Big data: architectures and data analytics

Spark Basic Concepts

Resilient Distributed Data sets (RDDs)

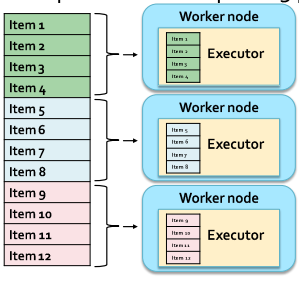
- RDDs are the primary abstraction in Spark
- RDDs are distributed collections of objects spread across the nodes of a clusters
  - They are split in partitions
  - Each node of the cluster that is used to run an application contains at least one partition of the RDD(s) that is (are) defined in the application

Resilient Distributed Data sets (RDDs)

- RDDs
  - Are stored in the main memory of the executors running in the nodes of the cluster (when it is possible) or on the local disk of the nodes if there is not enough main memory
  - Allow executing in parallel the code invoked on them
    - Each executor of a worker node runs the specified code on its partition of the RDD

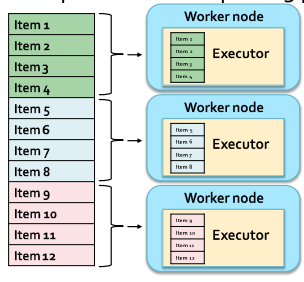
Resilient Distributed Data sets (RDDs)

- Example of an RDD split in 3 partitions



Resilient Distributed Data sets (RDDs)

- Example of an RDD split in 3 partitions



more partitions
=
more parallelism

Resilient Distributed Data sets (RDDs)

- RDDs
  - Are immutable once constructed
    - i.e., the content of an RDD cannot be modified
  - Track lineage information to efficiently recompute lost data
    - i.e., for each RDD, Spark knows how it as been constructed and can rebuilt it if a failure occurs
    - This information is represented by means of a DAG (Direct Acyclic Graph) connecting input data and RDDs

7

Resilient Distributed Data sets (RDDs)

- RDDs can be created
  - by parallelizing existing collections of the hosting programming language (e.g., collections and lists of Scala, Java, Pyhton, or R)
    - In this case the number of partition is specified by the user
  - from (large) files stored in HDFS or any other file system
    - In this case there is one partition per HDFS block
  - by transforming an existing RDDs
    - The number of partitions depends on the type of transformation

8

Resilient Distributed Data sets (RDDs)

- Spark programs are written in terms of operations on resilient distributed data sets
  - Transformations
    - map, filter, join, …
  - Actions
    - count, collect, save, …

Spark Framework

- Spark
  - Manages scheduling and synchronization of the jobs
  - Manages the split of RDDs in partitions and allocates RDDs' partitions in the nodes of the cluster
  - Hides complexities of fault-tolerance and slow machines
    - RDDs are automatically rebuilt in case of machine failure

Spark Programs

Supported languages

- Spark supports many programming languages
  - Scala
    - The same language that is used to develop the Spark framework and all its components (Spark Core, Sparl SQL, Spark Streaming, Mllib, GraphX)
  - Java
  - Python
  - R

12

Supported languages

- Spark supports many programming languages
  - Scala
    - The same language that is used to develop the Spark framework and all its components (Spark Core, Sparl SQL, Spark Streaming, Mllib, GraphX)
  - Java ← We will use Java
  - Python
  - R

13

Structure of Spark programs

- The Driver program
  - Contains the main method
  - "Defines" the workflow of the application
  - Accesses Spark through the SparkContext object
    - The SparkContext object represents a connection to the cluster
  - Creates Resilient Distributed Datasets (RDDs) that are "allocated" on the nodes of the cluster
  - Invokes parallel operations on RDDs

14

Structure of Spark programs

- The Driver program defines
  - Local variables
    - The standard variables of the Java programs
  - RDDs
    - Distributed "variables" stored in the nodes of the cluster
  - The SparkContext object allows
    - Creating RDDs
    - "Submitting" executors (processes) that execute in parallel specific operations on RDDs
      - Transformations and Actions

