterable[String]))]
```

- This is the same as

```scala
val xg = xrdd.groupByKey
val yg = yrdd.groupByKey
val grouped = xg.join(yg)
```

Idioms

- Text data parsing: Scala pattern match Idiom

```
case class Movie(id:Int, title:String, genres: Array[String])
val txt = sc.textFile("moviedata")
val movies = txt.map{ line =>  line.split("::") match {
  case Array(id, title, genres) =>
   new Movie(id.toInt, title, genres.split("|"))
    }
  }
```

- The use of the pattern matcher avoids array size exceptions
- Allows naming fields of the split Array directly at the case statement

Idioms

- Self-contained text formatting and print statement

```
println(s"""First items are: ${xrdd.take(5).mkString(", ")}
And the mean is ${xrdd.reduce(_+_) / xrdd.count}""")
```

- `s"$var"` (String interpolation)
  allows complex code inside a String
- Multiline Strings (`"""`) help make this readable
- Separate mkString and other string formatting logic from main program logic (to reduce clutter)
- Keep printing and formatting-related code in a single place

Bring argument class fields to scope

```
// x is a
case class Movie(movieId:Int, title:String, genres:Array[String])
something.map{ x =>
  import x._
  movieId -> title
}
```

Tips: Changing data path and master

- Main program structure for running on a cluster plus testing locally
- Give -Dspark.master=local[2] in VM args in Eclipse, or command line replace with -Dspark.master=spark://ukko123.hpc.cs.helsinki.fi:7077

```
main(args: Array[String]) {
  val dataPath = args(0)
  val conf = new
    SparkConf().setAppName(getClass.getName)
  val sc = new SparkContext(conf)
  val dataRdd = sc.textFile(dataPath)
  //...
}
```

Converting Pearson to Spark

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

- Start with the math (Wikipedia)
- Required components:
- Two datasets with equal length (n)
- Mean of both datasets (mx and my)
- Upper:
  - Product of difference from mean at each index i of both datasets
- Lower:
  - Standard deviation (sqrt of square difference sum) of each dataset separately, multiplied

Converting Pearson to Spark

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

- Mean is needed before calculating SD and the upper side, so do it separately:

```
val xdata = Array(0.0, 2.0, 4.0, 6.0, 8.0, 10.0)
val xrdd = sc.parallelize(xdata)
val ydata = Array(1.0, 3.0, 5.0, 7.0, 9.0, 9.0)
// Correct result for these is r=0.982
val yrdd = sc.parallelize(ydata)
val mx = xrdd.reduce(_+_) / xrdd.count // 5.0
val my = yrdd.reduce(_+_) / xrdd.count // 5.67
```

Converting Pearson to Spark

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

- Upper part needs us to combine both datasets by index. We do this with zip:

```
val both = xrdd.zip(yrdd)
val upper = both.map{ pair =>
 (pair._1 - mx)*(pair._2 - my)}.reduce(_+_) // 60.0
```

- The lower part has similar components, but the difference is squared before summing up:

  ```
  val (lowerx, lowery) = both.map{ pair =>
    math.pow((pair._1 - mx), 2) -> math.pow((pair._2 – my), 2)}
    .reduce((a, b) => (a._1+b._1, a._2+b._2)) // 70.0, 53.33
  ```

```
val r = upper / (math.sqrt(lowerx) * math.sqrt(lowery))
// 0.9819805060619657
```

- Correct result.

Optimizing Pearson

- We ran three map-reduces (mean, upper, lower). What if we could do it in two? (mean, upper+lower)

```
val (upper, lowerx, lowery) = xrdd.zip(yrdd).map { pair =>
   val up = (pair._1 - mx) * (pair._2 - my)
   val lowx = math.pow((pair._1 - mx), 2)
   val lowy = math.pow((pair._2 - my), 2)
  (up, lowx, lowy)
}.reduce { (a, b) => (a._1 + b._1, a._2 + b._2, a._3 + b._3) }
// 60.0, 70.0, 53.33
val r = upper / (math.sqrt(lowerx) * math.sqrt(lowery))
// 0.9819805060619657
```

Whole thing on one slide

