Lecture 9 – Big Data Analytics

Luca Morandini
Data Architect – AURIN Project
University of Melbourne
luca.morandini@unimelb.edu.au

Outline of the Lecture

- **Part 1:** Introduction to big data analytics
  - Types of analysis performed
  - Distributed computing on big data
- **Part 2:** Apache Hadoop
  - The Hadoop ecosystem
  - Hadoop Distributed File System
  - Hadoop architecture
  - Programming with Hadoop
- **Part 3:** Apache Spark
  - Why Spark
  - Spark Architecture
  - Programming in Spark
- **Part 4:** Apache Spark Workshop

Part 1: Introduction to Big Data Analytics

Big Data Analytics

- There would not be much point in amassing vast amount of data without being able to analyse it, hence the blossoming of large-scale *business intelligence* and more complex *machine learning* algorithms.

- There is a good deal of overlap and confusion among terms such as *business intelligence*, *machine learning*, *statistics*, and *data mining*. For the sake of clarity, we just use the more general term *(big) data analytics.*

Examples of Analytics

A non-exhaustive list of analysis typically performed on big data:

- Full-text searching (similar to the Google search engine)
- Aggregation of data (e.g. summing up the number of hits by web-page, day, month, etc.)
- Clustering (e.g. grouping customer into classes based on their spending habits)
- Sentiment analysis (e.g. deciding whether a tweet on a given topic express a positive or negative sentiment)
- Recommendations (suggesting additional products to a client during checkout based on the choices made by other clients who bought a similar product)

Challenges of Big Data Analytics

A framework for analysing big data has to distribute both data and processing over many nodes, which implies:

- Reading and writing distributed datasets
- Preserving data in the presence of failing data nodes
- Supporting the execution of MapReduce tasks
- Being fault-tolerant (a few failing compute nodes may slow down the processing, but not stop it)
- Coordinating the execution of tasks across a cluster

Tools for Analytics

- There are many tools that can perform data analytics, such as:
  - Statistical packages (R, Stata, SAS, SPSS, …)
  - Business Intelligence tools (Tableau, Business Objects, Pentaho, …)
  - Information retrieval tools (ElasticSearch, IBM TextExtender, …)

- However, when it comes to big data, the majority of applications are built on top of an open-source framework: **Apache Hadoop**

- In the following sections, the basics of Hadoop are presented

Part 2: Apache Hadoop

A bit of History

- Apache Hadoop started as an offshoot of the Apache Nutch project (circa 2005) which aimed at developing a complete open source search engine based on Lucene (the same sea4ch e fine of CouchDB)
- It become apparent that the amount of parallelism needed by web searches required a framework dedicated to managing hundreds (if not thousands) of nodes, hence the birth of Hadoop, later to be adopted by Yahoo!
- The Nutch project spawned another big data tool we mentioned: ElasticSearch
- Hadoop has kept on growing, and it is now the foundation of many big data solutions, both open source and proprietary

The Hadoop Ecosystem

Apache Hadoop started as a way to distribute files over a cluster and execute MapReduce tasks, but many tools have now been built on that foundation to add further functionality

- DBMSs (Hive, HBase, Accumulo)
- Business Intelligence tools (Apache Pig),
- Distributed configuration tools (Apache Zookeeper)
- Distributed resource management tools (YARN, Mesos)
- Machine learning libraries (Mahout)
- Object storage (Ozone): a key-value store optimised for big objects

Hadoop Distributed File System (HDFS)