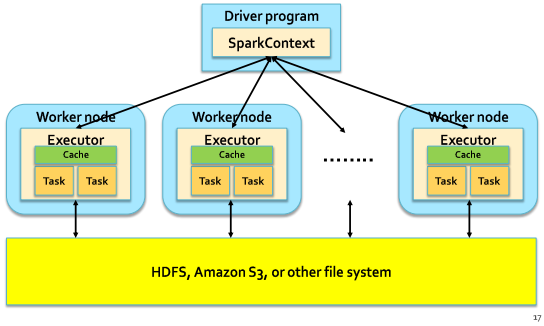
15

Structure of Spark programs

- The worker nodes of the cluster are used to run the application by means of the executors
- Each executor runs on its partition of the RDD(s) the operations that are specified in the driver
  - All the code, also the one that is executed in the nodes of the cluster, is usually defined in the Driver

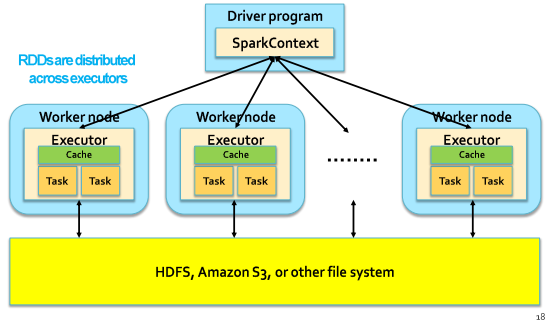
16

Distributed execution of Spark



17

Distributed execution of Spark



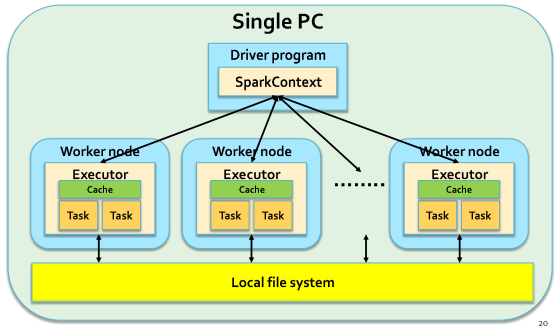
18

Local execution of Spark

- Spark programs can also be executed locally
  - Local threads are used to parallelize the execution of the application on RDDs on a single PC
    - Local threads can be seen are "pseudo-worker" nodes
  - It is useful to develop and test the applications before deploying them on the cluster
  - A local scheduler is launched to run Spark programs locally

19

Local execution of Spark



20

Spark official terminology

- Application
  - User program built on Spark. Consists of a driver program and executors on the cluster.
- Application jar
  - A jar containing the user's Spark application.
- Driver program
  - The process running the main() function of the application and creating the SparkContext

Based on http://spark.apache.org/docs/latest/cluster-overview.html

21

Spark official terminology

- Cluster manager
  - An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN)
- Deploy mode
  - Distinguishes where the driver process runs. In "cluster" mode, the framework launches the driver inside of the cluster. In "client" mode, the submitter launches the driver outside of the cluster.
- Worker node
  - Any node of the cluster that can run application code in the cluster

22

Spark official terminology

- Executor
  - A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them. Each application has its own executors.
- 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)

23

Spark official terminology

- 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)

24

Spark Programs: Examples

Spark program: Count line

- Count the number of lines of the input file
- The name of the file is specified by using the a command line parameter (i.e., args[o])
- Print the results on the standard output

26

Spark program: Count line

```
package it.polito.bigdata.spark.linecount;

import org.apache.spark.api.java.*;
import org.apache.spark.SparkConf;

public class DriverSparkBigData {
    public static void main(String[] args) {

        String inputFile;
        long numLines;

        inputFile=args[o];

        // Create a configuration object and set the name of the application
        SparkConf conf=new SparkConf().setAppName("Spark Line Count");

        // Create a Spark Context object
        JavaSparkContext sc = new JavaSparkContext(conf);
```

27

Spark program: Count line

```
        // Build an RDD of Strings from the input textual file
        // Each element of the RDD is a line of the input file
        JavaRDD<String> lines=sc.textFile(inputFile);

        // Count the number of lines in the input file
        // Store the returned value in the local variable numLines
        numLines=lines.count();

        // Print the output in the standard output (stdout)
        System.out.println("Number of lines="+numLines);

        // Close the Spark Context object
        sc.close();
    }
}
```

