Introduction of Apache Spark

TANG Gen

Outline

- Design of Apache Spark
  - SparkContext
  - Spark architecture
- Design of **RDD**
  - Conception
  - Transformations
  - Actions
- Shared variables
- Key-value pairs

Design of Apache Spark

/SparkContext

- The first thing that a Spark application does is to create a *SparkContext* object.

- *SparkContext* tells Spark how to access a cluster.

- RDDs are created by *SparkContext* object.

- In the Spark shells, Scala or Python, *SparkContext* has already been created, which is sc variable.

- Other programs must use a constructor to create a new *SparkContext*

Design of Apache Spark

/SparkContext

```
1  /*
2   * load the CHANGES.txt in the spark directory.
3   * Then put it into memory and do some interactive
4   * search for various patterns
5  */
6
7  //Created an RDD
8  val docs = sc.textFile("file:/Users/tgbaggio/spark/spark-1.3.0-bin-hadoop2.4/CHANGES.txt")
9
10 //Transformation on RDD
11 val sparkChange = docs.filter(_.contains("SPARK"))
12
13 //Put data in memory
14 sparkChange.cache()
15
16 //First action
17 sparkChange.count()
18
19 //Second action
20 sparkChange.filter(_.contains("PySpark")).collect().foreach(println)
```

Design of Apache Spark

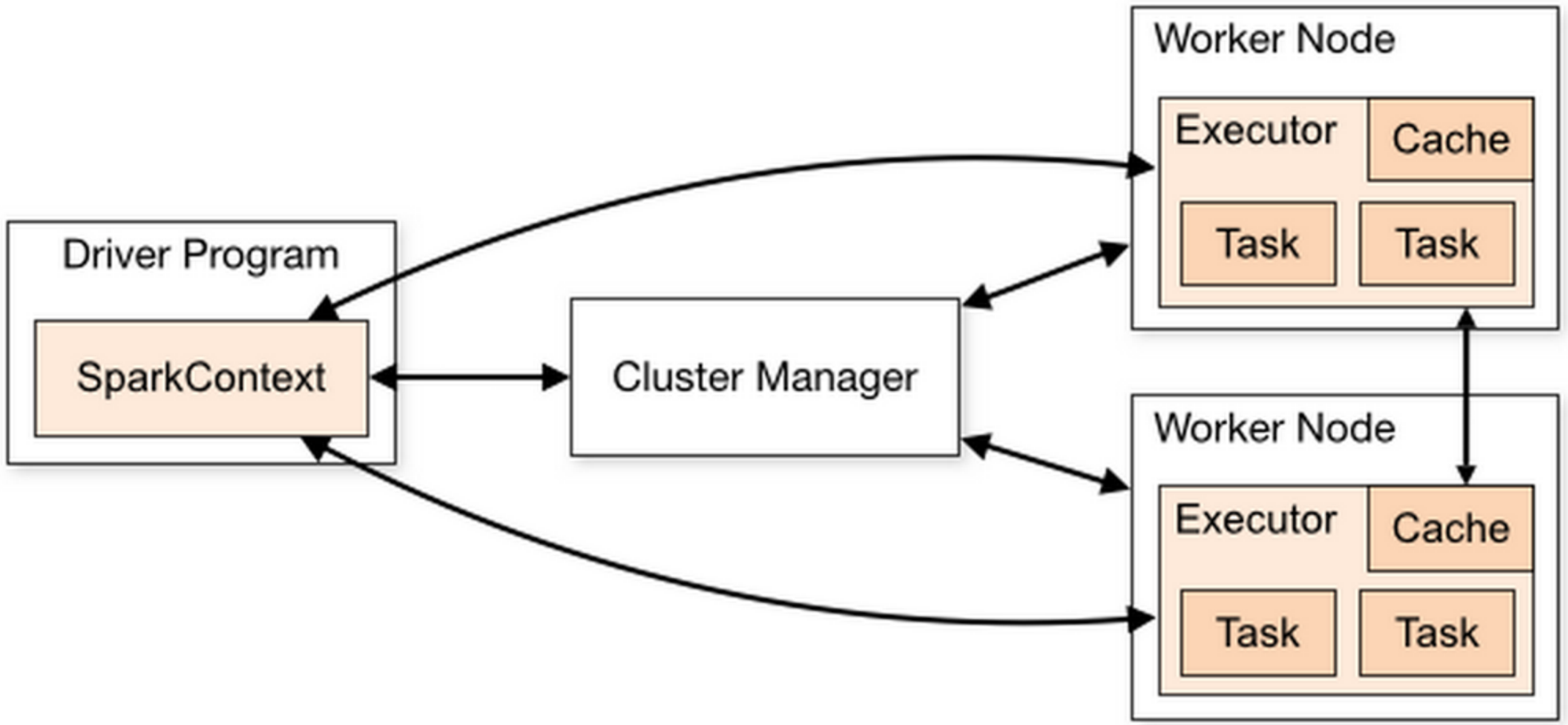
/SparkContext

```scala
package spark.training

import org.apache.spark.{SparkConf, SparkContext}

object TextMining extends Serializable {
  def main(args: Array[String]): Unit = {

    //Created SparkContext
    val sc = new SparkContext(new SparkConf().setAppName("TextMining"))

    //Created an RDD
    val docs = sc.textFile("file:/Users/tgbaggio/spark/spark-1.3.0-bin-hadoop2.4/CHANGES.txt")

    //Transformation on RDD
    val sparkChange = docs.filter(_.contains("SPARK"))

    //Put data in memory
    sparkChange.cache()

    //First action
    println("There are " + sparkChange.count().toString + " lines containing SPARK")

    //Second action
    sparkChange.filter(_.contains("PySpark")).collect().foreach(println)
  }
}
```