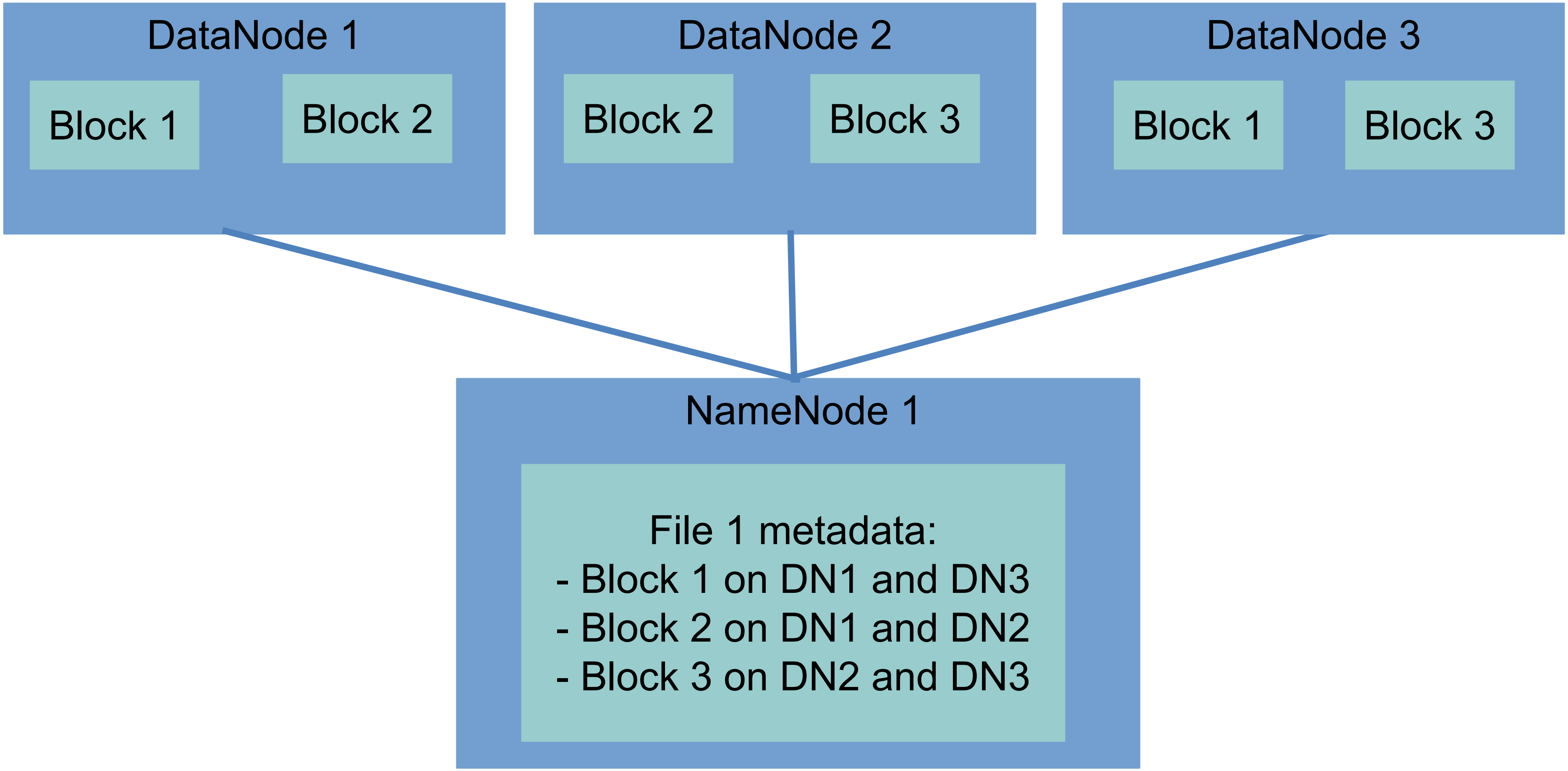
The core of Hadoop is a fault tolerant file system that has been explicitly designed to span many nodes

HDFS blocks are much larger than blocks used by an ordinary file system (say, 4 KB versus 128MB), the reasons for this unusual size are:

- Reduced need for memory to store information about where the blocks are (metadata)
- More efficient use of the network (with a large block, a reduced number network connections needs to be kept open)
- Reduced need for seek operations on big files
- Efficient when most data of a block have to be processed

HDFS Architecture

A HDFS file is a collection of blocks stored in ***datanodes***, with metadata (such as the position of those blocks) that is stored in ***namenodes***



DataNode 1

Block 1

Block 2

DataNode 2

Block 2

Block 3

DataNode 3

Block 1

Block 3

NameNode 1

File 1 metadata:
- Block 1 on DN1 and DN3
- Block 2 on DN1 and DN2
- Block 3 on DN2 and DN3

The Hadoop Resource Manager (YARN)

- The other main component of Hadoop is the MapReduce task manager, **YARN** (*Yet Another Resource Negotiator*)
- YARN deals with executing MapReduce jobs on a cluster. It is composed of a central *Resource Manager* (on the master) and many *Node Managers* that reside on slave machines.
- Every time a MapReduce job is scheduled for execution on a Hadoop cluster, YARN starts an *Application Master* that negotiates resources with the Resource Manager and starts *Containers* on the slave nodes (Note: Containers are the processes were the actual processing is done, *not to be confused with Docker containers*).

