İstanbul Kalkınma Ajansı

Türkiye Cumhuriyeti Kalkınma Bakanlığı

BAU Bahçeşehir University

İstanbul Big Data Eğitim ve Araştırma Merkezi

Spark Programming Essentials

Bu eğitim sunumları İstanbul Kalkınma Ajansı'nın 2016 yılı Yenilikçi ve Yaratıcı İstanbul Mali Destek Programı kapsamında yürütülmekte olan TR10/16/YNY/0036 no'lu İstanbul Big Data Eğitim ve Araştırma Merkezi Projesi dahilinde gerçekleştirilmiştir. İçerik ile ilgili tek sorumluluk Bahçeşehir Üniversitesi'ne ait olup İSTKA veya Kalkınma Bakanlığı'nın görüşlerini yansıtmamaktadır.

Spark Essentials:

using, Spak Shell:

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

```
./bin/pyspark
```

alternatively, with IPython Notebook:

```
IPYTHON_OPTS="notebook --pylab inline" ./bin/pyspark
```

Spark Essentials: *SparkContext*

First thing that a Spark program does is create a `SparkContext` object, which tells Spark how to access a cluster

In the shell for either Scala or Python, this is the `sc` variable, which is created automatically

Other programs must use a constructor to instantiate a new `SparkContext`

Then in turn `SparkContext` gets used to create other variables

Spark Essentials: *SparkContext*

Scala:

```
scala> sc
res: spark.SparkContext = spark.SparkContext@470d1f30
```

Python:

```
>>> sc
<pyspark.context.SparkContext object at 0x7f7570783350>
```

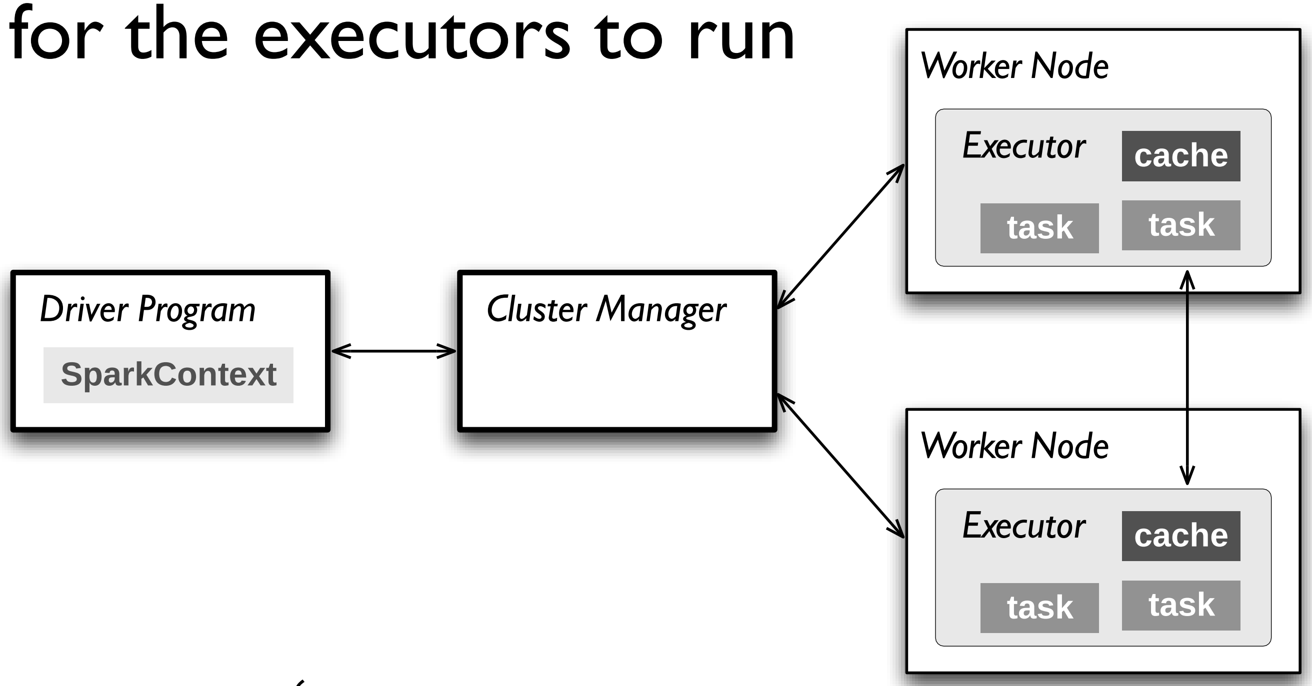
Spark Essentials: *Master*

The `master` parameter for a `SparkContext` determines which cluster to use

master	***description***
