CS523 - BDT
Big Data Technologies

Apache Spark
(In-Memory Cluster Computation)

What is Apache Spark?

- Fast and general purpose engine for large scale data processing
- Provides a framework that supports in-memory cluster computing
- Well suited for graph and machine learning algorithms (iterative computation) and interactive data mining.
- Rich set of APIs in Java, Scala, Python and R for performing many common data processing tasks, such as joins.
- Integrated higher level libraries
- Spark also comes with a REPL (read-eval-print-loop) for both Scala and Python, which makes it quick and easy to explore datasets. (Spark shell)
- Spark uses MapReduce idea but not implementation. It has its own distributed runtime for executing work on a cluster.

Spark Motivation

- Matei Zaharia created Spark for his PhD at University of California Berkeley, in response to the limitations he had seen in MapReduce while working in summer internships at early Hadoop users, including Facebook.

- MapReduce couldn't do interactive queries and it couldn't handle advanced algorithms, such as machine learning.
- As big data analytics evolves beyond simple batch jobs, there is a need for both more complex multi-stage applications (e.g. machine learning algorithms) and more interactive ad-hoc queries.
- Spark makes efficient use of memory and can execute equivalent jobs 10 to 100 times faster than Hadoop's MapReduce.

Hadoop vs. Spark - An Answer to the Wrong Question

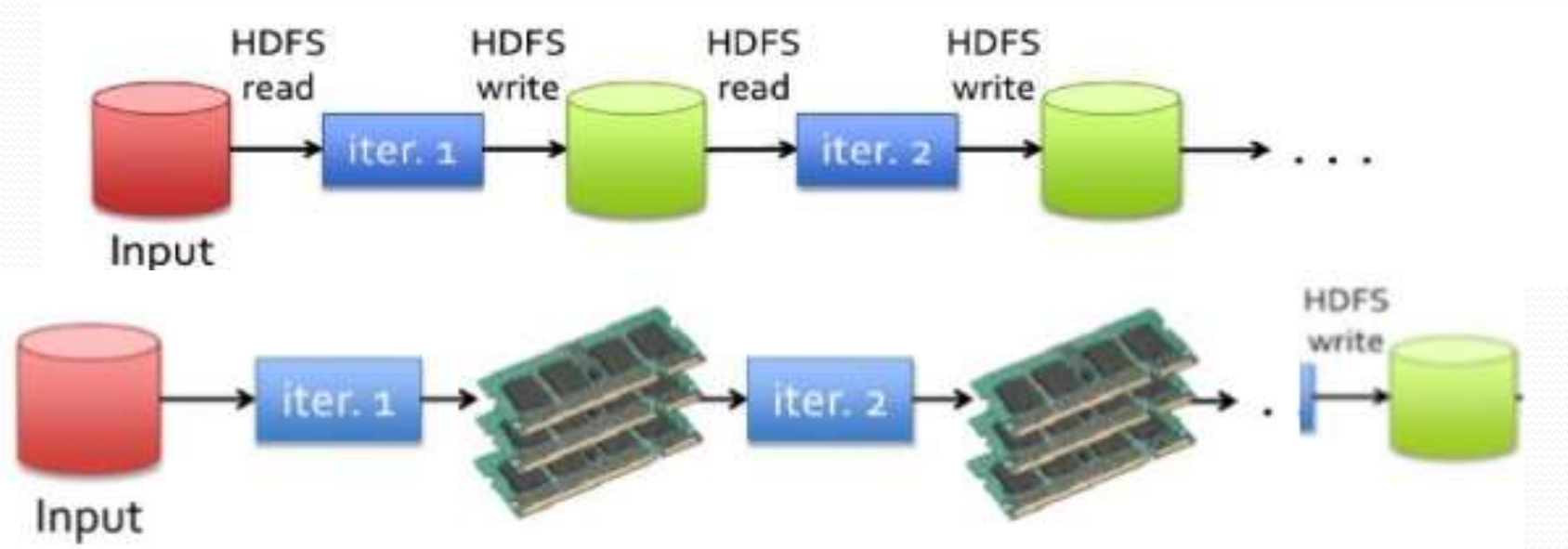
- Spark is not, despite the hype, a replacement for Hadoop. Nor is MapReduce dead.
- Spark can run on top of Hadoop, benefiting from Hadoop's cluster manager (YARN) and underlying storage (HDFS, HBase, etc.).
- Spark can also run completely separately from Hadoop, integrating with alternative cluster managers like Mesos and alternative storage platforms like Cassandra and Amazon S3.
- Much of the confusion around Spark's relationship to Hadoop dates back to the early years of Spark's development. At that time, Hadoop relied upon MapReduce for the bulk of its data processing.
- Hadoop MapReduce also managed scheduling and task allocation processes within the cluster; even workloads that were not best suited to batch processing were passed through Hadoop's MapReduce engine, adding complexity and reducing performance.

Hadoop vs. Spark - An Answer to the Wrong Question

- MapReduce is really a programming model. In Hadoop MapReduce, multiple MapReduce jobs would be strung together to create a data pipeline.
- In between every stage of that pipeline, the MR code would read data from the disk, and when completed, would write the data back to the disk.
- This process was inefficient because it had to read all the data from disk at the beginning of each stage of the process. This is where Spark comes in to play.
- Taking the same MR programming model, Spark was able to get an immediate 10x increase in performance, because it didn't have to store the data back to the disk, and all activities stayed in memory.

Hadoop vs. Spark - An Answer to the Wrong Question

- Spark offers a far faster way to process data than passing it through unnecessary Hadoop MapReduce processes.



Hadoop vs. Spark - An Answer to the Wrong Question

- Hadoop has since moved on with the development of the YARN cluster manager, thus freeing the project from its early dependence upon Hadoop MapReduce.
- Hadoop MapReduce is still best for running static batch processes.
- Other data processing tasks can be assigned to different processing engines (including Spark), with YARN handling the management and allocation of cluster resources.
- Spark is a viable alternative to Hadoop MapReduce in a range of other circumstances.
- Spark is not a replacement for Hadoop, but is instead a great companion to a modern Hadoop cluster deployment.