The HDFS Shell

- Managing the files on a HDFS cluster cannot be done on the operating system shell, hence a custom HDFS shell must be used.
- The HDFS file system shell replicates many of the usual commands (ls, rm, etc.), with some other commands dedicated to loading files from the operating system to the cluster (and back):

```
$HADOOP_HOME/bin/hadoop fs
  -copyFromLocal <localsrc> <dst>
```

- In addition, the status of the HDFS cluster and its contents can be seen on a web application (normally found on port `50070` of the master node).

Programming on Hadoop

The main programming language to write MapReduce jobs on Hadoop is Java, but many other languages can be used via different APIs.

Indeed any language that can read from standard input and write to standard output can be used.

Practically, the `hadoop` command is used to load the program (with the `-file` option) and send it to the cluster, and the mapper and reducer are specified with the `-mapper` and `-reducer` options (`aggregate` uses the Hadoop internal aggregator library):

```
$HADOOP_HOME/bin/hadoop jar \
 $HADOOP_HOME/hadoop-streaming.jar \
    -D mapreduce.job.reduces=12 \
    -input myInputDir \
    -output myOutputDir \
    -mapper myAggregatorForKeyCount.py \
    -reducer aggregate \
    -file myAggregatorForKeyCount.py
```

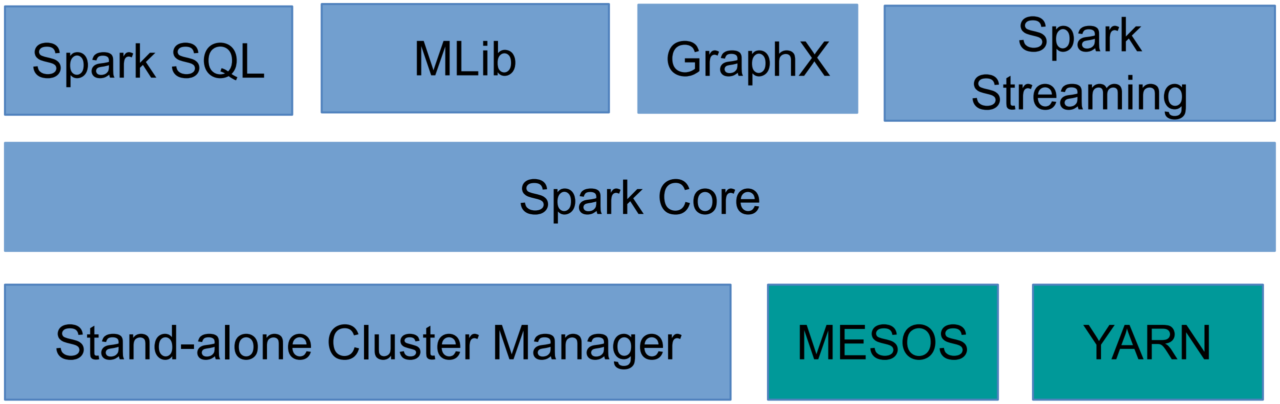
Part 3: Apache Spark

Why Spark?

- While Hadoop MapReduce works well, it is geared towards performing relatively simple jobs on large datasets.
- However, when complex jobs are performed (say, machine learning or graph-based algorithms), there is a strong incentive for caching data in memory and in having finer-grained control on the execution of jobs.
- Apache Spark was designed to reduce the latency inherent in the Hadoop approach for the execution of MapReduce jobs.
- Spark can operate within the Hadoop architecture, using YARN and Zookeeper to manage computing resources, and storing data on HDFS.

Spark Architecture

- One of the strong points of Spark is the tightly-coupled nature of its main components:



- Spark ships with a cluster manager of its own, but it can work with other cluster managers, such as YARN or MESOS.

Programming on Spark

- Spark is mostly written in Scala, and uses this language by default in its interactive shell. However, the APIs of Spark can be accessed by different languages: R, Python, and Java.

- Scala is a multi-paradigm language (both functional and object-oriented) that runs on the Java Virtual Machine and can use Java libraries and Java objects.

- The most popular languages used to develop Spark applications are Java and Python.

- Other than Scala, there is a shell for Python (pySpark)

Getting Data In and Out of Spark #1

- Spark can read (and write) data in many formats, from text file to database tables, and it can use different file systems and DBMSs.