`local`	run Spark locally with one worker thread (no parallelism)
`local[K]`	run Spark locally with K worker threads (ideally set to # cores)
`spark://HOST:PORT`	connect to a Spark standalone cluster; PORT depends on config (7077 by default)
`mesos://HOST:PORT`	connect to a Mesos cluster; PORT depends on config (5050 by default)

Spark Essentials: *Clusters*

1. master connects to a *cluster manager* to allocate resources across applications
2. acquires *executors* on cluster nodes – processes run compute tasks, cache data
3. sends *app code* to the executors
4. sends *tasks* for the executors to run



Spark Essentials: *RDD*

Resilient **D**istributed **D**atasets (RDD) are the primary abstraction in Spark – a fault-tolerant collection of elements that can be operated on in parallel

- two types of operations on RDDs: *transformations* and *actions*
- transformations are lazy (not computed immediately)
- the transformed RDD gets recomputed when an action is run on it (default)
- however, an RDD can be *persisted* into storage in memory or disk

Spark Essentials: *RDD*

Scala:

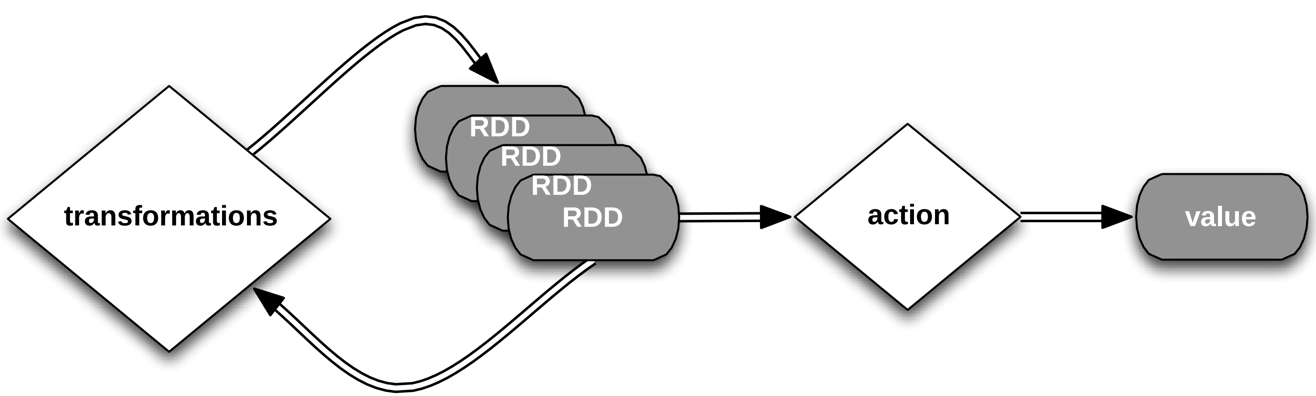
```
scala> val data = Array(1, 2, 3, 4, 5)
data: Array[Int] = Array(1, 2, 3, 4, 5)

scala> val distData = sc.parallelize(data)
distData: spark.RDD[Int] = spark.ParallelCollection@10d13e3e
```

Spark Essentials: *RDD*

Spark can create RDDs from any file stored in HDFS or other storage systems supported by Hadoop, e.g., local file system, Amazon S3, Hypertable, HBase, etc.

