CLOUD COMPUTING

Cloud Applications

Zeinab Zali
Isfahan University Of Technology

Spark

References:

“Spark: Cluster Computing with Working Sets”, Matei Zaharia, et. al. 2nd USENIX conference on HotCloud’10
“Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing”
Matei Zaharia, et. al. 9th USENIX conference on NSDI’12
spark.apache.org, www.edureka.co

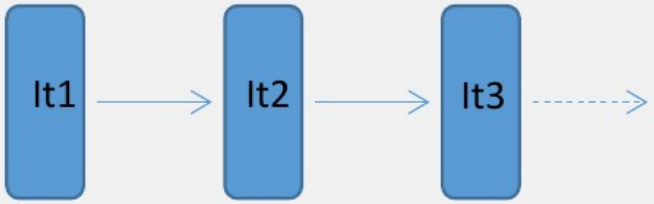
Spark Motivation (I)

- Extend the MapReduce model to better support two common classes of analytics apps:
  - Iterative algorithms (machine learning, graphs)
  - Interactive data mining
- Enhance programmability
  - Integrate into Scala programming language
  - Allow interactive use from Scala interpreter
- speeding up the Hadoop computational computing software process

Spark Motivation (II)

- Traditional MapReduce and classical parallel runtimes cannot solve iterative algorithms efficiently
  - MapReduce solution:
    - Split iteration into multiple MapReduce jobs.
    - Write a driver program for orchestration
  - Hadoop: Repeated data access to HDFS, no optimization to data caching and data transfers

```
Iterate {
   Map: for (each) i = 1 to M
      Compute();
   Reduce();
} Until converged();
```



Spark Features

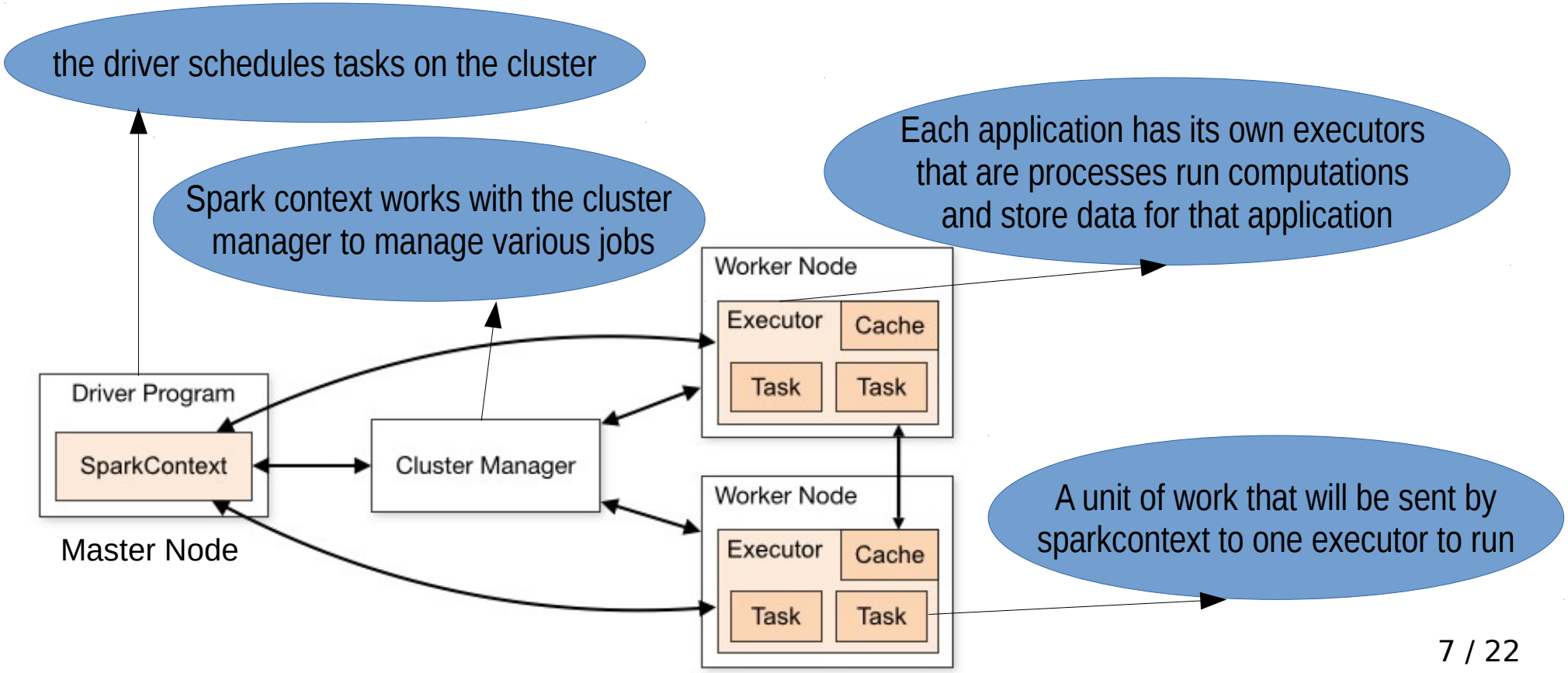
- The main feature of Spark is its in-memory cluster computing that increases the processing speed of an application
- cover a wide range of workloads
  - batch applications
  - iterative algorithms
  - interactive queries
  - streaming