- The simplest way to get data into Spark is reading them from a CSV file from the local file system:

  ```
  csv = sc.textFile("file.csv")
  ```

- With a small change, Spark can be made to read from HDFS file systems (or Amazon S3):

  ```
  csv = sc.textFile("hdfs://file.csv")
  csv= sc.textFile("s3://myBucket/myFile.csv")
  ```

- Of course, the text lines in the file would have to be parsed and put into objects before they can be used by Spark.

Getting Data In and Out of Spark #2

- Another popular format is JSON, which can be parsed (and streamed back into a file) using Java libraries such as Jackson or Gson.

- An efficient data format that is unique to Hadoop is the *sequence file*. This is a flat file composed of key/value pairs.

- Another option to load/save data is the use of serialised Java objects (the Kryo library, rather than the native Java serialization is commonly used).

- While this option is simple to implement (the majority of Java objects can be serialised), it is neither fast nor robust (any change to the original object structure would make the serialised file impossible to read back).

Getting Data In and Out of Spark #3

- HDFS or distributed DBMSs (such as Hive, Cassandra or Accumulo) can be used in conjunction with Spark.

- SQL queries can also be used to extract data:

```
df = sqlContext.sql("SELECT * FROM table")
```

- Relational DBMSs can be a source of data too, e.g. via JDBC.

- CouchDB, MongoDB and ElasticSearch connectors are also available.

The Spark Shell

- The Spark Shell allows to send commands to the cluster interactively in either Scala or Python.

- A simple program* in Python to count the occurrences of the word “Spark” in the README of the framework.

```
./bin/pyspark
>>> textFile = sc.textFile("README.md")
>>> textFile.filter(lambda line: "Spark"
                     in line).count()
15
```

- While the shell can be extremely useful, it prevents Spark from deploying all of its optimizations, leading to poor performance.

* *Taken from the Spark documentation*

Non-interactive Jobs in Spark

The usual word count, but in Java 7 (from the Spark website).

```
JavaRDD<String> input = sc.textFile(inputFile);
JavaRDD<String> words = input.flatMap(
      new FlatMapFunction<String, String>() {
        public Iterable<String> call(String x) {
          return Arrays.asList(x.split(" "));
        }});
JavaPairRDD<String, Integer> counts =
words.mapToPair(new PairFunction<String, String,
 Integer>(){
     public Tuple2<String, Integer> call(String x){
          return new Tuple2(x, 1);
   }}).reduceByKey(new Function2<Integer, Integer, Integer>() {
  public Integer call(Integer x, Integer y){ return x + y;}});
counts.saveAsTextFile(outputFile);
```