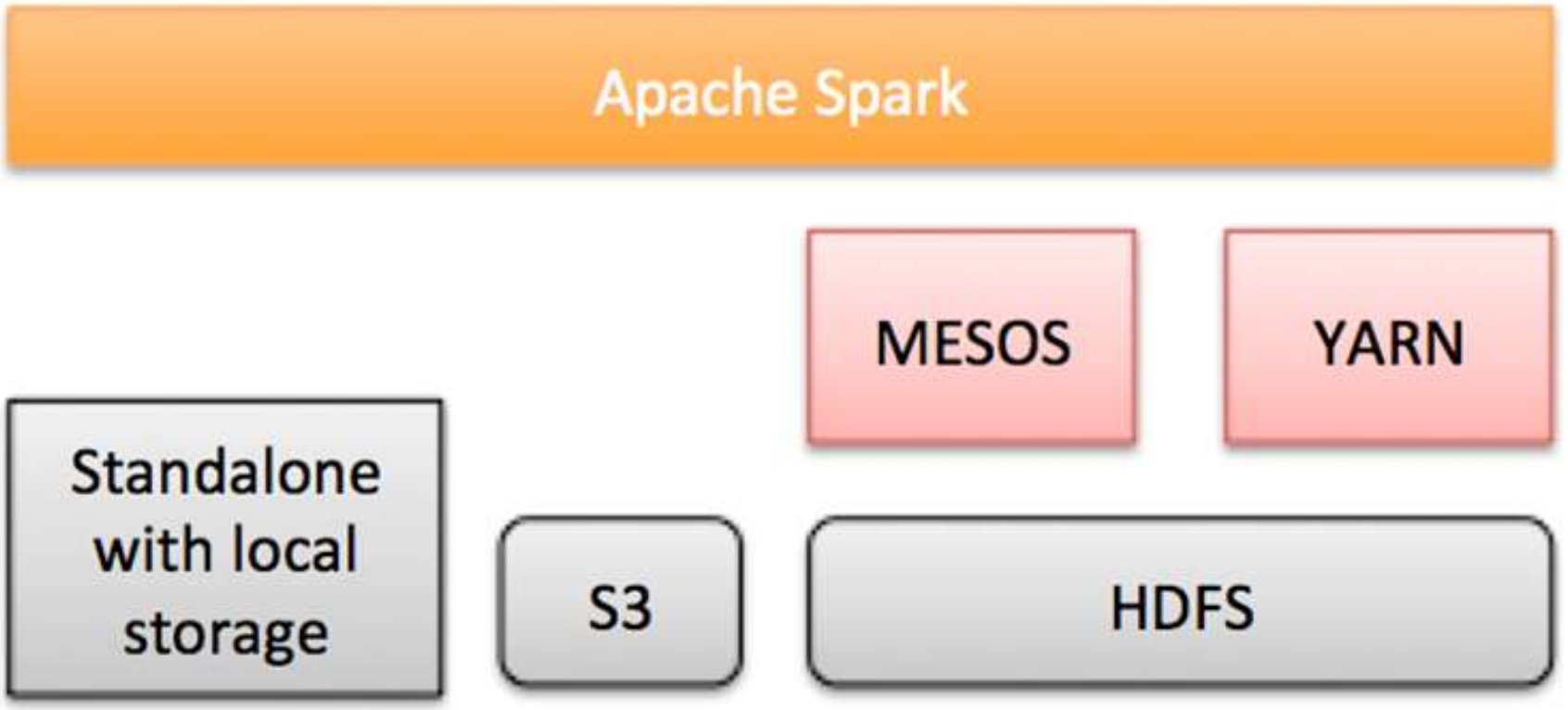
What Hadoop Gives Spark

- Apache Spark is often deployed in conjunction with a Hadoop cluster, and Spark is able to benefit from a number of capabilities as a result.
- **YARN resource manager**, which takes responsibility for scheduling tasks across available nodes in the cluster;
- **Distributed File System**, which stores data when the cluster runs out of free memory, and which persistently stores historical data when Spark is not running;
- **Disaster Recovery capabilities**, inherent to Hadoop, which enable recovery of data when individual nodes fail.
- **Data Security**, which becomes increasingly important as Spark tackles production workloads in regulated industries such as healthcare and financial services. Projects like Apache Knox and Apache Ranger offer data security capabilities that augment Hadoop.
- **A distributed data platform**, meaning that Spark jobs can be deployed on available resources anywhere in a distributed cluster, without the need to manually allocate and track those individual jobs.