28

Spark program: Count line

```
package it.polito.bigdata.spark.linecount;

import org.apache.spark.api.java.*;
import org.apache.spark.SparkConf;

public class DriverSparkBigData {
    public static void main(String[] args) {

        String inputFile;
        long numLines;

        inputFile=args[o];

        // Create a configuration object and set the name of the application
        SparkConf conf=new SparkConf().setAppName("Spark Line Count");

        // Create a Spark Context object
        JavaSparkContext sc = new JavaSparkContext(conf);
```

Local Java variables.
They are allocated in the main memory of the same process of the object instancing the Driver Class

29

Spark program: Count line

```
        // Build an RDD of Strings from the input textual file
        // Each element of the RDD is a line of the input file
        JavaRDD<String> lines=sc.textFile(inputFile);

        // Count the number of lines in th
        // Store the returned value in the
        numLines=lines.count();

        // Print the output in the standard output (stdout)
        System.out.println("Number of lines="+numLines);

        // Close the Spark Context object
        sc.close();
    }
}
```

RDD.
It is allocated in the main memory or in the local disk of the executors of the worker nodes

Local Java variables.
They are allocated in the main memory of the same process of the object instancing the Driver Class

30

Spark program: Count line

- Local variables
  - Can be used to store only "small" objects/data
    - The maximum size is equal to the main memory of the process associated with the Driver
- RDDs
  - Are used to store "big/large" collections of objects/data in the nodes of the cluster
    - In the main memory of the nodes, when it is possible
    - On the local disks of the worker nodes, when it is necessary

31

Spark program: Word Count

- Word Count implemented by means of Spark
- The name of the input file is specified by using the a command line parameter (i.e., args[0])
- The output of the application (i.e., the pairs (word, num. of occurrences) is stored in and output folder (i.e., args[1])
- **Note:** Do not worry about the datails

32

Spark program: Word Count

```
package it.polito.bigdata.spark.wordcount;

import java.util.Arrays;
import org.apache.spark.api.java.*;
import org.apache.spark.api.java.function.*;
import org.apache.spark.SparkConf;
import scala.Tuple2;

public class SparkWordCount {
    @SuppressWarnings("serial")
    public static void main(String[] args) {

        String inputFile=args[0];
        String outputPath=args[1;]

        // Create a configuration object and set the name of the application
        SparkConf conf=new SparkConf().setAppName("Spark Line Count");

        // Create a Spark Context object
        JavaSparkContext sc = new JavaSparkContext(conf);
```

33

Spark program: Word Count

```
// Build an RDD of Strings from the input textual file
// Each element of the RDD is a line of the input file
JavaRDD<String> lines=sc.textFile(inputFile);

// Split/transform the content of the lines RDD in a
// list words an store in the words RDD
JavaRDD<String> words = lines.flatMap(
                new FlatMapFunction<String, String>() {
                        @Override
                        public Iterable<String> call(String s) {
                                return Arrays.asList(s.split("\\s+"));
                        }
                });
```

34

Spark program: Word Count

```
// Map/transform each word in the words RDD
// to a pair (word,1) an store the result in the words_one RDD
JavaPairRDD<String, Integer> words_one =
        words.mapToPair(
                new PairFunction<String, String, Integer>() {
                    @Override
                    public Tuple2<String, Integer> call(String s) {
                            return new Tuple2<String, Integer>(s.toLowerCase(), 1);
                    }
                });
```

35

Spark program: Word Count

```
// Count the occurrence of each word.
// Reduce by key the pairs of the words_one RDD and store
// the result (the list of pairs (word, num. of occurrences)
// in the counts RDD
JavaPairRDD<String, Integer> counts =
        words_one.reduceByKey(
                new Function2<Integer, Integer, Integer {
                    @ Override
                    public Integer call(Integer i1, Integer i2) {
                            return i1 + i2;
                    }
                });
```

36

Spark program: Word Count

```
        // Store the result in the output folder
        counts.saveAsTextFile(outputPath);

        // Close the Spark Context object
        sc.close();
    }
}
```

37