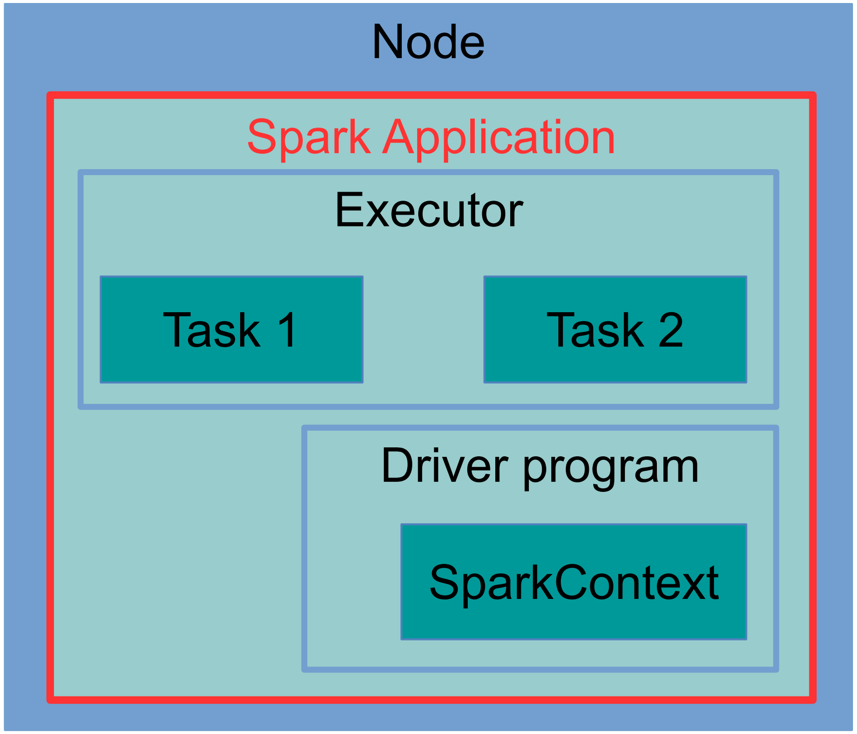
Spark Runtime Architecture

- Applications in Spark are composed of different components including:
  - ***Job***: the data processing that has to be performed on a dataset
  - ***Task***: a single operation on a dataset
  - ***Executors***: the processes in which tasks are executed
  - ***Cluster Manager***: the process assigning tasks to executors
  - ***Driver program***: the main logic of the application
  - ***Spark application***: Driver program + Executors
  - ***Spark Context***: the general configuration of the job

- These different components can be arranged in three different deployment modes across the cluster.

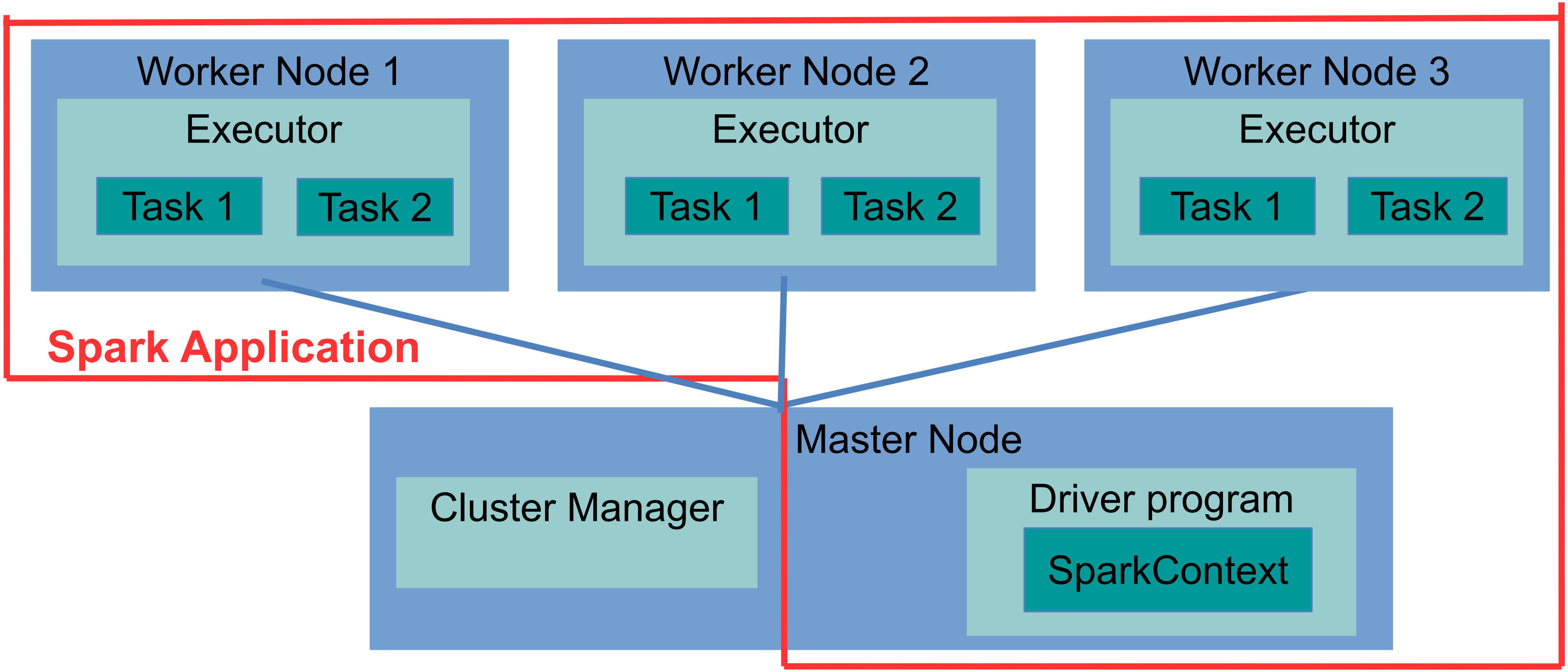
Spark Runtime Architecture: Local Mode

In local mode, every Spark component runs within the same JVM. However, the Spark application can still run in parallel, as there may be more than one executor active. (Local mode is good when developing/debugging)