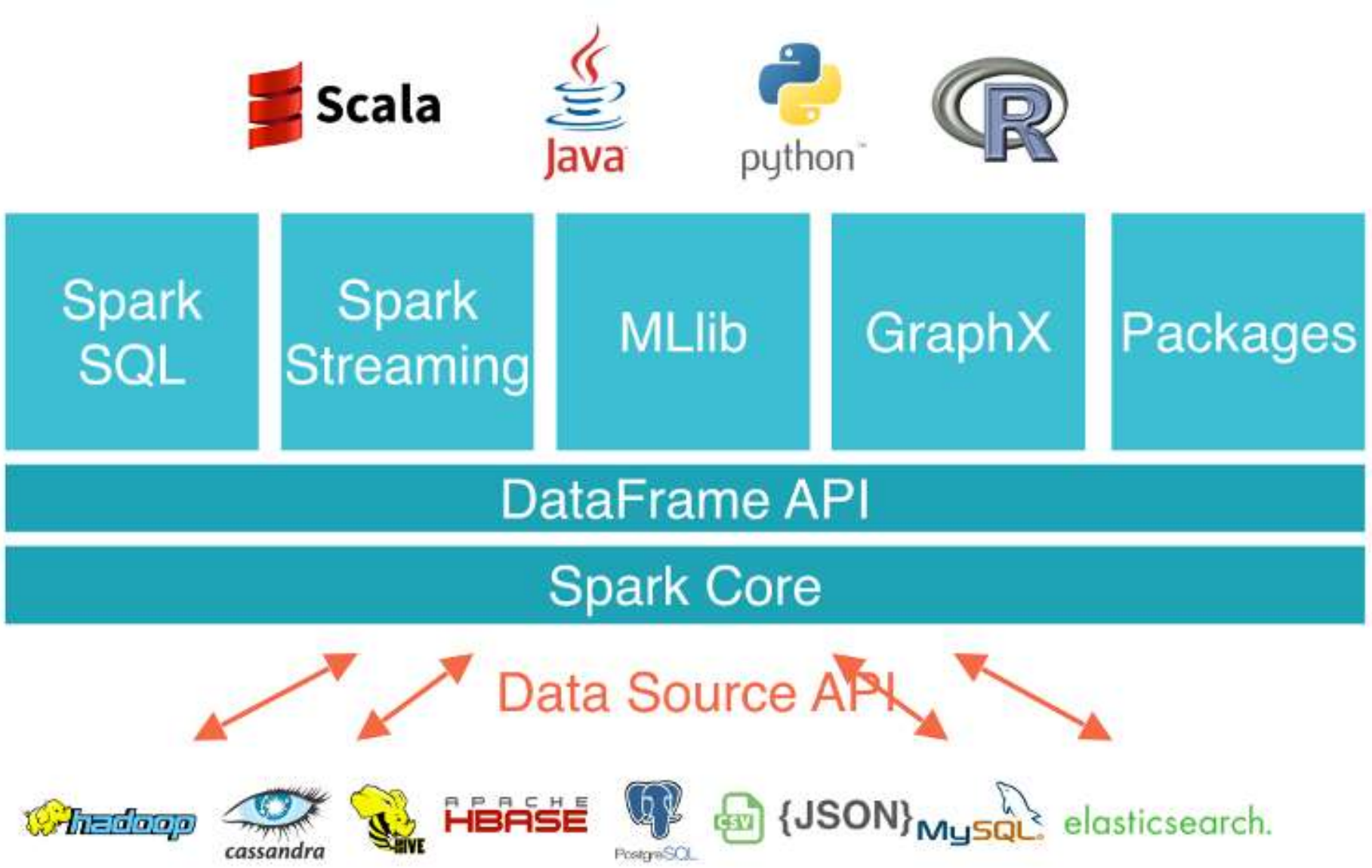
What Spark Gives Hadoop

- Hadoop has come a long way since its early versions which were essentially concerned with facilitating the batch processing of MapReduce jobs on large volumes of data stored in HDFS. Particularly since the introduction of the YARN resource manager, Hadoop is now better able to manage a wide range of data processing tasks, from batch processing to streaming data and graph analysis.
- Spark is able to contribute, via YARN, to Hadoop-based jobs. In particular, Spark's machine learning module delivers capabilities not easily exploited in Hadoop without the use of Spark.
- Spark's original design goal, to enable rapid in-memory processing of sizeable data volumes, also remains an important contribution to the capabilities of a Hadoop cluster.
- In certain circumstances, Spark's SQL capabilities, streaming capabilities (otherwise available to Hadoop through Storm, for example), and graph processing capabilities (otherwise available to Hadoop through Neo4J or Giraph) may also prove to be of value in enterprise use cases.

Executing Spark Applications



Spark Components



Spark Modules

- **Spark Core :-** is the heart of Spark, and is responsible for management functions such as task scheduling.
  - Spark Core implements and depends upon a programming abstraction known as Resilient Distributed Datasets (RDDs).
- **Spark SQL :-** is Spark's module for working with structured data, and it is designed to support workloads that combine familiar SQL database queries with more complicated, algorithm-based analytics.
  - Spark SQL supports the open source Hive project, and its SQL-like HiveQL query syntax.
  - Spark SQL also supports JDBC and ODBC connections, enabling a degree of integration with existing databases, data warehouses and business intelligence tools.
  - JDBC connectors can also be used to integrate with Apache Drill, opening up access to an even broader range of data sources.

Spark Modules

- **Spark Streaming :-** This module supports scalable and fault-tolerant processing of streaming data, and can integrate with established sources of data streams like Flume (optimized for data logs) and Kafka (optimized for distributed messaging).
  - Spark Streaming's design, and its use of Spark's RDD abstraction, are meant to ensure that applications written for streaming data can be repurposed to analyze batches of historical data with little modification.
- **Spark Mlib :-** This is Spark's scalable machine learning library, which implements a set of commonly used machine learning and statistical algorithms.
  - These include correlations and hypothesis testing, classification and regression, clustering, and principal component analysis.

Spark Modules

- **Spark GraphX**: **:-** This module began life as a separate UC Berkeley research project, which was eventually donated to the Apache Spark project.
  - GraphX supports analysis of and computation over graphs of data, and supports a version of graph processing's Pregel API.
  - GraphX includes a number of widely understood graph algorithms, including PageRank.
- **Spark R :-** This module was added to the 1.4.x release of Apache Spark, providing data scientists and statisticians using R with a lightweight mechanism for calling upon Spark's capabilities.

DataFrames API

- An additional DataFrames API was added to Spark in 2015.
- DataFrames **offer**:
  - Ability to scale from kilobytes of data on a single laptop to petabytes on a large cluster
  - Support for a wide array of data formats and storage systems
  - State-of-the-art optimization and code generation through the Spark SQL Catalyst optimizer
  - Seamless integration with all big data tooling and infrastructure via Spark APIs for Python, Java, Scala, and R
- This extended API will ease application development, while helping to improve performance via the optimizations and code generation.