Spark Ecosystem

- Components
  - Core, streaming, SQL, GraphX, Mlib, SparkR
- APIs
  - Scala, Java, Python, R

Spark Architecture (I)

- Spark applications run as independent sets of processes on a cluster coordinated by the SparkContext object in the main program (driver).



Spark concepts

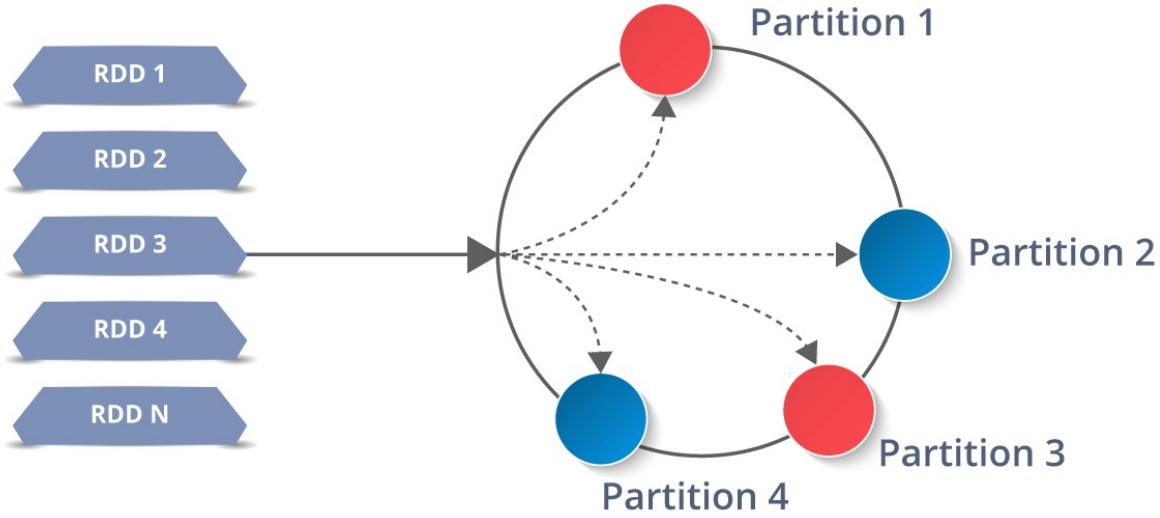
- Driver Program: The process running the main() function of the application and creating the SparkContext
  - In the interactive shell, the shell acts as the driver program.
- Task: A unit of work that will be sent to one executor
- Job: A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g. save, collect);
- Stage: Each job gets divided into smaller sets of tasks called stages that depend on each other (similar to the map and reduce stages in MapReduce);

Cluster Manager types

- Standalone: a simple cluster manager included with Spark that makes it easy to set up a cluster.
- Apache Mesos: a general cluster manager that can also run Hadoop MapReduce and service applications.
- Hadoop YARN: the resource manager in Hadoop 2.
- Kubernetes: an open-source system for automating deployment, scaling, and management of containerized applications.