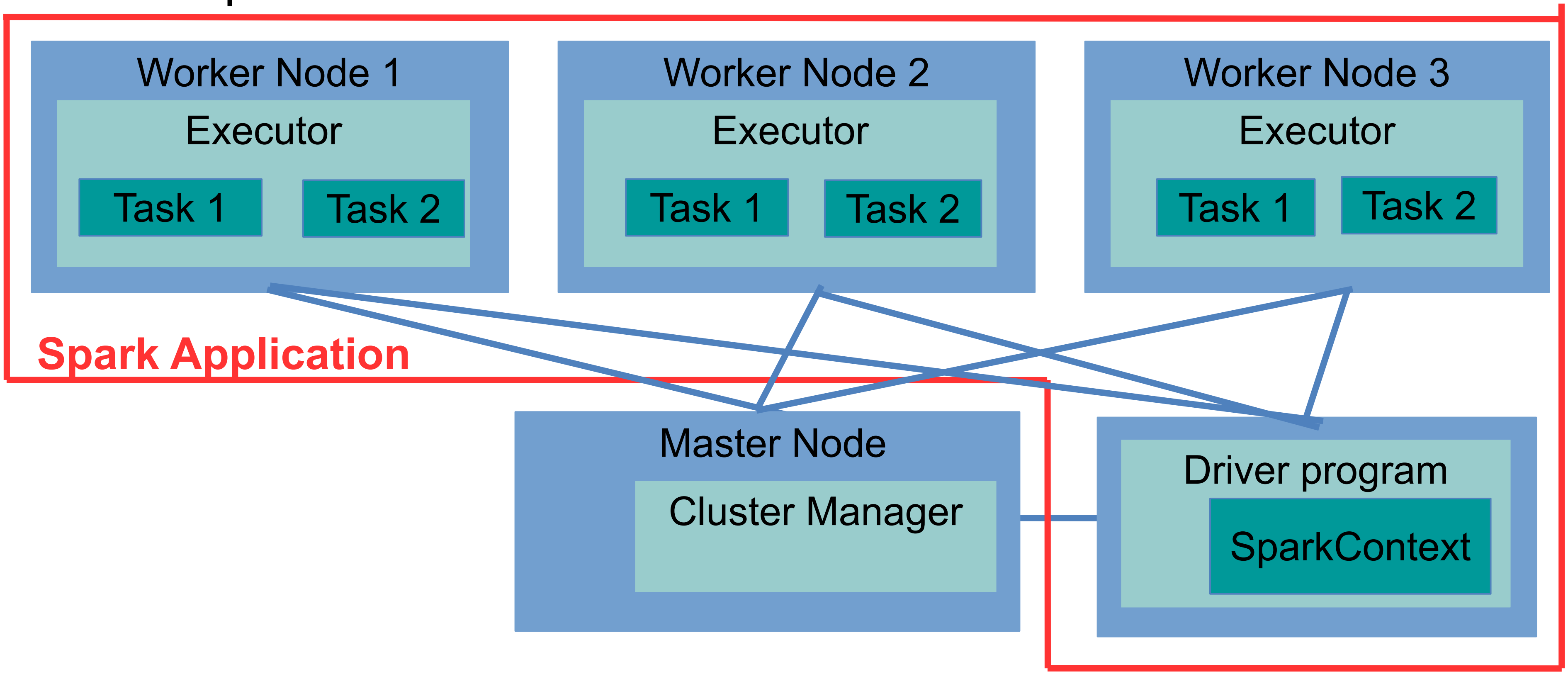
Spark Runtime Architecture: Cluster Mode

In cluster mode, every component, including the driver program, is executed on the cluster; hence, upon launching, the job can run autonomously. This is the common way of running non-interactive Spark jobs.



Spark Runtime Architecture: Client Mode

In client mode, the driver program talks directly to the executors on the worker nodes. Therefore, the machine hosting the driver program has to be connected to the cluster until job completion. Client mode must be used when the applications are interactive, as happens in the Python or Scala Spark shells.