Resilient Distributed Dataset (RDD)

- Central abstraction in Spark: a read-only distributed collection of objects that is partitioned across multiple machines in a cluster. (In Spark terminology, each HDFS block is called *partitions (used to be called as splits)*)
- RDD is the representation of your data in object format that is coming into your system and allows you to perform computations on that data.
- Once created, RDDs never change
- RDDs are built through parallel transformations (map, filter, reduce, groupBy etc.) - Generate RDD from other RDD
- Lazy operations that builds a DAG (Directed Acyclic Graph)
- Automatically rebuilt on failure using lineage.
- Controllable persistence (RAM, HDFS, etc.)

Resilient Distributed Dataset (RDD)

- A collection (array) like this [1,2,3,4] would become like [1], [2], [3], [4] distributed over the cluster.
- The fact that the collection is *distributed* on a number of machines is transparent to its users, so working with RDDs is very similar to working with ordinary local collections like plain old lists, maps, sets, and so on.
- Fault-tolerance is achieved, in part, by tracking the lineage of transformations applied to coarse grained sets of data.
- In addition to automatic fault tolerance and distribution, the RDD provides an elaborate API, which allows you to work with a collection in a functional style.
- You can filter the collection; map over it with a function; reduce it to a cumulative value; subtract, intersect, or create a union with another RDD, and so on.

RDD Traits

- In memory
- Immutable
- Lazily evaluated
- Typed
- Parallel
- Partitioned
- Cached

Create RDD

There are 2 ways to create RDD:

- Parallelizing an existing collection, example an array.
  - **Parallelized Collection**

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

- Create from an existing storage, example a file in local or distributed file system
  - **External Data Set**

```
val distFile = sc.textFile (“data.txt”)
```

Transformations & Actions

- **2 types of RDD operations**: **Transformations & actions**
- Transformation generates a new RDD from an existing one, while an action triggers a computation on an RDD and does something with the results—either returning them to the user, or saving them to external storage.
- In a typical Spark program, one or more RDDs are loaded as input and through a series of transformations are turned into a set of target RDDs, which have an action performed on them (such as computing a result or writing them to persistent storage).
- Loading an RDD or performing a transformation on one does not trigger any data processing; it merely creates a plan for performing a computation. The computation is only triggered when an action (like foreach, count, collect, reduce, save) is performed on an RDD.
- Actions have an immediate effect, but transformations do not—they are lazy, in the sense that they don't perform any work until an action is performed on the transformed RDD.

Transformations & Actions Example

- For example, the following commands lowercases lines in a text file:

```
val text = sc.textFile(inputPath)
val lower: RDD[String] = text.map(_.toLowerCase())
lower.foreach(println(_))
```

- The `map()` method is a transformation, which Spark represents internally as a function (*`toLowerCase()`*) to be called at some later time on each element in the input RDD (`text`).
- The function is not actually called until the *`foreach()`* method (which is an action) is invoked and Spark runs a job to read the input file and call *`toLowerCase()`* on each line in it, before writing the result to the console.
- One way of telling if an operation is a transformation or an action is by looking at its return type: if the return type is RDD, then it's a transformation; otherwise, it's an action.
- A transformed RDD can be persisted in memory so that subsequent operations on it are more efficient.

Spark Context

- The Spark context is the entry point for interacting with Spark.
- You use it for things like connecting to Spark from an application, configuring a session, managing job execution, loading or saving a file, and so on.

```
JavaSparkContext sc = new JavaSparkContext(new
                      SparkConf().setAppName("wordCount")
                                 .setMaster("local"));
```

Transformations & Actions

Transformations
map (func)
flatMap(func)
filter(func)
groupByKey(func)
reduceByKey(func)
mapValues(func)
.....

Actions
take (N)
count()
collect()
reduce(func)
takeOrdered(N)
top(N)
.....

https://training.databricks.com/visualapi.pdf

Additional Transformations and Actions

Where	Function	Description
SparkContext	doubleRDDToDoubleRDDFunctions	Extra functions available on RDDs of Doubles
SparkContext	numericRDDToDoubleRDDFunctions	Extra functions available on RDDs of Doubles
SparkContext	rddToPairRDDFunctions	Extra functions available on RDDs of (key, value) pairs
SparkContext	hadoopFile()	Get an RDD for a Hadoop file with an arbitrary InputFormat
SparkContext	hadoopRDD()	Get an RDD for a Hadoop file with an arbitrary InputFormat
SparkContext	makeRDD()	Distribute a local Scala collection to form an RDD
SparkContext	parallelize()	Distribute a local Scala collection to form an RDD
SparkContext	textFile()	Read a text file from a file system URI
SparkContext	wholeTextFiles()	Read a directory of text files from a file system URI

Spark Word Count in Java 7

```
// Load our input data
JavaRDD<String> input = sc.textFile(inputFile);

// Split up into words
JavaRDD<String> words = input.flatMap(new FlatMapFunction<String, String>() {
   public Iterator<String> call(String x) {
      return Arrays.asList(x.split(" ")).iterator();
   }
});

// Transform into word and count
JavaPairRDD<String, Integer> counts =
      words.mapToPair(new PairFunction<String, String, Integer>() {
         public Tuple2<String, Integer> call(String x) {
            return new Tuple2<String, Integer>(x, 1);
         }
      }).reduceByKey(new Function2<Integer, Integer, Integer>() {
         public Integer call(Integer x, Integer y) {
            return x + y;
         }
      });

// Save the word count back out to a text file, causing evaluation.
counts.saveAsTextFile(outputFile);
```

Spark Word Count in Java 8

```
// Load our input data
JavaRDD<String> lines = sc.textFile(args[0]);

// Calculate word count
JavaPairRDD<String, Integer> counts = lines
        .flatMap(line -> Arrays.asList(line.split(" ")).iterator())
        .mapToPair(w -> new Tuple2<String, Integer>(w, 1))
        .reduceByKey((x, y) -> x + y);

// Save the word count back out to a text file, causing evaluation
counts.saveAsTextFile(args[1]);
```

Spark Word Count in Scala and Python

Scala

```scala
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Python

```python
text_file = sc.textFile("hdfs://...")
counts = text_file.flatMap(lambda line: line.split(" ")) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

RDD Persistence

- Most RDD operations are lazy. Think of an RDD as a description of a series of operations. An RDD is not data. Consider the following line:

```
JavaRDD<String> logData = sc.textFile(logFile)
```

- It does nothing. It creates an RDD that says "we will need to load this file". The file is not loaded at this point.