Spark supports text files, SequenceFiles, and any other Hadoop `InputFormat`, and can also take a directory or a glob (e.g. `/data/201404*`)



Spark Essentials: *RDD*

Scala:

```
scala> val distFile = sc.textFile("README.md")
distFile: spark.RDD[String] = spark.HadoopRDD@1d4cee08
```

Transformations create a new dataset from an existing one

All transformations in Spark are *lazy*: they do not compute their results right away – instead they remember the transformations applied to some base dataset

- optimize the required calculations
- recover from lost data partitions

Spark Transformations

```
map()                          intersection()    cartesion()
flatMap()                      distinct()        pipe()
filter()                       groupByKey()      coalesce()
mapPartitions()                reduceByKey()     repartition()
mapPartitionsWithIndex()       sortByKey()       partitionBy()
sample()                       join()            ...
union()                        cogroup()
```

Spark Essentials: *Transformations*

transformation	***description***
map(*func***)**	return a new distributed dataset formed by passing each element of the source through a function *func*
filter(*func***)**	return a new dataset formed by selecting those elements of the source on which *func* returns true
flatMap(*func***)**	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)
sample(*withReplacement, fraction, seed***)**	sample a fraction *fraction* of the data, with or without replacement, using a given random number generator *seed*
union(*otherDataset***)**	return a new dataset that contains the union of the elements in the source dataset and the argument
distinct([*numTasks***]))**	return a new dataset that contains the distinct elements of the source dataset

Spark Essentials: *Transformations*

transformation	***description***
groupByKey([*numTasks*])	when called on a dataset of `(K, V)` pairs, returns a dataset of `(K, Seq[V])` pairs
reduceByKey(*func*, [*numTasks*])	when called on a dataset of `(K, V)` pairs, returns a dataset of `(K, V)` pairs where the values for each key are aggregated using the given reduce function
sortByKey([*ascending*], [*numTasks*])	when called on a dataset of `(K, V)` pairs where `K` implements `Ordered`, returns a dataset of `(K, V)` pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument
join(*otherDataset*, [*numTasks*])	when called on 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(*otherDataset*, [*numTasks*])	when called on datasets of type `(K, V)` and `(K, W)`, returns a dataset of `(K, Seq[V], Seq[W])` tuples – also called `groupWith`
cartesian(*otherDataset*)	when called on datasets of types `T` and `U`, returns a dataset of `(T, U)` pairs (all pairs of elements)

Spark Essentials: *Transformations*

Scala:

```
val distFile = sc.textFile("README.md")
distFile.map(l => l.split(" ")).collect()
distFile.flatMap(l => l.split(" ")).collect()
```

distFile is a collection of lines

looking at the output, how would you compare results for map() vs. flatMap() ?

Spark Actions

```
reduce()
collect()
count()
first()
take()
takeSample()
saveToCassandra()
takeOrdered()
saveAsTextFile()
saveAsSequenceFile()
saveAsObjectFile()
countByKey()
foreach()
...
```

Spark Essentials: *Actions*

action	***description***
reduce(*func*)	aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel
collect()	return all the elements of the dataset as an array at the driver program – 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
first()	return the first element of the dataset – similar to *take(1)*
take(*n*)	return an array with the first `n` elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements
takeSample(*withReplacement, fraction, seed*)	return an array with a random sample of `num` elements of the dataset, with or without replacement, using the given random number generator seed

Spark Essentials: *Actions*

action	*description*
`saveAsTextFile(`*`path`***`)`**	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
`saveAsSequenceFile(`*`path`***`)`**	write the elements of the dataset as a Hadoop `SequenceFile` in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. Only available on RDDs of key-value pairs that either implement Hadoop's `Writable` interface or are implicitly convertible to `Writable` (Spark includes conversions for basic types like `Int`, `Double`, `String`, etc).
`countByKey()`	only available on RDDs of type `(K, V)`. Returns a `Map` of `(K, Int)` pairs with the count of each key
`foreach(`*`func`***`)`**	run a function *func* on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