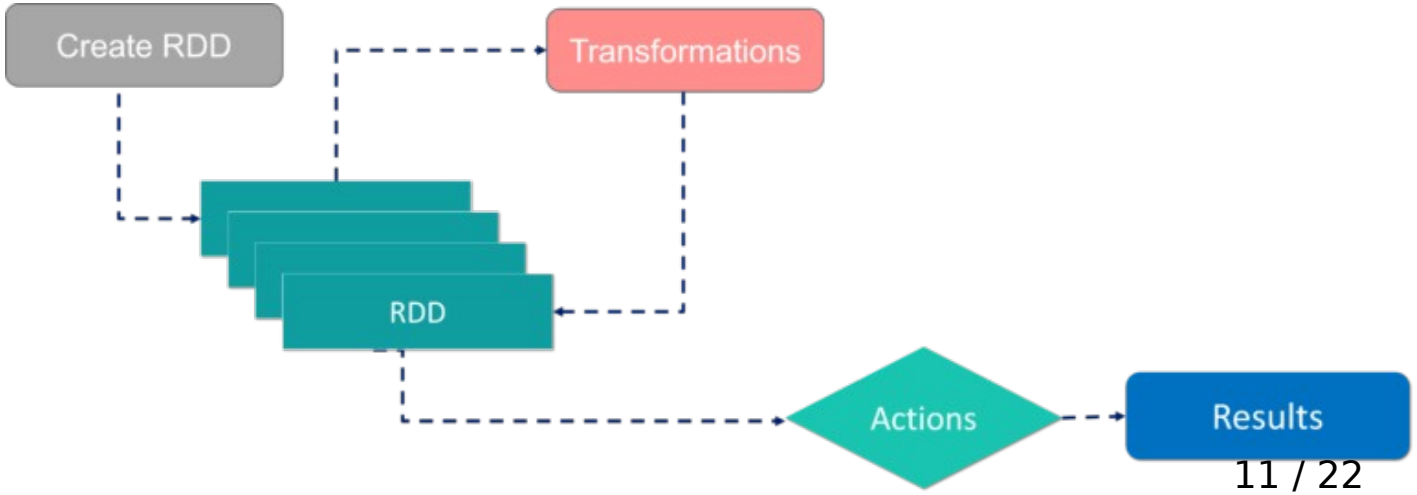
Spark main abstractions: RDD (I)

- RDD: a fault-tolerant immutable collection of elements that can be operated on in parallel
  - Resilient: Fault tolerant and is capable of rebuilding data on failure
  - Distributed: Distributed data among the multiple nodes in a cluster
  - Dataset: Collection of partitioned data with values



Spark main abstractions: RDD (II)

- Operations on RDD:
  - (1) Transformations, (2) Actions
- Two ways to create RDD:
  - (1) parallelizing an existing collection in the driver program, (2) referencing a dataset in an external storage system such as a shared filesystem, HDFS, ...



Spark main abstractions: DAG

- DAG (Directed Acyclic Graph): a sequence of computations performed on data where each node is an RDD partition and edge is a transformation on top of data.
  - The DAG abstraction helps eliminate the Hadoop MapReduce multi-stage execution model and provides performance enhancements over Hadoop

RDD Operations

- Transformations: They are the operations that are applied to create a new RDD.
- Actions: They are applied on an RDD to instruct Apache Spark to apply computation and pass the result back to the driver.

Spark Programming

Importing spark libraries

```scala
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
```

Scala

```java
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.SparkConf;
```

Java

```python
from pyspark import SparkContext, SparkConf
```

Python

Initializing Spark

```scala
val conf = new SparkConf().setAppName(appName).setMaster(master)
val sc =  new SparkContext(conf)
```

Scala

```java
SparkConf conf = new SparkConf().setAppName(appName).setMaster(master);
JavaSparkContext sc = new JavaSparkContext(conf);
```

Java

```python
conf = SparkConf().setAppName(appName).setMaster(master)
sc = SparkContext(conf=conf)
```

Python

Creating RDD

- Parallelized Collections

```
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
```

- Eexternal Datasets
  - Spark can create distributed datasets from any storage source supported by Hadoop, including your local file system, HDFS, Cassandra, HBase, Amazon S3
  - the file must also be accessible at the same path on worker nodes
  - Spark supports any Hadoop InputFormat

```
val distFile = sc.textFile("data.txt")
```

RDD Operations

- Transformations: create a new dataset from an existing one
- Actions: return a value to the driver program after running a computation on the dataset
- All transformations in Spark are **lazy**, in that they do not compute their results right away
  - each transformed RDD may be recomputed each time you run an action on it
  - you may also persist an RDD in memory using the persist (or cache) method

Transformations Examples

- 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.
- union(otherDataset): Return a new dataset that contains the union of the elements in the source dataset and the argument.
- intersection(otherDataset): Return a new RDD that contains the intersection of elements in the source dataset and the argument.

Actions Examples

- reduce(func): Aggregate the elements of the dataset using a function func (which takes two arguments and returns one)
- count(): Return the number of elements in the dataset.
- Collect(): Return all the elements of the dataset as an array at the driver program.
- foreach(func): Run a function func on each element of the dataset.

Example: Text Search

- Count the lines containing errors in a large log file stored in HDFS

```
val file = spark.textFile("hdfs://...")
val errs = file.filter(_.contains("ERROR"))
val ones = errs.map(_ => 1)
val count = ones.reduce(_+_)
```

Logistic Regression

```
// Read points from a text file and cache them
val points = spark.textFile(...)
                  .map(parsePoint).cache()
// Initialize w to random D-dimensional vector
var w = Vector.random(D)
// Run multiple iterations to update w
for (i <- 1 to ITERATIONS) {
  val grad = spark.accumulator(new Vector(D))
  for (p <- points) { // Runs in parallel
    val s = (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
    grad += s * p.x
  }
  w -= grad.value
}
```