- RDD operations that require observing the contents of the data cannot be lazy. (These are called actions.)
- An example is `RDD.count()` — to tell you the number of lines in the file, the file needs to be read. So if you write `logData.count()`, at this point the file will be read, the lines will be counted, and the count will be returned.

RDD Persistence

- What if you call `logData.count()`again? The same thing: the file will be read and counted again. Nothing is stored. An RDD is not data.

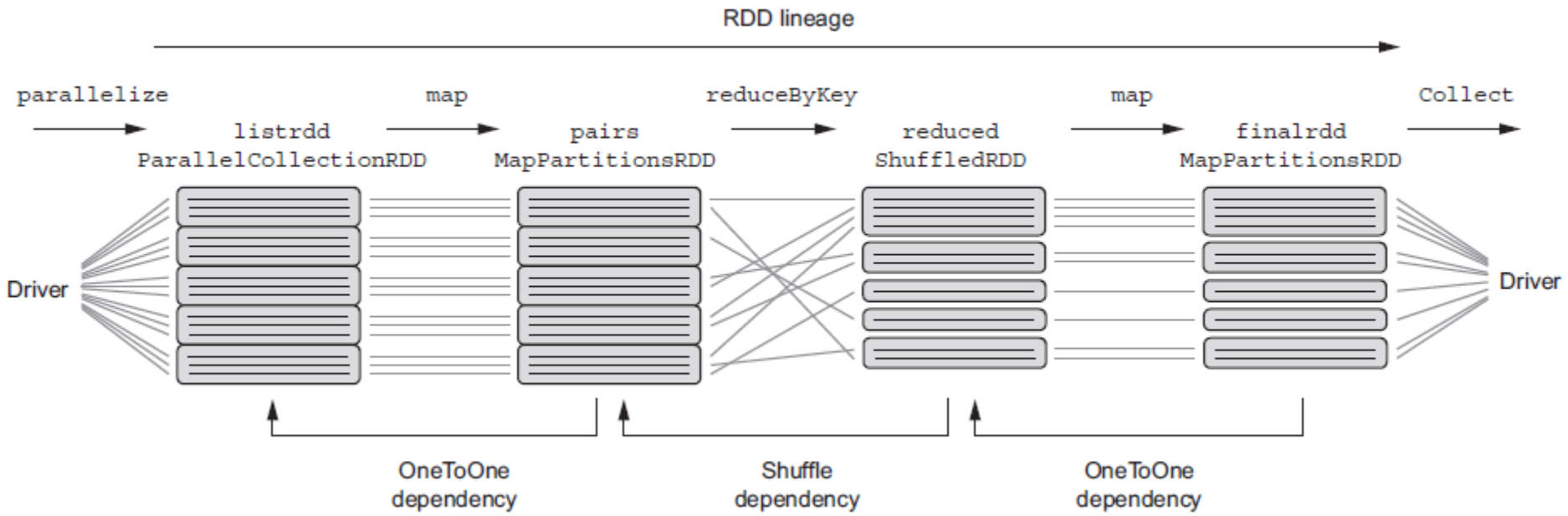
- So what does `logData.cache()` do?
- It does nothing. RDD.cache() is also a lazy operation. The file is still not read. But now the RDD says “we’ll need to read this file and then cache the contents".
- If you then run `logData.count()` the first time, the file will be loaded, cached, and counted. If you call `logData.count()` a second time, the operation will use the cache. It will just take the data from the cache and count the lines.

- The cache behavior depends on the available memory. If the file does not fit in the memory, for example, then `logData.count()` will fall back to the usual behavior and re-read the file.

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

public class SimpleApp {
  public static void main(String[] args) {
    String logFile = "input/logFile.txt";
    SparkConf conf = new SparkConf().setAppName("SimpleApp").setMaster
                                                              ("local");
    JavaSparkContext sc = new JavaSparkContext(conf);

    JavaRDD<String> logData = sc.textFile(logFile).cache();

    long numAs = logData.filter(s -> s.contains("a")).count();

    long numBs = logData.filter(s -> s.contains("b").count();

    System.out.println("Lines with a: " +numAs+ ", lines with b:"+numBs);