Spark Essentials: *Actions*

Scala:

```
val f = sc.textFile("README.md")
val words = f.flatMap(l => l.split(" ")).map(word => (word, 1))
words.reduceByKey(_ + _).collect.foreach(println)
```

Spark Essentials: *Persistence*

Spark can *persist* (or cache) a dataset in memory across operations

Each node stores in memory any slices of it that it computes and reuses them in other actions on that dataset – often making future actions more than 10x faster

The cache is *fault-tolerant*: if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it

Spark Essentials: *Persistence*

transformation	*description*
`MEMORY_ONLY`	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
`MEMORY_AND_DISK`	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
`MEMORY_ONLY_SER`	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
`MEMORY_AND_DISK_SER`	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
`DISK_ONLY`	Store the RDD partitions only on disk.
`MEMORY_ONLY_2`, `MEMORY_AND_DISK_2`, etc	Same as the levels above, but replicate each partition on two cluster nodes.

See:

http://spark.apache.org/docs/latest/programming-guide.html#rdd-persistence

Spark Essentials: *Persistence*

Scala:

```
val f = sc.textFile("README.md")
val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
w.reduceByKey(_ + _).collect.foreach(println)
```

Spark Essentials: *Broadcast Variables*

Broadcast variables let programmer keep a read-only variable cached on each machine rather than shipping a copy of it with tasks

For example, to give every node a copy of a large input dataset efficiently

Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost

Spark Essentials: *Broadcast Variables*

Scala:

```
val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar.value
```

Accumulators are variables that can only be "added" to through an *associative* operation

Used to implement counters and sums, efficiently in parallel

Spark natively supports accumulators of numeric value types and standard mutable collections, and programmers can extend for new types

Only the driver program can read an accumulator's value, not the tasks

Spark Essentials: *Accumulators*

Scala:

```
val accum = sc.accumulator(0)
sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)

accum.value
```

driver-side

Spark Essentials: *(K,V) pairs*

Scala:

```
val pair = (a, b)

    pair._1 // => a
    pair._2 // => b
```

Python:

```
pair = (a, b)

    pair[0] # => a
    pair[1] # => b
```

Java:

```
Tuple2 pair = new Tuple2(a, b);

    pair._1 // => a
    pair._2 // => b
```

Spark Programming - Deployment

Spark in Production:

In the following, let's consider the progression through a full software development lifecycle, step by step:

1. **build**
2. **deploy**
3. **monitor**

builds:

- SBT primer
- build/run a JAR using Scala + SBT

SBT is the **S**imple **B**uild **T**ool for Scala:

www.scala-sbt.org/

This is included with the Spark download, and does not need to be installed separately.

it provides for *incremental compilation* and an *interactive shell,* among other innovations.

SBT project uses *StackOverflow* for Q&A, that's a good resource to study further:

stackoverflow.com/tags/sbt

Spark in Production: *Build: SBT*

command	*description*
`clean`	delete all generated files (in the *target* directory)
`package`	create a JAR file
`run`	run the JAR (or main class, if named)
`compile`	compile the main sources (in *src/main/scala* and *src/main/java* directories)
`test`	compile and run all tests
`console`	launch a Scala interpreter
`help`	display detailed help for specified commands

builds:

- SBT primer
- build/run a JAR using Scala + SBT

Spark in Production: *Build: Scala*

The following sequence shows how to build a JAR file from a Scala app, using SBT

- First, this requires the "source" download, not the "binary"
- Connect into the `SPARK_HOME` directory
- Then run the following commands…

Spark in Production: *Build: Scala*

```
# Scala source + SBT build script on following slides

cd simple-app

../spark/bin/spark-submit \
  --class "SimpleApp" \
  --master local[*] \
  target/scala-2.10/simple-project_2.10-1.0.jar
```