Spark Context

- The deployment mode is set in the *Spark Context*, which is also used to set the configuration of a Spark application, including the cluster it connects to in cluster mode.
- For instance, this hard-coded Spark Context directs the execution to run locally, using 2 threads (usually, it is set to the number of cores):

```
sc = new SparkContext(new
 SparkConf().setMaster("local[2]"));
```

- This other hard-coded line directs the execution to a remote cluster:

```
sc = new SparkContext(new SparkConf()
.setMaster("spark://192.168.1.12:6066"));
```

- Spark Contexts can also be used to tune the execution by setting the memory, or the number of executors to use.

How to Submit Java Jobs to Spark

- For an application to be executed on Spark, either a shell or a submit script has to be used. The submit script is to be given all the information it needs:

```
./bin/spark-submit \
  --class <main-class>
  --master <master-url> \
  --deploy-mode <deploy-mode> \
  --conf <key>=<value> \
  <application-jar (an ubër-JAR)> \
  [application-arguments]
```

- The application JAR must be accessible from all the nodes in cluster deploy mode, hence it is usually put on HDFS.
- The submit script can be used to launch Python programs as well. Uber-JARs can be assembled by Maven with the *Shade* plugin.

Introducing the Resilient Distributed Dataset

Resilient Distributed Datasets (RDDs) are the way data are stored in Spark during computation, and understanding them is crucial to writing programs in Spark:

- **R**esilient (data are stored redundantly, hence a failing node would not affect their integrity)

- **D**istributed (data are split into chunks, and these chunks are sent to different nodes)

- **D**ataset (a dataset is just a collection of objects, hence very generic)

Properties of RDDs

- RDDs are *immutable*, once defined, they cannot be changed (this greatly simplifies parallel computations on them, and is consistent with the functional programming paradigm)

- RDDs are *transient*, they are meant to be used only once, then discarded (but they can be cached, if it improves performance)

- RDDs are *lazily-evaluated*, the evaluation process happens only when data cannot be kept in an RDD, as when the number of objects in an RDD has to be computed, or an RDD has to be written to a file (these are called *actions)*, but not when an RDD are transformed into another RDD (these are called *transformations*)

How to Build an RDD

- RDDs are usually created out of data stored elsewhere (HDFS, a local text file, a DBMS), as in:

```
JavaRDD<String> lines = sc.textFile("data.txt");

DataSet<Row> teenagers = sparkSession.sql(
"SELECT name FROM table WHERE age >= 13 AND age <=
19");
JavaRDD<Row> rddTeenagers= teenagers.javaRDD();
```

- RDDs can be created out of collections too, using the parallelize function:

```
List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);
JavaRDD<Integer> distData = sc.parallelize(data);
```

An Example of a Java 8 Driver Program using RDDs (counting words)

```
JavaRDD<String> input = sc.textFile("./README.md");
JavaRDD<String> words = input.flatMap(document -> {
  return Arrays.asList(document.split(" "));
}); // Yes, a Lambda expression!

JavaPairRDD<String, Integer> wordPairs =
  words.mapToPair((String w) -> {
    return new Tuple2<String, Integer>(w, 1);
  });

JavaPairRDD<String, Integer> results =
  wordPairs.reduceByKey((a, b) -> {
    return a + b;
  });

results.saveAsTextFile(outputFile);
```

Aside: Lambda Expressions

- Java 8 introduced *lambda expressions*, which are a functional programming feature of many languages – often known as *closures* or, in Javascript, as *callbacks*, or, in Scala and Python as *lambda functions*.

```
List<Integer> values = new ArrayList<Integer>();
Integer[] array = { 10, 20, 30 };
Collections.addAll(values, array);
List<Integer> result = new ArrayList<Integer>();
values.forEach((n) -> {
  result.add(n * 2);
});
```

- The list of parameters to the expression are listed before the *arrow token* (“->”) and their data types are (usually) inferred; the body of the expression follows the arrow token.
- Lambda expressions can access variables defined in the same scope of the expression.