    sc.close();
  }
}
```

RDD Lineage

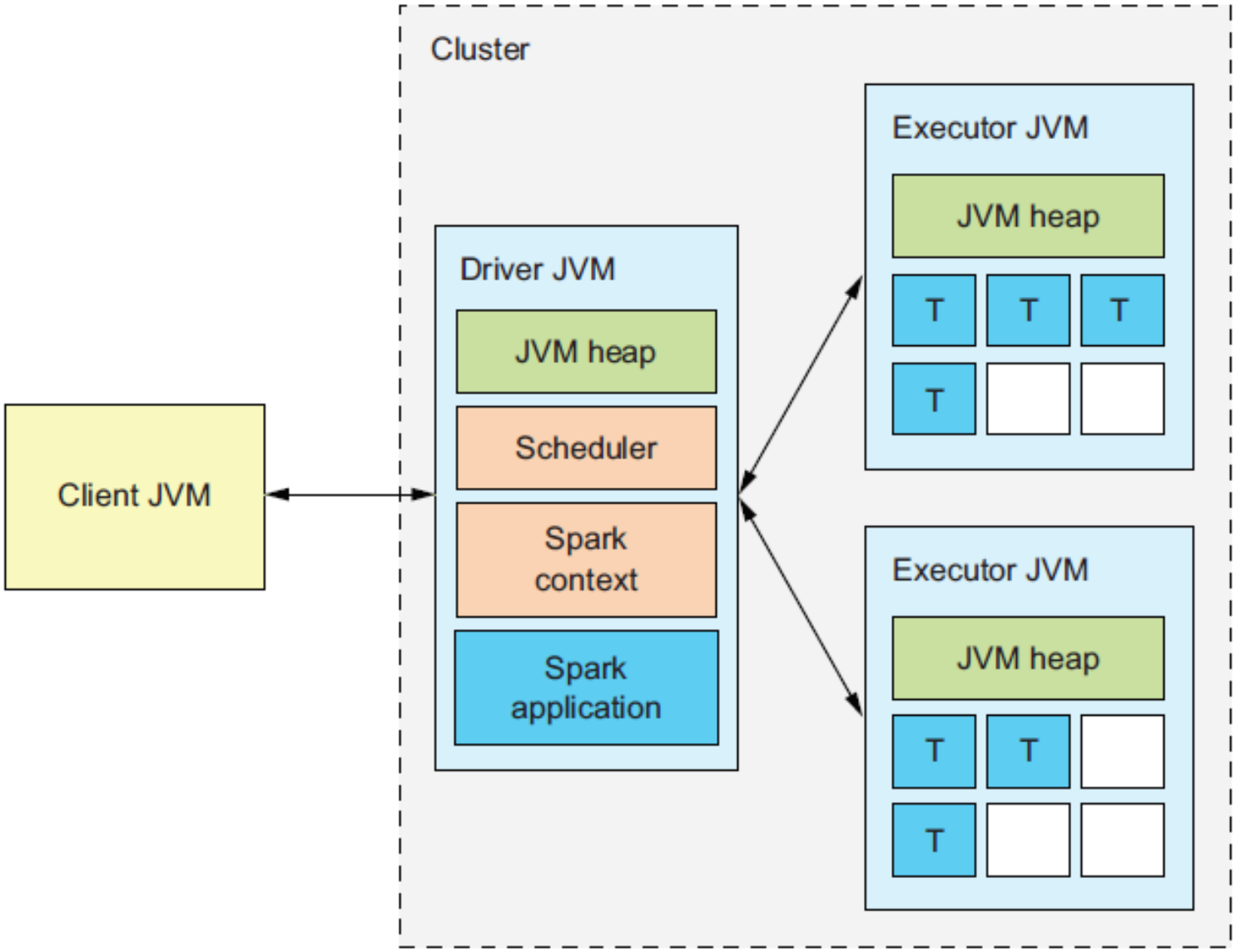
- Spark's execution model is based on *directed acyclic graphs* (DAGs) where RDDs are vertices and dependencies are edges.
- Every time a transformation is performed on an RDD, a new vertex (a new RDD) and a new edge (a dependency) are created.
- The new RDD depends on the old one, so the direction of the edge is from the child RDD to the parent RDD.
- This graph of dependencies is also called an RDD lineage.



Resilience of RDDs

- RDDs are *resilient* because of Spark's built-in fault recovery mechanics.
- Spark is capable of healing RDDs in case of node failure.
- RDDs provide fault tolerance by logging the transformations used to build a dataset (how it came to be) rather than the dataset itself.
- If a node fails, only a subset of the dataset that resided on the failed node needs to be recomputed.

Spark's Runtime Architecture



Spark's Runtime Components

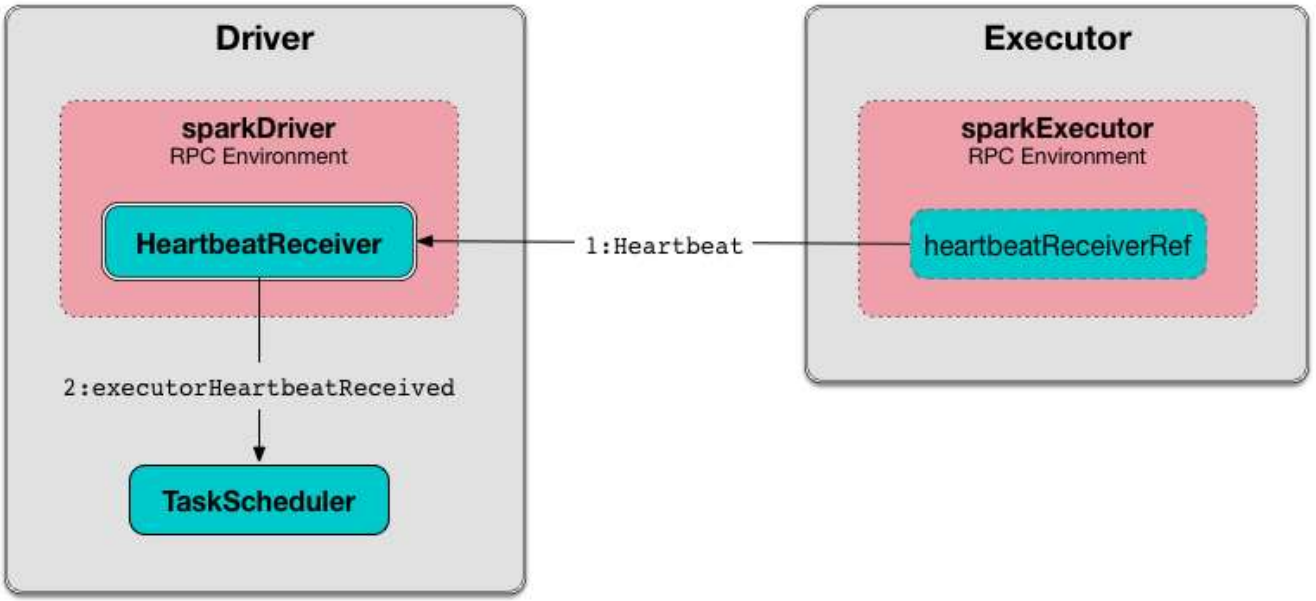
- **Client**: The *client process* starts the driver program.
  - The client process can be a spark-submit script for running applications, a spark-shell script, or a custom application using Spark API.
  - The client process prepares the classpath and all configuration options for the Spark application.
  - It also passes application arguments, if any, to the application running in the driver.

Spark's Runtime Components

- **DRIVER**: The driver orchestrates and monitors execution of a Spark application.
- There is always one driver per Spark application.
- You can think of the driver as a wrapper around the application.
- The driver and its subcomponents—the Spark context and scheduler - are responsible for the following:
  - Requesting memory and CPU resources from cluster managers
  - Breaking application logic into stages and tasks
  - Sending tasks to executors
  - Collecting the results

Spark's Runtime Components

- **Executors**: are JVM processes which accept tasks from the driver, execute those tasks, and return the results to the driver. There can be tens of thousands of executors in a Spark cluster.
  - Each executor has several *task slots* for running tasks in parallel.
  - Although these task slots are often referred to as CPU cores in Spark, they're implemented as threads and don't have to correspond to the number of physical CPU cores on the machine.
  - Resources for Spark applications are scheduled as executors (JVM processes) and CPU (task slots) and then memory is allocated to them.

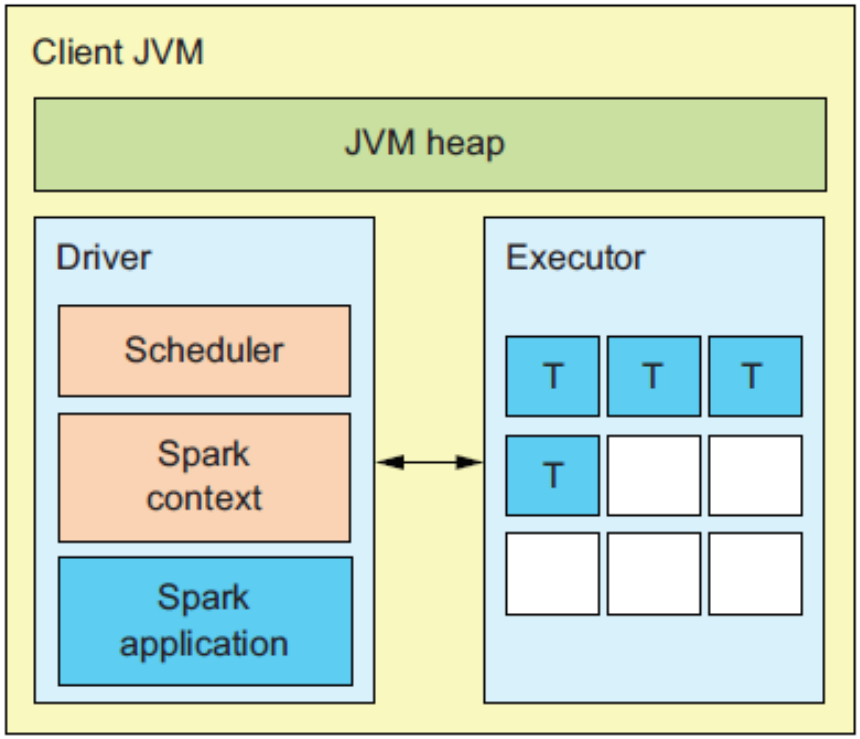


Creation Of The Spark Context

- Once the driver is started, it starts and configures an instance of SparkContext.
- When running a Spark REPL shell, the shell is the driver program.
- Your Spark context is already preconfigured and available as an sc variable.
- When running a standalone Spark application by submitting a JAR file or by using the Spark API from another program, your Spark application starts and configures the Spark context.