Design of Apache Spark

/Spark architecture

- Cluster mode overview



Design of Apache Spark

/Spark architecture

- In SparkContext, the *master* parameter is to determine which cluster to use.

master	description
local[K]	run Spark locally with K worker threads
spark://host:port	connect to a Spark standalone cluster: PORT depends on config(7077 by default)
mesos://host:port	connect to Mesos cluster; PORT depends on config(5050 by default)
yarn-client or yarn-cluster	Connect to Yarn cluster; the figuration is required

Design of Apache Spark

/Spark architecture

- Driver program which creates SparkContext connects to a cluster manager. a cluster manager is a processes to allocate resources across applications and usually runs on the master node

- Once connected, Spark acquires executors on cluster slave nodes. The executors are processes to run compute tasks, cache data

- Driver program sends app code to the executors via SparkContext

- Finally, SparkContext sends tasks for the executors to run

Design of **RDD**

/Conception

- RDD is Resilient Distributed Datasets.

- It is the primary abstraction in Spark by which Spark realise fault-tolerant and parallelised calculus.

- Currently, there are two ways to create an RDD:

    1. Take an existing collection and run actions on it in parallel. The created RDD is called parallelised collection.

        val rdd = sc.parallelize(1 to 10000)

    2. Take a file in Hadoop distributed file system or any other storage system supported by Hadoop and run actions on it in parallel. The created RDD is called Hadoop dataset

        val rdd = sc.textFile("hdfs://…")

Design of **RDD**

/Conception

- Recall

1. Two types of operations on RDDs: transformations and actions
2. Spark is lazy-evaluation, no immediate computation
3. The transformations get recomputed when an action is run on it (default)
4. An RDD can be persisted into storage in memory or disk

Design of **RDD**

/Transformations

- Transformations create a new RDD from an existing one
- All transformations in Spark are lazy: they just remember the transformations applied to base dataset and wait an actions to pull the trigger

    1. Optimisation: optimize the required calculations

    2. Fault-tolerant: recover from lost data partitions

Design of **RDD**

/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(RDD)	return a new dataset that contains the union of the elements in the source dataset and the argument
distinct()	return a new dataset that contains the distinct elements of the source dataset

Design of **RDD**

/Transformations

transformation	description
groupByKey()	when called on a dataset of (K, V) pairs, returns a dataset of (K, Seq[V]) pairs
reduceByKey(func)	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()	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()	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(RDD)	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(RDD)	when called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements)

Design of **RDD**

/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

Design of **RDD**

/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

Design of **RDD**

/Persistence

- Spark is a “In memory” technology: it can persist a dataset in memory across operations.

- Slave nodes in the cluster store some slices of dataset in memory and reuses them for other actions which makes future actions more than 10x faster

- The data in memory is also fault-tolerant. The lost data can be recomputed from original data

- There are several different level of persistence:
  MEMORY_ONLY
  MEMORY_AND_DISK
  MEMORY_ONLY_SER
  MEMORY_AND_DISK_SER
  DISK_ONLY

Shared variables

/Broadcast variables

- Normally, when a function passed to a Spark operation (such as map or reduce) is executed on a remote cluster node, it works on separate copies of all the variables used in the function.