```
val xrdd = sc.parallelize(Array(0.0, 2.0, 4.0, 6.0, 8.0, 10.0))
val yrdd = sc.parallelize(Array(1.0, 3.0, 5.0, 7.0, 9.0, 9.0))
val mx = xrdd.reduce(_ + _) / xrdd.count // 5.0
val my = yrdd.reduce(_ + _) / xrdd.count // 5.67
val (upper, lowerx, lowery) = xrdd.zip(yrdd).map { pair =>
  val up = (pair._1 - mx) * (pair._2 - my)
  val lowx = math.pow((pair._1 - mx), 2)
  val lowy = math.pow((pair._2 - my), 2)
 (up, lowx, lowy)
}.reduce { (a, b) => (a._1 + b._1, a._2 + b._2, a._3 + b._3) }
// 60.0, 70.0, 53.33
val r = upper / (math.sqrt(lowerx) * math.sqrt(lowery))
// 0.9819805060619657
```

Transformations

Create a new dataset from an existing dataset

All transformations are lazy and computed when the results are needed

Transformation history is retained in RDDs

- calculations can be optimized
- data can be recovered

Some operations can be given the **number of tasks**. This can be very important for performance. Spark and Hadoop prefer larger files and smaller number of tasks if the data is small. However, the number of tasks should always be **at least the number of CPU cores** in the computer / cluster running Spark.

Spark Transformations I/IV

Transformation	Description
map(**func**)	Returns a new RDD based on applying function **func** to the each element of the source
filter(**func**)	Returns a new RDD based on selecting elements of the source for which **func** is true
flatMap(**func**)	Returns a new RDD based on applying function **func** to each element of the source while **func** can return a sequence of items for each input element
mapPartitions(**func**)	Implements similar functionality to map, but is executed separately on each partition of the RDD. The function **func** must be of the type (Iterator <T>) => Iterator<U> when dealing with RDD type of T.
mapPartitionsWithIndx(**func**)	Similar to the above transformation, but includes an integer index of the partition with **func**. The function **func** must be of the type (Int, Iterator <T>) => Iterator<U> when dealing with RDD type of T.

Transformations II/IV

Transformation	Description
sample(**withReplac, frac**, **seed**)	Samples a fraction (**frac**) of the source data with or without replacement (**withReplac**) based on the given random **seed**
union(**other**)	Returns an union of the source dataset and the given dataset
intersection(**other**)	Returns elements common to both RDDs
distinct([**nTasks**])	Returns a new RDD that contains the distinct elements of the source dataset.

Spark Transformations III/IV

Transformation	Description
groupByKey([**numTask**])	Returns an RDD of (K, Seq[V]) pairs for a source dataset with (K,V) pairs.
reduceByKey(**func**, [**numTasks**])	Returns an RDD of (K,V) pairs for an (K,V) input dataset, in which the values for each key are combined using the given reduce function **func.**
aggregateByKey(**zeroVal**)(**seqOp**, **comboOp**, [**numTask**])	Given an RDD of (K,V) pairs, this transformation returns an RDD RDD of (K,U) pairs for which the values for each key are combined using the given combine functions and a neutral zero value.
sortByKey([**ascending**], [**numTasks**])	Returns an RDD of (K,V) pairs for an (K,V) input dataset where K implements *Ordered*, in which the keys are sorted in ascending or descending order (**ascending** boolean input variable).
join(**inputdataset**, [**numTask**])	Given datasets of type (K,V) and (K, W) returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key.
cogroup(**inputdataset**, [**numTask**])	Given datasets of type (K,V) and (K, W) returns a dataset of (K, Seq[V], Seq[W]) tuples.
cartesian(**inputdataset**)	Given datasets of types T and U, returns a combined dataset of (T, U) pairs that includes all pairs of elements.

Spark Transformations IV

Transformation	Description
pipe(**command**, [**envVars**])	Pipes each partition of the given RDD through a shell command (for example bash script). Elements of the RDD are written to the stdin of the process and lines output to the stdout are returned as an RDD of strings.
coalesce(**numPartitions**)	Reduces the number of partitions in the RDD to **numPartitions.**
repartition(**numPartitions**)	Facilitates the increasing or reducing the number of partitions in an RDD. Implements this by reshuffling data in a random manner for balancing.
repartitionAndSortWithinPartitions(**partitioner**)	Repartitions given RDD with the given partitioner sorts the elements by their keys. This transformation is more efficient than first repartitioning and then sorting.

Spark Actions I/II

Transformation	Description
reduce(**func**)	Combine the elements of the input RDD with the given function **func** that takes two arguments and returns one. The function should be commutative and associative for correct parallel execution.
collect()	Returns all the elements of the source RDD as an array for the driver program.
count()	Returns the number of elements in the source RDD.
first()	Returns the first element of the RDD. (Same as take(1))
take(**n**)	Returns an array with the first n elements of the RDD. Currently executed by the driver program (not parallel).
takeSample(**withReplac, frac, seed)**	Returns an array with a random sample of **frac** elements of the RDD. The sampling is done with or without replacement (**withReplac**) using the given random **seed**.
takeOrdered(**n**, [**ordering**])	Returns first n elements of the RDD using natural/custom ordering.

Spark Actions II

Transformation	**Description**
saveAsTextFile(**path**)	Saves the elements of the RDD as a text file to a given local/HDFS/Hadoop directory. The system uses toString on each element to save the RDD.
saveAsSequenceFile(**path**)	Saves the elements of an RDD as a Hadoop SequenceFile to a given local/HDFS/Hadoop directory. Only elements that conform to the Hadoop *Writable* interface are supported.
saveAsObjectFile(**path**)	Saves the elements of the RDD using Java serialization. The file can be loaded with SparkContext.objectFile().
countByKey()	Returns (K, Int) pairs with the count of each key
foreach(**func**)	Applies the given function **func** for each element of the RDD.

Spark API

https://spark.apache.org/docs/latest/api/scala/index.html

For Python

https://spark.apache.org/docs/latest/api/python/

Spark Programming Guide:

https://spark.apache.org/docs/latest/programming-guide.html

Check which version's documentation (stackoverflow, blogs, etc) you are looking at, the API had big changes after version 1.0.0, and since version 1.6.0, you no longer need to set storageFraction. Also, choice of master now happens via spark-submit, and some memory-related properties have been renamed.

Intro to Apache Spark: http://databricks.com

More information

These slides:

https://is.gd/bigdataalgo2017
https://is.gd/bigdataspark2017

Contact:

These slides:

Eemil Lagerspetz, Eemil.lagerspetz@cs.helsinki.fi

Big Data Frameworks course:

Mohammad Hoque, mohammad.hoque@cs.helsinki.fi

Course page:

https://www.cs.helsinki.fi/en/courses/582740/2017/k/k/1