- There can be only one Spark context per JVM.
- It's the main interface for accessing the Spark runtime.
- It's also possible for several users (multiple threads) to use the same SparkContext object simultaneously (SparkContext is thread-safe). In that case, several jobs of the same SparkContext compete for its executors' resources.

Local Mode

- This mode is convenient for testing purposes when you don't have access to a full cluster or you want to try something out quickly.
- In local mode, there is only one executor in the same client JVM as the driver, but this executor can spawn several threads to run tasks.



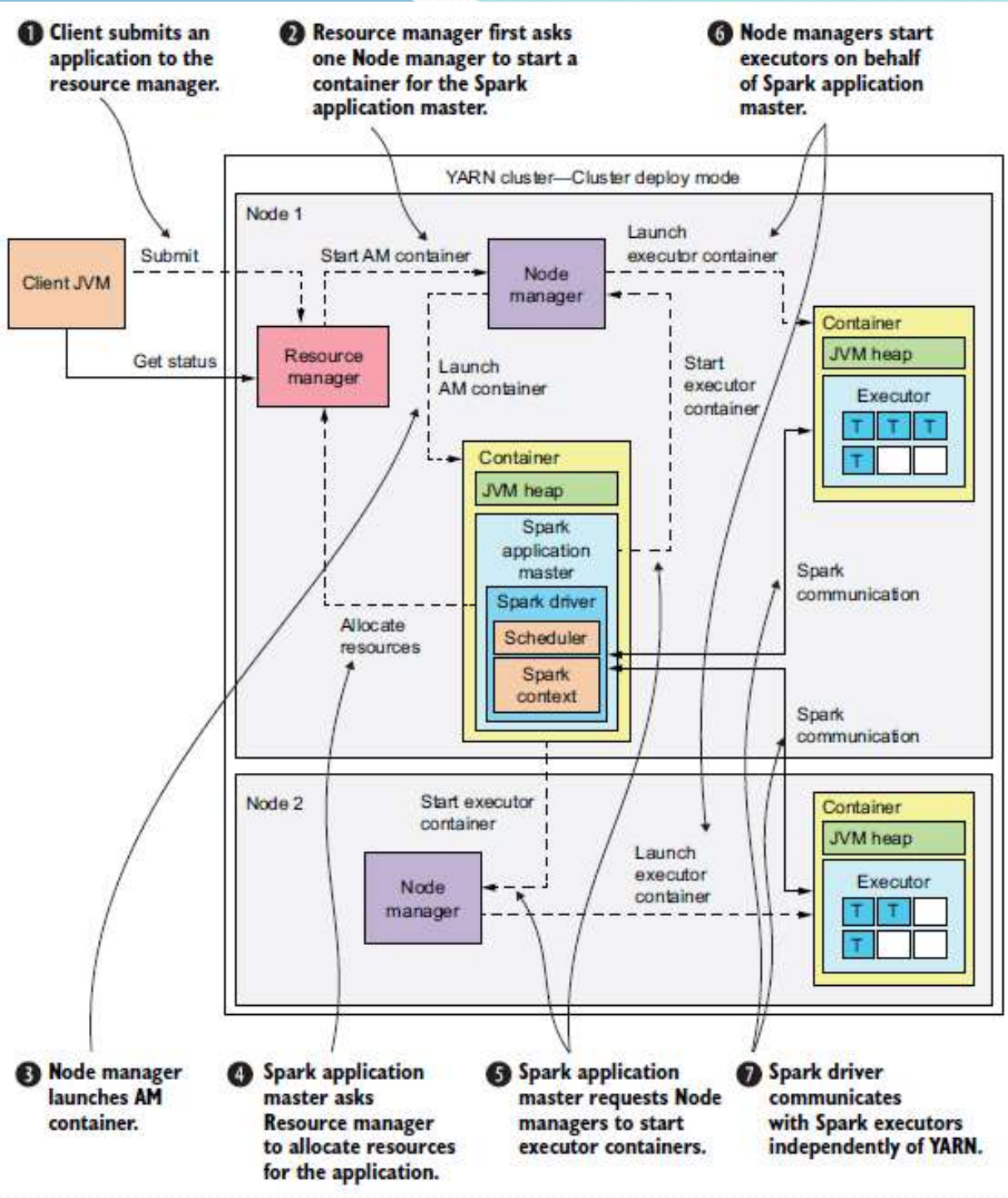
Local Mode

- To run Spark in local mode, set the master parameter to one of the following values:
- **local[<n>] -** Run a single executor using <n> threads, where <n> is a positive integer.
- **Local** - Run a single executor using one thread. This is the same as local[1].
- **local[*] -** Run a single executor using a number of threads equal to the number of CPU cores available on the local machine. In other words, use up all CPU cores.
- **local[<n>,<f>] -** Run a single executor using <n> threads, and allow a maximum of <f> failures per task. This is mostly used for Spark internal tests.
- If you start a spark-shell or spark-submit script with no --master parameter, (local[*]), local mode taking all CPU cores is assumed.

Local Mode

- NOTE:- If you use --master local with only one thread, you may notice that your log lines are missing from the driver's output.
- That's because in Spark Streaming, for example, that single thread is used to read streaming data from a source, and the driver wouldn't have any threads left to print out the results of your program.
- If you want the output printed to your log file, be sure to specify at least two threads (local[2]).

Spark On YARN



Executing Spark Applications

- There are two different ways you can interact with Spark.
  - **Static compiled programs**
    - Write a program in Scala, Java, or Python that uses Spark's library - that is, its API.
  - **REPL Shell -** use the Scala shell or the Python shell (pyspark).
    - Read – Eval – Print – Loop
    - REPL offers an interactive console that can be used for experimentation and idea testing.
    - There's no need for compilation and deployment just to find out something isn't working.
    - REPL can even be used for launching jobs on the full set of data.

Spark Shell

- A program written in the shell is discarded after you exit the shell.
- *Spark REPL* - It reads your input, evaluates it, prints the result, and then does it all over again—that is, after a command returns a result, it doesn't exit the scala> prompt; it stays ready for your next command (thus *loop*).
- When you start the shell, the SparkContext(sc) and the SQLContext (sqlContext) has already loaded.