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

- For instance,

```
val I = Array(1,2,3)
rdd.map(x => func(x, I))

val bVar = sc.broadcast(I)  (bVar.value == I)
rdd.map(x => func(x, bVar))
```

ship a copy of I with every task

ship a copy of I only once to every slave node

Shared variables

/Accumulators

- Accumulators are variables that can only be “added” to through an associative operation

- It is used to implement counters and sums, efficiently in parallel

- It can be only used by driver program, not tasks

- **Important**: Due to the mechanism of fault-tolerance, for accumulators used in **actions**, Spark applies each task’s update to each accumulator only once, but the **guarantee doesn’t exist in RDD transformations**

Shared variables

/Accumulators

- Scala

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

- Python

```
accum = sc.accumulator(0)
rdd = sc.parallelize(range(1, 4)
def f(x):
  global accum
  accum += x
rdd.foreach(f)
accum.value
```

Key-value pairs

Scala:

```scala
val pair = (a, b)

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

Python:

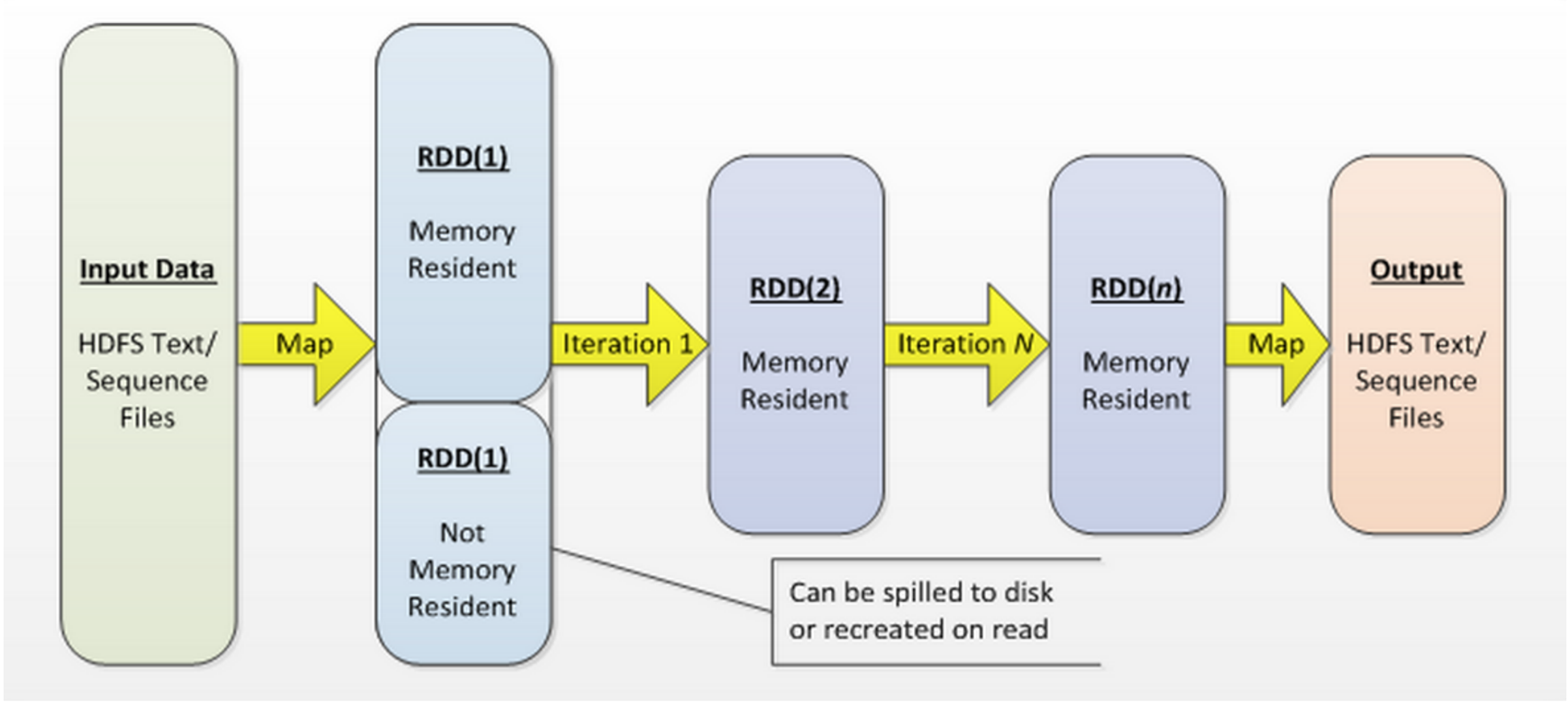
```python
pair = (a, b)

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

Java:

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

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

Case studies

/Spark at ebay



Case studies

/Spark at ebay

- Spark cluster at ebay
  - 2000 nodes,
  - 100TB of RAM,
  - 20,000 cores.