A Few Points About the Example

- RDD transformations use Lambda expressions (closures) to simplify programming

- The only *action* in this program is `saveAsTextFile`, all the others are *transformations*

- The transformations in the program (`flatMap`, `mapToPair`, `reduceByKey`) use lazy evaluation, hence Spark has the possibility of optimizing the process

- RDD variables are just placeholders until the action is encountered. Remember that the Spark application is not just the driver program, but all the RDD processing that takes place on the cluster

Let's Make the Execution Order Visible

```
System.out.println("Before read");
JavaRDD<String> input = sc.textFile("./a.txt");
System.out.println("After read");
JavaRDD<String> words = input.flatMap(document->{
  System.out.println("flatMap");
  return Arrays.asList(document.split(" "));
});
JavaPairRDD<String, Integer> wordPairs =
  words.mapToPair((String w) -> {
    System.out.println("mapToPair");
    return new Tuple2<String, Integer>(w, 1);
});
JavaPairRDD<String, Integer> results =
  wordPairs.reduceByKey((a, b) -> {
    System.out.println("reduceByKey");
    return a + b;
});
System.out.println("Before write");
results.saveAsTextFile("./counts");
System.out.println("After write");
```

```
Before read

After read
Before write
flatMap
mapToPair
mapToPair
flatMap
mapToPair
mapToPair
...
mapToPair
reduceByKey
mapToPair
reduceByKey
mapToPair
flatMap
mapToPair
reduceByKey
After write
```

Order of Execution of MapReduce Tasks

- While the execution order of Hadoop MapReduce is fixed, the lazy evaluation of Spark allows the developer to stop worrying about it, and have the Spark optimizer take care of it.

- In addition, the driver program can be divided into steps that are easier to understand without sacrificing performance (as long as those steps are composed of transformations).

Example of Transformations of RDDs

- **`rdd.filter(lambda)`** selects elements from an RDD

- **`rdd.distinct()`** returns an RDD without duplicated elements

- **`rdd.union(otherRdd)`** merges two RDDs

- **`rdd.intersection(otherRdd)`** returns elements common to both

- **`rdd.subtract(otherRdd)`** removes elements of otherRdd

- **`rdd.cartesian(otherRdd)`** returns the Cartesian product of both RDDs

Examples of Actions

- **`rdd.collect()`** returns all elements in an RDD

- **`rdd.count()`** returns the number of elements in an RDD

- **`rdd.reduce(lambda)`** applies the function to all elements repeatedly, resulting in one result (say, the sum of all elements. Not to be confused with the **`reduceByKey`** transformation)

- **`rdd.foreach(lambda)`** applies lambda to all elements of an RDD

Examples of Key/Value Pairs Transformations

- `rdd.map(lambda)` creates a key/value pair RDD by applying function lambda and returning one pair for element

- `rdd.flatMap(lambda)` applies a function to RDD elements and returns zero, one, or more pairs for element

- `rdd.reduceByKey(lambda)` processes all the pairs with the same key into one pair

- `rdd.join(otherRdd)` merges two key/value pair RDDs based on a key

Caching Intermediate Results

- `rdd.persist(storageLevel)` can be used to save an RDD either in memory and/or disk. The `storageLevel` can be tuned to a different mix of use of RAM or disk to store the RDD

- Note: since RDDs are immutable, the result of the final transformation is cached, not the input RDD. In other words, when this statement is executed

```
rddB = rddA.persist(DISK_ONLY)
```

only `rddB` has been written to disk.

Tuning the Degree of Parallelism

- Spark partitions data across the cluster according to some heuristic. However, sometimes it is useful to force it to partition data in a given number of pieces.

- Some transformations allow for a second parameter containing the desired number of partitions, e.g.

  ```
  sc.textFile(“bigfile.csv”,10)
  ```

- An RDD can also be re-partitioned explicitly:

  ```
  rdd.repartion(partitionNum)
  ```

- Another way to partition an RDD is to provide a *partitioner*, that is, a strategy to guide the partitioning. For instance,

  ```
  rdd.partitionBy(new HashPartitioner(100))
  ```

  splits RDD elements into 100 partitions according to theit keys

Spark Jobs, Tasks, and Stages

- A ***Job*** is the overall processing that Spark is directed to perform by a driver program

- A ***Task*** is a single transformation operating on a single partition of data on a single node

- A ***Stage*** is a set of tasks operating on a single partition

- A Job is composed of more than one stage when data are to be transferred across nodes (shuffling)

- The fewer the number of stages, the faster the computation (shuffling data across the cluster is slow)

References

- *Hadoop: The Definitive Guide, 4th edition*, Chris White, O'Reilly, 2015

- *Learning Spark,* H. Karau, A. Konwinski, P. Wendell, M. Zaharia, O'Reilly, 2015

- *Java The Complete Reference - Eleventh Edition*, by Herbert Schildt, Oracle Press, 2018

- *Functional JavaScript*, M. Fogus, O’Reilly, 2013