```
$ spark-shell
SLF4J: Found binding in [jar:file:/usr/lib/flume-ng/lib/slf4j-log4j12-1.7.5.jar!
]
SLF4J: Found binding in [jar:file:/usr/lib/parquet/lib/slf4j-log4j12-1.7.5.jar!/
SLF4J: Found binding in [jar:file:/usr/lib/avro/avro-tools-1.7.6-cdh5.12.0.jar!/
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.6.0
      /_/

Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_67)
Type in expressions to have them evaluated.
Type :help for more information.
```

When to consider using Spark?

- Spark needs pretty good Java programming skills. So the first decision should be how technical is the person trying to analyze the data? If the person knows SQL and that's about it, Hive is the obvious choice.
- The next decision is around how fast the data needs to be processed. Hive and Pig use batch oriented frameworks, which means your analytics jobs will run for many minutes or hours. Spark is faster, but also much lower level.
- Lastly you need to see how well formed is the data. Are you using something like a CSV file? Or is it some kind of messy web log? If its well structured, you'll probably spend far less time preparing the data to be analyzed by just loading it into Hive. If there's a lot of parsing and structuring you need to do with the data, Pig and Spark should then be considered.
- Spark uses more RAM instead of network and disk I/O. As it uses large RAM it needs a dedicated high end physical machine for producing effective results. So the decision will keep on changing dynamically with time.

Hands on Spark

- Spark Word Count in Java
  - Maven project in eclipse
  - Java word count program
  - pom.xml file

- Even if you plan on only using Spark from Python, you have to install Java, because Spark's Python API communicates with Spark running in a JVM.

Spark Programming Guide