Spark in Production: *Build: Scala*

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

object SimpleApp {
  def main(args: Array[String]) {
    val logFile = "README.md" // Should be some file on your system
    val sc = new SparkContext("local", "Simple App", "SPARK_HOME",
      List("target/scala-2.10/simple-project_2.10-1.0.jar"))
    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))
  }
}
```

Spark in Production: *Build: Scala*

```
name := "Simple Project"

version := "1.0"

scalaVersion := "2.10.4"

libraryDependencies += "org.apache.spark" % "spark-core_2.10" % "1.2.0"

resolvers += "Akka Repository" at "http://repo.akka.io/releases/"
```

Spark in Production: *Deploy*

deploy JAR to Hadoop cluster, using these alternatives:

- run on Apache Mesos
- run on EC2
- or, simply run the JAR on YARN

Spark in Production: *Deploy:Mesos*

Running Spark on **Apache Mesos**

spark.apache.org/docs/latest/running-on-mesos.html

For example:

```
./bin/spark-submit \
    --class org.apache.spark.examples.SparkPi \
    --master mesos://***.***.***.***:7077 \
    --deploy-mode cluster \
    --supervise \
    --executor-memory 20G \
    --total-executor-cores 100 \
    http://path/to/examples.jar \
    1000
```

Spark in Production: *Deploy:EC2*

Running Spark on AmazonAWS **EC2**:

blogs.aws.amazon.com/bigdata/post/Tx15AY5C50K70RV/Installing-Apache-Spark-on-an-Amazon-EMR-Cluster

Spark in Production: *Deploy: YARN*

spark.apache.org/docs/latest/running-on-yarn.html

- Simplest way to deploy Spark apps in production
- Does not require admin, just deploy apps to your Hadoop cluster

Exploring data sets loaded from HDFS…

1. launch a Spark cluster using EC2 script
2. load data files into HDFS
3. run Spark shell to perform *WordCount*

Spark in Production: *Deploy: HDFS examples*

```
# http://spark.apache.org/docs/latest/ec2-scripts.html
cd $SPARK_HOME/ec2

export AWS_ACCESS_KEY_ID=$AWS_ACCESS_KEY
export AWS_SECRET_ACCESS_KEY=$AWS_SECRET_KEY
./spark-ec2 -k spark -i ~/spark.pem -s 2 -z us-east-1b  launch foo

# can review EC2 instances and their security groups to identify master
# ssh into master
./spark-ec2 -k spark -i ~/spark.pem -s 2 -z us-east-1b login foo

# use ./ephemeral-hdfs/bin/hadoop to access HDFS
/root/hdfs/bin/hadoop fs           -mkdir /tmp
/root/hdfs/bin/hadoop fs           -put CHANGES.txt  /tmp

# now is the time when we Spark
cd /root/spark
export SPARK_HOME=$(pwd)

SPARK_HADOOP_VERSION=1.0.4 sbt/sbt  assembly

/root/hdfs/bin/hadoop fs           -put CHANGES.txt  /tmp
./bin/spark-shell
```

Spark in Production: *Deploy: HDFS examples*

```
/** NB: replace host IP with EC2 internal IP address **/

val f = sc.textFile("hdfs://10.72.61.192:9000/foo/CHANGES.txt")
val counts =
 f.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey(_ + _)

counts.collect().foreach(println)
counts.saveAsTextFile("hdfs://10.72.61.192:9000/foo/wc")
```

Let's check the results in HDFS…

```
root/hdfs/bin/hadoop fs            -cat /tmp/wc/part-*

(Adds,1)
(alpha,2)
(ssh,1)
(graphite,1)
(canonical,2)
(ASF,3)
(display,4)
(synchronization,2)
(instead,7)
(javadoc,1)
(hsaputra/update-pom-asf,1)

…
```

review UI features

spark.apache.org/docs/latest/monitoring.html

http://<master>:8080/

http:/ / <master>:50070/

- verify: is my job still running?
- drill-down into *workers* and *stages*
- diagnose / troubleshoot